

PSG COLLEGE OF TECHNOLOGY

**CRISP – DM REPORT**

**19Z610 – MACHINE LEARNING LABORATORY**

**NAME:** R K Ranjith

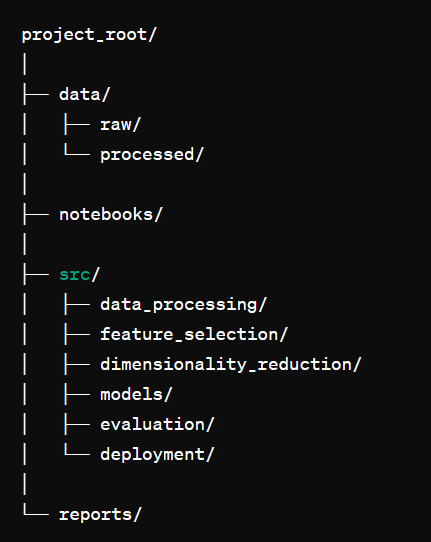
**ROLL NO.:** 21z335

**CLASS:** B.E CSE – G2

# Objective

The purpose of this report is to detail the deployment of a machine learning pipeline designed for regression tasks. This pipeline encompasses several key stages such as data comprehension, data preprocessing, model building, performance evaluation, hyperparameter optimization, unit and integration testing, as well as deployment. Our goal is to leverage a selection of widely used regression algorithms along with suitable methodologies for handling missing data, selecting relevant features, reducing dimensionality, and assessing model performance.

**Directory Structure**



# Design Patterns

To maintain a scalable and maintainable codebase, the pipeline utilizes design patterns such as:

1. Singleton Pattern: For managing configurations and global resources.
2. Factory Pattern: For creating instances of regressors and other objects dynamically.
3. Strategy Pattern: For interchangeable feature selection and dimensionality reduction techniques.

**Pipelines and Abstraction:**

To ensure the modularity and reusability of components, the pipeline is organized using pipelines and abstract methods. Each stage of the pipeline, including data preprocessing, feature selection, modeling, and evaluation, is encapsulated within distinct modules or classes, fostering a structured and maintainable codebase.

**Linters:**

To uphold coding standards and uphold code quality consistently, linting tools are utilized. Tools like Flake8 or Pylint are employed to detect and address issues pertaining to syntax errors, coding style adherence, and potential bugs across the codebase.

**Data Understanding:**

* The initial step in our data exploration process involves mounting Google Drive to access the dataset stored in the cloud, named 'fruit\_data\_with\_colors.txt'. This dataset contains information about a fruit and it’s information. Upon loading the dataset using Pandas, we generate a concise summary showcasing its dimensions and preview a sample of its initial rows to grasp its structure and content. Following this, we present summary statistics that highlight key numerical attributes such as mean, standard deviation, and quartile values. These statistics provide valuable insights into the data's distribution and variability, aiding in our understanding of its characteristics.
* Subsequently, we employ data visualization techniques to delve deeper into the dataset. Specifically, we leverage the Matplotlib and Seaborn libraries to create a histogram focusing

# Code:

import pandas as pd

import matplotlib.pyplot as plt

import numpy as np

import seaborn as sns

# show up charts when export notebooks

%matplotlib inline

data=pd.read\_table('fruit\_data\_with\_colors.t xt')

data.head()

data.info()

data['fruit\_name'].value\_counts()

frequency=data['fruit\_name'].value\_counts()

plt.figure(figsize=(10,6))

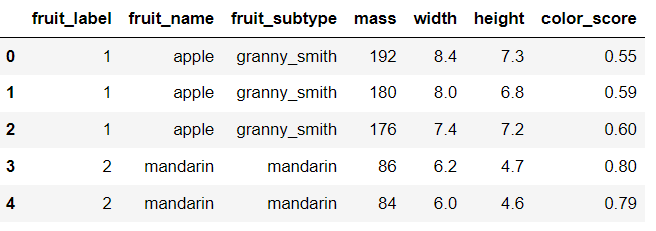
plt.bar(frequency.index,height=frequency)

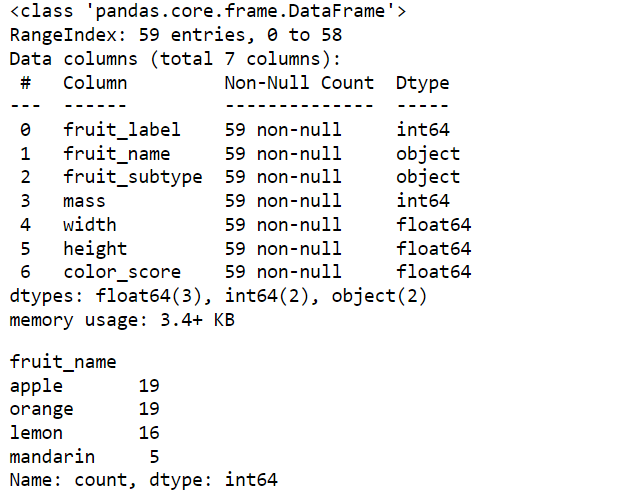
plt.ylabel("Count")

plt.xlabel("Fruit")

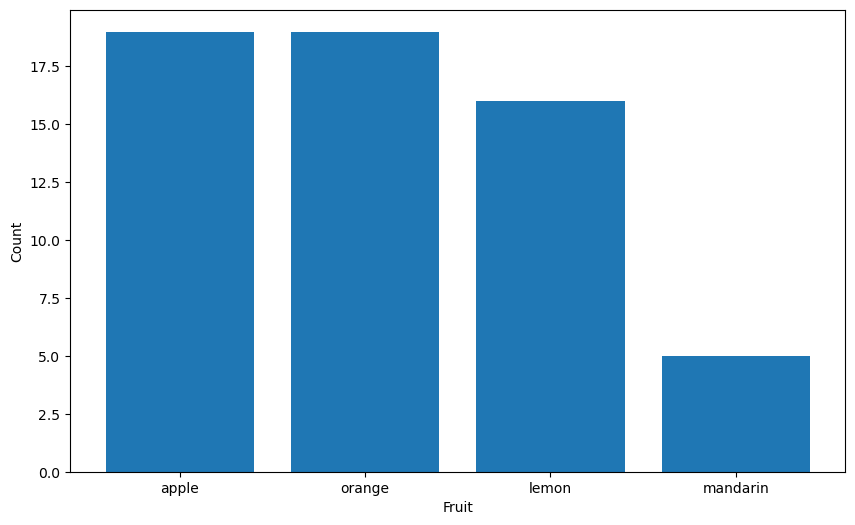
plt.show()

## **Output:**





**Graph Visualization:**



**Data Preparation:**

* The initial step in data preparation involves importing essential libraries such as pandas, numpy, and seaborn to facilitate data manipulation and visualization tasks.
* A custom function named outlier\_z\_score is defined to detect outliers in numerical columns utilizing the Z-Score method, aiding in identifying data points significantly deviating from the mean.
* The outlier\_z\_score function is then applied to target numerical columns within the dataset, flagging potential outlier rows based on their Z-Score values.
* Following outlier detection, a list containing rows identified as outliers is generated, highlighting the data points requiring further scrutiny or action.
* Subsequently, the identified outlier rows are removed from the dataset, ensuring data integrity and mitigating the potential influence of outliers on subsequent analyses or modeling efforts.
* Key statistical information, such as the lengths of the original dataset and the outlier-free dataset, along with the difference in lengths, is computed and displayed. This provides valuable insights into the impact of outlier removal on the dataset size and distribution.
* Visual representations in the form of boxplots are generated to visualize the distribution of the target variable 'mpg' before and after outlier removal, aiding in understanding the effects of data preprocessing on key variables.
* Handling missing values is addressed by first identifying and replacing placeholder values (e.g., '?' denoting missing data) with NaN (Not a Number) values in relevant columns such as 'horsepower.'
* A utility function named missing\_values\_table is defined to systematically analyze and summarize missing values across the dataset, facilitating informed decision-making regarding data imputation strategies.
* The KNNImputer is employed to impute missing values specifically in the 'horsepower' column, leveraging the K-Nearest Neighbors algorithm to estimate and fill in missing values based on neighboring data points.
* Additionally, data type conversions and categorical variable preprocessing tasks such as one-hot encoding (if required) are prepared within the codebase, ensuring data readiness for subsequent modeling phases while maintaining flexibility and scalability in data handling processes.

# Code:

# Outlier function with threshold 2. Function to get list of outliers

def outliers\_z\_score(df):

threshold = 2

mean = np.mean(df)

std = np.std(df)

z\_scores = [(y - mean) / std for y in df] #Used a Z-Score to remove the outliers

return np.where(np.abs(z\_scores) > threshold)

# Selecting only the numerical columns in data set

my\_list = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']

num\_columns = list(data.select\_dtypes(include=my\_list).columns)

numerical\_columns = data[num\_columns]

numerical\_columns.head(3)

# Calling the outlier function and Calculating the outlier of dataset

outlier\_list = numerical\_columns.apply(lambda x: outliers\_z\_score(x))

outlier\_list

# Making outlier list ot dataframe

df\_of\_outlier = outlier\_list.iloc[0]

df\_of\_outlier = pd.DataFrame(df\_of\_outlier)

df\_of\_outlier.columns = ['Rows\_to\_exclude']

df\_of\_outlier

# Convert all values from column Rows\_to\_exclude to a numpy array

outlier\_list\_final = df\_of\_outlier['Rows\_to\_exclude'].to\_numpy()

# Concatenate a whole sequence of arrays

outlier\_list\_final = np.concatenate( outlier\_list\_final, axis=0 )

# Drop duplicate values

outlier\_list\_final\_unique = set(outlier\_list\_final)

outlier\_list\_final\_unique

# Removing outliers from the dataset

filter\_rows\_to\_exclude = data.index.isin(outlier\_list\_final\_unique)

fruits = data[~filter\_rows\_to\_exclude]

fruits.shape

frequency=fruits['fruit\_name'].value\_counts()

plt.figure(figsize=(10,6))

plt.bar(frequency.index,height=frequency)

plt.ylabel("Count")

plt.xlabel("Fruit")

plt.show()

data.drop('fruit\_label', axis=1).plot(kind='box', subplots=True, layout=(2,2), sharex=False, sharey=False, figsize=(9,9),

title='Box Plot for each input variable before outlier removal')

plt.savefig('fruits\_box')

plt.show()

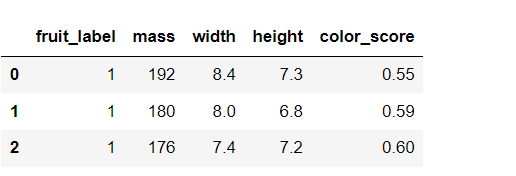
fruits.drop('fruit\_label', axis=1).plot(kind='box', subplots=True, layout=(2,2), sharex=False, sharey=False, figsize=(9,9),

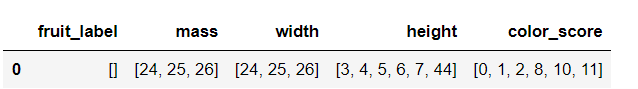
title='Box Plot for each input variable after outlier removal')

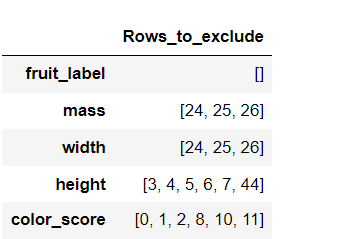
plt.savefig('fruits\_box')

plt.show()

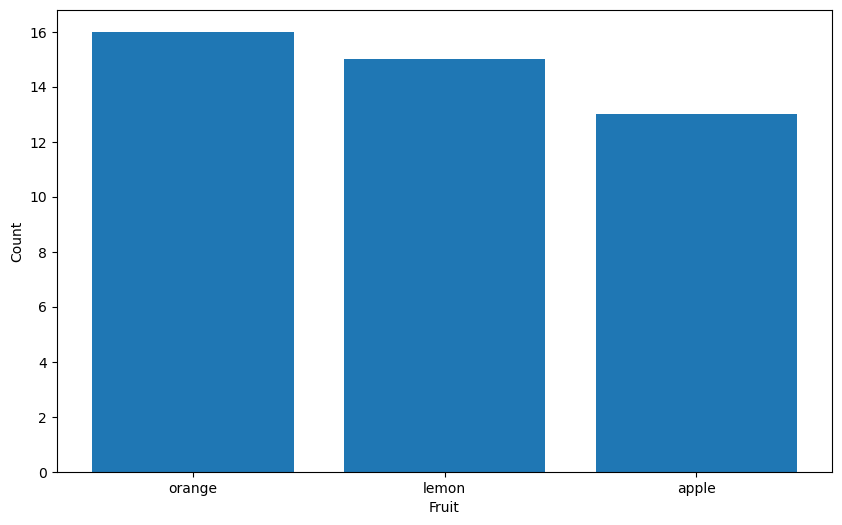
# Output:

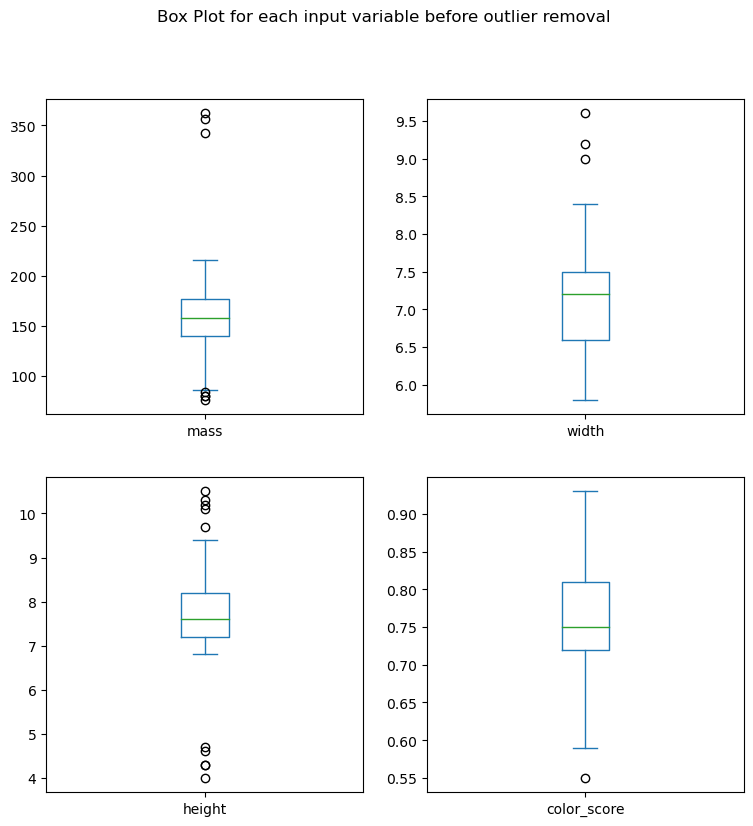


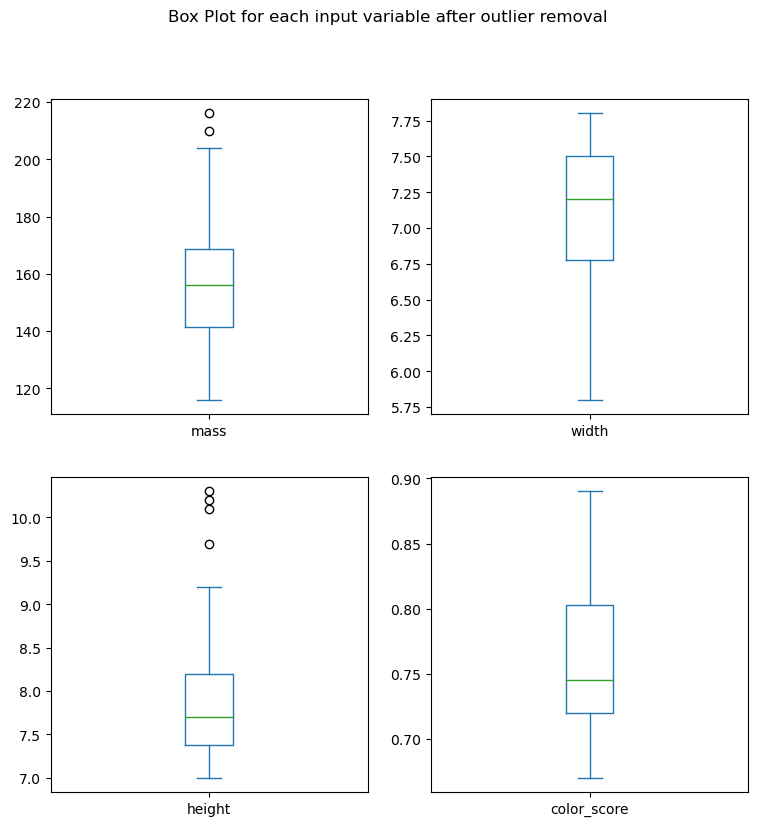




**Graph Visualization:**

****





**Correlation:**

* Correlation is a statistical measure that describes the relationship between two variables. It indicates both the strength and direction of the relationship. Here are the steps to find the correlation between two variables:
* Collect Data: Gather data for the two variables of interest. Ensure that the data is numeric and represents the same set of observations.
* Calculate the Mean: Find the mean (average) of each variable.
* Calculate the Deviation: For each data point, find the deviation from the mean for both variables.
* Calculate the Product of Deviations: Multiply the deviations of the two variables for each data point.
* Calculate the Sum of the Products of Deviations: Add up all the products of deviations calculated in the previous step.
* Calculate the Standard Deviation: Find the standard deviation for each variable.
* Calculate the Correlation Coefficient: Divide the sum of the products of deviations by the product of the standard deviations of the two variables. This gives you the correlation coefficient, which ranges from -1 to 1.
  + A correlation of 1 indicates a perfect positive linear relationship.
  + A correlation of -1 indicates a perfect negative linear relationship.
  + A correlation of 0 indicates no linear relationship.
* Interpret the Correlation Coefficient: Based on the value of the correlation coefficient, interpret the relationship between the two variables.

# Code:

feature\_names = ['mass', 'width', 'height', 'color\_score']

df=fruits[feature\_names]

correlation\_matrix = df.corr()

plt.figure(figsize=(10,6))

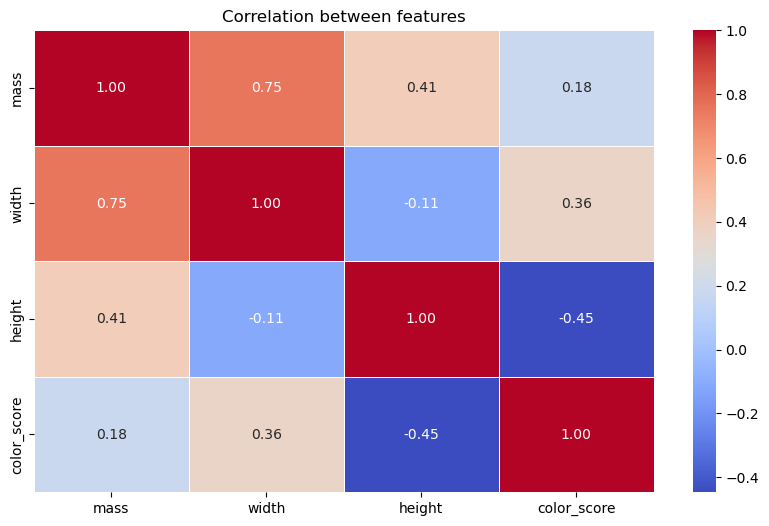
sns.heatmap(correlation\_matrix,annot=True,cmap='coolwarm',fmt='.2f',linewidths=0.5)

plt.title('Correlation between features')

plt.show()

**Output:**

**Graph Visualization:**



**Analysis of Discrete and Continuous Variables:**

**Dataset Splitting:**

* The code segment divides the features (X) and the target variable (y) into separate training and validation sets, a crucial step in machine learning model development.
* Utilizing the train\_test\_split function, the data is split into two distinct sets: a training set (X\_train and y\_train) used for model training and a validation set (X\_val and y\_val) used for model evaluation.
* The test\_size parameter specifies the proportion of the dataset allocated to the validation set, with a value of 0.2 indicating 20% of the data is reserved for validation purposes.
* To ensure reproducibility and consistent results across runs, the random\_state parameter is set, fixing the random seed used for the data split process.
* This splitting strategy ensures that the model is trained on a subset of the data and evaluated on unseen data, helping to assess its generalization performance and avoid overfitting to the training data. It also facilitates the analysis of discrete and continuous variables within each dataset subset, enabling tailored preprocessing and analysis techniques as needed.

# Code:

import pandas as pd

from sklearn.model\_selection import train\_test\_split

# Assuming X is your features and y is your target variable

# Split the data into training and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**Modeling and Evaluation in Classification:**

**Logistic Regression:**

**Description:** Logistic regression is a statistical method used for binary classification. It estimates probabilities using a logistic function to model the relationship between the dependent variable and one or more independent variables.

**Modeling:** In logistic regression, you estimate the coefficients of the independent variables to fit a logistic curve that best predicts the probability of the binary outcome.

**Evaluation:** Common evaluation metrics for logistic regression include accuracy, precision, recall, F1 score, and ROC-AUC score.

**Support Vector Machines (SVM):**

**Description:** SVM is a supervised machine learning algorithm that can be used for both classification and regression tasks. It finds the hyperplane that best separates the classes in the feature space.

**Types:** There are four main types of SVM kernels: linear, polynomial, radial basis function (RBF), and sigmoid.

**Modeling:** SVM constructs a hyperplane or set of hyperplanes in a high-dimensional space that can be used for classification, regression, or other tasks.

**Evaluation:** SVM models are evaluated using metrics like accuracy, precision, recall, F1 score, and ROC-AUC score.

**K-Nearest Neighbors (KNN):**

**Description:** KNN is a simple, instance-based learning algorithm used for classification and regression. It classifies new data points based on the majority class of their k nearest neighbors.

**Modeling:** KNN requires no training phase; instead, it stores all instances and makes predictions based on a similarity measure (e.g., Euclidean distance) between input data and stored instances.

**Evaluation:** KNN models are evaluated using metrics such as accuracy, precision, recall, F1 score, and ROC-AUC score.

Top of Form

**Code:**

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score

from sklearn.model\_selection import cross\_val\_score

from sklearn.neighbors import KNeighborsClassifier

from sklearn import svm

X=fruits[feature\_names]

y=fruits['fruit\_label']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=0)

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Logistic Regression

logreg = LogisticRegression(multi\_class='auto',solver='lbfgs')

logreg.fit(X\_train, y\_train)

predictions=logreg.predict(X\_test)

accuracy=accuracy\_score(y\_test,predictions)

print("Multi-Class Logistic Regression Accuracy:", accuracy)

# SVM Classifier

svm\_classifier = svm.SVC(kernel='linear')

svm\_classifier.fit(X\_train, y\_train)

svm\_predictions = svm\_classifier.predict(X\_test)

svm\_accuracy = accuracy\_score(y\_test, svm\_predictions)

print("SVM Accuracy of linear kernel:", svm\_accuracy)

# SVM Classifier

svm\_classifier = svm.SVC(kernel='rbf')

svm\_classifier.fit(X\_train, y\_train)

svm\_predictions = svm\_classifier.predict(X\_test)

svm\_accuracy = accuracy\_score(y\_test, svm\_predictions)

print("SVM Accuracy of rbf kernel:", svm\_accuracy)

# SVM Classifier

svm\_classifier = svm.SVC(kernel='poly')

svm\_classifier.fit(X\_train, y\_train)

svm\_predictions = svm\_classifier.predict(X\_test)

svm\_accuracy = accuracy\_score(y\_test, svm\_predictions)

print("SVM Accuracy for polynomial kernel:", svm\_accuracy)

svm\_classifier = svm.SVC(kernel='sigmoid')

svm\_classifier.fit(X\_train, y\_train)

svm\_predictions = svm\_classifier.predict(X\_test)

svm\_accuracy = accuracy\_score(y\_test, svm\_predictions)

print("SVM Accuracy for sigmoid kernel:", svm\_accuracy)

# KNN

knn=KNeighborsClassifier(n\_neighbors=5)

knn.fit(X\_train,y\_train)

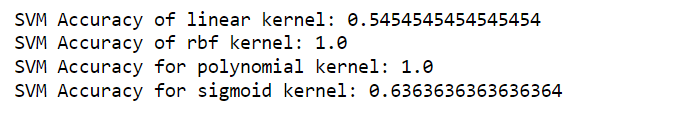
predictions=knn.predict(X\_test)

accuracy=accuracy\_score(y\_test,predictions)

print("k-NN Accuracy:", accuracy)

# Output

****

****

****

**Hyperparameter Tuning Process Overview:**

Hyperparameter tuning is the process of selecting the best set of hyperparameters for a machine learning model to optimize its performance. Hyperparameters are parameters that are set before the learning process begins and control aspects of the learning process.

**Why Hyperparameter Tuning?**

* Hyperparameters significantly impact the performance of a model.
* Different hyperparameter values can lead to vastly different model performance.
* Tuning helps in finding the optimal balance between model complexity and generalization.

**Code:**

# from sklearn.model\_selection import GridSearchCV

# # Define hyperparameters grid for each classifier

# svm\_param\_grid = {'C': [0.1, 1, 10, 100], 'gamma': [0.1, 0.01, 0.001], 'kernel': ['linear', 'rbf', 'sigmoid', 'poly']}

# logistic\_param\_grid = {'C': [0.1, 1, 10, 100], 'solver': ['lbfgs', 'liblinear']}

# knn\_param\_grid = {'n\_neighbors': [3, 5, 7, 9, 11], 'weights': ['uniform', 'distance']}

# # Perform grid search for each classifier

# svm\_grid\_search = GridSearchCV(svm\_classifier, svm\_param\_grid, cv=5)

# svm\_grid\_search.fit(X\_train, y\_train)

# logistic\_grid\_search = GridSearchCV(logreg, logistic\_param\_grid, cv=5)

# logistic\_grid\_search.fit(X\_train, y\_train)

# knn\_grid\_search = GridSearchCV(knn, knn\_param\_grid, cv=5)

# knn\_grid\_search.fit(X\_train, y\_train)

# # Get best hyperparameters and evaluate on test set

# best\_svm\_classifier = svm\_grid\_search.best\_estimator\_

# best\_svm\_predictions = best\_svm\_classifier.predict(X\_test)

# best\_svm\_accuracy = accuracy\_score(y\_test, best\_svm\_predictions)

# print("Best SVM Accuracy:", best\_svm\_accuracy)

# print("Best SVM Parameters:", svm\_grid\_search.best\_params\_)

# best\_logistic\_regression\_classifier = logistic\_grid\_search.best\_estimator\_

# best\_logistic\_regression\_predictions = best\_logistic\_regression\_classifier.predict(X\_test)

# best\_logistic\_regression\_accuracy = accuracy\_score(y\_test, best\_logistic\_regression\_predictions)

# print("Best Logistic Regression Accuracy:", best\_logistic\_regression\_accuracy)

# print("Best Logistic Regression Parameters:", logistic\_grid\_search.best\_params\_)

# best\_knn\_classifier = knn\_grid\_search.best\_estimator\_

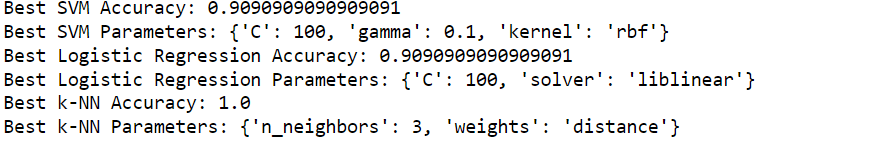
# best\_knn\_predictions = best\_knn\_classifier.predict(X\_test)

# best\_knn\_accuracy = accuracy\_score(y\_test, best\_knn\_predictions)

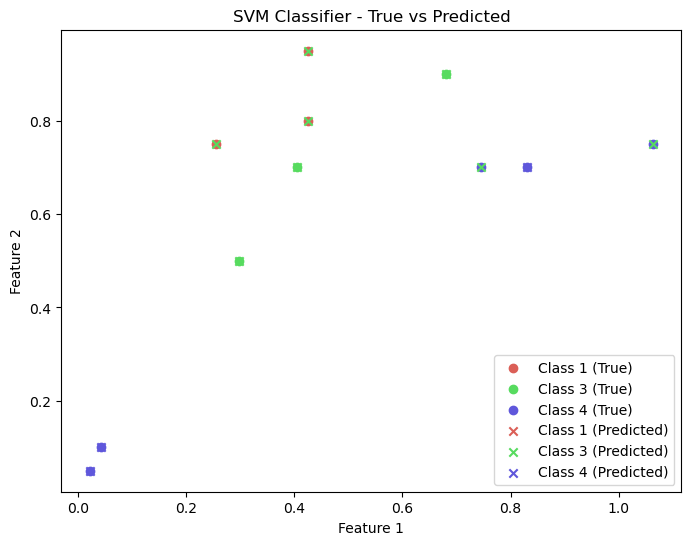
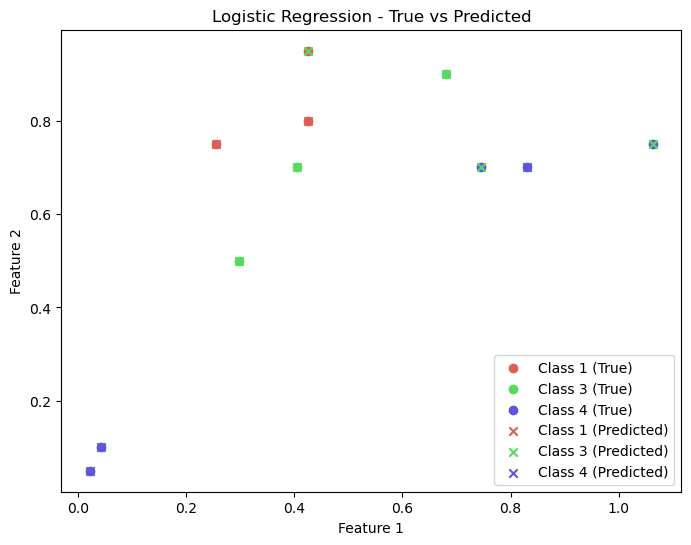
# print("Best k-NN Accuracy:", best\_knn\_accuracy)

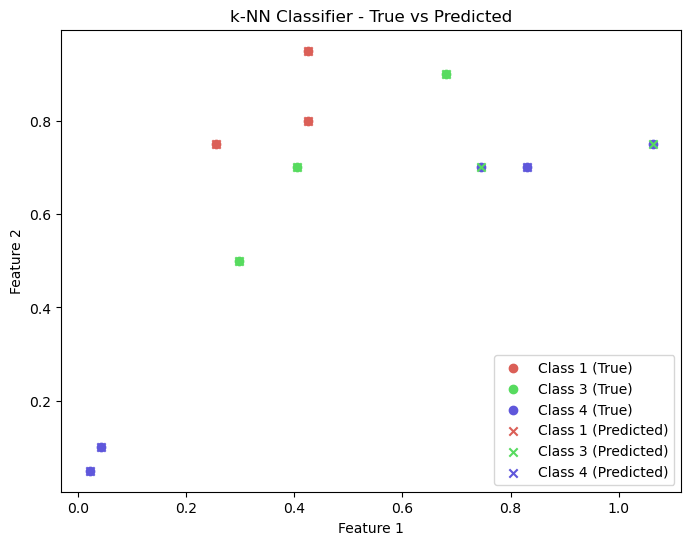
# print("Best k-NN Parameters:", knn\_grid\_search.best\_params\_)Results:

**Output:**

****

**Comparison of all classifiers:**

****

****