Outliers_in_DS

August 8, 2025

1 Outliers

1.1 What are Outliers?

We all have heard of the idiom 'odd one out which means something unusual in comparison to the others in a group. Similarly, an Outlier is an observation in a given dataset that lies far from the rest of the observations. That means an outlier treatment is vastly larger or smaller than the remaining values in the set.

1.2 Why Do they Occur?

An outlier may occur due to the variability in the data, or due to experimental error/human error. They may indicate an experimental error or heavy skewness in the data(heavy-tailed distribution).

1.3 What Do They Affect?

In statistics, we have three measures of central tendency namely Mean, Median, and Mode. They help us describe the data.

- Mean is the accurate measure to describe the data when we do not have any outliers present.
- Median is used if there is an outlier in the dataset.
- Mode is used if there is an outlier AND about ½ or more of the data is the same.

Mean' is the only measure of central tendency that is affected by the outlier treatment which in turn impacts Standard deviation.

1.4 Example

Consider a small dataset, sample= [15, 101, 18, 7, 13, 16, 11, 21, 5, 15, 10, 9]. By looking at it, one can quickly say '101' is an outlier that is much larger than the other values.

From the above calculations, we can clearly say the Mean is more affected than the Median.

1.5 Detecting Outliers

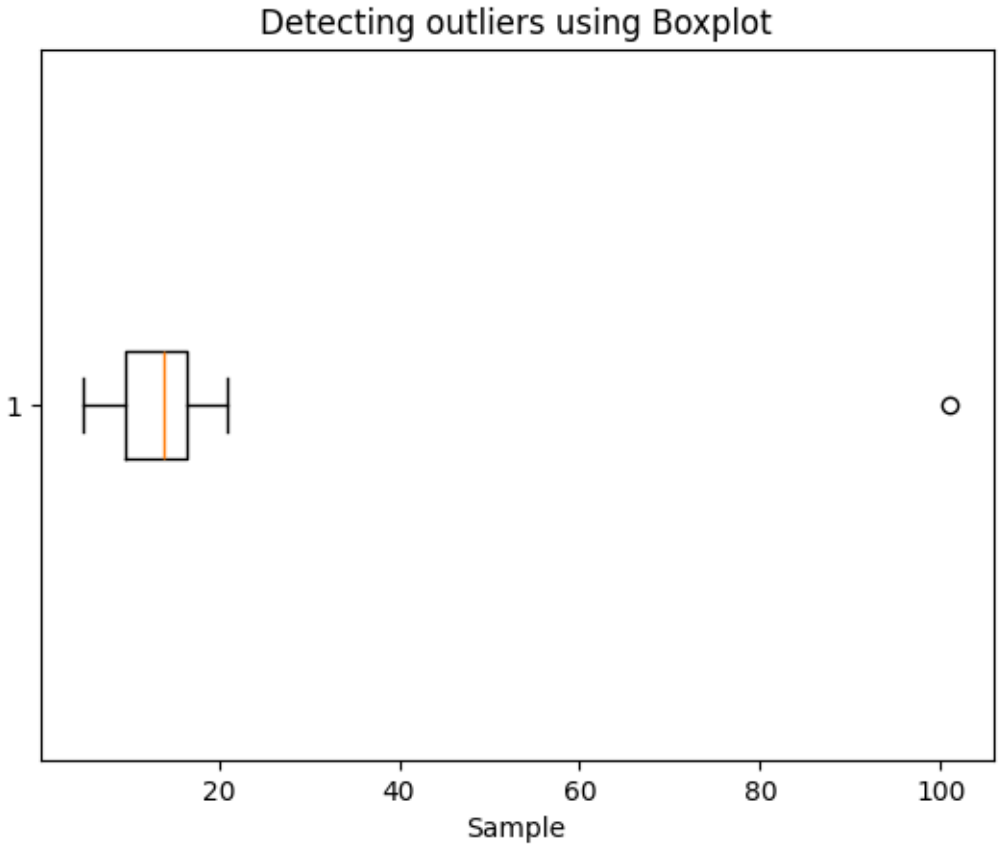
If our dataset is small, we can detect the outlier by just looking at the dataset. But what if we have a huge dataset, how do we identify the outliers then? We need to use visualization and mathematical techniques.

Below are some of the techniques of detecting outliers

- Boxplots
- Z-score
- Inter Quantile Range(IQR)

1.5.1 1.Detecting Outliers Using Boxplot

[ ]:
```
import matplotlib.pyplot as plt

sample= [15, 101, 18, 7, 13, 16, 11, 21, 5, 15, 10, 9]
plt.boxplot(sample, vert=False)
plt.title("Detecting outliers using Boxplot")
plt.xlabel('Sample')
plt.show()
```



1.5.2 2.Detecting Outliers using the Z-scores

Criteria: any data point whose Z-score falls out of 3rd standard deviation is an outlier treatment.

Steps: * Loop through all the data points and compute the Z-score using the formula (Xi-

mean)/std. * Define a threshold value of 3 and mark the datapoints whose absolute value of Z-score is greater than the threshold as outliers.

[ ]:
```
import numpy as np
outliers = []
```

[ ]:
```
# titanic.dropna(axis=0, inplace=True)
titanic['Age'] = titanic['Age'].replace(np.nan, titanic['Age'].mean()
data = list(titanic['Age'])
```

[ ]:
```
def detect_outliers_zscore(data):
    thres = 2
    mean = np.mean(data)
    std = np.std(data)
    # print(mean, std)
    for i in data:
        z_score = (i-mean)/std
        print(z_score)
        if (np.abs(z_score) > thres):
            outliers.append(i)
    return outliers# Driver code
```

[ ]:
```
sample_outliers = detect_outliers_zscore(data)
print("Outliers from Z-scores method: ", sample_outliers)
```

```
-0.20502261723677698
3.2635567432280412
-0.08402566280195775
-0.5276811623962949
-0.28568725352665647
-0.16469029909183724
-0.366351889816536
0.03697129163286148
-0.6083457986861744
-0.20502261723677698
-0.4066842079614757
-0.44701652610641546
Outliers from Z-scores method:  [101]
```

1.5.3 3.Detecting Outliers using the Inter Quantile Range(IQR)

Criteria: Data points that lie 1.5 times of IQR above Q3 and below Q1 are outliers. This shows in detail about outlier treatment in Python.

Steps:

- Sort the dataset in ascending order

- Calculate the 1st and 3rd quartiles(Q1, Q3)
- Compute IQR=Q3-Q1
- Compute lower bound = (Q1–1.5*IQR), upper bound = (Q3+1.5*IQR)
- Loop through the values of the dataset and check for those who fall below the lower bound and above the upper bound and mark them as outlier treatment in python.

[ ]:
```
outliers_iqr = []
```

[ ]:
```
def detect_outliers_iqr(data):
    data = sorted(data)
    q1 = np.percentile(data, 25)
    q3 = np.percentile(data, 75)
    print(q1, q3)
    IQR = q3-q1
    print(IQR)
    lwr_bound = q1-(1.5*IQR)
    upr_bound = q3+(1.5*IQR)
    print(lwr_bound, upr_bound)
    for i in data:
        if (i<lwr_bound or i>upr_bound):
            outliers_iqr.append(i)
    return outliers_iqr# Driver code
```

[ ]:
```
sample_outliers = detect_outliers_iqr(sample)
print("Outliers from IQR method: ", sample_outliers)
```

```
9.75 16.5
6.75
-0.375 26.625
Outliers from IQR method:  [101]
```

1.6 How to Handle Outliers?

Till now we learned about detecting the outliers handling. The main question is how to deal with outliers?

Below are some of the methods of treating the outliers:

1.6.1 Step 1: Trimming/Remove the outliers

In this technique, we remove the outliers from the dataset. Although it is not a good practice to follow.

Python code to delete the outlier treatment and copy the rest of the elements to another array.

[ ]:
```
# Trimming
sample_trimmed_1 = [x for x in data if x not in outliers] # Use list␣
 ↪comprehension to create a new list without outliers
sample_trimmed_2 = [x for x in data if x not in outliers_iqr] # Use list␣
 ↪comprehension to create a new list without outliers
```

```
# print(sample_trimmed)
print(len(data))
print(len(sample_trimmed_1))
print(len(sample_trimmed_2))
```

```
[15, 18, 7, 13, 16, 11, 21, 5, 15, 10, 9]
11
```

1.6.2 Step 2: Quantile Based Flooring and Capping (Winsorization)

Winsorization is a statistical technique used to reduce the impact of outliers on data analysis. It involves replacing extreme values in a dataset with less extreme values, typically those at a certain percentile, rather than removing them entirely. This creates a winsorized dataset, which is then used for further calculations, like finding a winsorized mean.

In this technique, the outlier is capped at a certain value above the 90th percentile value or floored at a factor below the 10th percentile value. Python code to delete the outlier and copy the rest of the elements to another array.

[ ]:
```
# Computing 10th, 90th percentiles and replacing the outlier treatment in python
# print(sample)
tenth_percentile = np.percentile(data, 10)
ninetieth_percentile = np.percentile(data, 90)
print(tenth_percentile, ninetieth_percentile)
```

```
[15, 101, 18, 7, 13, 16, 11, 21, 5, 15, 10, 9]
7.2 20.700000000000003
```

[ ]:
```
# from scipy.stats.mstats import winsorize

# # Apply winsorization
# new_sample = sample.copy()
# new_sample = winsorize(new_sample, limits=[tenth_percentile,␣
 ↪ninetieth_percentile])
```

[ ]:
```
b = np.where(data<tenth_percentile, tenth_percentile, sample)
b = np.where(b>ninetieth_percentile, ninetieth_percentile, b)
print("Sample:", sample)
print("New array:",b)
```

```
Sample: [15, 101, 18, 7, 13, 16, 11, 21, 5, 15, 10, 9]
New array: [15.  20.7 18.   7.2 13.  16.  11.  20.7  7.2 15.  10.   9. ]
```

The data points that are lesser than the 10th percentile are replaced with the 10th percentile value and the data points that are greater than the 90th percentile are replaced with 90th percentile value.

1.6.3 Step 3: Mean/Median Imputation

As the mean value is highly influenced by the outlier treatment, it is advised to replace the outliers with the median value.

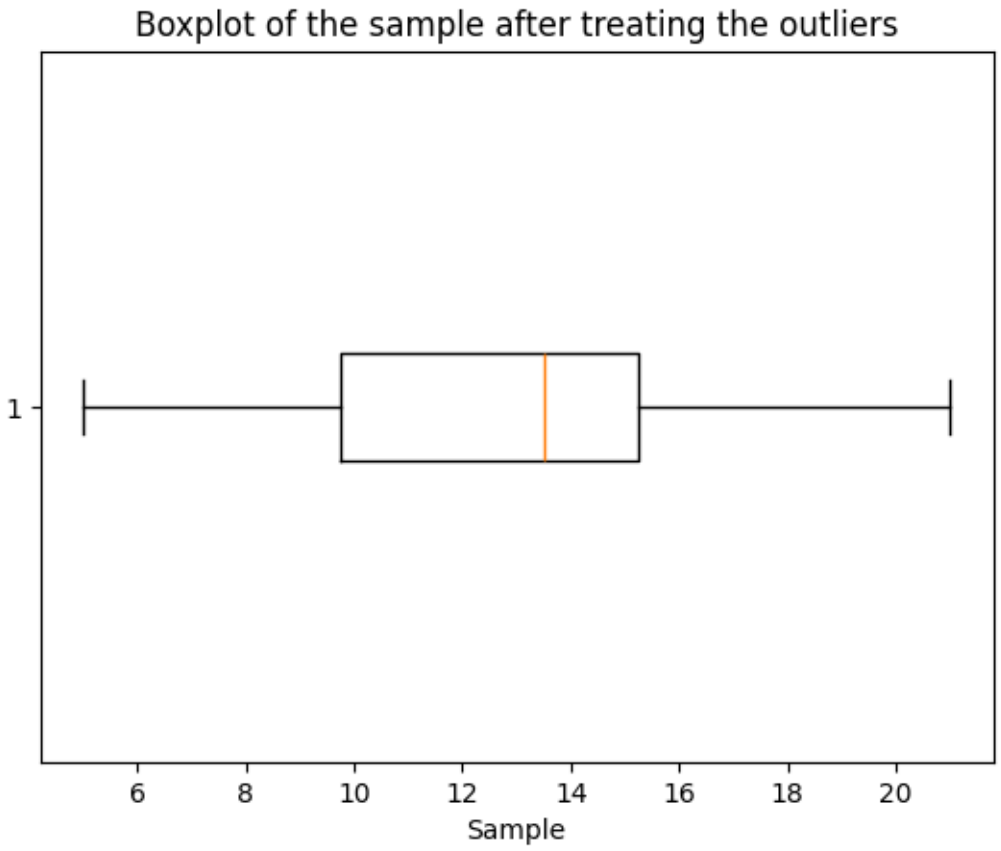
[ ]:
```
median = np.median(sample)
print(sample_outliers)
# Replace with median (using a more efficient method)
c = np.where(np.isin(sample, sample_outliers), median, sample)

print("Sample: ", sample)
print("New array: ", c)
```

```
[101]
Sample:  [15, 101, 18, 7, 13, 16, 11, 21, 5, 15, 10, 9]
New array:  [15. 14. 18.  7. 13. 16. 11. 21.  5. 15. 10.  9.]
```

1.7 Visualizing the Data after Treating the Outlier

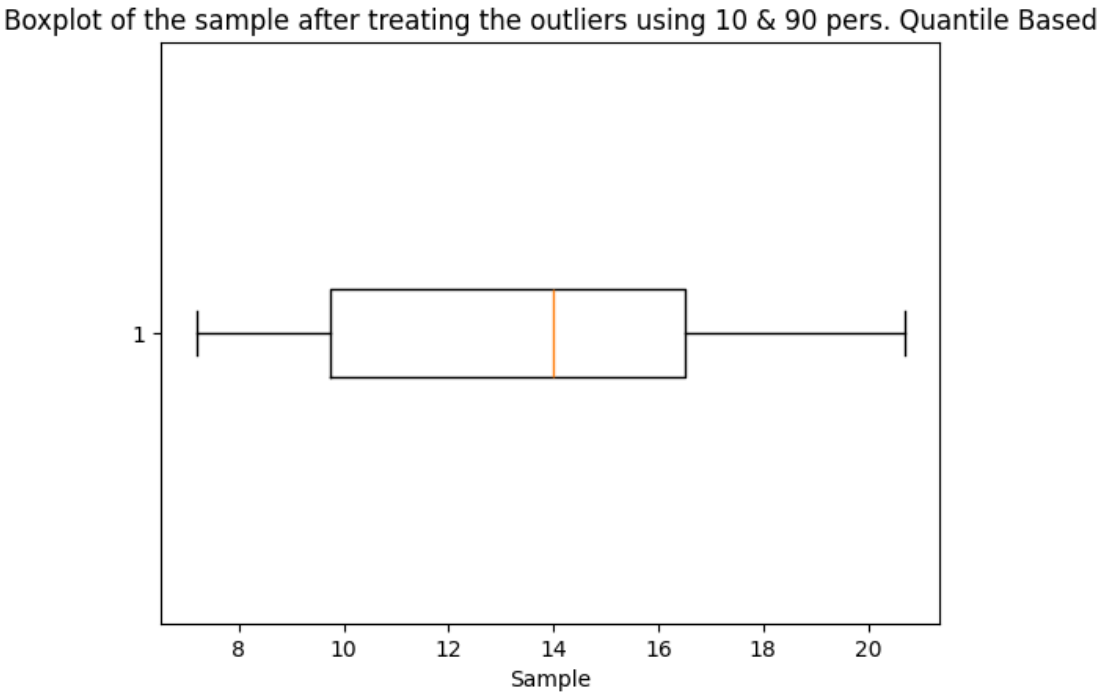
[ ]:
```
plt.boxplot(c, vert=False)
plt.title("Boxplot of the sample after treating the outliers")
plt.xlabel("Sample")
```

[ ]:
```
Text(0.5, 0, 'Sample')
```



[ ]:
```
plt.boxplot(b, vert=False)
plt.title("Boxplot of the sample after treating the outliers using 10 & 90 pers.
  ↪ Quantile Based")
plt.xlabel("Sample")
```

[ ]:
```
Text(0.5, 0, 'Sample')
```



[ ]:
```

```

Central_Limit_Theorem

August 8, 2025

1 Central Limit Theorem

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```
df = pd.read_csv("/content/drive/MyDrive/PPSU/Clt-data.csv")
df
```

[ ]:
```
      Wall Thickness
0          12.354875
1          12.617417
2          12.369719
3          13.223345
4          13.159193
…                …
8995       12.995322
8996       13.060026
8997       12.795001
8998       12.777421
8999       13.014160

[9000 rows x 1 columns]
```

```
<google.colab._quickchart_helpers.SectionTitle at 0x7dcbbbe27410>
```

```
from matplotlib import pyplot as plt
df['Wall Thickness'].plot(kind='hist', bins=20, title='Wall Thickness')
plt.gca().spines[['top', 'right',]].set_visible(False)
```

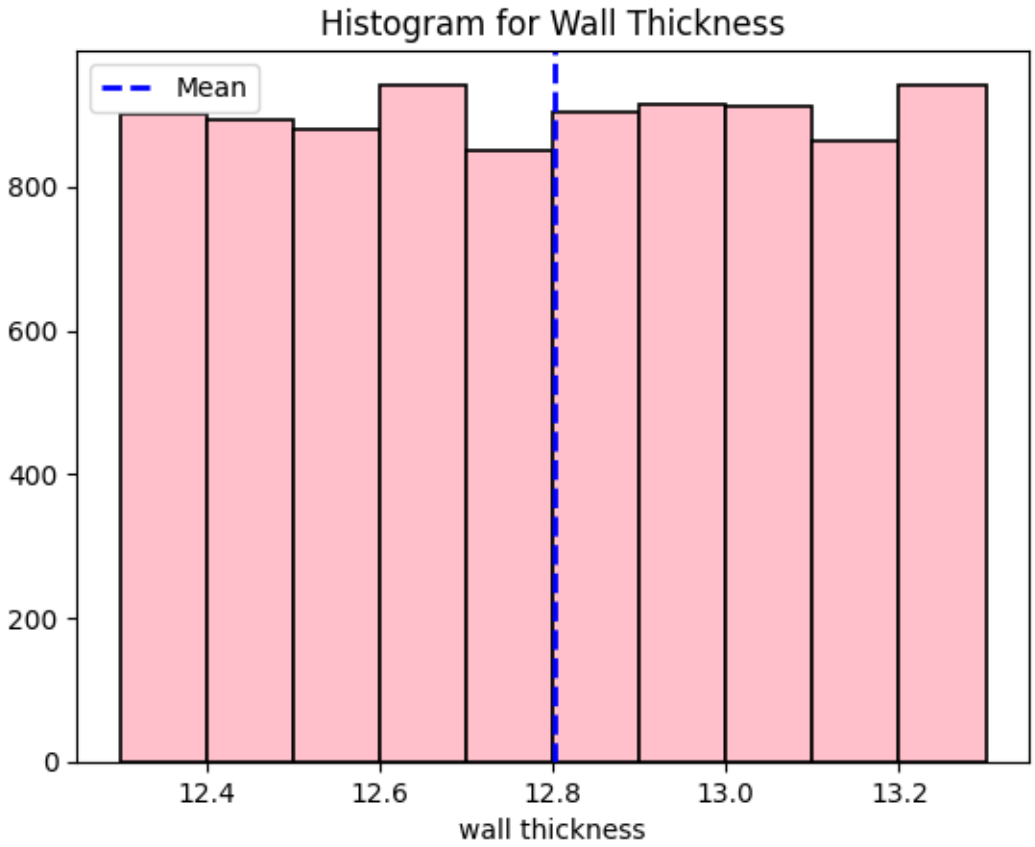
```
<google.colab._quickchart_helpers.SectionTitle at 0x7dcbba5e5e90>
```

```
from matplotlib import pyplot as plt
df['Wall Thickness'].plot(kind='line', figsize=(8, 4), title='Wall Thickness')
plt.gca().spines[['top', 'right']].set_visible(False)
```

[ ]:
```
#Calculate the population mean
pop_mean = np.mean(df['Wall Thickness'])
print(pop_mean)
```

```
12.802049245535558
```

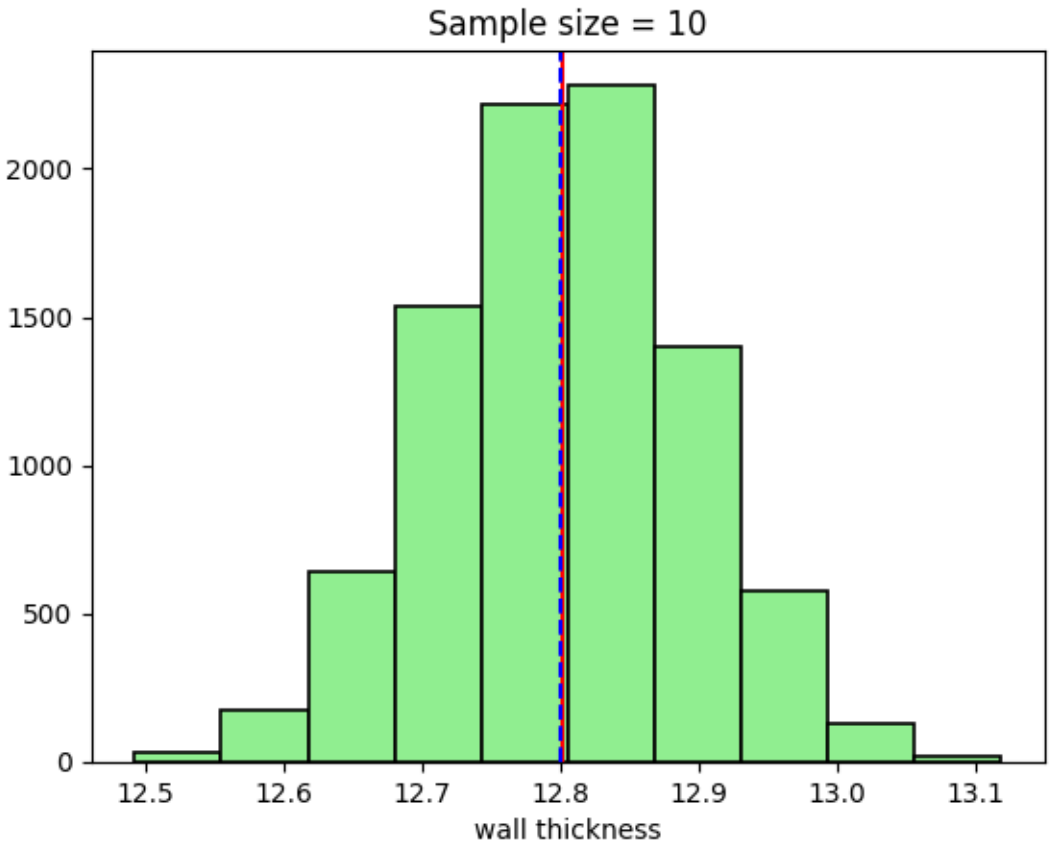
[ ]:
```
#Plot all the observations in the data
plt.hist(df["Wall Thickness"],color= "pink", edgecolor = "black", linewidth=1.2)
plt.title("Histogram for Wall Thickness")
plt.xlabel("wall thickness")
# draw a vertical line at the mean
plt.axvline(pop_mean, color='blue', linestyle='dashed', linewidth=2,␣
 ↪label='Mean')
plt.legend()
plt.show()
```



[ ]:
```
s10 = []
n = 9000
for i in range(n):
```

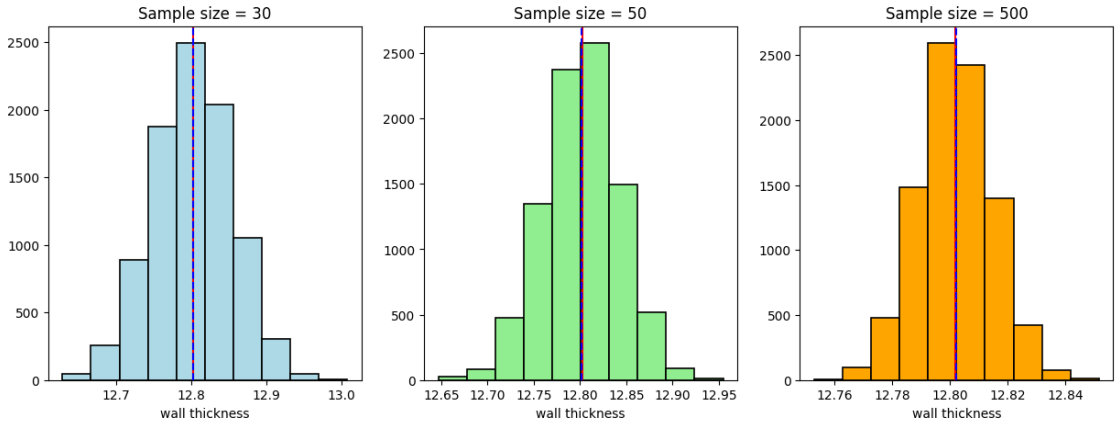
```
  s10.append(np.mean(np.random.choice(df['Wall Thickness'], size=10,␣
 ↪replace=True)))
```

[ ]:
```
#We will take sample size=10, samples=9000
#Calculate the arithmetice mean and plot the mean of sample 9000 times

# s10<-c()
# n=9000
# for (i in 1:n) {
# s10[i] = mean(sample(data$Wall.Thickness,10, replace = TRUE))}
# hist(s10, col ="lightgreen", main="Sample size =10",xlab = "wall thickness")
# abline(v = mean(s10), col = "Red")
# abline(v = 12.8, col = "blue")


plt.hist(s10, color='lightgreen', edgecolor = "black", linewidth=1.2)
plt.title('Sample size = 10')
plt.xlabel('wall thickness')
plt.axvline(np.mean(s10), color='red')
plt.axvline(12.8, color='blue', linestyle='dashed')
plt.show()
```



[ ]:
```
s30 = []
s50 = []
s500 = []
n = 9000
for i in range(n):
  s30.append(np.mean(np.random.choice(df['Wall Thickness'], size=30,␣
 ↪replace=True)))
  s50.append(np.mean(np.random.choice(df['Wall Thickness'], size=50,␣
 ↪replace=True)))
  s500.append(np.mean(np.random.choice(df['Wall Thickness'], size=500,␣
 ↪replace=True)))
```

[ ]:
```
plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
plt.hist(s30, color ="lightblue", edgecolor = "black", linewidth=1.2)
plt.title('Sample size = 30')
plt.xlabel('wall thickness')
plt.axvline(np.mean(s30), color='red')
plt.axvline(pop_mean, color='blue', linestyle='dashed')

plt.subplot(1, 3, 2)
plt.hist(s50, color ="lightgreen", edgecolor = "black", linewidth=1.2)
plt.title('Sample size = 50')
plt.xlabel('wall thickness')
plt.axvline(np.mean(s50), color='red')
plt.axvline(pop_mean, color='blue', linestyle='dashed')

plt.subplot(1, 3, 3)
plt.hist(s500, color ="orange", edgecolor = "black", linewidth=1.2)
plt.title('Sample size = 500')
plt.xlabel('wall thickness')
plt.axvline(np.mean(s500), color='red')
plt.axvline(pop_mean, color='blue', linestyle='dashed')

plt.show()
```


Sample size = 30
2500
2000
1500
1000
500
0
12.7
12.8
12.9
13.0
wall thickness
Sample size = 50
2500
2000
1500
1000
500
0
12.65
12.70
12.75
12.80
12.85
12.90
12.95
wall thickness
Sample size = 500
2500
2000
1500
1000
500
0
12.76
12.78
12.80
12.82
12.84
wall thickness

[ ]:

Central Limit Theorem

August 8, 2025

[17]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

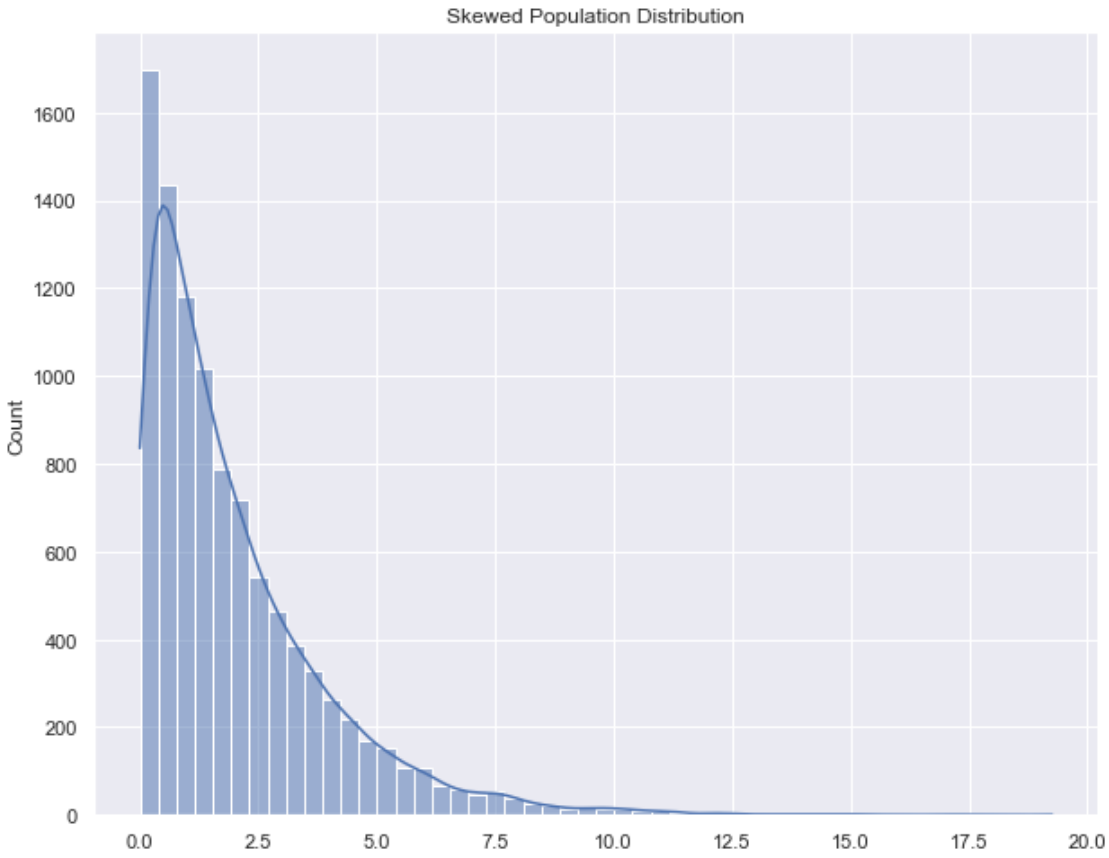
0.0.1 Generate a right-skewed population of 10,000 data points using numpy.random.exponential(scale=2.0).

[18]:
```
# Step a: Create a skewed population
population = np.random.exponential(scale=2.0, size=10000)
population
```

[18]:
```
array([1.83927822, 4.84749774, 0.70896141, …, 2.49130997, 0.26468994,
       0.32092804])
```

0.0.2 Plot the histogram of the population distribution.

[4]:
```
# Step b: Plot population distribution
sns.set(rc={'figure.figsize':(10, 8)})
sns.histplot(population, kde=True, bins=50)
plt.title("Skewed Population Distribution")
plt.show()
```



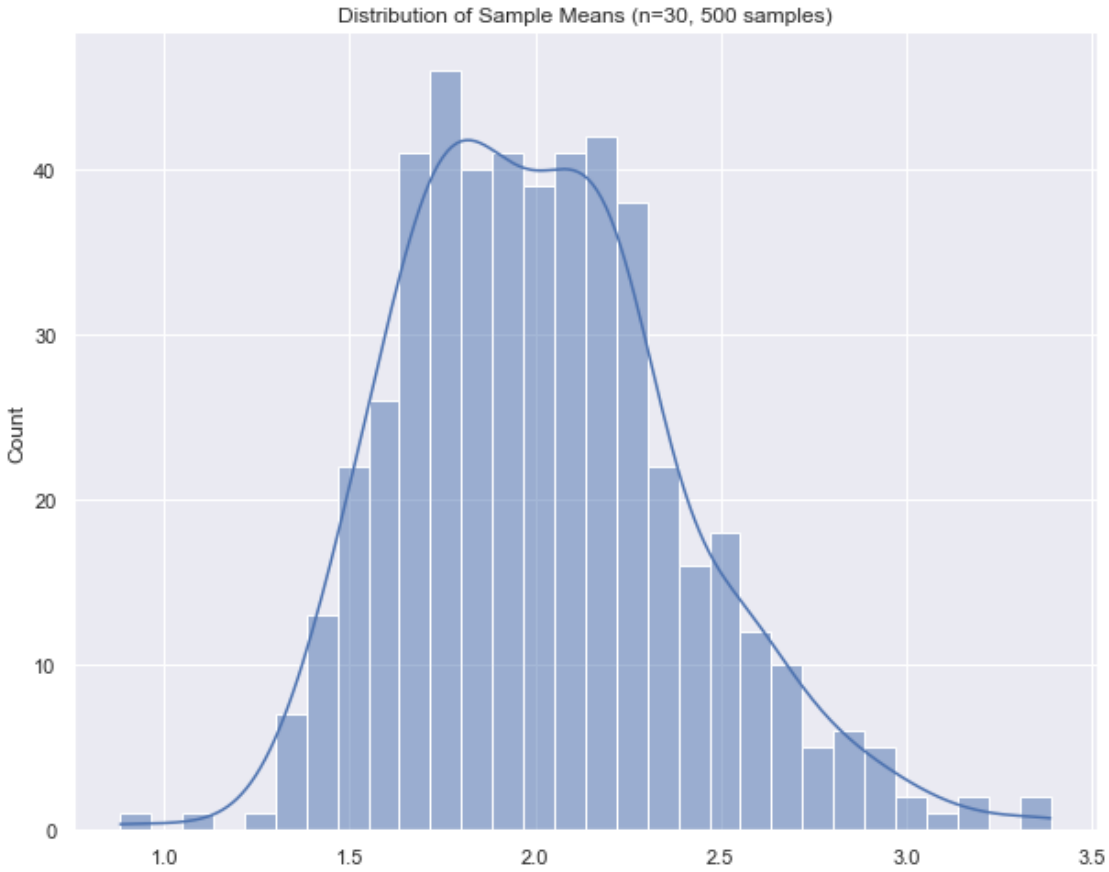
1 Take 500 samples, each of size 30, and compute their means.

[5]:
```
# Step c & d: Sampling
sample_means = []
for _ in range(500):
    sample = np.random.choice(population, size=30)
    sample_means.append(np.mean(sample))

# print(sample_means)
print(len(sample_means))
```

```
500
```

1.0.1 Plot the histogram of these sample means and show that it is approximately normal.

[6]:
```
# Plot sample mean distribution
sns.histplot(sample_means, kde=True, bins=30)
plt.title("Distribution of Sample Means (n=30, 500 samples)")
plt.show()
```



1.0.2 Compare the population mean and the mean of sample means.

[7]:
```
# Step e: Compare means
print("Population Mean:", np.mean(population))
print("Mean of Sample Means:", np.mean(sample_means))
```

```
Population Mean: 1.9987165723454539
Mean of Sample Means: 2.0188857627755006
```

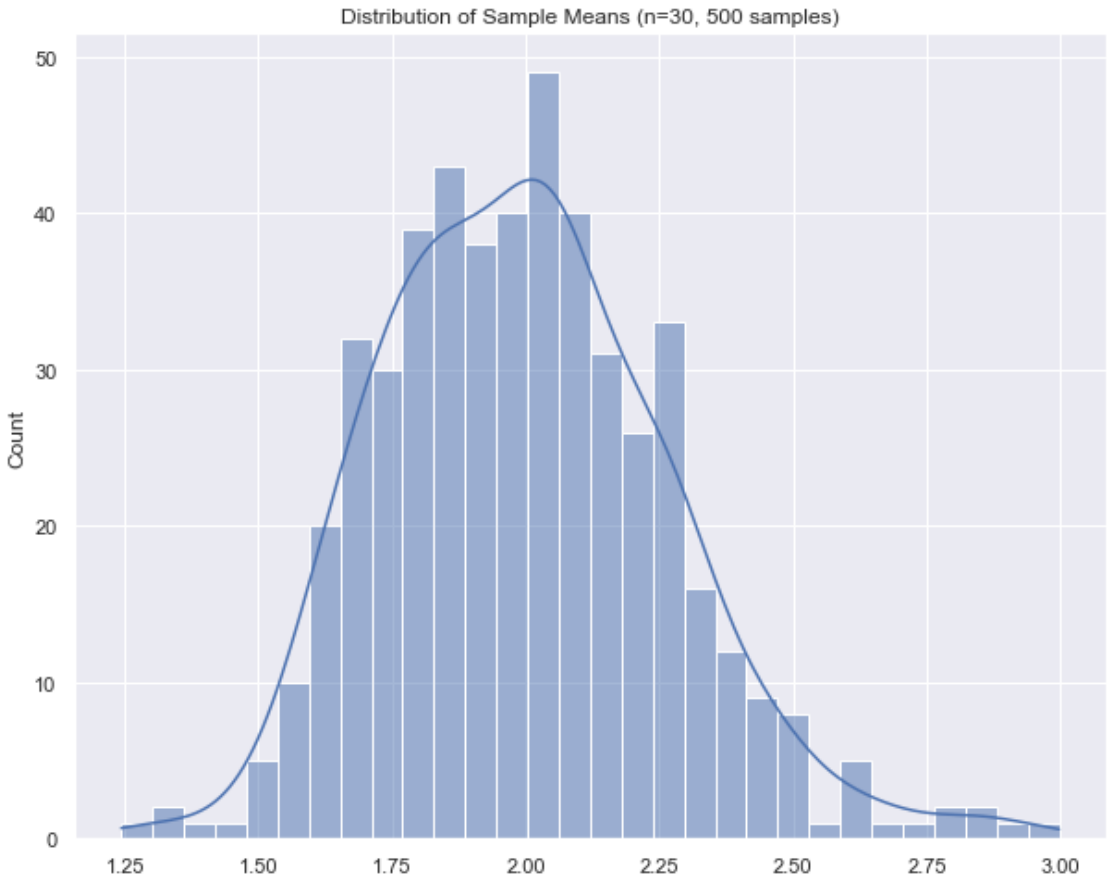
2 Take 500 samples, each of size 50, and compute their means.

[11]:
```
# Step c & d: Sampling
sample_means_2 = []
for _ in range(500):
    sample = np.random.choice(population, size=50)
    sample_means_2.append(np.mean(sample))

print(len(sample_means_2))
```

```
500
```

2.0.1 Plot the histogram of these sample means and show that it is approximately normal.

[12]:
```
# Plot sample mean distribution
sns.histplot(sample_means_2, kde=True, bins=30)
plt.title("Distribution of Sample Means (n=30, 500 samples)")
plt.show()
```



2.0.2 Compare the population mean and the mean of sample means.

[13]:
```
# Step e: Compare means
print("Population Mean:", np.mean(population))
print("Mean of Sample Means:", np.mean(sample_means_2))
```

```
Population Mean: 1.9987165723454539
Mean of Sample Means: 1.9968908223833546
```

3 Real Data CLT Exploration

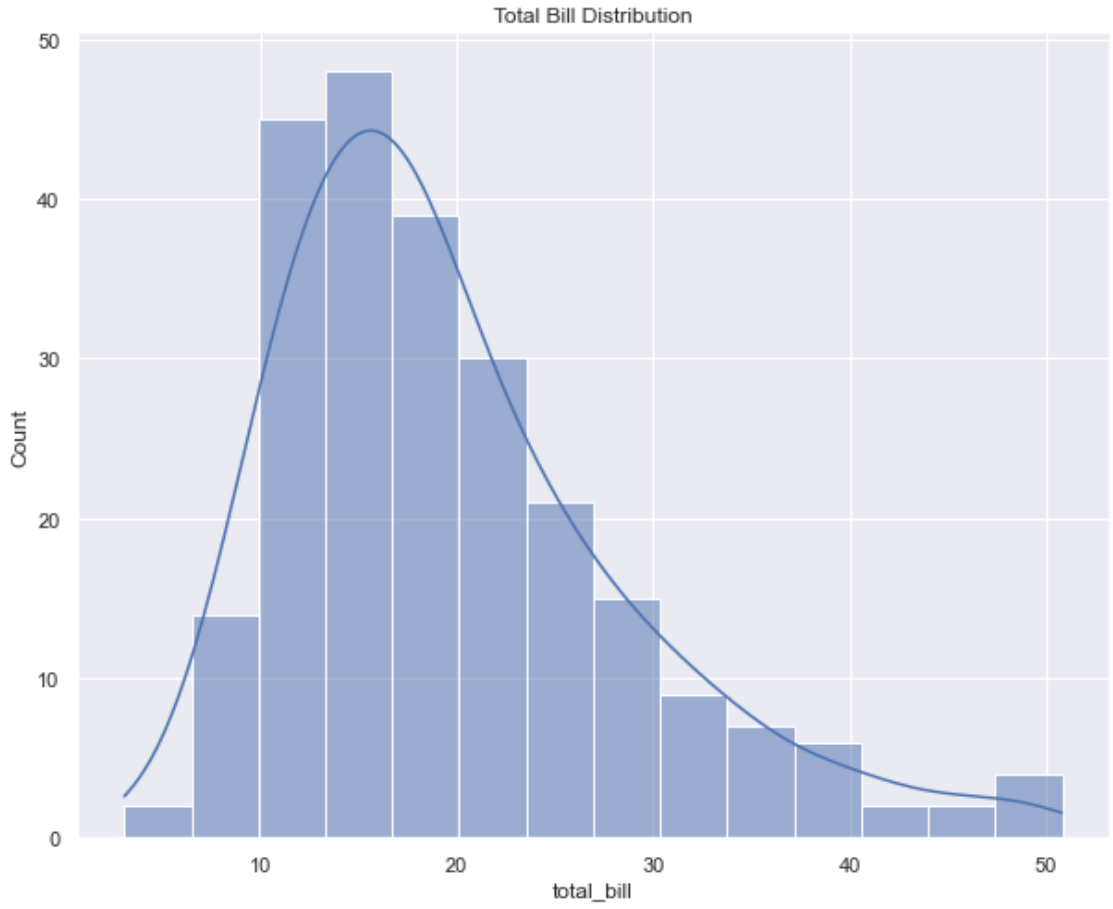
[19]:
```
# Load data
tips = sns.load_dataset('tips')
total_bill = tips['total_bill']
tips
```

[19]:

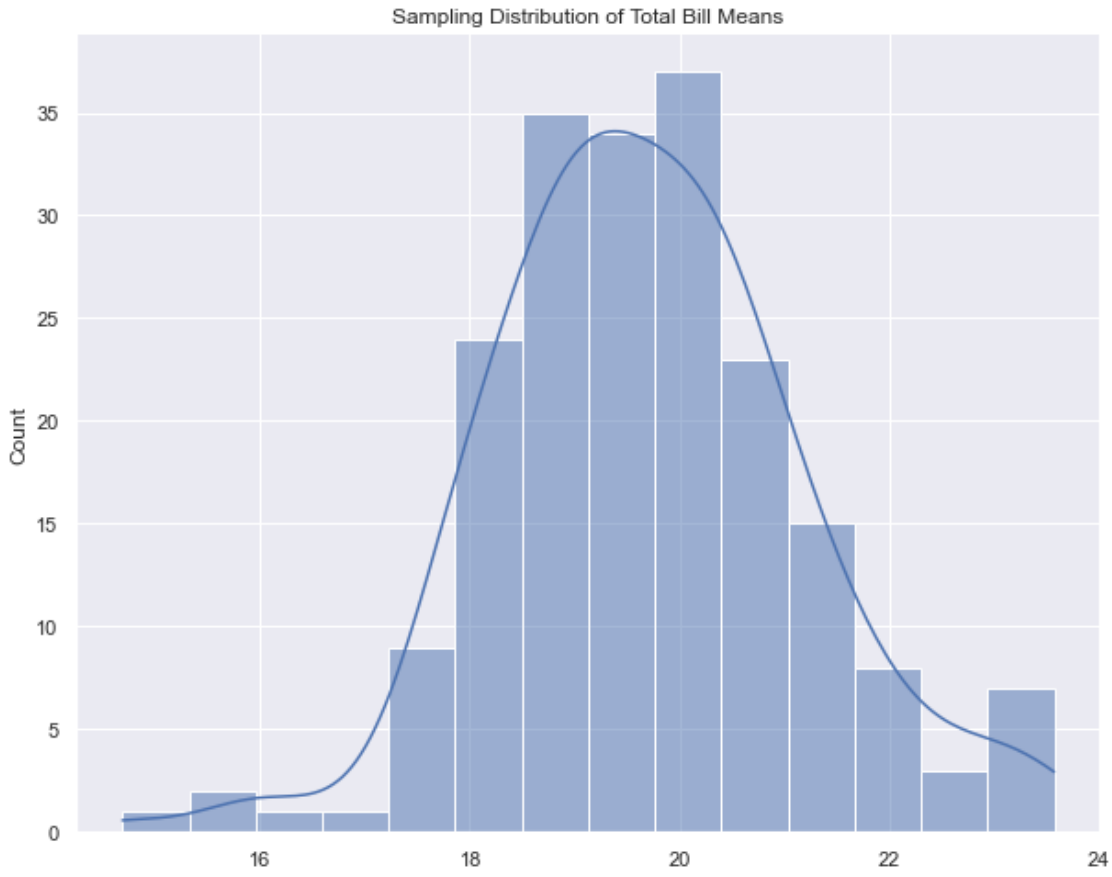
	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4
..	…	…	…	…	…	…	…
239	29.03	5.92	Male	No	Sat	Dinner	3
240	27.18	2.00	Female	Yes	Sat	Dinner	2
241	22.67	2.00	Male	Yes	Sat	Dinner	2
242	17.82	1.75	Male	No	Sat	Dinner	2
243	18.78	3.00	Female	No	Thur	Dinner	2

[244 rows x 7 columns]

[15]:
```
# a) Plot population distribution
sns.histplot(total_bill, kde=True)
plt.title("Total Bill Distribution")
plt.show()
```



[20]:
```
# b & c) Sampling
sample_means_tips = [np.mean(np.random.choice(total_bill, size=40)) for _ in␣
↪range(200)]
sns.histplot(sample_means_tips, kde=True)
plt.title("Sampling Distribution of Total Bill Means")
plt.show()
```



[21]:
```
# d) Print and comment
print("Population Mean:", np.mean(total_bill))
print("Mean of Sample Means:", np.mean(sample_means_tips))
```

```
Population Mean: 19.785942622950824
Mean of Sample Means: 19.69337625
```

[ ]:
```

```

Linear_Regression

August 8, 2025

1 Linear Regression

In the most simple words, Linear Regression is the supervised Machine Learning model in which the model finds the best fit linear line between the independent and dependent variable i.e it finds the linear relationship between the dependent and independent variable.

Linear Regression is of two types: Simple and Multiple. Simple Linear Regression is where only one independent variable is present and the model has to find the linear relationship of it with the dependent variable

Whereas, In Multiple Linear Regression there are more than one independent variables for the model to find the relationship.

Equation of Simple Linear Regression, where bo is the intercept, b1 is coefficient or slope, x is the independent variable and y is the dependent variable.

Equation of Multiple Linear Regression, where bo is the intercept, b1,b2,b3,b4…,bn are coefficients or slopes of the independent variables x1,x2,x3,x4…,xn and y is the dependent variable.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

[3]:
```
# Generating some random data
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

[4]:
```
X.shape
```

[4]:
```
(100, 1)
```

[5]:
```
print("X values:", X)
print("y valies: ", y)
```

```
X values: [[0.74908024]
 [1.90142861]
 [1.46398788]
 [1.19731697]
 [0.31203728]
 [0.31198904]
 [0.11616722]
 [1.73235229]
 [1.20223002]
 [1.41614516]
 [0.04116899]
 [1.9398197 ]
 [1.66488528]
 [0.42467822]
 [0.36364993]
 [0.36680902]
 [0.60848449]
 [1.04951286]
 [0.86389004]
 [0.58245828]
 [1.22370579]
 [0.27898772]
 [0.5842893 ]
 [0.73272369]
 [0.91213997]
 [1.57035192]
 [0.39934756]
 [1.02846888]
 [1.18482914]
 [0.09290083]
 [1.2150897 ]
 [0.34104825]
 [0.13010319]
 [1.89777107]
 [1.93126407]
 [1.6167947 ]
 [0.60922754]
 [0.19534423]
 [1.36846605]
 [0.88030499]
 [0.24407647]
 [0.99035382]
 [0.06877704]
 [1.8186408 ]
 [0.51755996]
```

[1.32504457]
[0.62342215]
[1.04013604]
[1.09342056]
[0.36970891]
[1.93916926]
[1.55026565]
[1.87899788]
[1.7896547]
[1.19579996]
[1.84374847]
[0.176985]
[0.39196572]
[0.09045458]
[0.65066066]
[0.77735458]
[0.54269806]
[1.65747502]
[0.71350665]
[0.56186902]
[1.08539217]
[0.28184845]
[1.60439396]
[0.14910129]
[1.97377387]
[1.54448954]
[0.39743136]
[0.01104423]
[1.63092286]
[1.41371469]
[1.45801434]
[1.54254069]
[0.1480893]
[0.71693146]
[0.23173812]
[1.72620685]
[1.24659625]
[0.66179605]
[0.1271167]
[0.62196464]
[0.65036664]
[1.45921236]
[1.27511494]
[1.77442549]
[0.94442985]
[0.23918849]
[1.42648957]
[1.5215701]

```
 [1.1225544 ]
 [1.54193436]
 [0.98759119]
 [1.04546566]
 [0.85508204]
 [0.05083825]
 [0.21578285]]
y valies:  [[ 6.33428778]
 [ 9.40527849]
 [ 8.48372443]
 [ 5.60438199]
 [ 4.71643995]
 [ 5.29307969]
 [ 5.82639572]
 [ 8.67878666]
 [ 6.79819647]
 [ 7.74667842]
 [ 5.03890908]
 [10.14821022]
 [ 8.46489564]
 [ 5.7873021 ]
 [ 5.18802735]
 [ 6.06907205]
 [ 5.12340036]
 [ 6.82087644]
 [ 6.19956196]
 [ 4.28385989]
 [ 7.96723765]
 [ 5.09801844]
 [ 5.75798135]
 [ 5.96358393]
 [ 5.32104916]
 [ 8.29041045]
 [ 4.85532818]
 [ 6.28312936]
 [ 7.3932017 ]
 [ 4.68275333]
 [ 9.53145501]
 [ 5.19772255]
 [ 4.64785995]
 [ 9.61886731]
 [ 7.87502098]
 [ 8.82387021]
 [ 5.88791282]
 [ 7.0492748 ]
 [ 7.91303719]
 [ 6.9424623 ]
 [ 4.69751764]
```

```
[ 5.80238342]
[ 5.34915394]
[10.20785545]
[ 6.34371184]
[ 7.06574625]
[ 7.27306077]
[ 5.71855706]
[ 7.86711877]
[ 7.29958236]
[ 8.82697144]
[ 8.08449921]
[ 9.73664501]
[ 8.86548845]
[ 6.03673644]
[ 9.59980838]
[ 3.4686513 ]
[ 5.64948961]
[ 3.3519395 ]
[ 7.50191639]
[ 5.54881045]
[ 5.30603267]
[ 9.78594227]
[ 4.90965564]
[ 5.91306699]
[ 8.56331925]
[ 3.23806212]
[ 8.99781574]
[ 4.70718666]
[10.70314449]
[ 7.3965179 ]
[ 3.87183748]
[ 4.55507427]
[ 9.18975324]
[ 8.49163691]
[ 8.72049122]
[ 7.94759736]
[ 4.67652161]
[ 6.44386684]
[ 3.98086294]
[11.04439507]
[ 8.21362168]
[ 4.79408465]
[ 5.03790371]
[ 4.89121226]
[ 6.73818454]
[ 9.53623265]
[ 7.00466251]
[10.28665258]
```

```
 [ 7.24607048]
 [ 5.53962564]
 [10.17626171]
 [ 8.31932218]
 [ 6.61392702]
 [ 7.73628865]
 [ 6.14696329]
 [ 7.05929527]
 [ 6.90639808]
 [ 4.42920556]
 [ 5.47453181]]
```

TRAINING DATA

[6]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2,
                                                    random_state=42)
```

[7]:
```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(80, 1)
(20, 1)
(80, 1)
(20, 1)
```

train_test_split Function train_test_split is a function from the sklearn.model_selection module in the scikit-learn library. It is used to split your dataset into two parts: one part for training the model and another part for testing the model's performance.

X:

Represents the features or independent variables of the dataset. This is the input data that the model will learn from. In the case of simple linear regression, X might be a single feature. For multiple linear regression, X could have multiple features.

y:

Represents the target variable or dependent variable. This is the output that the model is trying to predict based on the features in X.

test_size=0.2:

Determines the proportion of the dataset that should be allocated to the test set. test_size=0.2 means 20% of the data will be used for testing, and the remaining 80% will be used for training.

random_state=42:

Controls the randomness of the data splitting. By setting random_state=42, you ensure that the split is reproducible. If you run the code multiple times with the same random_state, you'll get

the same training and testing sets every time. The value 42 is arbitrary; any integer can be used, or you can omit this parameter to have random splits every time you run the code.

X_train: Contains the features for the training set. This is the subset of X that will be used to train the model. 80% of the data (if test_size=0.2) from X will go into X_train.

X_test: Contains the features for the test set. This subset of X will be used to evaluate how well the model performs on unseen data. 20% of the data (if test_size=0.2) from X will go into X_test.

y_train: Contains the target variable values corresponding to X_train. This is what the model will learn to predict during training. 80% of the data (if test_size=0.2) from y will go into y_train.

y_test: Contains the target variable values corresponding to X_test. This is used to evaluate the model's predictions after it has been trained. 20% of the data (if test_size=0.2) from y will go into y_test.

Purpose of Splitting the Data

Training Set (X_train, y_train): Used to fit the model, allowing it to learn the relationship between the features and the target.

Test Set (X_test, y_test): Used to evaluate the model's performance on new, unseen data. This helps to assess how well the model generalizes to new data, which is crucial for predicting real-world outcomes.

Train the Model USING LINEAR REGRESSION

[9]:
```
model = LinearRegression()
model.fit(X_train, y_train)
```

[9]:
```
LinearRegression()
```

LinearRegression(): This creates an instance of the LinearRegression class from the scikit-learn library.

model: This is now a Linear Regression model that you can train on your data

fit Method: This is used to train the model. The fit method takes two arguments:

X_train: The features (input data) used to train the model. This dataset contains the independent variables (e.g., size of the house, number of rooms).

y_train: The target variable (output data) that the model is trying to predict. This dataset contains the dependent variable (e.g., house prices).

What Happens During Training

The fit method calculates the best-fit line (or hyperplane in multiple dimensions) that minimizes the difference between the predicted values and the actual values in y_train.

Make Predictions

[10]:
```
y_pred = model.predict(X_test)
print(y_pred)
```

model.predict Method:

Purpose: This method is used to make predictions based on the features provided.

Input: It takes as input the new data for which you want to predict the target variable. In this case, X_test is used.

y_pred: This is an array of predicted values for the target variable based on the test set features (X_test).

Content: Each value in y_pred corresponds to the predicted target value for each row in X_test.

Evaluate the Model

[11]:
```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
print(f"R-squared: {r2}")
```

```
Mean Squared Error: 0.6536995137170021
R-squared: 0.8072059636181392
```

mean_squared_error Function: This function is from the sklearn.metrics module and computes the Mean Squared Error between the true values and the predicted values.

Inputs: *y_test: The actual target values from the test set.*

y_pred: The predicted values from the model for the test set.

MSE = Σ(y_test - y_pred)^2/n

MSE measures the average squared difference between the actual and predicted values.

Lower MSE: Indicates better model performance, as it means the predicted values are closer to the actual values.

Higher MSE: Indicates poorer model performance, as the predicted values are farther from the actual values.

r2_score Function: This function is also from the sklearn.metrics module and calculates the R-squared score, which measures how well the model's predictions match the actual values.

Inputs: y_test: The actual target values from the test set.

y_pred: The predicted values from the model for the test set.

^2 = 1 - (sum squared regression (SSR) ÷ total sum of squares (SST)),

= 1 - [Σ(y_test - y_pred)^2 ÷ Σ(y_test - y_mean)^2]

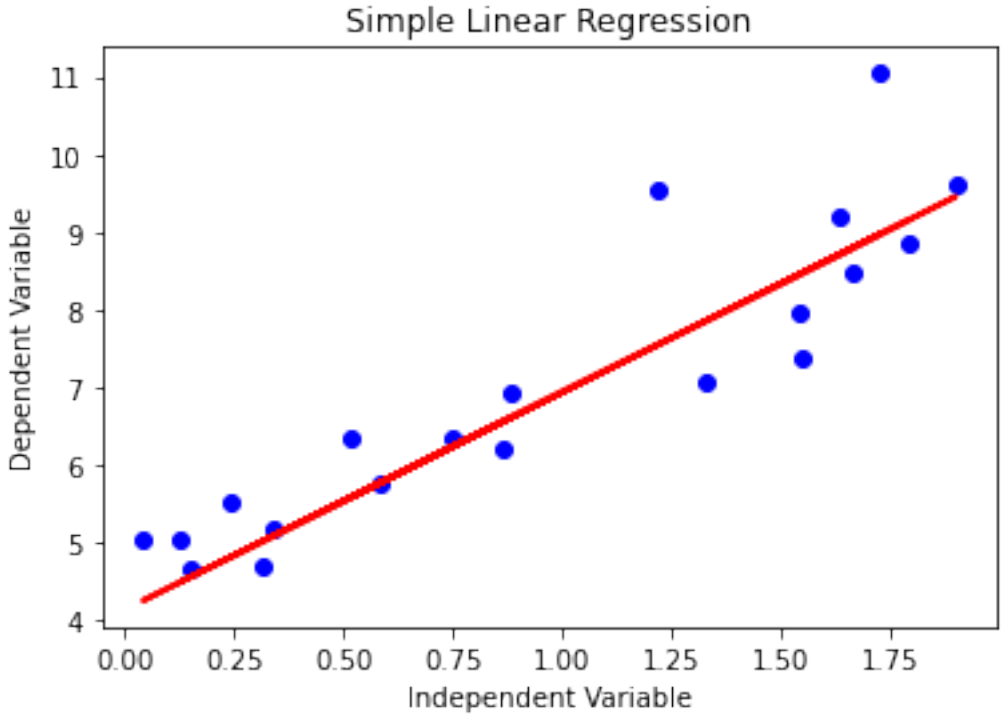
$R^2 = 1$: The model perfectly predicts the target variable.

$R^2 = 0$: The model performs no better than simply predicting the mean of the target variable.

Negative R^2: The model performs worse than predicting the mean value.

Visualize the Results

[12]:
```
plt.scatter(X_test, y_test, color='blue')
plt.plot(X_test, y_pred, color='red', linewidth=2)
plt.title('Simple Linear Regression')
plt.xlabel('Independent Variable')
plt.ylabel('Dependent Variable')
plt.show()
```



[13]:
```
m = model.coef_
b = model.intercept_
print(m)
print(b)
```

```
[[2.79932366]]
[4.14291332]
```

[ ]:
```

```

Linear Regression Model

August 8, 2025

1 Simple Linear Regression

[1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

[2]:
```
df = pd.read_csv('placement.csv')
df
```

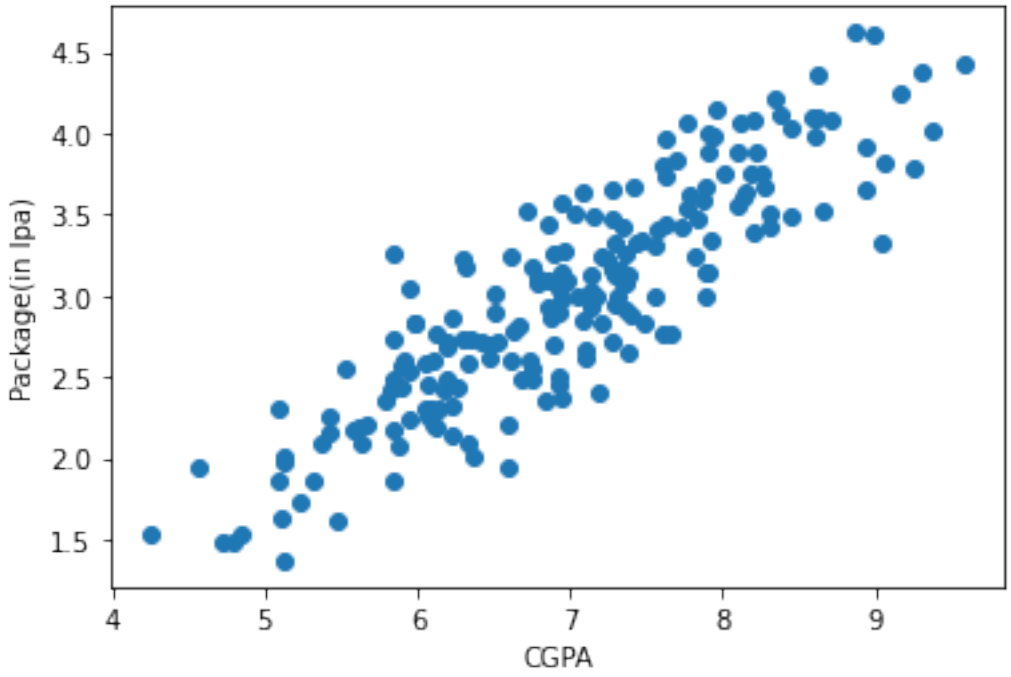
[2]:

	cgpa	package
0	6.89	3.26
1	5.12	1.98
2	7.82	3.25
3	7.42	3.67
4	6.94	3.57
..	…	…
195	6.93	2.46
196	5.89	2.57
197	7.21	3.24
198	7.63	3.96
199	6.22	2.33

[200 rows x 2 columns]

[3]:
```
plt.scatter(df['cgpa'], df['package'])
plt.xlabel('CGPA')
plt.ylabel('Package(in lpa)')
```

[3]:
```
Text(0, 0.5, 'Package(in lpa)')
```



[4]:
```
X = df.iloc[:,0:1]
y = df.iloc[:,-1]
```

[5]:
```
y
```

[5]:
```
0      3.26
1      1.98
2      3.25
3      3.67
4      3.57
       …
195    2.46
196    2.57
197    3.24
198    3.96
199    2.33
Name: package, Length: 200, dtype: float64
```

[6]:
```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.
 ↪2,random_state=2)
```

[7]:
```
from sklearn.linear_model import LinearRegression
```

[8]:
```
lr = LinearRegression()
```

[9]:
```
lr.fit(X_train,y_train)
```

[9]:
```
LinearRegression()
```

[11]:
```
# Check the predicted value for any point in X_test
lr.predict(X_test.iloc[0].values.reshape(1,1))
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\base.py:450: UserWarning: X
does not have valid feature names, but LinearRegression was fitted with feature
names
  warnings.warn(
```

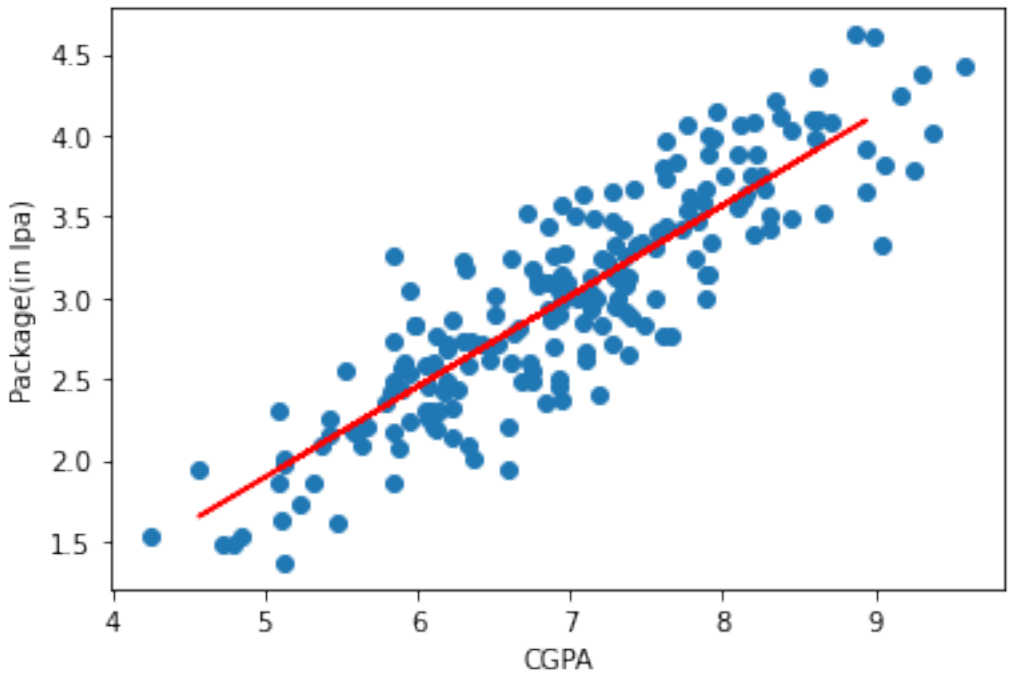
[11]:
```
array([3.89111601])
```

[12]:
```
X_test.shape
```

[12]:
```
(40, 1)
```

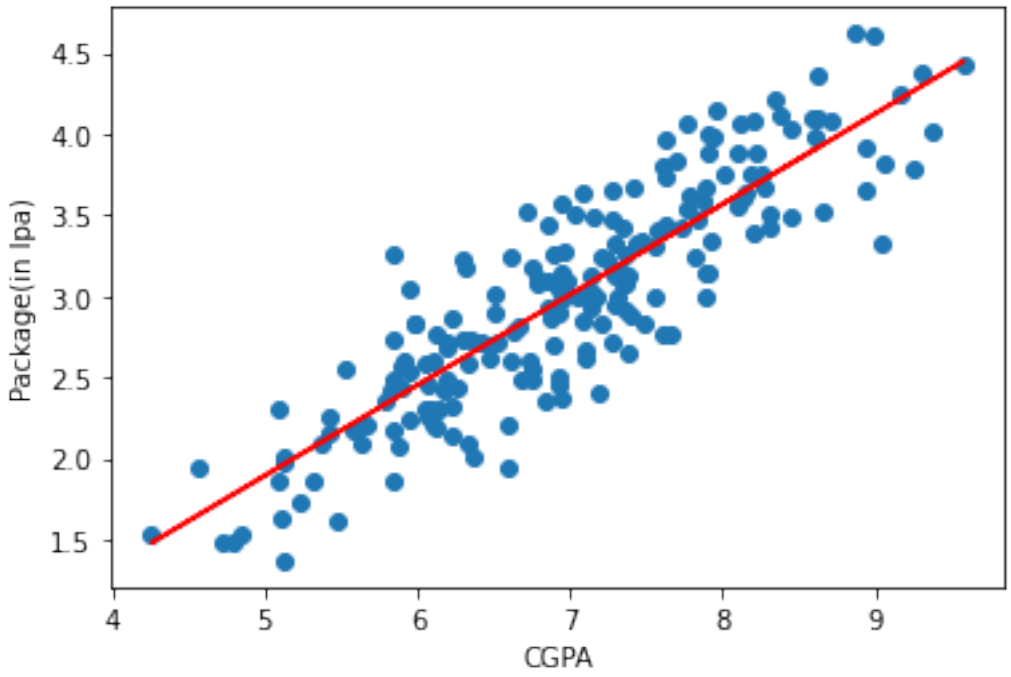
[13]:
```
plt.scatter(df['cgpa'],df['package'])
plt.plot(X_test['cgpa'], lr.predict(X_test), color='red')
plt.xlabel('CGPA')
plt.ylabel('Package(in lpa)')
```

[13]:
```
Text(0, 0.5, 'Package(in lpa)')
```



[14]:
```
plt.scatter(df['cgpa'],df['package'])
plt.plot(X_train['cgpa'], lr.predict(X_train), color='red')
plt.xlabel('CGPA')
plt.ylabel('Package(in lpa)')
```

[14]: Text(0, 0.5, 'Package(in lpa)')



[15]:
```
m = lr.coef_
b = lr.intercept_
```

[16]:
```
print(m)
print(b)
```

```
[0.55795197]
-0.8961119222429144
```

[17]:
```
# y = mx + b

m * 8.58 + b
```

[17]: array([3.89111601])

2 Multiple Linear Regression

[18]:
```
from sklearn.datasets import make_regression
X,y = make_regression(n_samples=100,
                      n_features=2,
                      n_informative=2,
                      n_targets=1, noise=50)
```

[19]:
```
print(X.shape)
print(y.shape)
```

```
(100, 2)
(100,)
```

[20]:
```
df = pd.DataFrame({'feature1':X[:,0],
                   'feature2':X[:,1],
                   'target':y})
```

[21]:
```
df
```

[21]:

	feature1	feature2	target
0	0.884692	2.126007	304.033786
1	1.626508	0.606918	117.748062
2	0.129176	1.574805	101.124911
3	0.887112	2.703314	340.697576
4	0.181254	-0.643160	-58.009669
..	…	…	…
95	-1.165216	-0.859034	-143.266410
96	0.614785	-0.070305	56.496736
97	-2.174535	-0.247711	-117.779025
98	-2.058572	-0.346620	-200.440260
99	1.930348	1.527551	302.557835

[100 rows x 3 columns]

[22]:
```
import plotly.express as px
import plotly.graph_objects as go
```

[44]:
```
import plotly.io as pio
pio.renderers.default = 'browser'
```

[23]:
```
fig = px.scatter_3d(df,
                    x='feature1',
                    y='feature2',
                    z='target')
fig.show()
```

[24]:
```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,
                                                 random_state=3)
```

[25]:
```
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(80, 2)
(80,)
(20, 2)
(20,)
```

[26]:
```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
```

[27]:
```
lr.fit(X_train,y_train)
```

[27]:
```
LinearRegression()
```

[28]:
```
y_pred = lr.predict(X_test)
print(y_pred.shape)
```

```
(20,)
```

[29]:
```
from sklearn.metrics import mean_absolute_error,mean_squared_error,r2_score

print("MAE",mean_absolute_error(y_test,y_pred))
print("MSE",mean_squared_error(y_test,y_pred))
print("R2 score",r2_score(y_test,y_pred))
```

```
MAE 38.67915528700148
MSE 2271.0706266966017
R2 score 0.8594373770057168
```

[30]:
```
x = np.linspace(-5, 5, 10)
y = np.linspace(-5, 5, 10)
xGrid, yGrid = np.meshgrid(y, x)
final = np.vstack((xGrid.ravel().reshape(1,100),yGrid.ravel().reshape(1,100))).T

z_final = lr.predict(final).reshape(10,10)

z = z_final
```

[31]:
```
fig = px.scatter_3d(df, x='feature1', y='feature2', z='target' )
fig.add_trace(go.Surface(x = x, y = y, z =z ))
fig.show()
```

[32]:
```
lr.coef_
```

[32]:
```
array([84.12427054, 81.96373401])
```

[33]:
```
lr.intercept_
```

[33]:
```
-6.377059542606279
```

3 Multiple Linear Regression on Real World dataset

[34]:
```
import numpy as np
from sklearn.datasets import load_diabetes
import pandas as pd
```

[35]:
```
# Load the diabetes dataset
diabetes = load_diabetes()

# Convert to DataFrame
df = pd.DataFrame(data=diabetes.data, columns=diabetes.feature_names)

# Add the target variable as a new column
df['target'] = diabetes.target
df
```

[35]:

	age	sex	bmi	bp	s1	s2	s3	\
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	
2	0.085299	0.050680	0.044451	-0.005671	-0.045599	-0.034194	-0.032356	
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	
..	…	…	…	…	…	…	…	
437	0.041708	0.050680	0.019662	0.059744	-0.005697	-0.002566	-0.028674	
438	-0.005515	0.050680	-0.015906	-0.067642	0.049341	0.079165	-0.028674	
439	0.041708	0.050680	-0.015906	0.017282	-0.037344	-0.013840	-0.024993	
440	-0.045472	-0.044642	0.039062	0.001215	0.016318	0.015283	-0.028674	
441	-0.045472	-0.044642	-0.073030	-0.081414	0.083740	0.027809	0.173816	

	s4	s5	s6	target
0	-0.002592	0.019908	-0.017646	151.0
1	-0.039493	-0.068330	-0.092204	75.0
2	-0.002592	0.002864	-0.025930	141.0
3	0.034309	0.022692	-0.009362	206.0
4	-0.002592	-0.031991	-0.046641	135.0
..	…	…	…	…
437	-0.002592	0.031193	0.007207	178.0
438	0.034309	-0.018118	0.044485	104.0
439	-0.011080	-0.046879	0.015491	132.0

```
440  0.026560  0.044528 -0.025930   220.0
441 -0.039493 -0.004220  0.003064    57.0

[442 rows x 11 columns]
```

[36]:
```
X = df.iloc[:,0:10]
y = df.iloc[:,-1]
```

[37]:
```
X.shape
```

[37]:
```
(442, 10)
```

[38]:
```
y.shape
```

[38]:
```
(442,)
```

[39]:
```
from sklearn.model_selection import train_test_split
```

[40]:
```
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.
↪2,random_state=2)
```

[41]:
```
print(X_train.shape)
print(X_test.shape)
```

```
(353, 10)
(89, 10)
```

[42]:
```
from sklearn.linear_model import LinearRegression
```

[43]:
```
reg = LinearRegression()
```

[44]:
```
reg.fit(X_train,y_train)
```

[44]:
```
LinearRegression()
```

[45]:
```
y_pred = reg.predict(X_test)
```

[46]:
```
from sklearn.metrics import r2_score
r2_score(y_test,y_pred)
```

[46]:
```
0.4399387660024644
```

[47]:
```
reg.coef_
```

[47]:
```
array([  -9.16088483, -205.46225988,  516.68462383,  340.62734108,
       -895.54360867,  561.21453306,  153.88478595,  126.73431596,
        861.12139955,   52.41982836])
```

[48]:
```
reg.intercept_
```

[48]: 151.88334520854633

[ ]:

Logistic_Regression

August 8, 2025

1 Logistic Regression

- Logistic regression is a great general-purpose classifier, striking an excellent balance between accurate classifications and real-world interpretability.
- I think of it as kind of a nonbinary version of SVM, one that scores points with probabilities based on how far they are from the hyperplane, rather than using that hyperplane as a definitive cutoff.
- If the training data is almost linearly separable, then all points that aren't near the hyperplane will get a confident prediction near 0 or 1.
- But if the two classes bleed over the hyperplane a lot, the predictions will be more muted, and only points far from the hyperplane will get confident scores.

Key Components:

- **Binary Outcomes:** Logistic regression predicts the probability of an outcome, which is limited to two categories (e.g., pass/fail, win/lose).
- **Sigmoid Function:** The logistic regression model uses the sigmoid function (or logistic function) to map predicted values to probabilities. The sigmoid function is defined as:

The output of the sigmoid function is a probability between 0 and 1.

- **Decision Boundary:** The predicted probability can be converted into a binary outcome by selecting a threshold (commonly 0.5). If the predicted probability is greater than 0.5, the output is classified as 1, otherwise, it is 0.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
```

1.1 Steps in Logistic Regression:

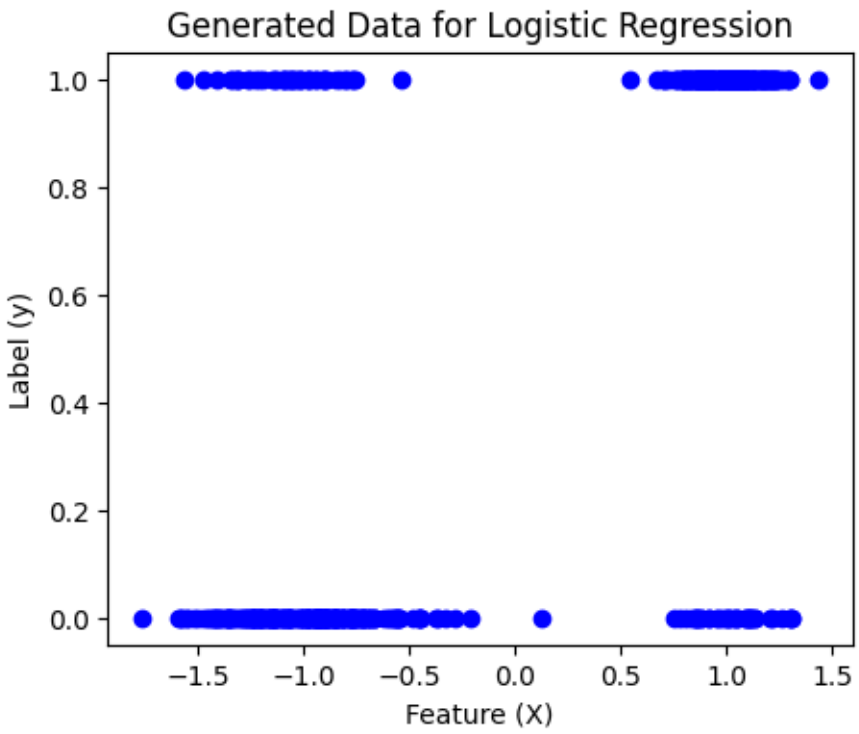
GENERATE THE DATA:

- make_classification(): This function is used to generate a random dataset for classification tasks. The generated X contains the feature values, and y contains the binary labels (0 or 1).
- n_samples: Number of data points to generate.
- n_features: Number of input features (here, it's set to 1 for simplicity, like hours of study).
- n_informative: Number of informative features (contributing to the decision boundary).
- n_redundant: Redundant features are set to 0 to avoid adding unnecessary data.
- flip_y: Proportion of labels that are randomly flipped to introduce some noise.
- random_state: Seed to make the data generation process reproducible.

[4]:
```
# Generate synthetic data for binary classification
# X: input features (study hours, etc.), y: output labels (0 or 1 for pass/fail)
X, y = make_classification(n_samples=500,      # Number of samples
                           n_features=1,       # Number of features (e.g.,␣
 ↪study hours)
                           n_informative=1,    # Number of informative features
                           n_redundant=0,      # No redundant features
                           n_clusters_per_class=1,
                           flip_y=0.25,         # Add noise by flipping 10% of␣
 ↪labels
                           random_state=42)    # Ensure reproducibility
print("Range of the input data(X): ", {np.max(X), np.min(X)})
print(X.shape)
# print(y)
```

```
Range of the input data(X):  {1.435835927800221, -1.7645319571614881}
(500, 1)
```

[5]:
```
# Visualize the generated data
fig = plt.figure(figsize=(5, 4))

plt.scatter(X, y, color='blue')
plt.title('Generated Data for Logistic Regression')
plt.xlabel('Feature (X)')
plt.ylabel('Label (y)')
plt.show()
```



TRAINING & TEST DATA

[6]:
```
# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

Like Linear Regression, we split the dataset into training and testing sets. Use the train_test_split function from the sklearn.model_selection module to divide the dataset into input features (X) and output labels (y).

Train the Model USING LOGISTIC REGRESSION

[7]:
```
model = LogisticRegression()
model.fit(X_train, y_train)
```

[7]:
```
LogisticRegression()
```

Fit the Logistic Regression Model:

The logistic regression model is created using the LogisticRegression() class from the sklearn.linear_model module. The model learns the relationship between the features and the binary outcome during training.

Make Predictions:

After training, the model can predict the probability that a given input belongs to class 1. The predict_proba() function gives the probabilities, while predict() gives the final classification (0 or

1).

[8]:
```
y_pred_prob = model.predict_proba(X_test)  # Predict probabilities
y_pred = model.predict(X_test)  # Predict binary outcome (0 or 1)


print(y_pred)
```

```
[1 1 1 0 1 0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 0 0 0 0 1 0 1 1 0 0 1 1 1 1 0 0
 1 0 0 1 1 1 1 0 1 0 0 0 1 0 1 1 0 0 0 1 0 1 0 0 0 0 0 1 0 0 1 1 1 1 0 0 0
 0 1 1 0 0 0 0 1 0 0 0 0 1 1 0 1 1 0 0 1 1 0 0 1 1 0]
```

[9]:
```
# Compute model slope and intercept

a = model.coef_
b = model.intercept_,
print("Estimated model slope, a:" , a)
print("Estimated model intercept, b:" , b)
```

```
Estimated model slope, a: [[1.49524877]]
Estimated model intercept, b: (array([0.06724698]),)
```

So, our fitted regression line is

y = 1.49524877 * x - 0.06724698

That is our linear model.

Evaluate the Model:

- Common evaluation metrics for Logistic Regression include: Accuracy: The percentage of correct predictions.
- Confusion Matrix: A matrix that shows the true positives, false positives, true negatives, and false negatives. Precision and Recall: Metrics for evaluating how well the model handles each class.

[10]:
```
from sklearn.metrics import accuracy_score, confusion_matrix
```

[11]:
```
accuracy = accuracy_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

print(f"Accuracy: {accuracy}")
print("Confusion Matrix:")
print(cm)
```

```
Accuracy: 0.84
Confusion Matrix:
[[42  6]
 [10 42]]
```

Accuracy

From all the classes (positive and negative), how many of them we have predicted correctly. In this case, it will be 4/7.

Accuracy should be **high** as possible.

- **y_true: array-like of shape (n_samples,)** Ground truth (correct) target values.
- **y_pred: array-like of shape (n_samples,)** Estimated targets as returned by a classifier.

Returns:

- **C: ndarray of shape (n_classes, n_classes)** Confusion matrix whose i-th row and j-th column entry indicates the number of samples with true label being i-th class and predicted label being j-th class.

Confusion Matrix

A confusion matrix is a two-dimensional matrix used in classification experiments to evaluate the performance of a system by showing the number of correctly and wrongly classified data, helping to identify which classes of data are most often misplaced.

The above equation can be explained by saying, from all the positive classes, how many we predicted correctly.

Recall should be high as possible.

The above equation can be explained by saying, from all the classes we have predicted as positive, how many are actually positive.

Precision should be high as possible.

F-Score

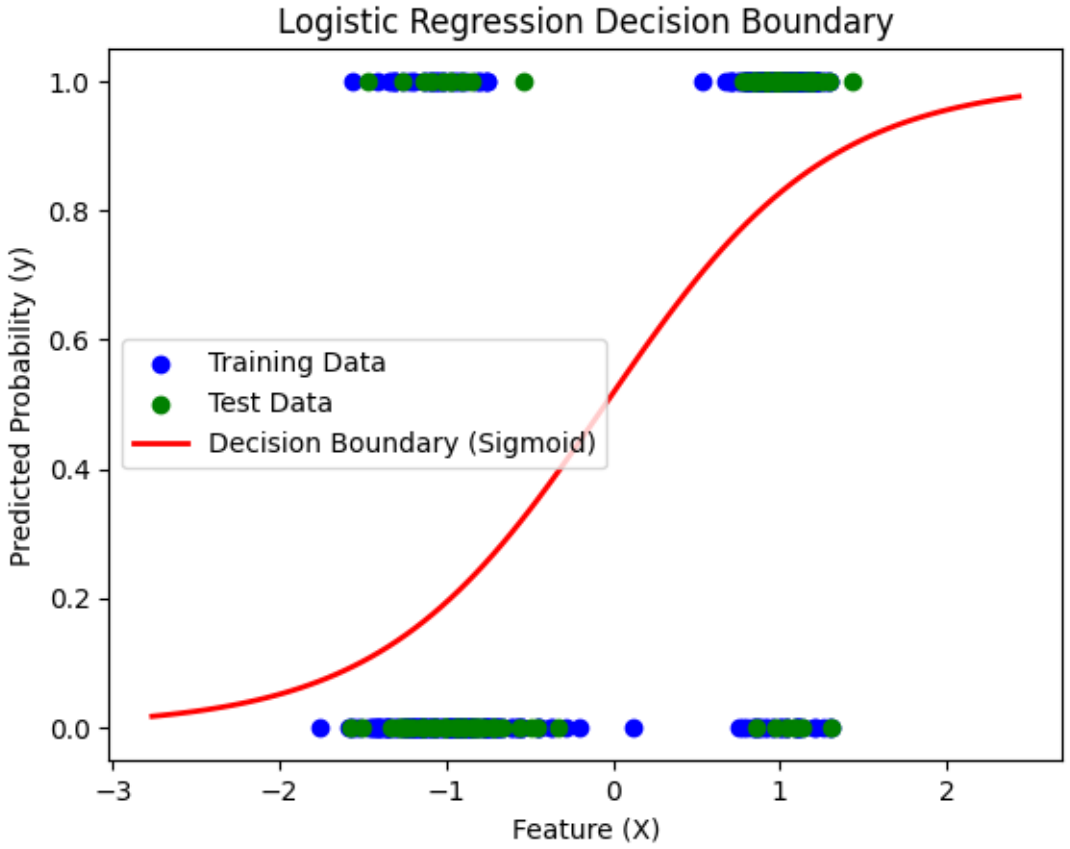
It is difficult to compare two models with low precision and high recall or vice versa. So to make them comparable, we use F-Score. F-score helps to measure Recall and Precision at the same time.

F1-score is a harmonic mean of Precision and Recall, and so it gives a combined idea about these two metrics. It is maximum when Precision is equal to Recall.

Visualize the Results

[12]:
```
# Generate a range of values for X to visualize the decision boundary
X_values = np.linspace(X.min() - 1, X.max() + 1, 100).reshape(-1, 1)

# Predict the probabilities for these values using the logistic regression model
y_pred_prob = model.predict_proba(X_values)[:, 1]

plt.scatter(X_train, y_train, color='blue', label='Training Data')
plt.scatter(X_test, y_test, color='green', label='Test Data')

# Plot the predicted probabilities (decision boundary)
plt.plot(X_values, y_pred_prob, color='red', linewidth=2, label='Decision␣
 ↪Boundary (Sigmoid)')
```

```
plt.title('Logistic Regression Decision Boundary')
plt.xlabel('Feature (X)')
plt.ylabel('Predicted Probability (y)')
plt.legend()
plt.show()
```



[ ]:

Decision_Tree

August 8, 2025

1 Decision Tree Model

A Decision Tree is a non-parametric supervised learning algorithm, widely used for both classification and regression tasks. It works by splitting the data into subsets based on the most important feature at each node, forming a tree-like structure.

Decision trees are simple to understand and interpret, making them a popular choice in machine learning.

1.0.1 Key Concepts:

- **Nodes:**
  - Root Node: The top node of the tree that represents the entire dataset. The first feature split happens here.
  - Internal Nodes: These represent feature-based splits and have children nodes.
  - Leaf Nodes: The terminal nodes of the tree which give the final prediction (for classification, they contain class labels).
- **Splitting:** At each node, the algorithm selects the feature that best splits the data into different classes or values. The goal is to reduce impurity (for classification) or variance (for regression).
- **Impurity Measures:**
  - Gini Impurity: A commonly used metric for classification trees, measuring how often a randomly chosen element from the set would be incorrectly classified.

    Where is the probability of a class in the node.
  - Entropy: Another impurity metric, derived from information theory. It quantifies the uncertainty of a random variable.

- Variance Reduction: Used in regression trees to measure the reduction in variance after a split.

- **Recursive Partitioning:** The tree grows by recursively splitting the data based on the most informative features until a stopping condition is met, such as reaching a maximum depth or a minimum number of samples in a node.
- **Pruning:** Decision trees are prone to **overfitting** if they grow too deep. Pruning is the process of trimming the tree to reduce its size and improve generalization on unseen data. It can be:
- Pre-pruning: Stop growing the tree early, based on criteria like maximum depth or minimum samples.
- Post-pruning: Prune the tree after it has fully grown.

[ ]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt
```

1.0.2 Steps in Decision Tree Modeling:

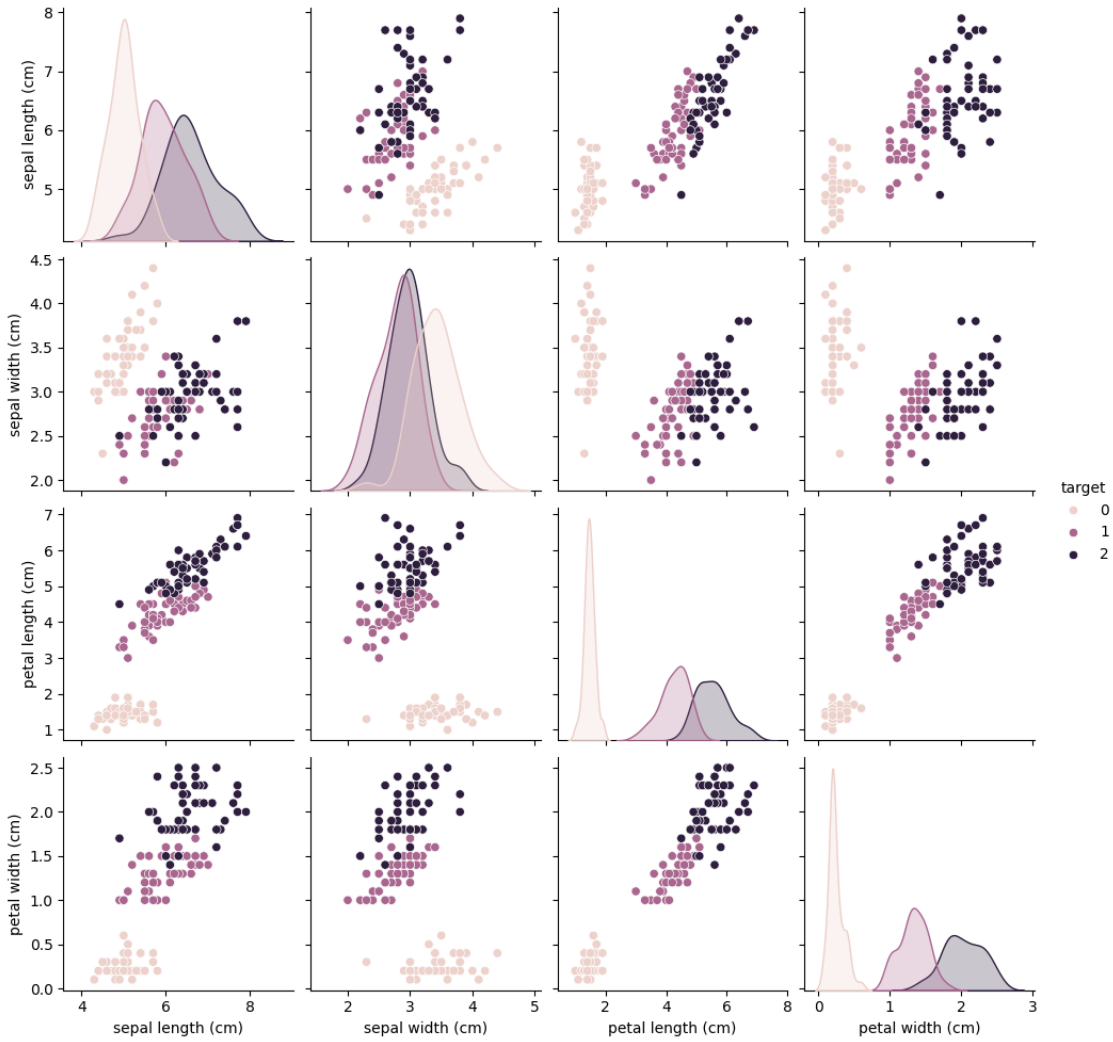
1. **Data Preparation:** Like with other models, we first prepare the dataset by splitting it into training and testing sets.

[ ]:
```
# Load the dataset
iris = load_iris()
X, y = iris.data, iris.target
print(X.shape)
print(y.shape)
```

```
(150, 4)
(150,)
```

visualizing the dataset

[ ]:
```
import pandas as pd
import seaborn as sns

# Create a DataFrame from the data
df = pd.DataFrame(data=iris.data,
                  columns=iris.feature_names)
df['target'] = iris.target

# Visualize the data using seaborn
sns.pairplot(df, hue='target')
plt.show()
```



[ ]:
```
# Split the data
X_train, X_test, y_train, y_test =
 train_test_split(X, y, test_size=0.2,
                  random_state=42)
```

[ ]:
```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(120, 4)
(30, 4)
(120,)
(30,)
```

2. **Model Training:** We use the DecisionTreeClassifier for classification tasks or Decision-

TreeRegressor for regression tasks.

[ ]:
```
# Train a Decision Tree Classifier
model = DecisionTreeClassifier(random_state=42)
model.fit(X_train, y_train)
```

[ ]:
```
DecisionTreeClassifier(random_state=42)
```

3. **Prediction:** After training, we can use the model to predict outcomes for the test set.

[ ]:
```
# Predict the outcomes
y_pred = model.predict(X_test)
```

[ ]:
```
print(y_pred)
```

```
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0]
```

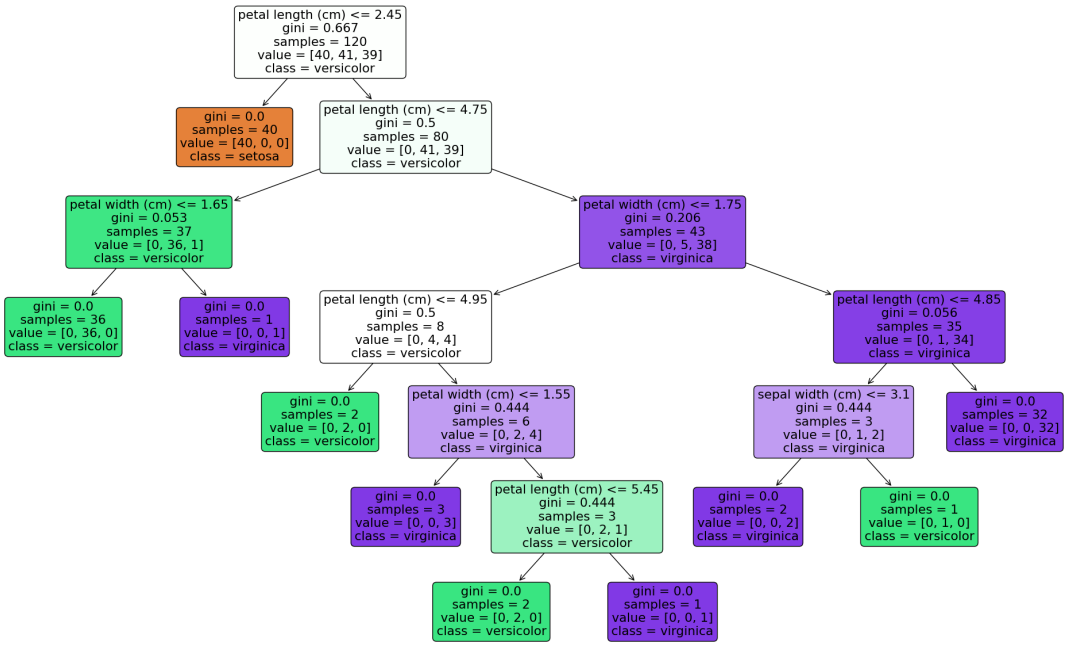
4. **Evaluation:** Common metrics for classification trees include accuracy, precision, recall, and the confusion matrix. For regression trees, metrics such as Mean Squared Error (MSE) or R-squared are used.

[ ]:
```
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)
```

```
Accuracy: 1.0
```

5. **Visualization:** Decision trees can be visualized to understand the splits. The plot_tree function from the sklearn.tree module can be used for this purpose.

[ ]:
```
# Visualize the tree
plt.figure(figsize=(25,15))
plot_tree(model, filled=True,
          feature_names=iris.feature_names,
          class_names=iris.target_names,
          rounded=True)
plt.show()
```



[ ]:
```
X = df.iloc[:,0:1]
y = df  .iloc[:,-1]
```

By Misha Patel

Naive Bayes Model

Naive Bayes is a family of probabilistic algorithms based on Bayes' Theorem with the “naive” assumption of independence between every pair of features. It is widely used for classification tasks, particularly when the dimensionality of the input is high. Naive Bayes is known for its simplicity, efficiency, and good performance with small datasets.

Key Concepts:

1. **Bayes' Theorem:** Bayes' Theorem provides a way to update the probability estimate for a hypothesis as more evidence or information becomes available. It is given by:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Where:

- $P(A|B)$ is the posterior probability of class A given predictor B.
- $P(B|A)$ is the likelihood, the probability of predictor B given class A.
- $P(A)$ is the prior probability of class A.
- $P(B)$ is the prior probability of predictor B.

1. **Naive Assumption:** Naive Bayes assumes that the features are independent of each other, given the class label. This simplification allows the model to be highly efficient but might not hold true in real-world scenarios.

2. **Types of Naive Bayes Classifiers:**

   - **Gaussian Naive Bayes:** Assumes that the features follow a normal (Gaussian) distribution. It is used for continuous data.
   - **Multinomial Naive Bayes:** Used for discrete data and is particularly effective for text classification tasks like document classification.
   - **Bernoulli Naive Bayes:** Similar to the multinomial variant but designed for binary/boolean features.

Steps to Implement Naive Bayes:

1. Import Necessary Libraries:

In [ ]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

2. Load Dataset: Here, we use the Iris dataset for simplicity.

In [ ]:
```
data = load_iris()
X = data.data
y = data.target

print(X.shape)
print(y.shape)
```

```
(150, 4)
(150,)
```

3. Split the Dataset:

In [ ]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(120, 4)
(30, 4)
(120,)
(30,)
```

4. Train the Naive Bayes Model:

- Calculate the prior probability for each class.
- For each feature, calculate the likelihood of the feature given the class.
- Multiply the prior and the likelihoods to compute the posterior probability for each class.
- The class with the highest posterior probability is the predicted class.

In [ ]:
```
model = GaussianNB()
model.fit(X_train, y_train)
```

Out[ ]:
```
GaussianNB()
```

1. **Make Predictions:**
    - For a new instance, calculate the posterior probability for each class using the formula:

$$P(C|X) = \frac{P(X|C) \cdot P(C)}{P(X)}$$

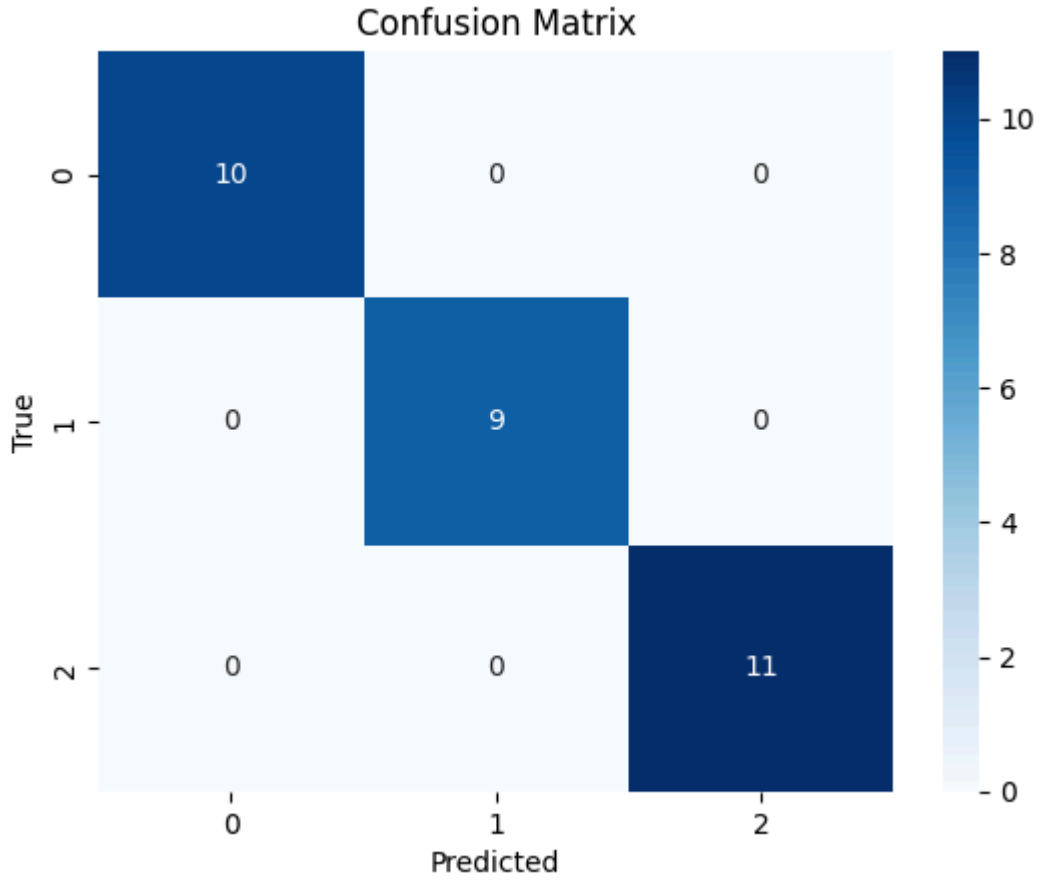
Where X represents the features of the new instance, and C represents the class.

In [ ]:
```
y_pred = model.predict(X_test)
print(y_pred)
```

```
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0]
```

6. Evaluate the Model:

In [ ]:
```
# accuracy = accuracy_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)
# print(f"Accuracy: {accuracy}")
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```



In [ ]:
```
# from sklearn.metrics import classification_report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

```
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

Advantages of Naive Bayes:

1. Fast and Efficient: Naive Bayes is computationally efficient and works well with large datasets.
2. Handles Missing Data: Naive Bayes can handle missing data naturally by ignoring the missing values during probability calculations.
3. Performs Well with High Dimensionality: It performs well with high-dimensional data, especially in text classification tasks.
4. Requires Less Training Data: The model can be trained effectively with a small amount of data.

Disadvantages of Naive Bayes:

1. Naive Assumption of Independence: The assumption that all features are independent is often not true in real-world scenarios, which can affect the model's performance.
2. Zero Probability Problem: If a class and a feature value never occur together in the training data, the model will assign zero probability to that class.
3. Not Ideal for Continuous Features: Although Gaussian Naive Bayes can handle continuous features, it's not ideal for this purpose compared to other algorithms.

In [ ]:

Introduction to Support Vector Machines (SVM)

Support Vector Machines (SVM) are a set of supervised learning algorithms used for both **classification and regression** tasks. They are particularly popular for classification problems due to their ability to create clear decision boundaries between classes. SVMs are based on the idea of finding a hyperplane that best divides a dataset into two classes.

Key Concepts in SVM

1. Hyperplane:

In an n-dimensional space, a hyperplane is a flat affine subspace that divides the data points into different classes. For binary classification, the goal of SVM is to find the hyperplane that best separates the two classes.

- In 2D, a hyperplane is simply a line.
- In 3D, a hyperplane becomes a plane.
- In higher dimensions, it is generalized as an n-1 dimensional hyperplane.

2. Margin:

The margin refers to the distance between the hyperplane and the nearest data points of any class, called support vectors. SVM aims to maximize this margin to create the most optimal separation between classes.

- Maximizing the margin ensures better generalization of the model on unseen data.

3. Support Vectors:

These are the data points that are closest to the hyperplane. They influence the position and orientation of the hyperplane. The model is named after these critical points, as they "support" the hyperplane.

Soft Margin vs. Hard Margin

- Hard Margin SVM: Assumes that the data is linearly separable, meaning that a perfect hyperplane can be found that divides the two classes without any misclassifications. However, this may not be practical in real-world scenarios with noisy data.
- Soft Margin SVM: Allows for some misclassification of data points by introducing a penalty term (C) that controls the trade-off between maximizing the margin and minimizing the classification error.

Kernel Trick

SVMs can efficiently handle data that is not linearly separable using kernels. The kernel trick allows SVMs to operate in a **higher-dimensional space** without explicitly calculating the transformation, making it computationally efficient.

- Linear Kernel: Suitable when the data is linearly separable. The decision boundary is a straight line or hyperplane.
- Polynomial Kernel: Suitable for data that follows a polynomial distribution. It allows the hyperplane to be curved.
- Radial Basis Function (RBF) Kernel: One of the most commonly used kernels. It maps the data into an infinite-dimensional space, allowing it to handle highly complex data structures.
- Sigmoid Kernel: Often used in neural networks and can mimic the behavior of a two-layer perceptron.

Mathematical Formulation

Given a dataset with n data points, where each data point belongs to one of two classes:

$$D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$$

where $x_i \in \mathbb{R}^n$ is a feature vector and $y_i \in \{-1, 1\}$ is the class label.

- The SVM aims to find a hyperplane $w \cdot x + b = 0$ that separates the classes with the largest margin, where w is the weight vector, and b is the bias.

The optimization problem is:

$$\min \frac{1}{2}\|w\|^2$$

subject to $y_i(w \cdot x_i + b) \geq 1$ for all i, which ensures that all data points are correctly classified and at least 1 unit away from the hyperplane.

For **soft-margin SVM**, the objective becomes:

$$\min \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \xi_i$$

where $\xi_i \geq 0$ are slack variables that allow some misclassifications, and C is a parameter controlling the trade-off between maximizing the margin and minimizing the error.

Step 1: Import the Necessary Libraries and Dataset

In [ ]:
```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
```

```
from sklearn.svm import SVC
from sklearn.metrics import classification_report,confusion_matrix
```

In [ ]:
```
# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data[:, :2]  # Taking the first two features for simplicity (sepal length,
y = iris.target

# We will only classify two species for binary classification
X = X[y != 2]  # Remove the third class for binary classification
y = y[y != 2]
print(X.shape)
print(y.shape)
```

```
(100, 2)
(100,)
```

In [ ]:
```
# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=42)
```

Step 2: Train the SVM Classifier

In [ ]:
```
# Initialize the SVM classifier
svm_model = SVC(kernel='linear', C=1.0)

# Fit the model using the training data
svm_model.fit(X_train, y_train)
```

Out[ ]:
```
SVC
SVC(kernel='linear')
```

Step 3: Make Predictions and Evaluate the Model

In [ ]:
```
# Predict the labels of the test set
y_pred = svm_model.predict(X_test)

# Evaluate the model's performance
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
[[12  0]
 [ 0  8]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        12
           1       1.00      1.00      1.00         8

    accuracy                           1.00        20
   macro avg       1.00      1.00      1.00        20
weighted avg       1.00      1.00      1.00        20
```

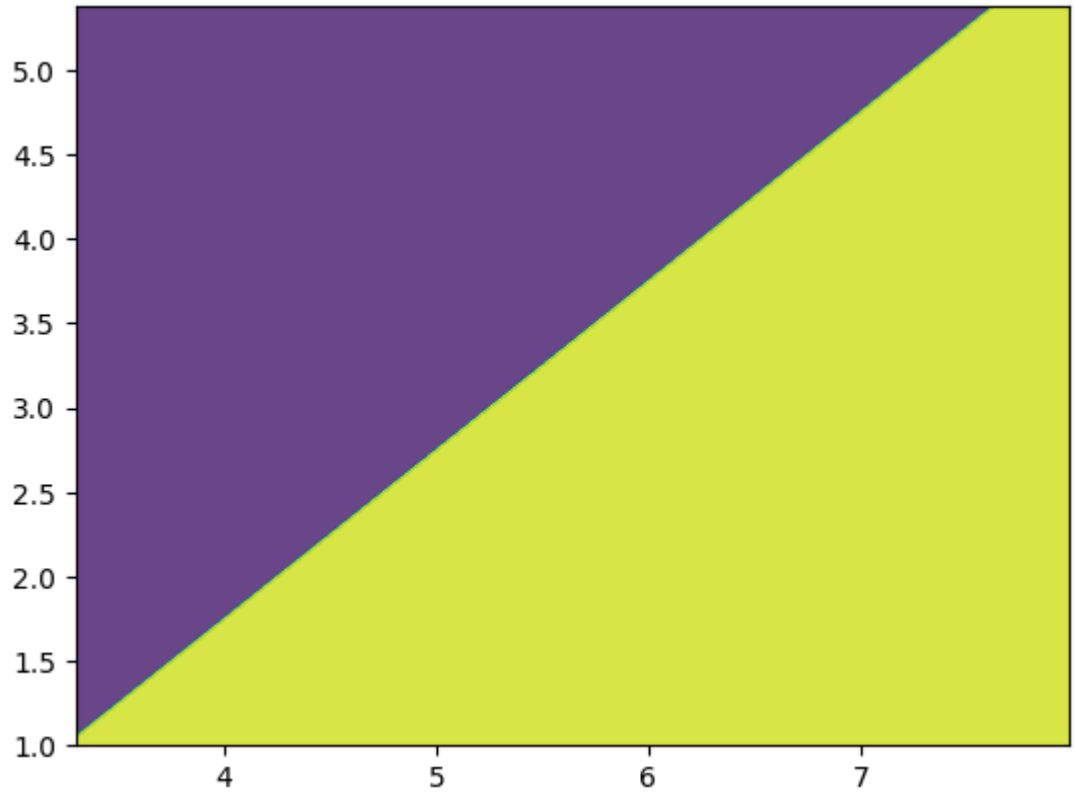
Step 4: Visualize the Decision Boundary

In [ ]:
```
# Create a mesh grid to plot decision boundary
h = .02  # step size in the mesh
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
```

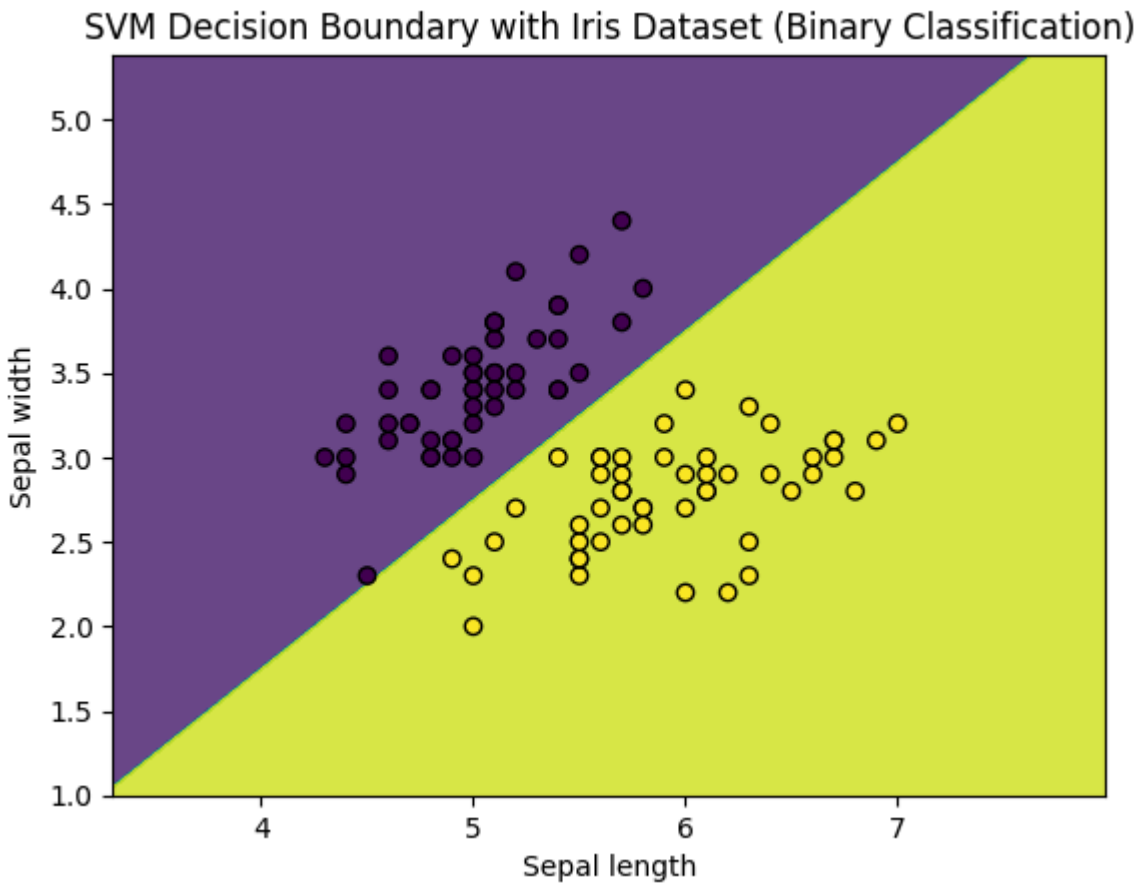
```
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
```

In [ ]:
```
# Plot decision boundary
Z = svm_model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, alpha=0.8)
```

Out[ ]:
```
<matplotlib.contour.QuadContourSet at 0x78d81346f690>
```



In [ ]:
```
# Create a mesh grid to plot decision boundary
h = .02  # step size in the mesh
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
# Plot decision boundary
Z = svm_model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, alpha=0.8)
# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', marker='o')
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.title('SVM Decision Boundary with Iris Dataset (Binary Classification)')
plt.show()
```



Advantages of SVM

- Effective in high-dimensional spaces, especially when the number of dimensions exceeds the number of samples.
- Uses a subset of the training points (support vectors), making it memory efficient.
- Works well for both linearly and non-linearly separable data through kernel trick.

Disadvantages of SVM

- SVMs are not suitable for very large datasets, as training can be slow.
- SVM models require careful tuning of hyperparameters such as the penalty parameter (C) and the kernel parameters (e.g., gamma in RBF kernel).
- It can be less effective when the classes are not well separated or when there's a lot of noise in the data.

Applications of SVM

- Text Classification: Commonly used for tasks like spam detection and sentiment analysis.
- Image Classification: SVM is used in image recognition, particularly in tasks like object detection.

- Bioinformatics: SVMs are used in gene expression classification, protein structure prediction, etc.

In [ ]: