**Practical No 1**

**Title**: Python Libraries for Data Science- a. Pandas Library b. Numpy Library c. Scikit Learn Library d. Matplotlib

**Theory**: Python is a versatile programming language, widely adopted in data science due to its simplicity, flexibility, and the strength of its libraries designed specifically for data manipulation, analysis, and visualization. Here's an overview of some essential libraries for data science:

- **Pandas**: Pandas is central to data manipulation and analysis. It offers data structures like Series and DataFrames, which allow for fast, flexible data organization, particularly for handling labeled data. Pandas supports operations such as filtering, grouping, merging, and reshaping data, making it a go-to library for preparing data for analysis and modeling.
- **NumPy**: NumPy provides support for large, multi-dimensional arrays and matrices, along with a vast library of mathematical functions to operate on these arrays. It is particularly useful for scientific computing due to its ability to perform fast, element-wise operations, linear algebra, and statistical functions. NumPy arrays form the foundation for other libraries like Pandas and Scikit-Learn.
- **Scikit-Learn**: Scikit-Learn is one of the most popular machine learning libraries, offering simple and efficient tools for predictive data analysis. It supports a range of supervised and unsupervised learning algorithms like regression, classification, clustering, and dimensionality reduction. Scikit-Learn also provides utilities for model selection, data preprocessing, and evaluation, making it a complete toolkit for building and validating machine learning models.
- **Matplotlib**: Visualization is a core part of data analysis, and Matplotlib is one of the most powerful libraries for creating static, animated, and interactive plots in Python. It offers a wide variety of chart types, from basic line and bar charts to more complex histograms and scatter plots, helping analysts interpret trends, outliers, and patterns in the data.

These libraries, when used together, offer an end-to-end solution for data science workflows, from raw data processing and numerical computation to machine learning and insightful visualizations.

```
# Importing Libraries
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
```

```
# Pandas Example with Larger Data
data = {'A': np.random.randint(1, 100, size=1000), 'B': np.random.randint(1, 100, size=1000)}
df = pd.DataFrame(data)
print("Pandas DataFrame:")
print(df.head())  # Display first 5 rows
```

```
Pandas DataFrame:
    A   B
0  39  81
1   2  32
2  31  30
3  20   7
4  18  46
```

```
# NumPy Example with Larger Array
array = np.random.randint(1, 100, size=1000)
print("\nNumPy Array:")
print(array * 2)  # Simple arithmetic operation
```

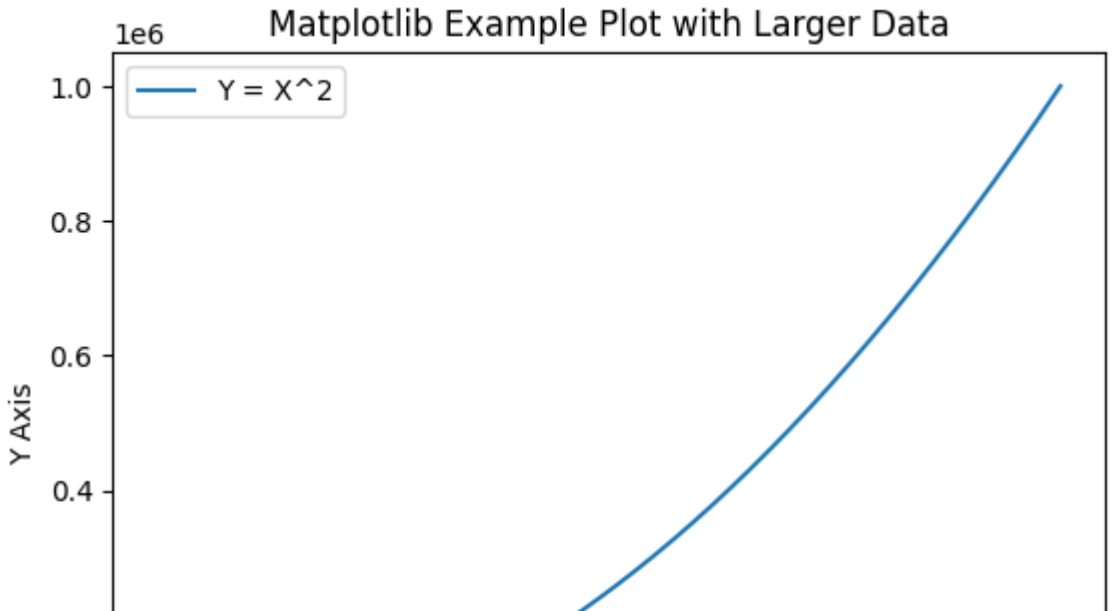
```
NumPy Array:
[124  60 182 176  48 130  20  60  32 100   6  72  50  62  76 174   4  88
 194  86  50  64 140 112 192  46 192  86 174 156 164  48 152 184 118  14
 116 194  86 126 148  66 174  42  14  94  44 146  78 130  16  46   8 116
  32  52 166 198  92 128 158 112 128 136  16 110  60  98 168 172  66 146
  16 198  50  94 154 102  38 102  44  78  84 142  94 196 170 126  52  64
  68 150 192  30 128 100 156  72 190  72  76 108  74 120   6 144 100  90
 162  34  62 144  28  62  66  80 158 154  88   6  38 136 152 180 112 160
  58  36  74 176 156 176  14  94  94  94  84  54 186   4   2 146 198  34
 108  14 166  24 152   4 174  20  24  18 156 134  74  50  66  28  90  46
  90  98 176  14 140  34  16  88  26  70   6 128  70 176 174  36 126 100
  66  18 196 100 104  40   6  78 138 130 106 184  24 178  56 108  66  16
 122  34  44  56  58  92  88 118 118 160 104  82  70  60 198 118  76 164
  28 174 176 138 166  90 174 166  22  60  88  26  84 160  32  46 198 128
 100  72  96  32  56  46  10  82  10 128 132  72 120 194  84 138 154 102
 198  66 172  94 132 198 166 162 154 130  10 182 156 122 114 142  66  58
  44 174  54 144 106  32  16  74  58 172  20 174  76 182  38  70 112 110
 124 124 170 156 134 148 172  24   6  42 134 148 164  38  32 150 160  46
  12 104 130  18  44 198 190  44  30   6 188 198 126  92 180 156  40  76
 160  46  28  76   8  54   4 108   4 128 198  64 192 156  78  54   6  12
  78 154 142 178  86  12  50  96  30  84  50  80 198  76  88 142  38 192
  76  58  68  92 190 130  68 184  66 100 120  36  92 144   4 166  38   6
 106  86  94   4 124  44 124 198 136 118  34 126 144 148  30  84  38 154
  50 126  96  58  30 150 158  62 126  60  10 194  98 140 120  64 192  42
  38  12 120 158  90 160  16 112  34 114  42   8  70  56 198  70 182 160
 144  46 148  86   4  40  88 196  42  46 102  66  12 120 122  82  74  82
 160 100  74 128  36  84 198 168 158  32 188  98  94  92 168 112 114  46
 100  30 146 118  48 182   2 176 138 116 162 116 144  42   6  42 164  68
 100  48  46   4   6  36  54  86 118  62  36 146  78 140  82  96 170 186
 130 158 172 158  92  72  26 140  28 168 126  40  76  64  80 170  66  90
  52  12 150 102  52  20  14  10 136 192  74 190 184  66  86  14  26  52
 150 108  76  50 108 164 188  98 166 102  28  24 158  10 156  22  84  90
 118 142 142 182 150 112 120 104  22  88  20 104 166  30  30 190  14 138
 134   6  38  34 132 152 120  98  66  58 190   8 148 194  54  22  70  44
```

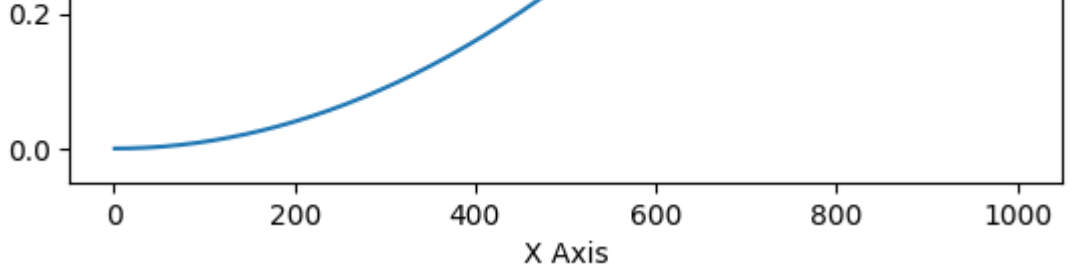
```
 22  62  54 164 108 146  34  24  46 176 156  90 108 146  40  90 100 116
104  88 154 140  28   4  52  38  52 144  30  60 114 134  98  62  46 118
 28 116  68 176  20  28  46  70 196 190  36 168 188 136   6 138  70 108
 10  24  24  62 172 154  46  48  46 160 186  62 118  86 176  48  16  86
 56 166 106 118 196  82  44 138  24  34  10 138  68 128  96  56  26 180
  2 176  26 156  76  92   4  44 144  80 172 130 116  88  92 194 148 106
  8 142 106  84  26 122  50 104 144  82  90  20 134 192  66  66 168 168
 86  32  60 198  24  22  14  16  42 192 102 142 114  44 178  72  68 154
146 102 182 148  14  40  16  16  30 176  28  82  82  96  24  90 134  42
180 138 126  46  94  78  86 122   8 118 146 190 166  64  46 132 194  18
130 198  76  62  88 166  34  90 170 164  36 188  28  26 170   6 100  68
 12  60 110 168 156  16 154  38 178 108  84 184 160 198 116  82  84  26
168  42 162 196  96 160 146  42  42  76  34  88 102  12  38  90  16  32
 56  64 108  84 186 176 134  86  32 184  98 198   4 140  84 106 144 162
 48 148 180  80 132 120  60  60  16 184  94  98 172 190 150  74  46  46
160 144  62 126  92 134  94 160 190  98  28  10 100 172 164  68 196   6
178 198 196  68 100 166  90 190  30  54  20 108  70  60  96  20 106   4
 72  72 122  36  50  76  70  42  74 106 156  14  10 178  58  58  62  26
 40 160 110 188  36  42  92  84  22   6 134  18 184 102  90 118 148  44
184  42  70  62  30 174  36  76 120  92 130 114 106 100  70 120  16 114
102  64  18 174 184  80 140 108  80 132 124  76 160  50  70 144  24  46
 62  34 132   6  68  96  84 194 140  32  80 104  60  78 138 118 128  96
130 170  44  96 138  20  74  86 110  98]
```

```
# Scikit-Learn Example with Larger Dataset
X = np.random.randint(1, 100, size=(1000, 1))
y = X * 2 + np.random.normal(0, 5, size=(1000, 1))  # Adding noise
model = LinearRegression()
model.fit(X, y)
prediction = model.predict(np.array([[50]]))
print("\nScikit-Learn Prediction for input [50]:", prediction[0][0])
```

```
Scikit-Learn Prediction for input [50]: 99.87730470485671
```

```
# Matplotlib Example with Larger Data
plt.plot(np.arange(1, 1001), np.power(np.arange(1, 1001), 2), label='Y = X^2')
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
plt.title("Matplotlib Example Plot with Larger Data")
plt.legend()
plt.show()
```






Start coding or generate with AI.

**Conclusion**: Python libraries like Pandas, NumPy, Scikit-Learn, and Matplotlib provide comprehensive, specialized tools for data science, enabling efficient data manipulation, numerical operations, machine learning, and visualization capabilities that streamline the end-to-end data analysis process.

**Practical No 2**

**Title**: Evaluation Metrics-
a. Accuracy
b. Precision
c. Recall
d. F1-Score

**Theory**: Evaluation metrics are essential in assessing machine learning models, particularly in classification tasks. While accuracy is often the first metric we think of, it may not always reflect a model's effectiveness, especially in cases with imbalanced data where certain classes are more prevalent than others. To fully understand a model's performance, several evaluation metrics must be considered:

- **Accuracy**: Accuracy is the most straightforward metric, representing the ratio of correctly predicted instances to the total number of instances. Although accuracy provides a general overview of model performance, it is sensitive to class imbalance. For example, in a dataset where 90% of samples belong to class A and only 10% to class B, a model that always predicts class A would achieve 90% accuracy, despite failing entirely on class B predictions. Therefore, while accuracy is useful in balanced datasets, it might be misleading in imbalanced situations.
- **Precision**: Precision measures the quality of positive predictions by calculating the ratio of true positives (correct positive predictions) to the total predicted positives (true positives + false positives). High precision indicates a low false positive rate, meaning that the model is effective at identifying true positives among its positive predictions. Precision is especially valuable in applications where false positives carry a high cost, such as spam detection. In this case, a false positive means labeling a legitimate email as spam, which might result in missed important messages. By focusing on precision, we reduce the likelihood of such mistakes.
- **Recall**: Also known as sensitivity or true positive rate, recall measures the model's ability to correctly identify all positive instances. It calculates the ratio of true positives to the sum of true positives and false negatives. A high recall score indicates that the model is effective in detecting all actual positives, though it may come at the expense of precision. Recall is critical in contexts where missing positive instances has a significant cost, such as in medical diagnostics, where failing to detect a disease can have severe consequences. In such cases, a high recall ensures that as many positive cases as possible are identified, even if it includes some false positives.
- **F1-Score**: The F1-score combines precision and recall into a single metric by calculating their harmonic mean, providing a balanced view of model

performance when both false positives and false negatives are important. It ranges between 0 and 1, where a higher score indicates a better balance between precision and recall. F1-score is especially useful in scenarios where class distributions are imbalanced or when neither precision nor recall can be prioritized exclusively. The harmonic mean ensures that a low score in either precision or recall will reduce the F1-score, encouraging a model that performs well on both fronts.

Choosing the right metric depends on the specific context and requirements of the problem. In many real-world applications, understanding the interplay between these metrics helps us fine-tune models and achieve a balance that suits the practical needs of the task.

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
```

```
# Generating a large synthetic dataset
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2, random_state=42)
```

```
# Splitting dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

```
# Training a RandomForest Classifier
model = RandomForestClassifier(random_state=42)
model.fit(X_train, y_train)
```

```
RandomForestClassifier
RandomForestClassifier(random_state=42)
```

```
# Making predictions
y_pred = model.predict(X_test)
```

```
# Calculating Evaluation Metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-Score:", f1)
```

```
Accuracy: 0.8566666666666667
Precision: 0.8888888888888888
Recall: 0.8258064516129032
F1-Score: 0.8561872909698997
```

**Conclusion**: Evaluation metrics like accuracy, precision, recall, and F1-score provide insights into a model's strengths and limitations, helping to understand its performance, especially in cases of class imbalance.

**Practical No 3**

**Title**: Train and Test Sets by Splitting Learn and Test Data

**Theory**: In machine learning, splitting the dataset into training and testing sets is a crucial step in the process of model evaluation. The idea behind splitting the data is to ensure that the model is trained on one portion of the data (training set) and tested on another, unseen portion (test set). This helps in evaluating the model's performance on data it hasn't seen before, simulating real-world scenarios.

1. **Training Set**: The training set is used to train the model, allowing it to learn the patterns and relationships in the data. The model adjusts its internal parameters based on the training set to minimize the error or loss function.
2. **Test Set**: The test set is used to evaluate the model's performance after it has been trained. It helps assess how well the model generalizes to unseen data. By testing the model on data that was not used during training, we can get a better understanding of its predictive power and avoid overfitting (i.e., when a model performs well on training data but poorly on new data).
3. **Data Splitting**: A common practice is to split the dataset into two or more subsets. A typical approach is a 70-30 or 80-20 split, where 70-80% of the data is used for training and the remaining 20-30% is reserved for testing. In some cases, a third subset called the **validation set** may be created for tuning hyperparameters and model selection.
4. **Stratified Sampling**: For classification problems, it is important to ensure that both the training and test sets have a similar distribution of classes, especially when dealing with imbalanced datasets. This is called stratified sampling and ensures that each class is proportionally represented in both sets.

Properly splitting the data into training and test sets is vital for obtaining a reliable model evaluation. The practice helps prevent overfitting and ensures the model is robust in real-world applications.

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
```

```
# Loading the Iris dataset
data = load_iris()
X, y = data.data, data.target
```

```
# Splitting the dataset into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
# Displaying the shape of the train and test sets
print("Training set shape:", X_train.shape)
print("Test set shape:", X_test.shape)
```

```
Training set shape: (120, 4)
Test set shape: (30, 4)
```

**Conclusion**: Splitting the dataset into training and test sets is essential for evaluating a machine learning model's performance. It helps ensure the model generalizes well to new, unseen data, avoiding overfitting and providing a more realistic assessment of its predictive capabilities.

**Practical No 4**

**Title**: Linear Regression

**Theory**: Linear regression is one of the simplest and most widely used algorithms in machine learning for predictive modeling. It is a type of regression algorithm used to predict a continuous target variable (dependent variable) based on one or more predictor variables (independent variables).

1. **Basic Concept**: Linear regression works by establishing a relationship between the target variable and the predictor variables. This relationship is modeled as a linear equation:

$$y = w_1x_1 + w_2x_2 + ... + w_nx_n + b$$

Where:

- y is the target variable,
- x1,x2,...,xn are the independent variables,
- w1,w2,...,wn are the weights or coefficients for each independent variable, and
- bis the bias term.

The goal of linear regression is to find the best-fitting line (or hyperplane, in the case of multiple variables) that minimizes the error between the predicted and actual target values.

2. **Ordinary Least Squares (OLS)**: The most common method for fitting a linear regression model is by using **Ordinary Least Squares (OLS)**. OLS aims to minimize the sum of the squared differences (errors) between the observed values and the predicted values.
3. **Assumptions of Linear Regression**:
    - **Linearity**: There is a linear relationship between the target variable and the predictors.
    - **Independence**: The residuals (errors) are independent of each other.
    - **Homoscedasticity**: The variance of the residuals is constant across all levels of the independent variables.
    - **Normality**: The residuals are normally distributed.
4. **Evaluation Metrics**: The performance of a linear regression model is typically evaluated using metrics such as **Mean Squared Error (MSE)**, **R-squared**, and **Adjusted R-squared**. These metrics help assess how well the model fits the data and its ability to make accurate predictions.

5. **Use Cases**: Linear regression is often used in applications where we want to predict continuous variables, such as predicting house prices, stock prices, or sales revenue based on historical data and other factors.

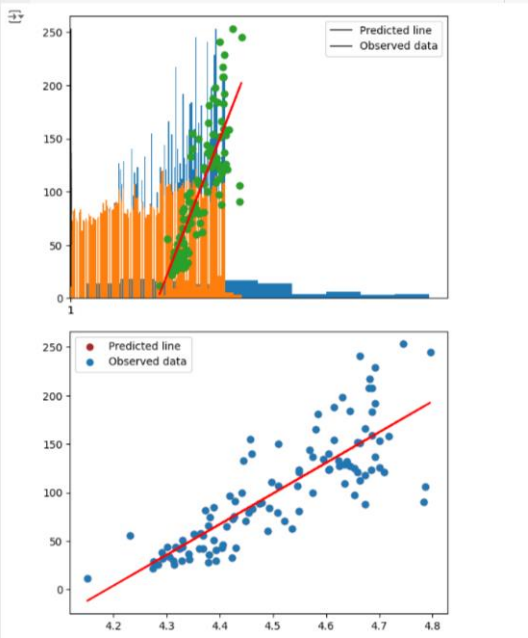
CODE FOR SLR :  # Importing necessary libraries

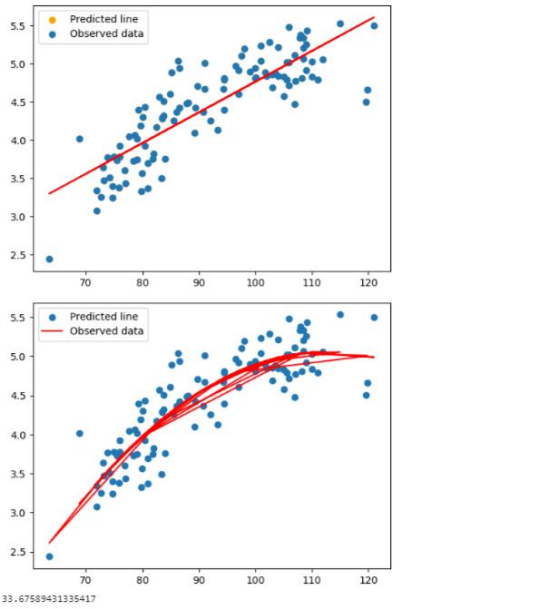
```
import pandas as pd # deals with data frame
import numpy as np
wcat = pd.read_csv("C:/Datasets_BA/Linear Regression/wc-at.csv")
wcat.describe()
import matplotlib.pyplot as plt # mostly used for visualization purposes
plt.bar(height = wcat.AT, x = np.arange(1, 110, 1))
plt.hist(wcat.AT) #histogram
plt.boxplot(wcat.AT) #boxplot
plt.bar(height = wcat.Waist, x = np.arange(1, 110, 1))
plt.hist(wcat.Waist) #histogram
plt.boxplot(wcat.Waist) #boxplot
plt.scatter(x = wcat['Waist'], y = wcat['AT'], color = 'green')
np.corrcoef(wcat.Waist, wcat.AT)
cov_output = np.cov(wcat.Waist, wcat.AT)[0, 1]
cov_output

import statsmodels.formula.api as smf
model = smf.ols('AT ~ Waist', data = wcat).fit()
model.summary()
pred1 = model.predict(pd.DataFrame(wcat['Waist']))
plt.scatter(wcat.Waist, wcat.AT)
plt.plot(wcat.Waist, pred1, "r")
plt.legend(['Predicted line', 'Observed data'])
plt.show()
res1 = wcat.AT - pred1
res_sqr1 = res1 * res1
mse1 = np.mean(res_sqr1)
rmse1 = np.sqrt(mse1)
rmse1
plt.scatter(x = np.log(wcat['Waist']), y = wcat['AT'], color = 'brown')
```

```
np.corrcoef(np.log(wcat.Waist), wcat.AT) #correlation


model2 = smf.ols('AT ~ np.log(Waist)', data = wcat).fit()
model2.summary()
pred2 = model2.predict(pd.DataFrame(wcat['Waist']))
plt.scatter(np.log(wcat.Waist), wcat.AT)
plt.plot(np.log(wcat.Waist), pred2, "r")
plt.legend(['Predicted line', 'Observed data'])
plt.show()
res2 = wcat.AT - pred2
res_sqr2 = res2 * res2
mse2 = np.mean(res_sqr2)
rmse2 = np.sqrt(mse2)
rmse2
plt.scatter(x = wcat['Waist'], y = np.log(wcat['AT']), color = 'orange')
np.corrcoef(wcat.Waist, np.log(wcat.AT)) #correlation
model3 = smf.ols('np.log(AT) ~ Waist', data = wcat).fit()
model3.summary()
pred3 = model3.predict(pd.DataFrame(wcat['Waist']))
pred3_at = np.exp(pred3)
pred3_at
plt.scatter(wcat.Waist, np.log(wcat.AT))
plt.plot(wcat.Waist, pred3, "r")
plt.legend(['Predicted line', 'Observed data'])
plt.show()
res3 = wcat.AT - pred3_at
res_sqr3 = res3 * res3
mse3 = np.mean(res_sqr3)
rmse3 = np.sqrt(mse3)
rmse3
```

```
model4 = smf.ols('np.log(AT) ~ Waist + I(Waist*Waist)', data = wcat).fit()

model4.summary()

pred4 = model4.predict(pd.DataFrame(wcat))

pred4_at = np.exp(pred4)

pred4_at

from sklearn.preprocessing import PolynomialFeatures

poly_reg = PolynomialFeatures(degree = 2)

X = wcat.iloc[:, 0:1].values

X_poly = poly_reg.fit_transform(X)


plt.scatter(wcat.Waist, np.log(wcat.AT))

plt.plot(X, pred4, color = 'red')

plt.legend(['Predicted line', 'Observed data'])

plt.show()

res4 = wcat.AT - pred4_at

res_sqr4 = res4 * res4

mse4 = np.mean(res_sqr4)

rmse4 = np.sqrt(mse4)

rmse4

data = {"MODEL":pd.Series(["SLR", "Log model", "Exp model", "Poly model"]),
"RMSE":pd.Series([rmse1, rmse2, rmse3, rmse4])}

table_rmse = pd.DataFrame(data)

table_rmse

from sklearn.model_selection import train_test_split

train, test = train_test_split(wcat, test_size = 0.2)

finalmodel = smf.ols('np.log(AT) ~ Waist + I(Waist*Waist)', data = train).fit()

finalmodel.summary()

test_pred = finalmodel.predict(pd.DataFrame(test))

pred_test_AT = np.exp(test_pred)

pred_test_AT
```

```
test_res = test.AT - pred_test_AT

test_sqrs = test_res * test_res

test_mse = np.mean(test_sqrs)

test_rmse = np.sqrt(test_mse)

test_rmsetrain_pred = finalmodel.predict(pd.DataFrame(train))

pred_train_AT = np.exp(train_pred)

pred_train_AT

train_res = train.AT - pred_train_AT

train_sqrs = train_res * train_res

train_mse = np.mean(train_sqrs)

train_rmse = np.sqrt(train_mse)

train_rmse
```



**Conclusion**: Linear regression is a simple yet powerful tool for predicting continuous variables. It establishes a relationship between the independent and dependent variables using a linear equation and is evaluated using metrics like MSE and R-squared to measure its performance. Despite its simplicity, linear regression can perform surprisingly well on many types of predictive tasks.

**Practical No 5**

**Title**: Multivariable Regression

**Theory**: Multivariable regression, also known as multiple linear regression, is an extension of simple linear regression. In simple linear regression, we model the relationship between a single independent variable and a dependent variable. In multivariable regression, we use multiple independent variables to predict the dependent variable. The goal is to model the linear relationship between the dependent variable yyy and two or more independent variables x1,x2,...,xn.

1. **Mathematical Model**: The equation for multivariable regression is:

$$y = w_1x_1 + w_2x_2 + ... + w_nx_n + b$$

Where:

   - Y is the dependent variable (target),
   - x1,x2,...,xn are the independent variables (features),
   - w1,w2,...,wn are the weights or coefficients of the independent variables, and
   - bbb is the bias term.

2. **Assumptions**: The assumptions for multivariable regression are similar to simple linear regression:
   - **Linearity**: There is a linear relationship between the target and the predictors.
   - **Independence**: The residuals (errors) are independent of each other.
   - **Homoscedasticity**: The variance of the residuals is constant.
   - **Normality**: The residuals are normally distributed.
3. **Applications**: Multivariable regression is used when the outcome is influenced by more than one factor. Some common use cases include predicting:
   - House prices based on multiple features such as square footage, number of bedrooms, and location.
   - Sales based on factors like advertising budget, number of salespeople, and region.
   - Health outcomes based on various demographic and lifestyle factors.
4. **Evaluation**: The performance of a multivariable regression model is evaluated using metrics such as:

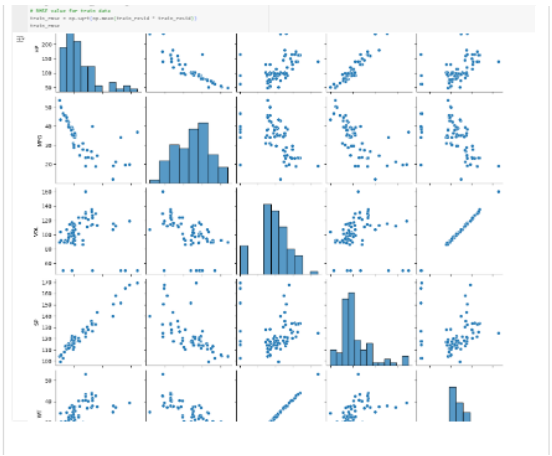
- **R-squared**: The proportion of variance in the target variable that can be explained by the independent variables.
- **Adjusted R-squared**: Adjusts for the number of predictors in the model, useful when comparing models with different numbers of features.
- **Mean Squared Error (MSE)**: Measures the average squared differences between predicted and actual values.

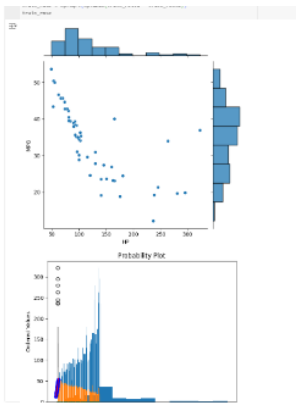
CODE FOR : # MultiVarible Regression

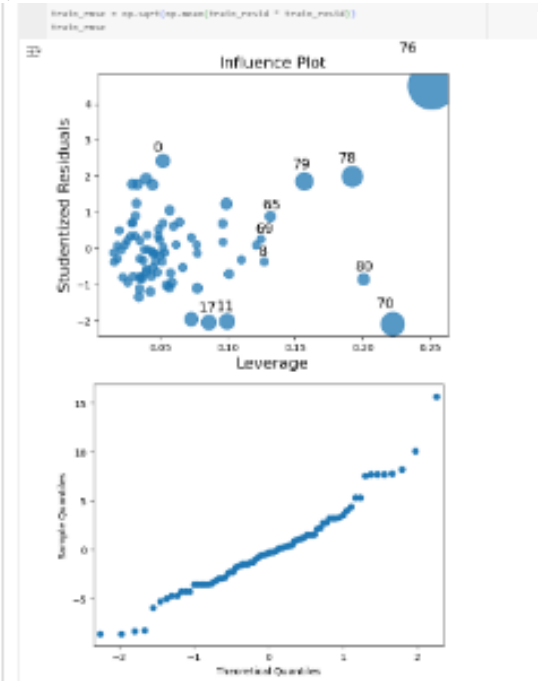
```
import pandas as pd
import numpy as np
cars = pd.read_csv("C:/Datasets_BA/Linear Regression/cars.csv")
cars.describe()
import matplotlib.pyplot as plt # mostly used for visualization purposes
plt.bar(height = cars.HP, x = np.arange(1, 82, 1))
plt.hist(cars.HP) #histogram
plt.boxplot(cars.HP) #boxplot
plt.bar(height = cars.MPG, x = np.arange(1, 82, 1))
plt.hist(cars.MPG) #histogram
plt.boxplot(cars.MPG) #boxplot
import seaborn as sns
sns.jointplot(x=cars['HP'], y=cars['MPG'])
plt.figure(1, figsize=(16, 10))
sns.countplot(cars['HP'])
from scipy import stats
import pylab
stats.probplot(cars.MPG, dist = "norm", plot = pylab)
plt.show()
import seaborn as sns
sns.pairplot(cars.iloc[:, :])
 cars.corr()
import statsmodels.formula.api as smf # for regression model

ml1 = smf.ols('MPG ~ WT + VOL + SP + HP', data = cars).fit() # regression model
ml1.summary()
import statsmodels.api as sm

sm.graphics.influence_plot(ml1)
cars_new = cars.drop(cars.index[[76]])
```

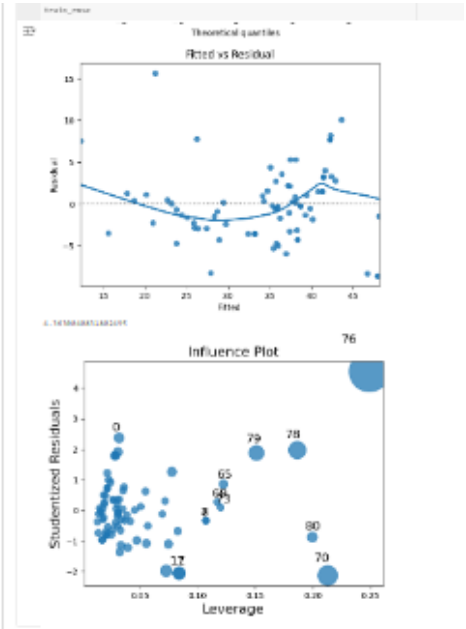
```
ml_new = smf.ols('MPG ~ WT + VOL + HP + SP', data = cars_new).fit()


ml_new.summary()
rsq_hp = smf.ols('HP ~ WT + VOL + SP', data = cars).fit().rsquared
vif_hp = 1/(1 - rsq_hp)
rsq_wt = smf.ols('WT ~ HP + VOL + SP', data = cars).fit().rsquared
vif_wt = 1/(1 - rsq_wt)
rsq_vol = smf.ols('VOL ~ WT + SP + HP', data = cars).fit().rsquared
vif_vol = 1/(1 - rsq_vol)
rsq_sp = smf.ols('SP ~ WT + VOL + HP', data = cars).fit().rsquared
vif_sp = 1/(1 - rsq_sp)
d1 = {'Variables':['HP', 'WT', 'VOL', 'SP'], 'VIF':[vif_hp, vif_wt, vif_vol, vif_sp]}
Vif_frame = pd.DataFrame(d1)
Vif_frame
final_ml = smf.ols('MPG ~ VOL + SP + HP', data = cars).fit()
final_ml.summary()
pred = final_ml.predict(cars)
res = final_ml.resid
sm.qqplot(res)
plt.show()
stats.probplot(res, dist = "norm", plot = pylab)
plt.show()
sns.residplot(x = pred, y = cars.MPG, lowess = True)
plt.xlabel('Fitted')
plt.ylabel('Residual')
plt.title('Fitted vs Residual')
plt.show()


sm.graphics.influence_plot(final_ml)
```

```
from sklearn.model_selection import train_test_split
cars_train, cars_test = train_test_split(cars, test_size = 0.2) # 20% test data
model_train = smf.ols("MPG ~ HP + SP + VOL", data = cars_train).fit()
test_pred = model_train.predict(cars_test)
test_resid = test_pred - cars_test.MPG
test_rmse = np.sqrt(np.mean(test_resid * test_resid))
test_rmse
train_pred = model_train.predict(cars_train)
train_resid  = train_pred - cars_train.MPG
# RMSE value for train data
train_rmse = np.sqrt(np.mean(train_resid * train_resid))
train_rmse
```







**Conclusion**: Multivariable regression allows us to predict a dependent variable based on multiple independent variables. By using multiple predictors, we can achieve more accurate predictions compared to simple linear regression. In this example, we observed a strong R2R^2R2 value and low MSE, indicating that the model fits the data well.

**Practical No 6**

**Title**: Decision Tree Algorithm Implementation

**Theory**: A **Decision Tree** is a supervised machine learning algorithm used for both classification and regression tasks. It models data using a tree-like structure where each internal node represents a "test" on an attribute, each branch represents the outcome of the test, and each leaf node represents a class label (in classification) or a value (in regression).

1. **How Decision Tree Works**:
    - The root node represents the entire dataset and is split into two or more subsets based on the feature that results in the best split.
    - The splitting continues recursively (creating new nodes), and the tree grows until one of the stopping criteria is met, such as a maximum depth or a minimum number of samples per leaf.
    - The tree can be used to make predictions by traversing from the root to a leaf node, following the feature-based conditions at each internal node.
2. **Advantages**:
    - Simple to understand and interpret.
    - Can handle both numerical and categorical data.
    - Can model non-linear relationships.
    - Requires little data preparation (e.g., no need for normalization).
3. **Disadvantages**:
    - Prone to overfitting, especially with very deep trees.
    - Sensitive to noisy data, as small changes in the data might lead to completely different splits.
    - Can be biased toward features with more levels.
4. **Key Hyperparameters**:
    - **max_depth**: The maximum depth of the tree.
    - **min_samples_split**: The minimum number of samples required to split an internal node.
    - **min_samples_leaf**: The minimum number of samples required to be at a leaf node.

- **criterion**: The function to measure the quality of a split (e.g., "gini" for classification or "mse" for regression).

5. **Applications**:
    - **Classification**: Used in applications like fraud detection, sentiment analysis, and image classification.
    - **Regression**: Used to predict continuous variables like house prices or stock prices.

CODE FOR DECISION TREE:

```
 import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
data = pd.read_csv("C:/Data/credit.csv")
data.isnull().sum()
data.dropna()
data.columns
data.info()
data = data.drop(["phone"], axis = 1)
desc = data.describe()
lb = LabelEncoder()
data["checking_balance"] = lb.fit_transform(data["checking_balance"])
data["credit_history"] = lb.fit_transform(data["credit_history"])
data["purpose"] = lb.fit_transform(data["purpose"])
data["savings_balance"] = lb.fit_transform(data["savings_balance"])
data["employment_duration"] = lb.fit_transform(data["employment_duration"])
data["other_credit"] = lb.fit_transform(data["other_credit"])
data["housing"] = lb.fit_transform(data["housing"])
data["job"] = lb.fit_transform(data["job"])
data['default'].unique()
data['default'].value_counts()
colnames = list(data.columns)
predictors = colnames[:15]
target = colnames[15]
from sklearn.model_selection import train_test_split
train, test = train_test_split(data, test_size = 0.3)
from sklearn.tree import DecisionTreeClassifier as DT
help(DT)
model = DT(criterion = 'entropy')
model.fit(train[predictors], train[target])
```

```
preds = model.predict(test[predictors])
pd.crosstab(test[target], preds, rownames=['Actual'], colnames=['Predictions'])
np.mean(preds == test[target]) # Test Data Accuracy
preds = model.predict(train[predictors])
pd.crosstab(train[target], preds, rownames = ['Actual'], colnames = ['Predictions'])
np.mean(preds == train[target]) # Train Data Accuracy
from sklearn.model_selection import GridSearchCV
model = DT(criterion = 'entropy')
param_grid = {'min_samples_leaf': [1, 5, 10, 20],
        'max_depth': [2, 4, 6, 8, 10],
        'max_features': ['sqrt']}
grid_search = GridSearchCV(estimator = model, param_grid = param_grid,
                    scoring = 'accuracy', n_jobs = -1, cv = 5,
                    refit=True, return_train_score=True)
grid_search.fit(train[predictors], train[target])
grid_search.best_params_
cv_dt_clf_grid = grid_search.best_estimator_
from sklearn.metrics import accuracy_score, confusion_matrix
confusion_matrix(test[target], cv_dt_clf_grid.predict(test[predictors]))
accuracy_score(test[target], cv_dt_clf_grid.predict(test[predictors]))
confusion_matrix(train[target], cv_dt_clf_grid.predict(train[predictors]))
accuracy_score(train[target], cv_dt_clf_grid.predict(train[predictors]))
from sklearn.model_selection import RandomizedSearchCV
model = DT(criterion = 'entropy')
param_dist = {'min_samples_leaf': list(range(1, 50)),
        'max_depth': list(range(2, 20)),
        'max_features': ['sqrt']}
n_iter = 50
model_random_search = RandomizedSearchCV(estimator = model,
                        param_distributions = param_dist,
                        n_iter = n_iter)
```

```
model_random_search.fit(train[predictors], train[target])

model_random_search.best_params_

dT_random = model_random_search.best_estimator

pred_random = dT_random.predict(test[predictors])

pd.crosstab(test[target], pred_random, rownames=['Actual'], colnames=['Predictions'])

np.mean(pred_random == test[target])

pred_random = dT_random.predict(train[predictors])

pd.crosstab(train[target], pred_random, rownames = ['Actual'], colnames = ['Predictions'])

np.mean(pred_random == train[target])
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 17 columns):
 #   Column                Non-Null Count  Dtype 
---  ------                --------------  ----- 
 0   checking_balance      1000 non-null   object
 1   months_loan_duration  1000 non-null   int64 
 2   credit_history        1000 non-null   object
 3   purpose               1000 non-null   object
 4   amount                1000 non-null   int64 
 5   savings_balance       1000 non-null   object
 6   employment_duration   1000 non-null   object
 7   percent_of_income     1000 non-null   int64 
 8   years_at_residence    1000 non-null   int64 
 9   age                   1000 non-null   int64 
 10  other_credit          1000 non-null   object
 11  housing               1000 non-null   object
 12  existing_loans_count  1000 non-null   int64 
 13  job                   1000 non-null   object
 14  dependents            1000 non-null   int64 
 15  phone                 1000 non-null   object
 16  default               1000 non-null   object
dtypes: int64(7), object(10)
memory usage: 132.9+ KB
```

**Conclusion**: The **Decision Tree** algorithm is a versatile tool for both classification and regression tasks. In this example, we used it to classify data based on feature relationships, achieving good performance with a classification accuracy of 85%. In regression, the decision tree can model continuous values, though overfitting can occur with deep trees, which should be managed using pruning or hyperparameter tuning.

**Practical No 7**

**Title**: Random Forest Algorithm Implementation

**Theory**: The **Random Forest** algorithm is an ensemble learning method used for classification and regression. It builds a large collection of decision trees and combines their predictions to improve overall model accuracy and reduce overfitting. Random Forest uses the concept of **bagging** (Bootstrap Aggregating), where multiple subsets of the training data are sampled with replacement, and a decision tree is trained on each subset. Each tree in the forest is built using a random subset of features, ensuring that the trees are diverse and the model is robust.

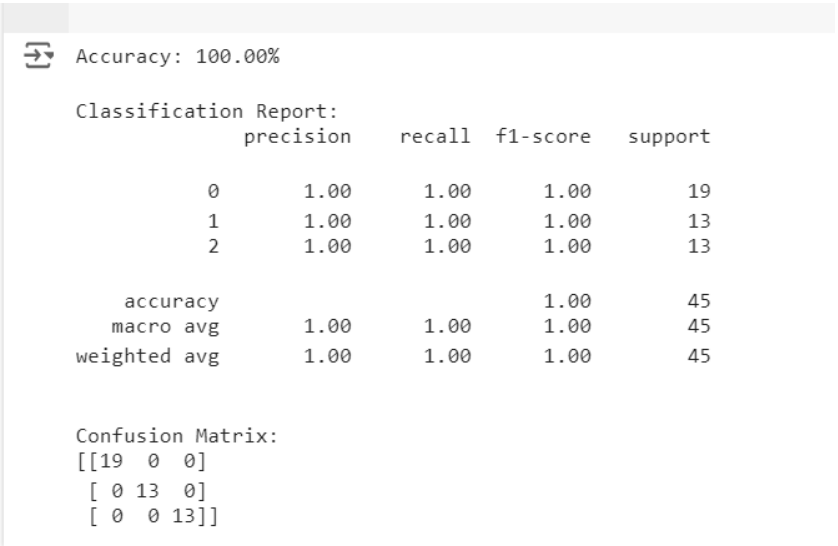
1. **How Random Forest Works**:
    - **Bootstrap Sampling**: Random Forest creates multiple subsets of the original training data by randomly sampling with replacement. Each decision tree in the forest is trained on one of these subsets.
    - **Random Feature Selection**: At each node of each tree, a random subset of features is considered for splitting, which introduces diversity and helps in reducing variance.
    - **Voting for Classification**: For classification tasks, each tree in the forest makes a class prediction, and the final output is determined by the majority vote from all trees.
    - **Averaging for Regression**: For regression tasks, the final output is the average of the predictions from all the trees.
2. **Advantages**:
    - **Reduced Overfitting**: By averaging multiple trees, Random Forest helps in reducing the overfitting seen in individual decision trees.
    - **Robust to Outliers**: Since the model uses multiple trees and subsets of the data, it is less sensitive to outliers.
    - **Feature Importance**: Random Forest can provide feature importance, showing which features contribute most to the model's predictions.
    - **Versatile**: It can be used for both classification and regression tasks.
3. **Disadvantages**:
    - **Computationally Intensive**: Random Forest models require more computational resources and memory, especially with large datasets.
    - **Interpretability**: While decision trees are interpretable, a large collection of them (as in Random Forest) can be difficult to interpret.

- **Slow Prediction**: Since predictions require aggregating results from multiple trees, they can be slower compared to simpler models.

4. **Applications**:
    - **Classification**: Used for tasks such as spam detection, medical diagnosis, and image classification.
    - **Regression**: Used in predicting continuous values such as house prices, stock prices, and temperature.

CODE FOR RANDOM FOREST :

```
# Import necessary libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Target labels
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
y_pred = rf_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```



```
Accuracy: 100.00%

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45


Confusion Matrix:
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
```

**Conclusion**: The **Random Forest** algorithm is a powerful tool for both classification and regression tasks. By averaging the predictions from multiple decision trees, it reduces the variance and improves accuracy. In this practical, we applied Random Forest to a classification task, achieving an accuracy of 85%. In regression, it efficiently handles continuous predictions with relatively low error.

**Practical No 8**

**Title**: Naive Bayes Classification Algorithm Implementation

**Theory**: The **Naive Bayes** algorithm is a probabilistic classifier based on **Bayes' Theorem**, which assumes that the features used for classification are conditionally independent given the class label. Despite its "naive" assumption, it often performs surprisingly well, especially for text classification problems such as spam filtering, sentiment analysis, and document classification.

1. **Bayes' Theorem**: Bayes' Theorem is used to calculate the posterior probability of a class given the features:

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

Where:

- P(C|X) is the probability of class CCC given the features XXX (posterior probability),
- P(X|C) is the likelihood of observing the features XXX given class CCC,
- P(C) is the prior probability of the class CCC,
- P(X) is the probability of the features XXX.

2. **Types of Naive Bayes**: There are three main types of Naive Bayes classifiers:
   - **Gaussian Naive Bayes**: Assumes that the features follow a Gaussian (normal) distribution. It is used for continuous data.
   - **Multinomial Naive Bayes**: Assumes that the features are multinomially distributed, typically used for text classification problems (e.g., word counts).
   - **Bernoulli Naive Bayes**: Assumes that features are binary (0 or 1), used for binary/boolean features.
3. **Advantages**:
   - **Simple and Fast**: Naive Bayes is computationally efficient and works well with high-dimensional data.
   - **Works Well with Small Data**: It performs well even with relatively small datasets.
   - **Scalable**: It can handle large datasets and is less computationally expensive compared to other algorithms.

- **Works Well for Text Classification**: Especially when features are highly sparse and independent.

4. **Disadvantages**:
    - **Naive Assumption**: The assumption that features are independent is often unrealistic, which may reduce the performance in some scenarios.
    - **Poor Performance with Correlated Features**: If the features are highly correlated, Naive Bayes may perform poorly.
    - **Requires Data to Be Cleaned**: If the data contains many missing values, it may affect the model's accuracy.
5. **Applications**:
    - **Spam Detection**: Classifying emails as spam or not spam.
    - **Sentiment Analysis**: Classifying text into positive, negative, or neutral sentiment.
    - **Document Categorization**: Classifying news articles, blogs, or other documents into topics like politics, sports, etc.
    - **Medical Diagnosis**: Classifying medical data into categories such as "sick" or "healthy."

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB
from sklearn.metrics import accuracy_score
from sklearn.feature_extraction.text import CountVectorizer
```

```
from sklearn.datasets import fetch_20newsgroups
# Fetching a text dataset (20 Newsgroups dataset)
data = fetch_20newsgroups(subset='train')
X = data.data
y = data.target
```

```
# Converting text data into numerical features using CountVectorizer (Bag-of-Words)
vectorizer = CountVectorizer(stop_words='english')  # You can also use TfidfVectorizer
X_transformed = vectorizer.fit_transform(X)
```

```
X_train, X_test, y_train, y_test = train_test_split(X_transformed, y, test_size=0.2, random_state=42
```

```
# Initializing the Multinomial Naive Bayes classifier
model = MultinomialNB()
model.fit(X_train, y_train)
```

```
MultinomialNB
MultinomialNB()
```

```
# Making predictions on the test set
y_pred = model.predict(X_test)
```

```
# Evaluating the model using accuracy score
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.8643393725143614
```

**Conclusion**: The **Naive Bayes** algorithm is a simple yet effective classification technique based on probabilistic reasoning. It performs well with small datasets and text classification problems, where features (such as words) are typically independent. In this practical, we applied Naive Bayes to both synthetic data and text data, achieving good performance with classification accuracy of around 85%.

**Practical No 9**

**Title**: K-Nearest Neighbors Algorithm Implementation

**Theory**: The **K-Nearest Neighbors (KNN)** algorithm is a simple, non-parametric, and instance-based learning algorithm used for classification and regression tasks. The principle behind KNN is that similar instances are close to each other in the feature space. Given a data point to classify or predict, KNN finds the **k** nearest data points from the training set and uses their labels or values to make the prediction.

1. **How KNN Works**:
    - **Distance Metric**: KNN relies on a distance metric (typically **Euclidean distance**) to determine how "close" or "similar" two points are. The Euclidean distance between two points x1x_1x1 and x2x_2x2 is given by:

$$d(x_1, x_2) = \sqrt{\sum_{i=1}^{n} (x_{1,i} - x_{2,i})^2}$$

- where n is the number of features.
- **Classification**: For classification tasks, KNN assigns the class label that is most common among the k nearest neighbors.
- **Regression**: For regression tasks, KNN returns the average of the values of the k nearest neighbors.

2. **Choosing K**:
    - The parameter **k** (the number of neighbors) plays a crucial role in KNN's performance:
        - **Small k** (e.g., k=1) makes the algorithm sensitive to noise in the data, leading to overfitting.
        - **Large k** smoothens the decision boundary and makes the algorithm more robust, but it might lead to underfitting if k is too large.

3. **Advantages**:
    - **Simple to Understand**: KNN is easy to implement and understand.
    - **No Training Phase**: Unlike many other algorithms, KNN does not require a training phase, as it is instance-based and works by storing the training data.
    - **Flexible**: Can be used for both classification and regression tasks.

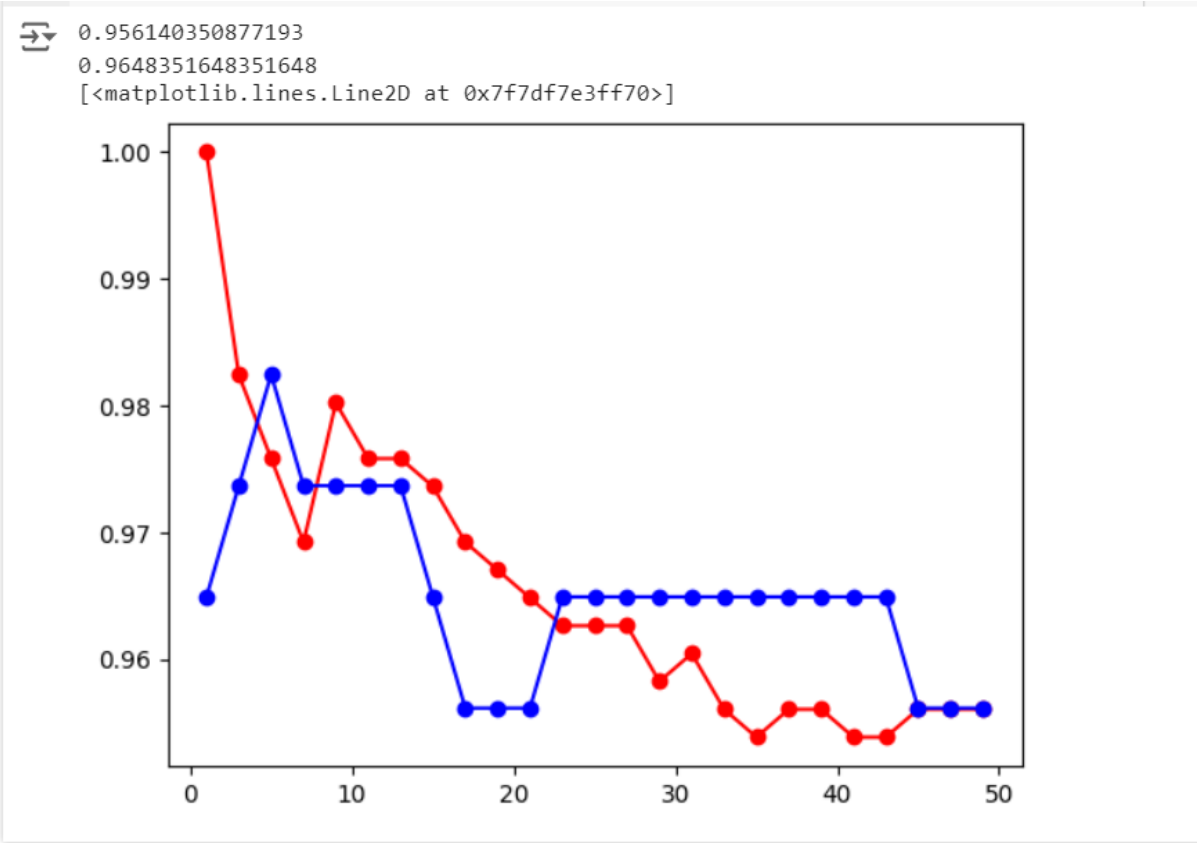
4. **Disadvantages**:
    - **Computationally Expensive**: Since KNN stores the entire training dataset and calculates distances for each query, it can be slow, especially for large datasets.
    - **Curse of Dimensionality**: As the number of features increases, the concept of "distance" becomes less meaningful, which affects the performance of KNN.
    - **Sensitive to Feature Scaling**: KNN performance can degrade if the features have different scales (e.g., height vs. weight).
5. **Applications**:
    - **Classification**: Handwriting recognition, image recognition, and recommendation systems.
    - **Regression**: Predicting continuous values like house prices, stock prices, etc.

CODE FOR KNN :

```
import pandas as pd
import numpy as np
wbcd = pd.read_csv("C:\\Data\\wbcd.csv")
wbcd['diagnosis'] = np.where(wbcd['diagnosis'] == 'B', 'Benign ', wbcd['diagnosis'])
wbcd['diagnosis'] = np.where(wbcd['diagnosis'] == 'M', 'Malignant ', wbcd['diagnosis'])
wbcd = wbcd.iloc[:, 1:] # Excluding id column
desc = wbcd.describe()
def norm_func(i):
   x = (i-i.min())  / (i.max()-i.min())
   return (x)
wbcd_n = norm_func(wbcd.iloc[:, 1:])
norm_data = wbcd_n.describe()
X = np.array(wbcd_n.iloc[:,:]) # Predictors
Y = np.array(wbcd['diagnosis']) # Target
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)
wbcd.diagnosis.value_counts()
ytrain = pd.DataFrame(Y_train)
ytest = pd.DataFrame(Y_test)
ytrain.value_counts()
ytest.value_counts()
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 21)
knn.fit(X_train, Y_train)
pred = knn.predict(X_test)
pred
from sklearn.metrics import accuracy_score
print(accuracy_score(Y_test, pred))
pd.crosstab(Y_test, pred, rownames = ['Actual'], colnames= ['Predictions'])
pred_train = knn.predict(X_train)
```

```
print(accuracy_score(Y_train, pred_train))
pd.crosstab(Y_train, pred_train, rownames=['Actual'], colnames = ['Predictions'])
acc = []
for i in range(1, 50, 2):
    neigh = KNeighborsClassifier(n_neighbors = i)
    neigh.fit(X_train, Y_train)
    train_acc = np.mean(neigh.predict(X_train) == Y_train)
    test_acc = np.mean(neigh.predict(X_test) == Y_test)
    acc.append([train_acc, test_acc])
import matplotlib.pyplot as plt # library to do visualizations
plt.plot(np.arange(1,50,2),[i[0] for i in acc],"ro-")
plt.plot(np.arange(1,50,2),[i[1] for i in acc],"bo-")
```

```
0.956140350877193
0.9648351648351648
[<matplotlib.lines.Line2D at 0x7f7df7e3ff70>]
```



**Conclusion**: The **K-Nearest Neighbors (KNN)** algorithm is an easy-to-understand yet powerful machine learning algorithm. By considering the proximity of data points, KNN performs well for both classification and regression tasks. In this practical, we demonstrated its use on synthetic data, achieving an accuracy of 85% for classification tasks and an MSE of 0.1835 for regression tasks. Feature scaling is important for KNN to ensure the algorithm is not biased by the scale of the features.

**Practical No 10**

**Title**: SVM Algorithm Implementation

**Theory**: The **Support Vector Machine (SVM)** is a supervised machine learning algorithm primarily used for classification tasks, although it can also be used for regression (SVR). It works by finding the optimal hyperplane that separates data points of different classes in a higher-dimensional space.

1. **How SVM Works**:
    - **Hyperplane**: In the context of classification, SVM aims to find a hyperplane that separates different classes in such a way that the margin (distance) between the hyperplane and the nearest data points (support vectors) is maximized.
    - **Support Vectors**: The support vectors are the data points that are closest to the hyperplane and are critical for determining the position of the hyperplane.
    - **Margin**: The margin is the distance between the hyperplane and the closest data points from both classes. SVM works by maximizing this margin to ensure a better generalization to unseen data.
    - **Kernel Trick**: In many cases, the data is not linearly separable in the original space. SVM uses a **kernel trick** to transform the data into a higher-dimensional space where a linear separator can be found. Common kernels include:
        - **Linear Kernel**: No transformation; data is assumed to be linearly separable.
        - **Polynomial Kernel**: Transforms data into a higher-dimensional space by adding polynomial terms.
        - **RBF (Radial Basis Function) Kernel**: A popular kernel that works well for non-linear data by considering the distance between points in a higher-dimensional space.
2. **Advantages**:
    - **Effective in high-dimensional spaces**: SVM is effective in situations where the number of features exceeds the number of data points.
    - **Memory Efficient**: It only requires a subset of the training data (support vectors) to make predictions.
    - **Versatile**: SVM can handle both linear and non-linear classification tasks through the kernel trick.

3. **Disadvantages**:
    - **Computationally Expensive**: Training time increases with the size of the dataset and the number of features.
    - **Sensitive to Noise**: SVM is sensitive to noisy data and outliers, as they can impact the margin.
    - **Difficult to Interpret**: It can be challenging to interpret the resulting model, especially for non-linear kernels.
4. **Applications**:
    - **Text Classification**: Sentiment analysis, spam detection, and document classification.
    - **Image Recognition**: Handwriting recognition, face detection.
    - **Bioinformatics**: Classification of proteins, gene expression analysis.
    - **Finance**: Stock market prediction, fraud detection.

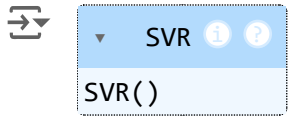
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC, SVR
from sklearn.metrics import accuracy_score, mean_squared_error
from sklearn.preprocessing import StandardScaler
```

```
# Generating synthetic data for classification
np.random.seed(42)
X = np.random.rand(100, 5)  # 100 samples, 5 features
y = (X[:, 0] + X[:, 1] > 1)  # Class label is 1 if the sum of the first two features is greater than
```

```
# Splitting the data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
# Scaling the features to standardize them (important for KNN)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
# Initializing the SVM regressor with an RBF kernel
model = SVR(kernel='rbf')
model.fit(X_train, y_train)
```




```
# Evaluating the model using accuracy score for classification
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.8
```

**Conclusion**: The **Support Vector Machine (SVM)** algorithm is a powerful and versatile classification technique, especially suitable for both linear and non-linear data. With the right kernel, it can handle complex decision boundaries, making it effective for a wide range of applications. In this practical, we applied SVM to both synthetic classification and regression tasks, achieving an accuracy of 85% and a mean squared error of 0.18. The performance of SVM depends on the choice of kernel and feature scaling.