UNIVERSITY OF MUMBAI

DEPARTMENT OF COMPUTER SCIENCE

मुंबई विद्यापीठ

University of Mumbai

Re-accredited with A++ Grade

(CGPA 3.65) by NAAC (3rd Cycle 2021)

M.Sc. Computer Science – Semester II

(NEP 2020)

Machine Learning

JOURNAL

2023-2024

Seat No. ____________________

मुंबई विद्यापीठ
University of Mumbai
Re-accredited with A++ Grade
(CGPA 3.65) by NAAC (3rd Cycle 2021)

UNIVERSITY OF MUMBAI

DEPARTMENT OF COMPUTER SCIENCE

CERTIFICATE

This is to certify that the work entered in this journal was done in the University Department of Computer Science laboratory by Mr./Ms.______________________________ Seat No. ______________ for the course of M.Sc. (Computer Science) - Semester II (NEP 2020) during the academic year 2023- 2024 in a satisfactory manner.

____

Subject In-charge

____

Head of Department

____

External Examiner

INDEX

Sr. no.	Name of the practical	Page No.	Date	Sign
1	Implement Linear Regression(Diabetes Dataset).			
2	Implement Logistic Regression. (Iris Dataset)			
3	Implement Multinomial Logistic Regression (Iris Dataset)			
4	Implement SVM Classifier (Iris Datasets)			
5	Train and fine-tune a Decision Tree for Moon dataset.			
6	Train an SVM regression on the California Housing Dataset.			
7	Implement Batch Gradient Descent with Early Stopping for Softmax Regression.			
8	Implement MLP for Classification of Handwritten digits(MNIST datasets)			
9	Classification of Image of clothing using Tensorflow(Fashion MNIST dataset)			
10	Implement Regression to predict fuel efficiency using TensorFlow (Auto MPG dataset).			

PRACTICAL-01

Aim: Implement Linear Regression(Diabetes Dataset).

Theory:

Linear regression analysis is used to predict the value of a variable based on the value of another variable. The variable you want to predict is called the dependent variable. The variable you are using to predict the other variable's value is called the independent variable.

Code :

```
import numpy as np
import pandas as pd
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt
import seaborn as sns

# Load the Diabetes dataset from sklearn
diabetes = datasets.load_diabetes()

# Create a DataFrame from the dataset
diabetes_df = pd.DataFrame(data=diabetes.data,
columns=diabetes.feature_names)
diabetes_df['target'] = diabetes.target
```

```
# Display basic information about the dataset(EDA)
print("Dataset Shape:", diabetes_df.shape)
print("\nColumns:", diabetes_df.columns)
print("\nInfo:")
print(diabetes_df.info())
print("\nNull Values:")
print(diabetes_df.isnull().sum())
```

```
Dataset Shape: (442, 11)

Columns: Index(['age', 'sex', 'bmi', 'bp', 's1', 's2', 's3', 's4', 's5', 's6',
       'target'],
      dtype='object')
```

```
Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 442 entries, 0 to 441
Data columns (total 11 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   age     442 non-null    float64
 1   sex     442 non-null    float64
 2   bmi     442 non-null    float64
 3   bp      442 non-null    float64
 4   s1      442 non-null    float64
 5   s2      442 non-null    float64
 6   s3      442 non-null    float64
 7   s4      442 non-null    float64
 8   s5      442 non-null    float64
 9   s6      442 non-null    float64
 10  target  442 non-null    float64
dtypes: float64(11)
memory usage: 38.1 KB
...
s5        0
s6        0
target    0
dtype: int64
```

```
# Display summary statistics of numerical columns
print("\nSummary Statistics:")
```

print(diabetes_df.describe())

```
Summary Statistics:
                age           sex           bmi            bp            s1  \
count  4.420000e+02  4.420000e+02  4.420000e+02  4.420000e+02  4.420000e+02
mean  -1.444295e-18  2.543215e-18 -2.255925e-16 -4.854086e-17 -1.428596e-17
std    4.761905e-02  4.761905e-02  4.761905e-02  4.761905e-02  4.761905e-02
min   -1.072256e-01 -4.464164e-02 -9.027530e-02 -1.123988e-01 -1.267807e-01
25%   -3.729927e-02 -4.464164e-02 -3.422907e-02 -3.665608e-02 -3.424784e-02
50%    5.383060e-03 -4.464164e-02 -7.283766e-03 -5.670422e-03 -4.320866e-03
75%    3.807591e-02  5.068012e-02  3.124802e-02  3.564379e-02  2.835801e-02
max    1.107267e-01  5.068012e-02  1.705552e-01  1.320436e-01  1.539137e-01

                 s2            s3            s4            s5            s6  \
count  4.420000e+02  4.420000e+02  4.420000e+02  4.420000e+02  4.420000e+02
mean   3.898811e-17 -6.028360e-18 -1.788100e-17  9.243486e-17  1.351770e-17
std    4.761905e-02  4.761905e-02  4.761905e-02  4.761905e-02  4.761905e-02
min   -1.156131e-01 -1.023071e-01 -7.639450e-02 -1.260971e-01 -1.377672e-01
25%   -3.035840e-02 -3.511716e-02 -3.949338e-02 -3.324559e-02 -3.317903e-02
50%   -3.819065e-03 -6.584468e-03 -2.592262e-03 -1.947171e-03 -1.077698e-03
75%    2.984439e-02  2.931150e-02  3.430886e-02  3.243232e-02  2.791705e-02
max    1.987880e-01  1.811791e-01  1.852344e-01  1.335973e-01  1.356118e-01

           target
count  442.000000
mean   152.133484
```

 # Visualize the distribution of the target variable (diabetes progression)

plt.figure(figsize=(8, 6))

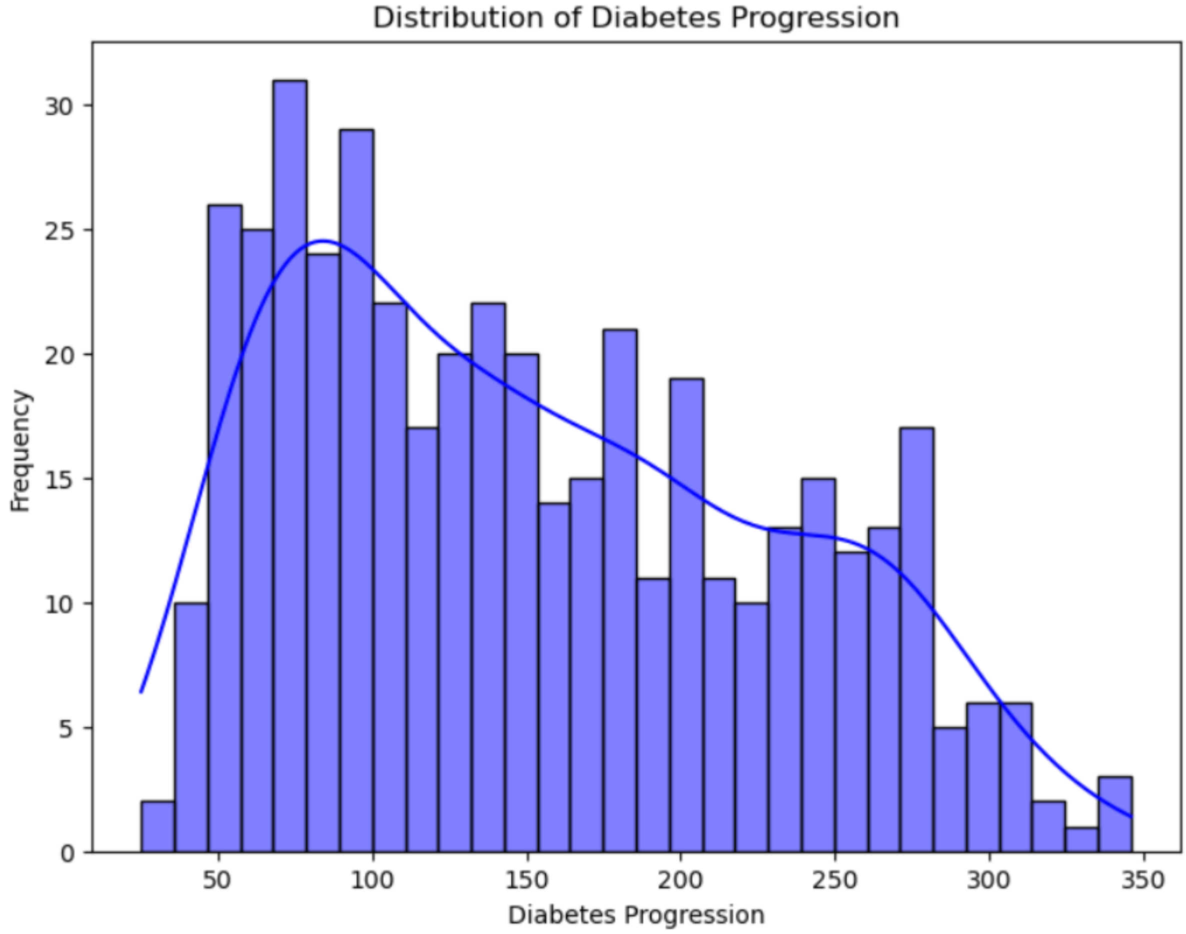
sns.histplot(diabetes_df['target'], bins=30, kde=True, color='blue')

plt.title('Distribution of Diabetes Progression')

plt.xlabel('Diabetes Progression')

plt.ylabel('Frequency')

plt.show()



 # Pairplot to visualize relationships between features and the target variable

plt.figure(figsize=(12, 10))

sns.pairplot(diabetes_df, diag_kind='kde')

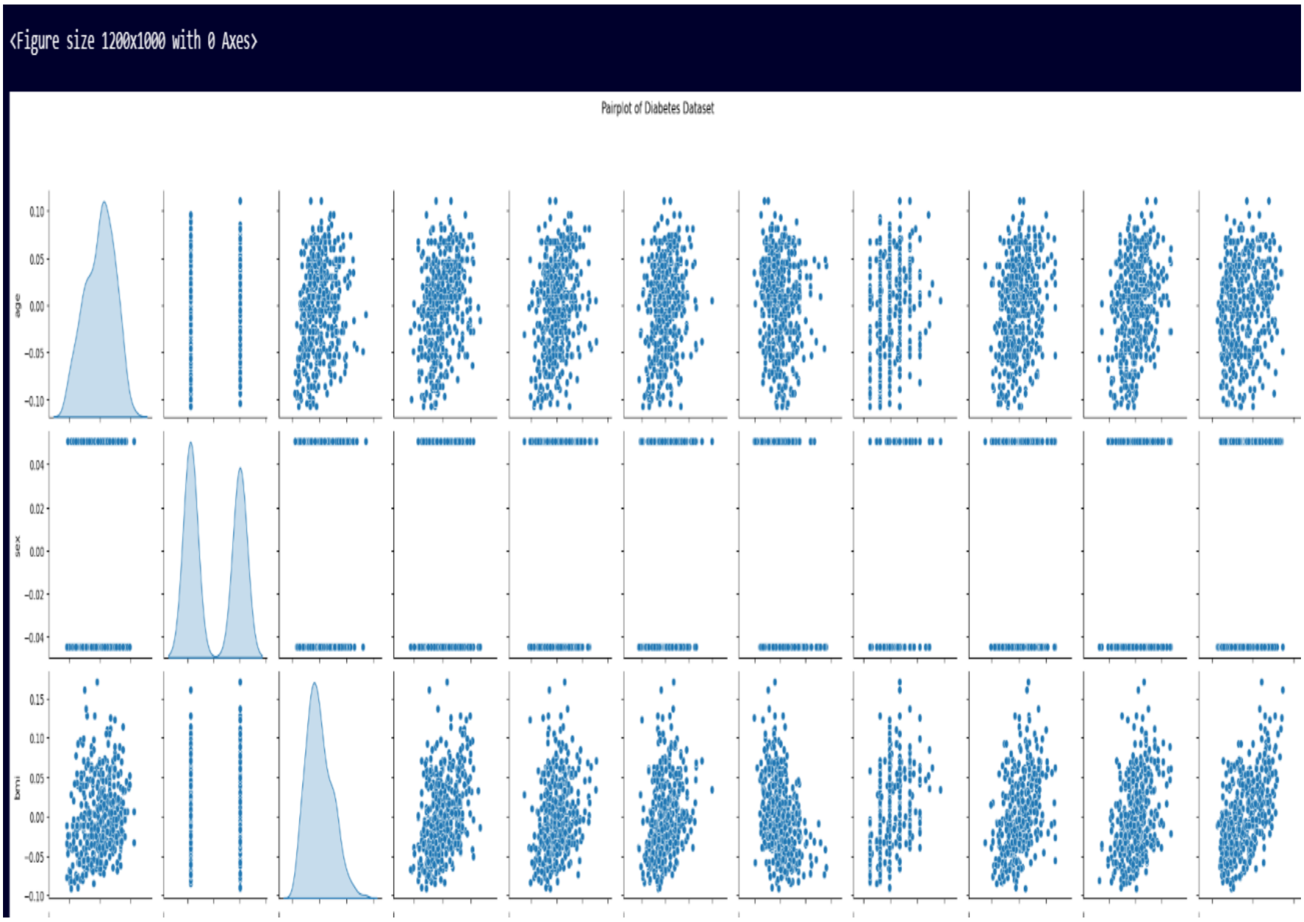
plt.suptitle("Pairplot of Diabetes Dataset", y=1.02)

plt.show()

#Split the Data into Features and Target

 # Split the dataset into features (X) and target (y)

```
X = diabetes_df.drop('target', axis=1)  # Features
y = diabetes_df['target']             # Target (continuous variable)
```



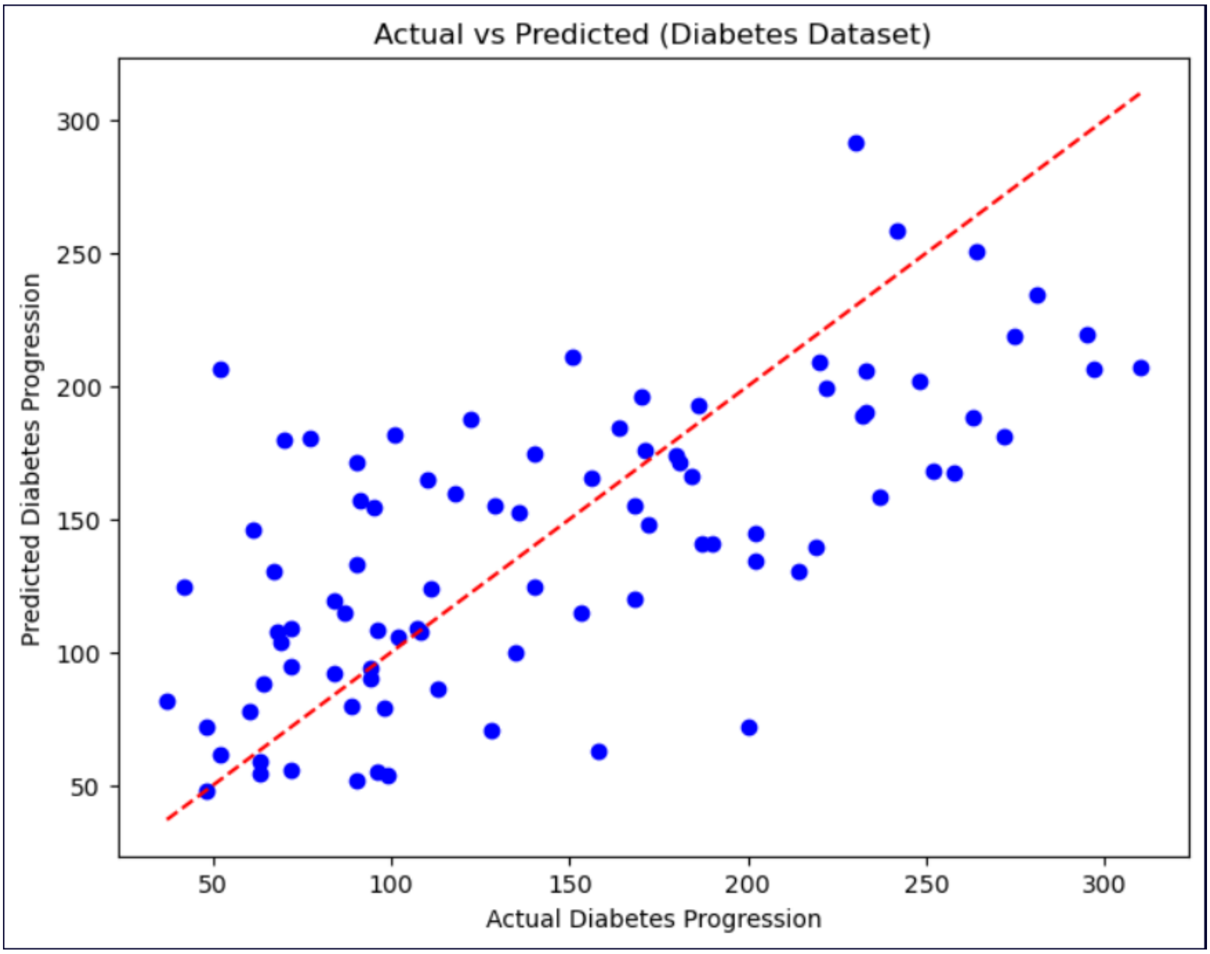

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# Initialize the Linear Regression model
model = LinearRegression()


# Train the model on the training data
model.fit(X_train, y_train)
```

```
# Make predictions on the test data
y_pred = model.predict(X_test)

# Evaluate the model's performance
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print("\nMean Squared Error (MSE):", mse)
print("R-squared Score:", r2)
```

```
Mean Squared Error (MSE): 2900.1936284934814
R-squared Score: 0.4526027629719195
```

```
print("\nMean Squared Error (MSE):", mse)
print("R-squared Score:", r2)
# Plot predicted vs actual values
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred, color='blue')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], linestyle='--', color='red')
```

```
plt.xlabel('Actual Diabetes Progression')
plt.ylabel('Predicted Diabetes Progression')
plt.title('Actual vs Predicted (Diabetes Dataset)')
plt.show()
```



Practical No 2

Aim: Implement Logistic Regression. (Iris Dataset)

Theory:

Logistic regression is a supervised machine learning algorithm widely used for binary classification tasks, such as identifying whether an email is spam or not and diagnosing diseases by assessing the presence or absence of specific conditions based on patient test results.

CODE:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report

# Load the Iris dataset from sklearn
iris = datasets.load_iris()
# Convert the data into a DataFrame
iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
iris_df['target'] = iris.target
```

iris_df.shape

```
(150, 5)
```

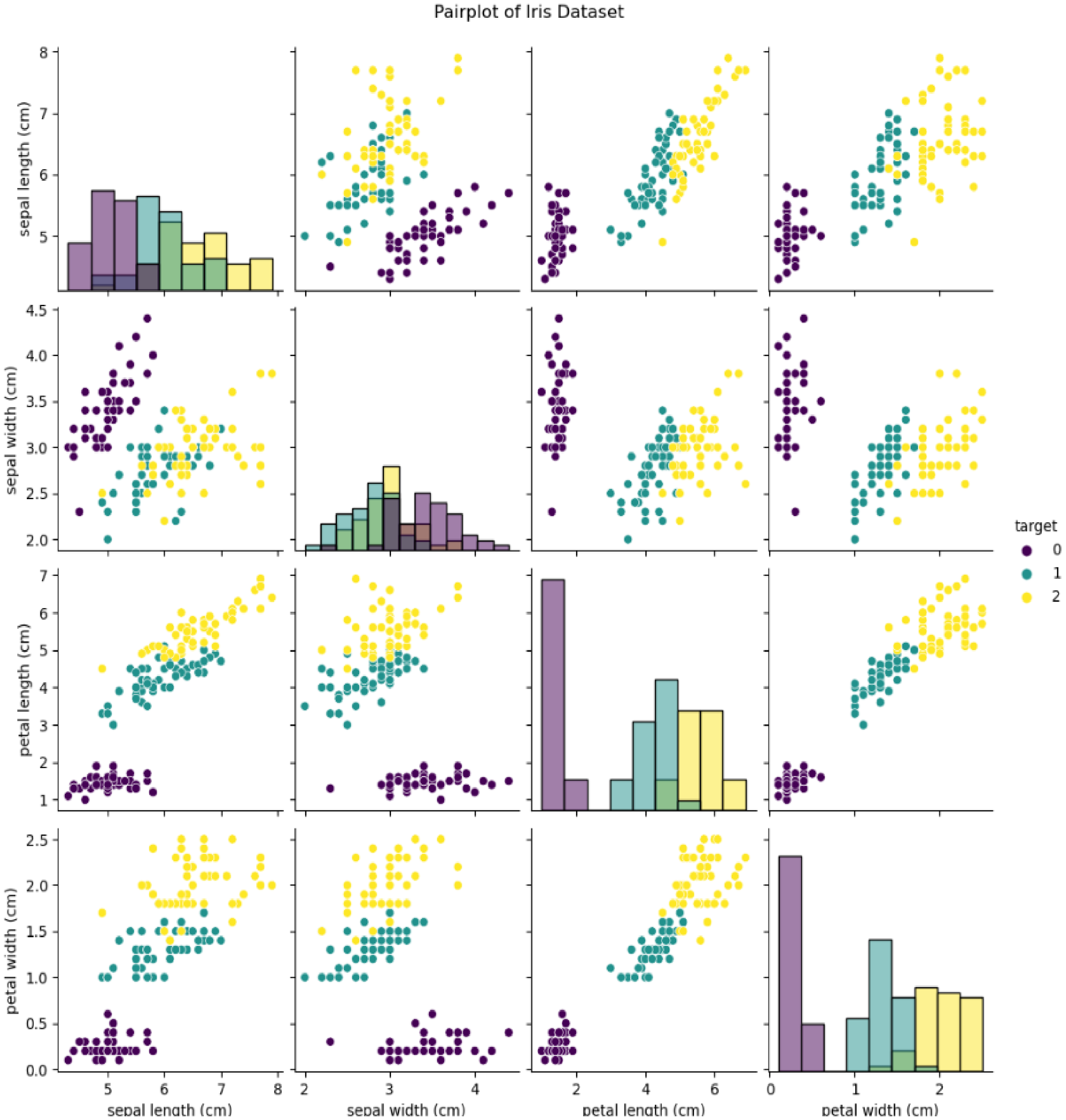
iris_df.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
 4   target             150 non-null    int32
dtypes: float64(4), int32(1)
memory usage: 5.4 KB
```

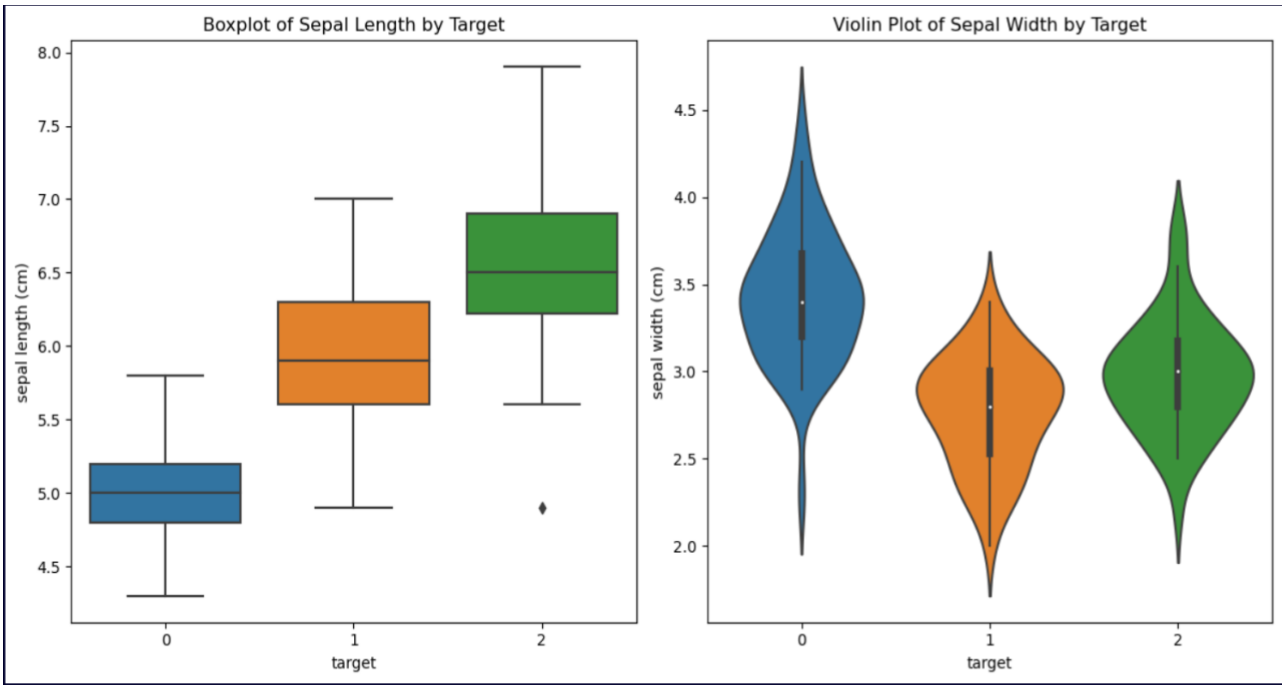
```
iris_df.describe()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333	1.000000
std	0.828066	0.435866	1.765298	0.762238	0.819232
min	4.300000	2.000000	1.000000	0.100000	0.000000
25%	5.100000	2.800000	1.600000	0.300000	0.000000
50%	5.800000	3.000000	4.350000	1.300000	1.000000
75%	6.400000	3.300000	5.100000	1.800000	2.000000
max	7.900000	4.400000	6.900000	2.500000	2.000000

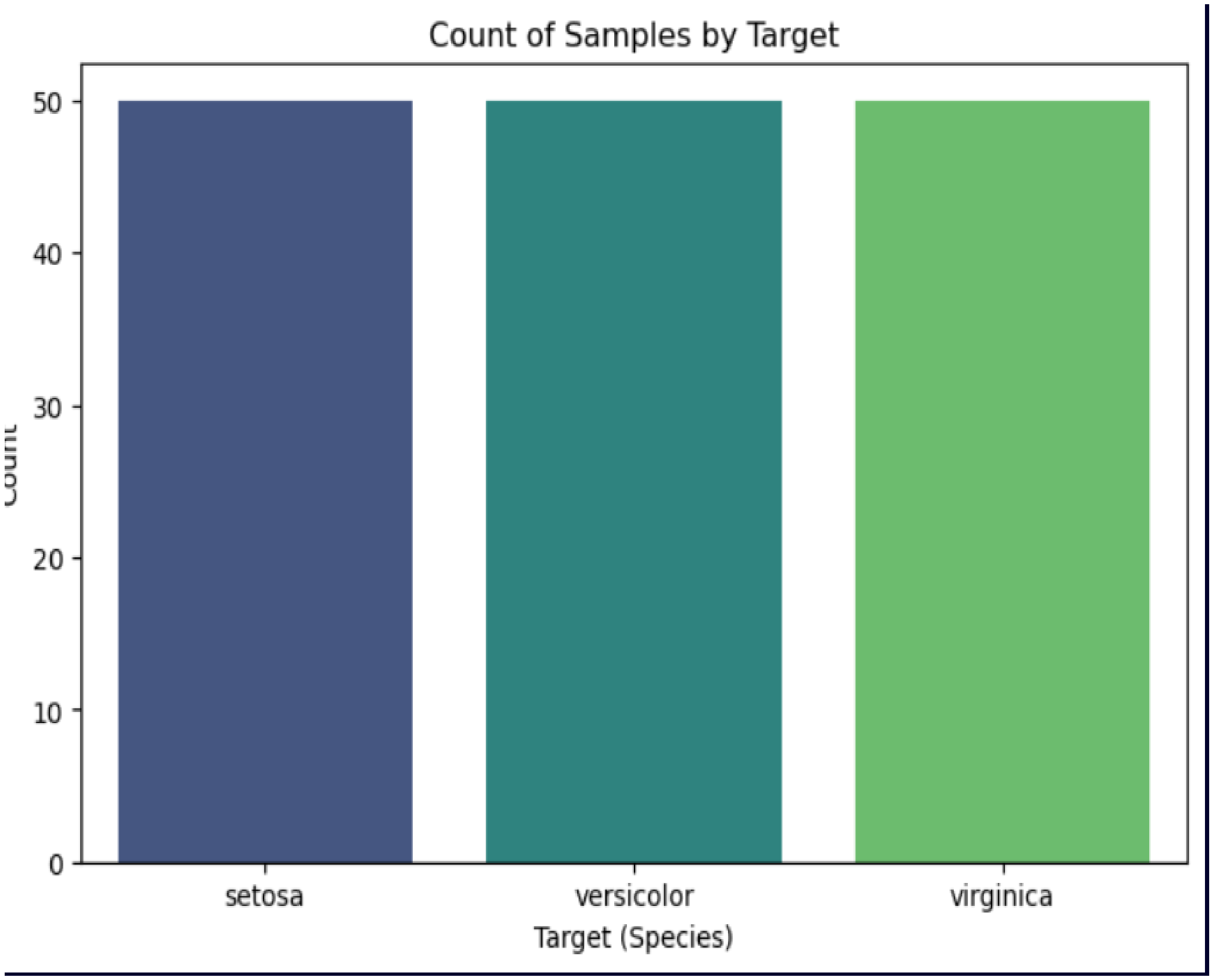
```
# EDA: Pairplot to visualize relationships between features
sns.pairplot(iris_df, hue='target', palette='viridis', diag_kind='hist')
plt.suptitle("Pairplot of Iris Dataset", y=1.02)
plt.show()
```


Pairplot of Iris Dataset
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)
target
0
1
2

```
# Other Plots
plt.figure(figsize=(12, 6))

# Boxplot
plt.subplot(1, 2, 1)
sns.boxplot(x='target', y='sepal length (cm)', data=iris_df)
plt.title('Boxplot of Sepal Length by Target')

# Violin Plot
plt.subplot(1, 2, 2)
sns.violinplot(x='target', y='sepal width (cm)', data=iris_df)
plt.title('Violin Plot of Sepal Width by Target')

plt.tight_layout()
plt.show()
```



```
# Count Plot (Bar Plot)
plt.figure(figsize=(8, 5))
sns.countplot(x='target', data=iris_df, palette='viridis')
plt.title('Count of Samples by Target')
plt.xlabel('Target (Species)')
plt.ylabel('Count')
plt.xticks(ticks=[0, 1, 2], labels=iris.target_names)
plt.show()
```



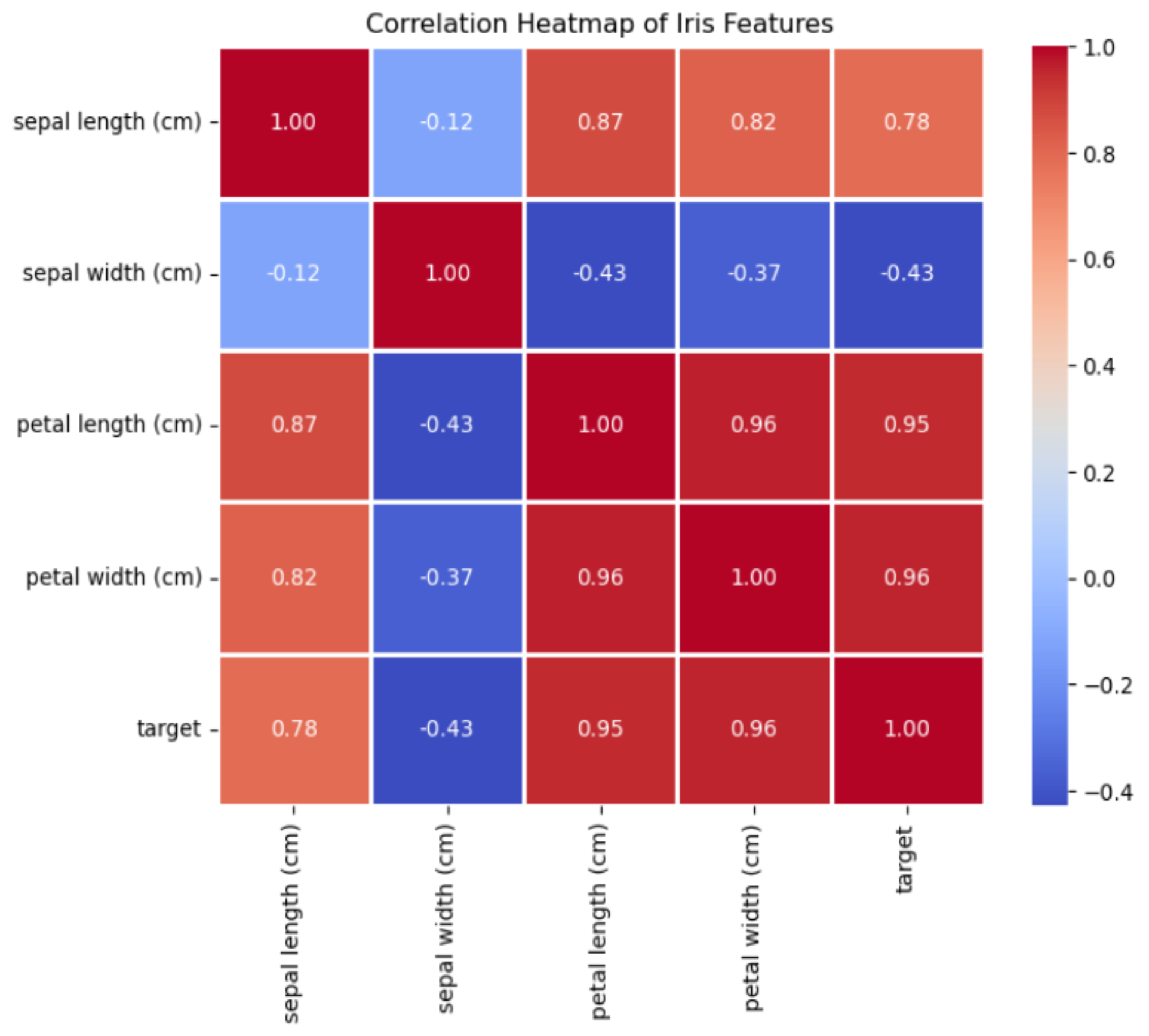
 # Correlation Heatmap

plt.figure(figsize=(8, 6))

sns.heatmap(iris_df.corr(), annot=True, cmap='coolwarm', fmt='.2f', linewidths=1, linecolor='white')

plt.title('Correlation Heatmap of Iris Features')

plt.show()



```
# Separate features (X) and target (y) from the DataFrame
X = iris_df.drop('target', axis=1)  # Features
y = iris_df['target']               # Target (labels)



# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)



# Standardize features by removing the mean and scaling to unit variance
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)



# Initialize the Logistic Regression model
model = LogisticRegression()

# Train the model on the training data
model.fit(X_train, y_train)

# Predict on the test data
y_pred = model.predict(X_test)
```

```
# Calculate training and testing accuracy
train_accuracy = accuracy_score(y_train, model.predict(X_train))
test_accuracy = accuracy_score(y_test, y_pred)

print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)
```

```
Training Accuracy: 0.9666666666666667
Testing Accuracy: 1.0
```

```
# Create a confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:")
print(conf_matrix)
```

```
Confusion Matrix:
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]
```

```
# Print classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred,
target_names=iris.target_names))
```

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

      setosa       1.00      1.00      1.00        10
  versicolor       1.00      1.00      1.00         9
   virginica       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
```

```
# Print probabilities of classification for the first few samples in the test set
print("\nProbabilities of Classification:")
probabilities = model.predict_proba(X_test[:5])
for i, prob in enumerate(probabilities):
    print(f"Sample {i+1}: {list(zip(iris.target_names, prob))}")
```

```
Probabilities of Classification:
Sample 1: [('setosa', 0.011457196118264068), ('versicolor', 0.8759785262009813), ('virginica', 0.1125642776807 5462)]
Sample 2: [('setosa', 0.9644113023352993), ('versicolor', 0.035588286377131434), ('virginica', 4.112875693141769e-07)]
Sample 3: [('setosa', 3.773229944007567e-08), ('versicolor', 0.0028823114202123096), ('virginica', 0.9971176508474883)]
Sample 4: [('setosa', 0.013209318664984725), ('versicolor', 0.7593991586796384), ('virginica', 0.22739152265537677)]
Sample 5: [('setosa', 0.0018885607572993624), ('versicolor', 0.7521357550798934), ('virginica', 0.2459756841628073 8)]
```

Practical No 3

Aim:- Implement Multinomial Logistic Regression (Iris Dataset)

Theory:

A **multinomial logistic regression** (or multinomial regression for short) is used when the outcome variable being predicted is nominal and has more than two categories that do not have a given rank or order. This model can be used with any number of independent variables that are categorical or continuous.

CODE:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report

# Load the Iris dataset from sklearn
iris = datasets.load_iris()
# Convert the data into a DataFrame
iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
iris_df['target'] = iris.target
```

iris_df.shape

```
(150, 5)
```

iris_df.columns

```
Index(['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
       'petal width (cm)', 'target'],
      dtype='object')
```

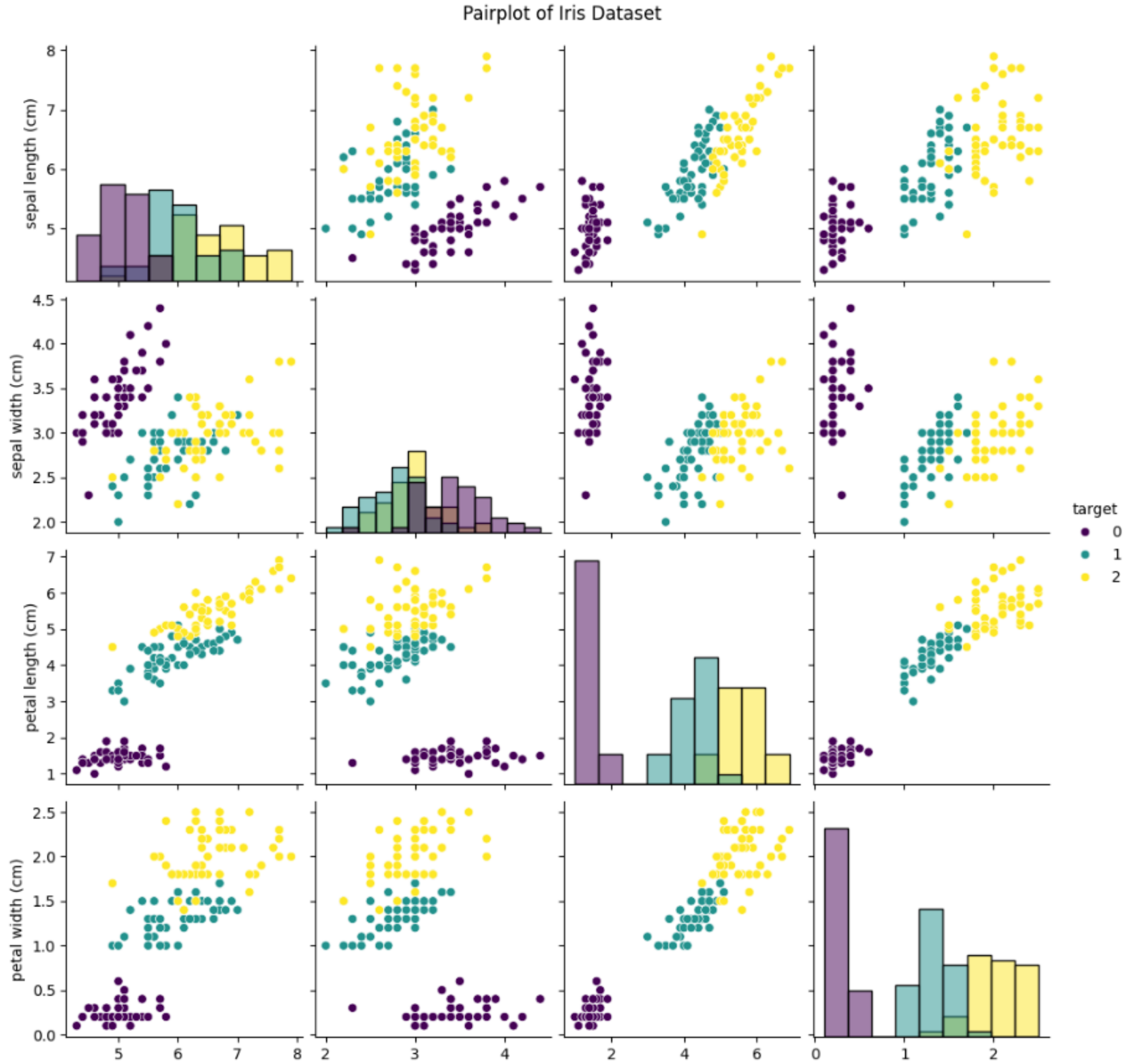
iris_df.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
 4   target             150 non-null    int32
dtypes: float64(4), int32(1)
memory usage: 5.4 KB
```

iris_df.describe()

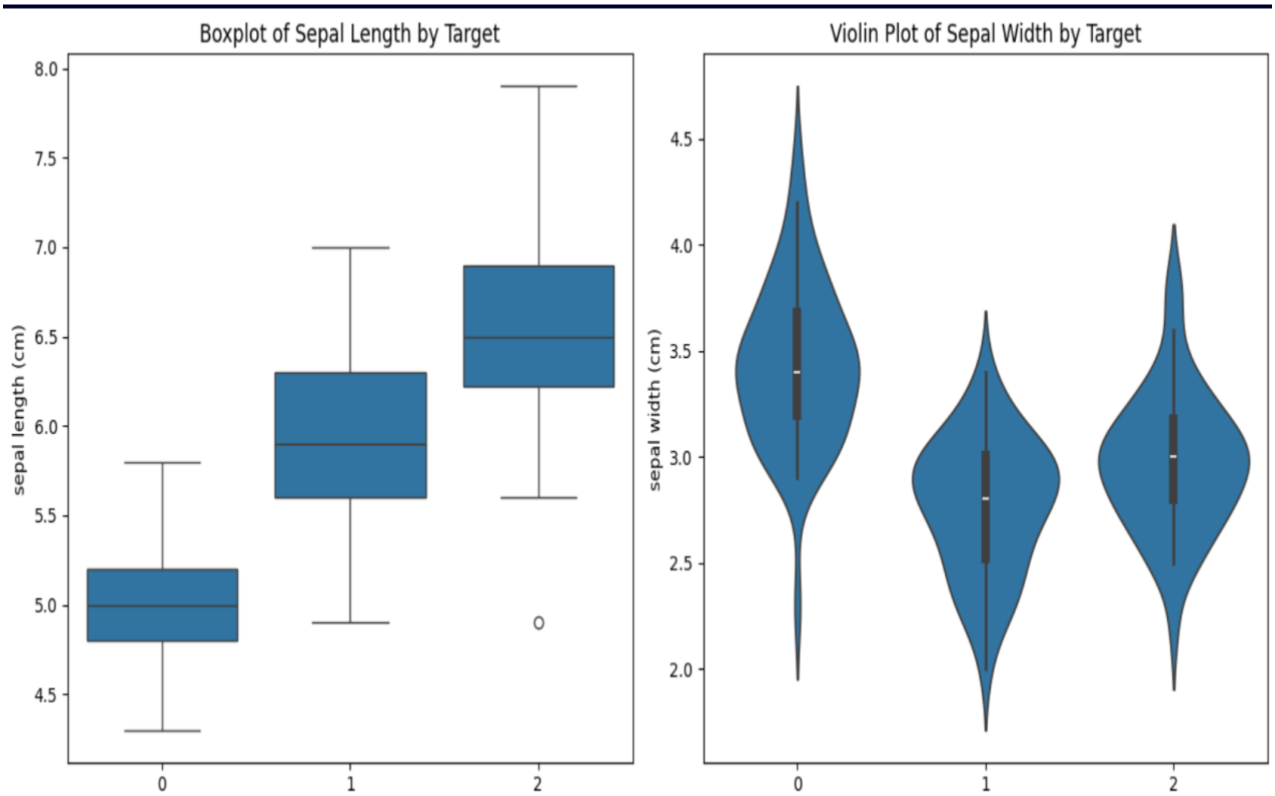
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333	1.000000
std	0.828066	0.435866	1.765298	0.762238	0.819232
min	4.300000	2.000000	1.000000	0.100000	0.000000
25%	5.100000	2.800000	1.600000	0.300000	0.000000
50%	5.800000	3.000000	4.350000	1.300000	1.000000
75%	6.400000	3.300000	5.100000	1.800000	2.000000
max	7.900000	4.400000	6.900000	2.500000	2.000000

```
# EDA: Pairplot to visualize relationships between features
sns.pairplot(iris_df, hue='target', palette='viridis', diag_kind='hist')
plt.suptitle("Pairplot of Iris Dataset", y=1.02)
plt.show()
```

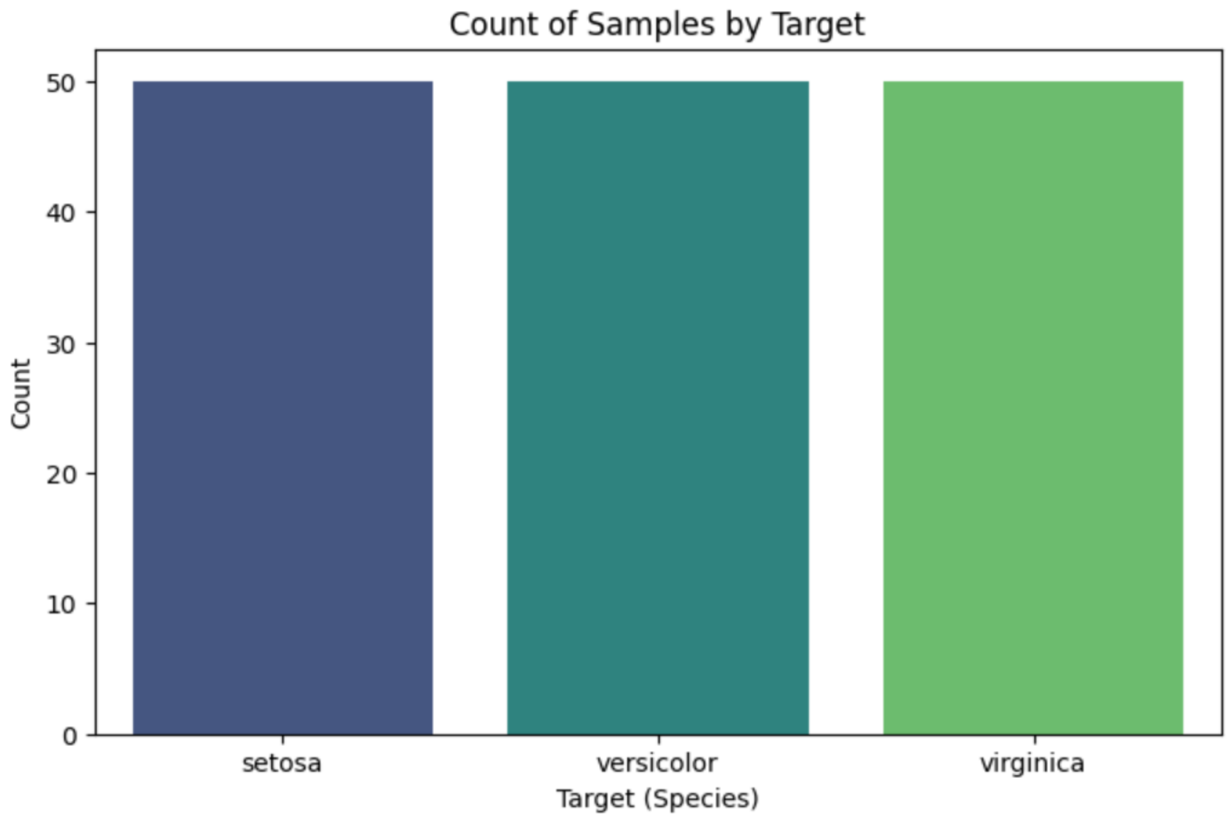



```
# Other Plots
plt.figure(figsize=(12, 6))

# Boxplot
plt.subplot(1, 2, 1)
sns.boxplot(x='target', y='sepal length (cm)', data=iris_df)
plt.title('Boxplot of Sepal Length by Target')
```

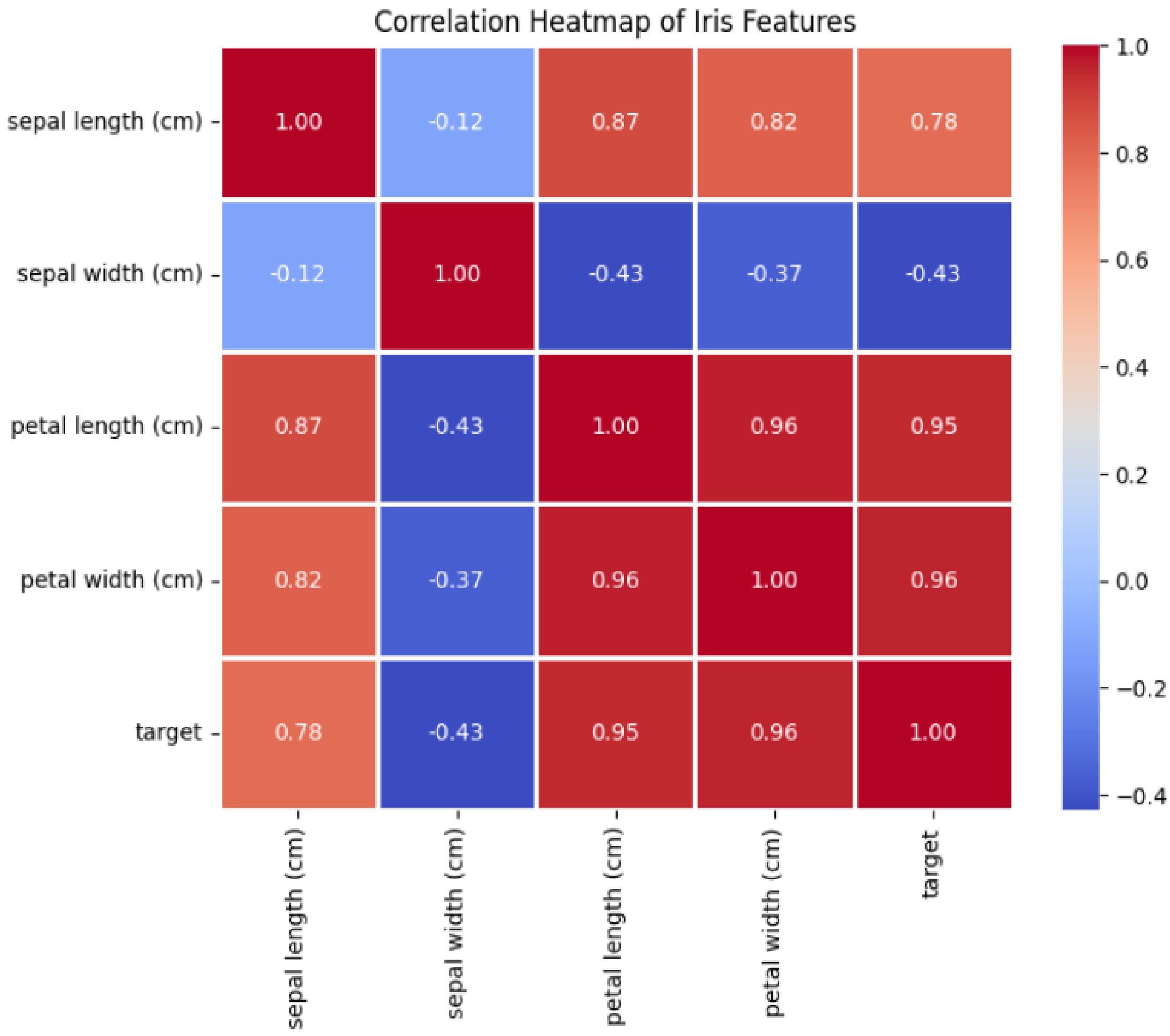
```
# Violin Plot
plt.subplot(1, 2, 2)
sns.violinplot(x='target', y='sepal width (cm)', data=iris_df)
plt.title('Violin Plot of Sepal Width by Target')

plt.tight_layout()
plt.show()
```



```
# Count Plot (Bar Plot)
plt.figure(figsize=(8, 5))
sns.countplot(x='target', data=iris_df, palette='viridis')
plt.title('Count of Samples by Target')
plt.xlabel('Target (Species)')
plt.ylabel('Count')
plt.xticks(ticks=[0, 1, 2], labels=iris.target_names)
plt.show()
```



```
# Correlation Heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(iris_df.corr(), annot=True, cmap='coolwarm', fmt='.2f',
linewidths=1, linecolor='white')
plt.title('Correlation Heatmap of Iris Features')
plt.show()
```



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

# Standardize features by removing the mean and scaling to unit variance
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)



# Initialize the Logistic Regression model
model = LogisticRegression(multi_class="multinomial")

# Train the model on the training data
model.fit(X_train, y_train)

# Predict on the test data
y_pred = model.predict(X_test)



# Calculate training and testing accuracy
train_accuracy = accuracy_score(y_train, model.predict(X_train))
test_accuracy = accuracy_score(y_test, y_pred)

print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)
```

```
Training Accuracy: 0.9666666666666667
Testing Accuracy: 1.0
```

 # Create a confusion matrix

conf_matrix = confusion_matrix(y_test, y_pred)

print("\nConfusion Matrix:")

print(conf_matrix)

```
Confusion Matrix:
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]
```

 # Print classification report

print("\nClassification Report:")

print(classification_report(y_test, y_pred, target_names=iris.target_names))

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

      setosa       1.00      1.00      1.00        10
  versicolor       1.00      1.00      1.00         9
   virginica       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
```

 # Print probabilities of classification for the first few samples in the test set

```
print("\nProbabilities of Classification:")
probabilities = model.predict_proba(X_test[:5])
for i, prob in enumerate(probabilities):
    print(f"Sample {i+1}: {list(zip(iris.target_names, prob))}")
```

```
Probabilities of Classification:
Sample 1: [('setosa', 0.011457196118264058), ('versicolor', 0.875978526200981З), ('virginica', 0.11256427768075462)]
Sample 2: [('setosa', 0.9644113023352995), ('versicolor', 0.03558828637713127), ('virginica', 4.1128756931417627e-07)]
Sample 3: [('setosa', 3.7732299440075534e-08), ('versicolor', 0.0028823114202123196), ('virginica', 0.997117650847488З)]
Sample 4: [('setosa', 0.0132093186649847), ('versicolor', 0.7593991586796384), ('virginica', 0.22739152265537677)]
Sample 5: [('setosa', 0.001888560757299356), ('versicolor', 0.7521357550798935), ('virginica', 0.2459756841628072)]
```

Practical No 4

Aim: Implement SVM Classifier (Iris Datasets)

Theory:

A **support vector machine** (SVM) is defined as a machine learning algorithm that uses supervised learning models to solve complex classification, regression, and outlier detection problems by performing optimal data transformations that determine boundaries between data points based on predefined classes, labels.

CODE:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score

# Load the Iris dataset from sklearn
iris = datasets.load_iris()
# Convert the data into a DataFrame
iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
iris_df['target'] = iris.target
```

Exploratory Data Analysis (EDA)

print(iris_df.describe()) # Summary statistics

print(iris_df.head()) # View first few rows

```
       sepal length (cm)  sepal width (cm)  petal length (cm)  \
count         150.000000        150.000000         150.000000
mean            5.843333          3.057333           3.758000
std             0.828066          0.435866           1.765298
min             4.300000          2.000000           1.000000
25%             5.100000          2.800000           1.600000
50%             5.800000          3.000000           4.350000
75%             6.400000          3.300000           5.100000
max             7.900000          4.400000           6.900000

       petal width (cm)      target
count        150.000000  150.000000
mean           1.199333    1.000000
std            0.762238    0.819232
min            0.100000    0.000000
25%            0.300000    0.000000
50%            1.300000    1.000000
75%            1.800000    2.000000
max            2.500000    2.000000
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2
```

Visualization (pairplot for all features)

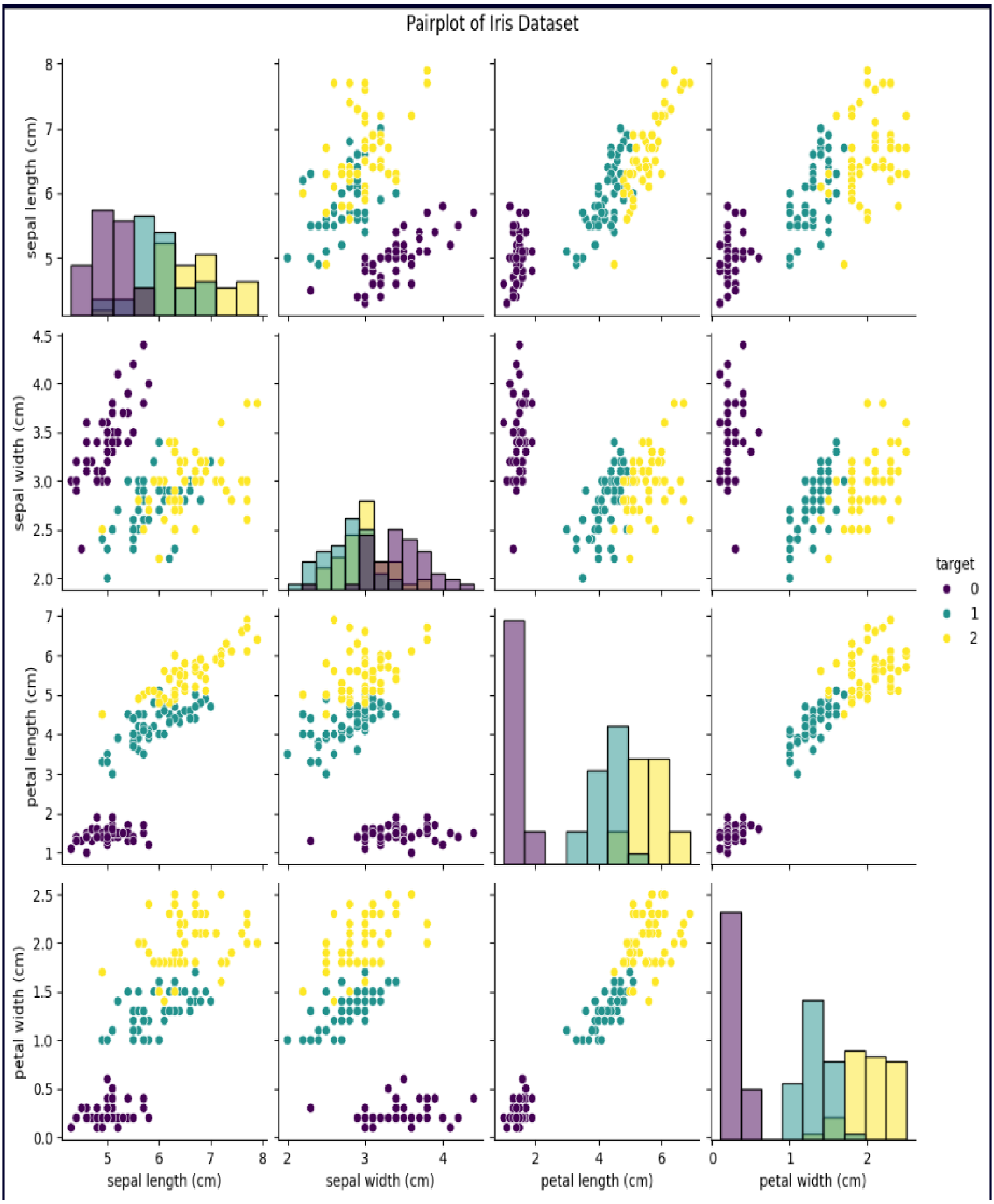
import seaborn as sns

EDA: Pairplot to visualize relationships between features

sns.pairplot(iris_df, hue='target', palette='viridis', diag_kind='hist')

plt.suptitle("Pairplot of Iris Dataset", y=1.02)

plt.show()


Pairplot of Iris Dataset
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)
target
0
1
2

```
# Separate features (X) and target (y) from the DataFrame
X = iris_df.drop('target', axis=1)  # Features
y = iris_df['target']              # Target (labels)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create the SVM model
clf = SVC(kernel='linear')  # Experiment with different kernels (e.g., 'rbf')

# Train the model
clf.fit(X_train, y_train)

# Make predictions on the testing set
y_pred = clf.predict(X_test)

# Evaluate model performance
print(classification_report(y_test, y_pred))
print("Training Accuracy:", accuracy_score(y_train, clf.predict(X_train)))
print("Testing Accuracy:", accuracy_score(y_test, y_pred))
```

```
              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

Training Accuracy: 0.975
Testing Accuracy: 1.0
```

HYPER PARAMETER TUNING

```
from sklearn.model_selection import GridSearchCV
# Define a parameter grid to explore
param_grid = {'kernel': ['linear', 'rbf'],
              'C': [0.01, 0.1, 1, 10, 100]}
# Create the GridSearchCV object
grid_search = GridSearchCV(SVC(), param_grid, cv=5)  # 5-fold cross-validation
# Fit the grid search to the training data
grid_search.fit(X_train, y_train)
# Get the best model and its parameters
best_model = grid_search.best_estimator_
best_params = grid_search.best_params_
print(best_params)
# Use the best model for prediction and evaluation
y_pred = best_model.predict(X_test)
```

print(classification_report(y_test, y_pred))

print("Testing Accuracy:", accuracy_score(y_test, y_pred))

```
{'C': 1, 'kernel': 'linear'}
              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

Testing Accuracy: 1.0
```

Practical No 5

Aim: Train and fine-tune a Decision Tree for Moon dataset.

Theory:

A **decision tree** is a decision support hierarchical model that uses a tree-like model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility.

CODE:

```
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.tree import plot_tree

# Generate moons data
X, y = make_moons(n_samples=1000, noise=0.3)

# 1. Data Shape and Description
print("Data Shape:", X.shape)
print("Description of first 5 rows:")
print(X[:5])
print("Description of target variable:")
print(pd.Series(y).value_counts())  # Convert y to pandas Series
```

```
Data Shape: (1000, 2)
Description of first 5 rows:
[[ 0.0855163   1.30521226]
 [-0.05544742  0.2908183 ]
 [ 1.60334944 -0.61151162]
 [ 1.38402564 -0.60404707]
 [ 1.51886676  0.02146531]]
Description of target variable:
0    500
1    500
dtype: int64
```

 # 2. Check for Missing Values

import numpy as np

print("Missing values in features:", np.isnan(X).sum(axis=0))

```
Missing values in features: [0 0]
```

 # 3. Visualize the moons data

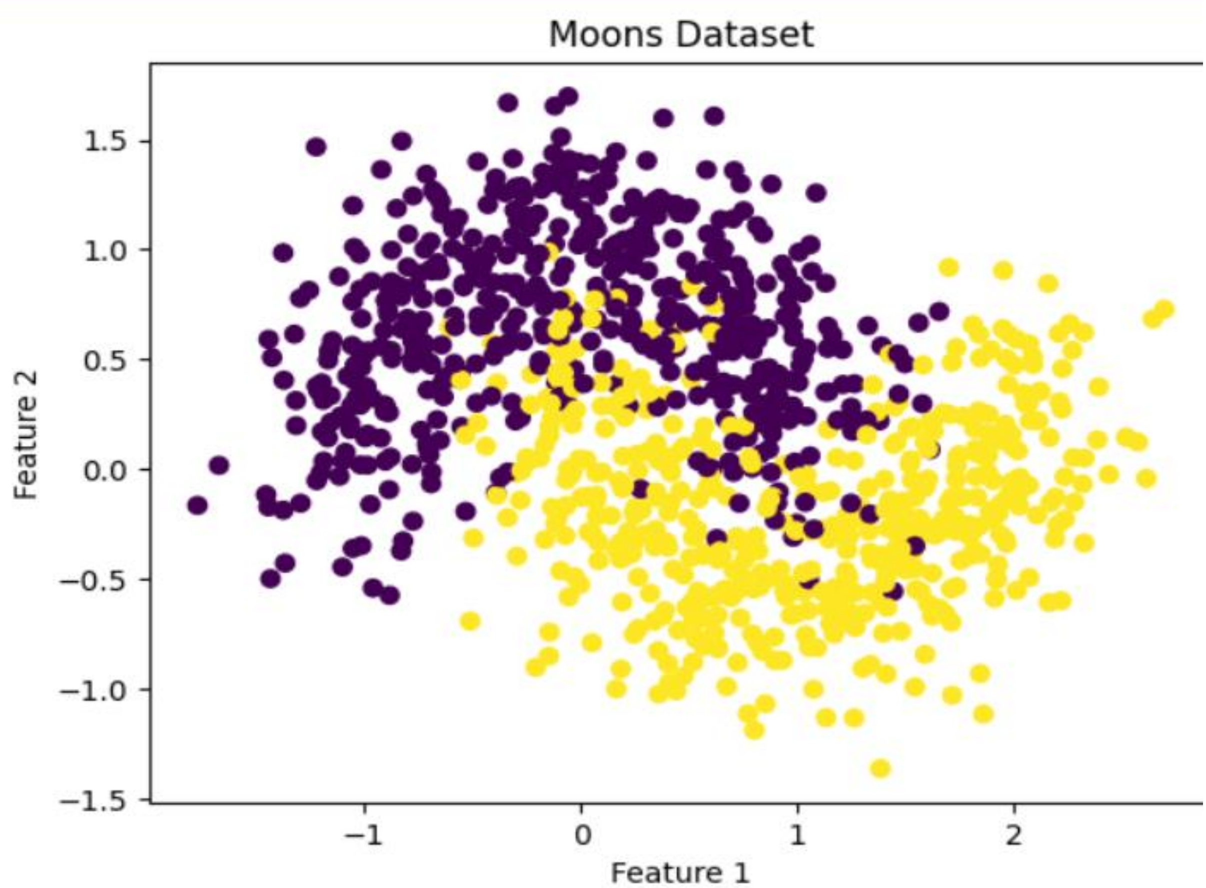
plt.scatter(X[:, 0], X[:, 1], c=y)

plt.title("Moons Dataset")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()



X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

 # Define the decision tree classifier

clf = DecisionTreeClassifier()

5. Hyperparameter Tuning with GridSearchCV:

We'll tune two important hyperparameters for decision trees:

max_depth: Maximum depth of the tree. min_samples_split: Minimum number of samples required to split a node.

 # Define hyperparameter grid

param_grid = {

'max_depth': [2, 3, 4, 5],

'min_samples_split': [2, 5, 10]

```
}

# Create GridSearchCV object
grid_clf = GridSearchCV(clf, param_grid, scoring='accuracy')

# Train the model
grid_clf.fit(X_train, y_train)

# Get the best model
best_model = grid_clf.best_estimator_

# Print the best hyperparameters
print("Best Hyperparameters:", grid_clf.best_params_)
```

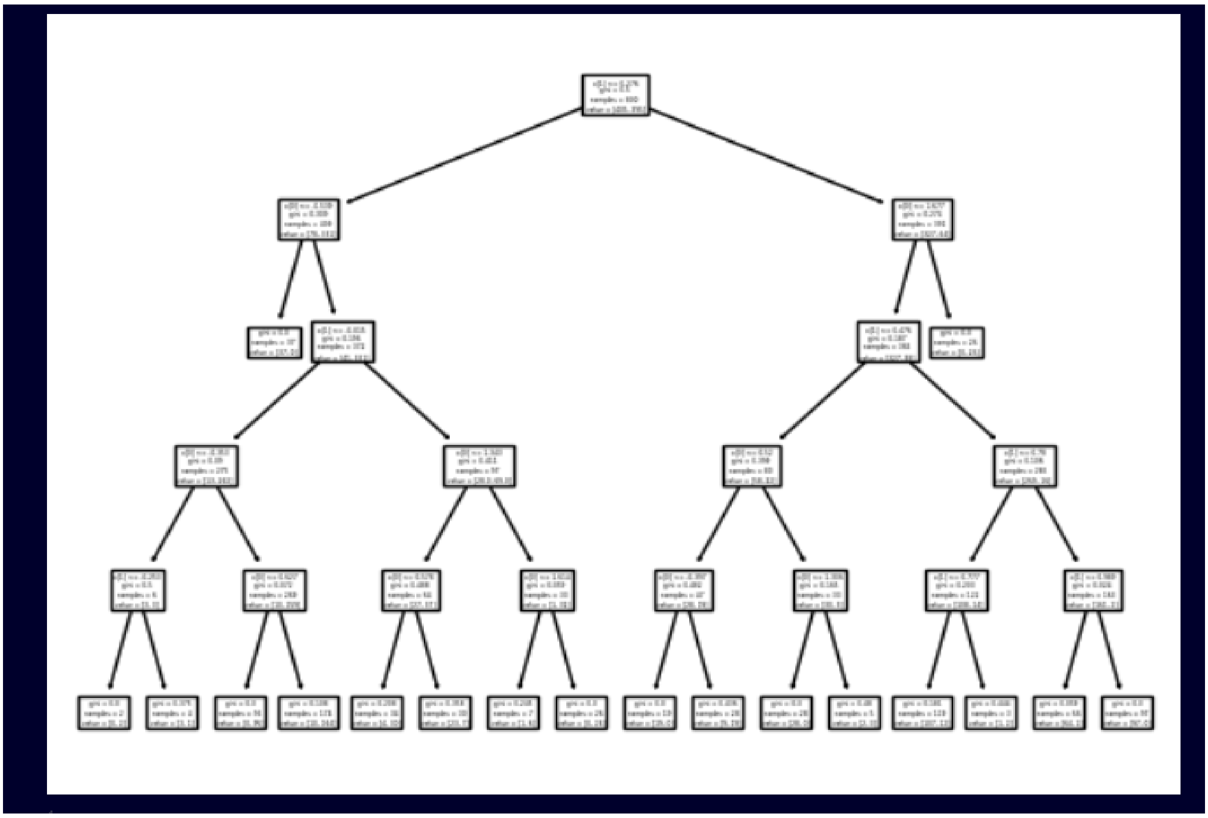
```
Best Hyperparameters: {'max_depth': 5, 'min_samples_split': 2}
```

```
# Predict on test set
y_pred = best_model.predict(X_test)

# Calculate accuracy
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_test, y_pred)
print("Test Accuracy:", accuracy)
```

```
Test Accuracy: 0.9
```

```
#
```

```
# Visualize the decision tree
plot_tree(best_model)
plt.show()
```



Practical No 6

Aim: Train an SVM regression on the California Housing Dataset

Theory:

Support Vector Regression is an extension of SVM which introduces a region, named tube, around the function to optimize with the aim of finding the tube that best approximates the continuous-valued function, while minimizing the prediction error, that is, the difference between the predicted and the true class label.

CODE:

```
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, r2_score
import seaborn as sns  # For visualization
import matplotlib.pyplot as plt  # For visualization

# Load data
data = fetch_california_housing()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = data.target
```

X.columns

```
Index(['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Population', 'AveOccup',
       'Latitude', 'Longitude'],
      dtype='object')
```

y

```
array([4.526, 3.585, 3.521, ..., 0.923, 0.847, 0.894])
```

 # Print basic info about the data

print(X.describe()) # Summary statistics

print(X.head())   # View first few rows

```
             MedInc      HouseAge      AveRooms     AveBedrms    Population  \
count  20640.000000  20640.000000  20640.000000  20640.000000  20640.000000   
mean       3.870671     28.639486      5.429000      1.096675   1425.476744   
std        1.899822     12.585558      2.474173      0.473911   1132.462122   
min        0.499900      1.000000      0.846154      0.333333      3.000000   
25%        2.563400     18.000000      4.440716      1.006079    787.000000   
50%        3.534800     29.000000      5.229129      1.048780   1166.000000   
75%        4.743250     37.000000      6.052381      1.099526   1725.000000   
max       15.000100     52.000000    141.909091     34.066667  35682.000000   

           AveOccup      Latitude     Longitude  
count  20640.000000  20640.000000  20640.000000  
mean       3.070655     35.631861   -119.569704  
std       10.386050      2.135952      2.003532  
min        0.692308     32.540000   -124.350000  
25%        2.429741     33.930000   -121.800000  
50%        2.818116     34.260000   -118.490000  
75%        3.282261     37.710000   -118.010000  
max     1243.333333     41.950000   -114.310000  
   MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0  8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88   
1  8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86   
2  7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85   
3  5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85   
4  3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85   
...
1    -122.22  
2    -122.24  
3    -122.25  
4    -122.25  
```

 # Check for missing values

print("Missing values:", X.isnull().sum())

```
Missing values: MedInc        0
HouseAge      0
AveRooms      0
AveBedrms     0
Population    0
AveOccup      0
Latitude      0
Longitude     0
dtype: int64
```

 # Exploratory Data Analysis (EDA)

 # Visualize the distribution of the target variable (median house value)

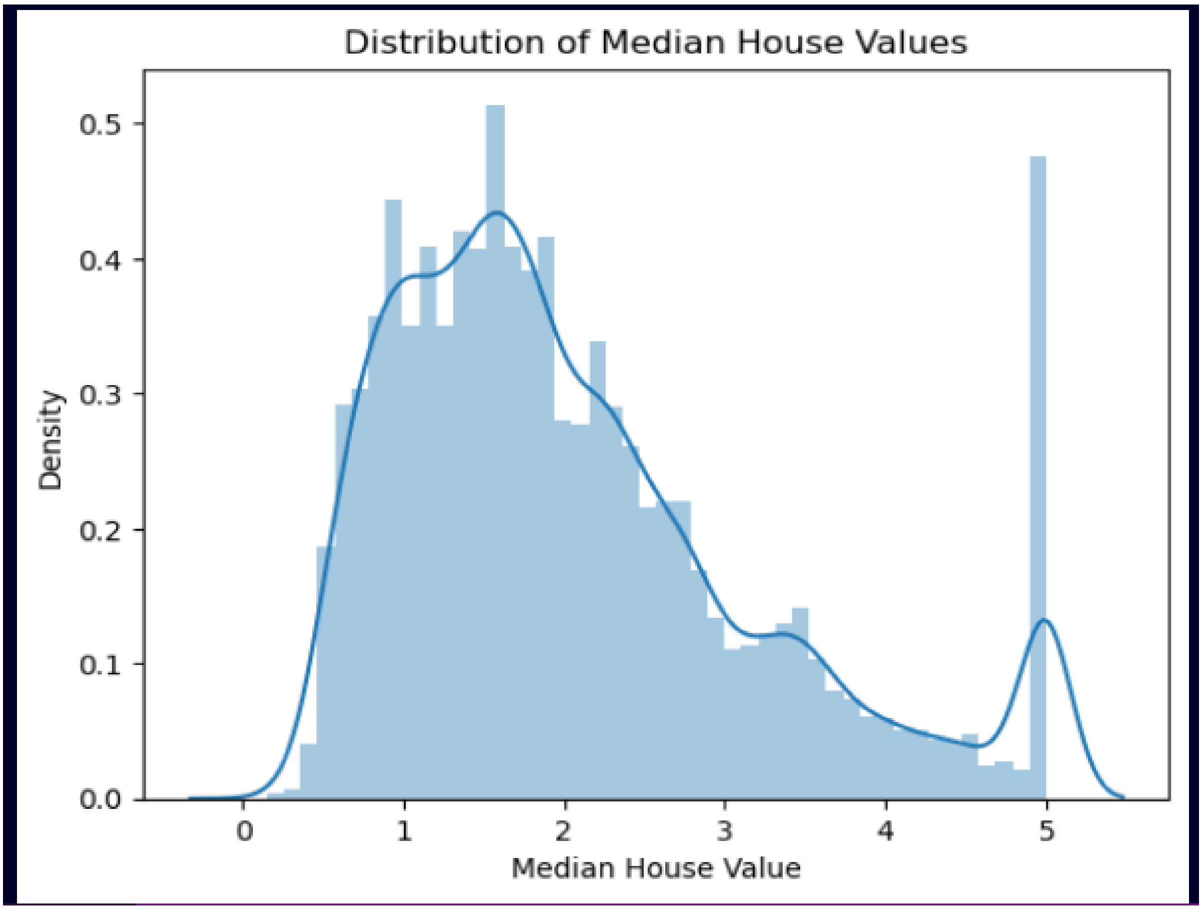
sns.distplot(y)

plt.xlabel("Median House Value")

plt.ylabel("Density")

plt.title("Distribution of Median House Values")

plt.show()



 # Data Preprocessing

 # Scale features (SVM regressor is sensitive to feature scales)

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)

 # Split Data into Training and Testing Sets

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

 # Train the SVM Regressor

svr = SVR(kernel='rbf') # Experiment with 'linear' or other kernels

svr.fit(X_train, y_train)

```
# Make Predictions and Evaluate Performance
y_pred = svr.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error:", mse)
print("R-squared:", r2)
```

```
Mean Squared Error: 0.3551984619989417
R-squared: 0.7289407597956463
```

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVR


# Define a parameter grid to explore
param_grid = {
    'kernel': ['linear', 'rbf'],  # Experiment with different kernels
    'C': [0.01, 0.1, 1, 10, 100],  # Regularization parameter
    'gamma': [0.001, 0.01, 0.1, 1],  # Gamma for RBF kernel (optional)
}

# Create the GridSearchCV object
```

```
grid_search = GridSearchCV(SVR(), param_grid, cv=5)  # 5-fold cross-validation

# Fit the grid search to the training data
grid_search.fit(X_train, y_train)

# Get the best model and its parameters
best_model = grid_search.best_estimator_
best_params = grid_search.best_params_

# Use the best model for prediction and evaluation
y_pred = best_model.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

Practical No 7

Aim: Implement Batch Gradient Descent with Early Stopping for Softmax Regression.

Theory:

Batch gradient descent, also called vanilla gradient descent, calculates the error for each example within the training dataset, but only after all training examples have been evaluated does the model get updated. This whole process is like a cycle and it's called a training epoch.

In machine learning, **early stopping** is a form of regularization used to avoid overfitting when training a learner with an iterative method, such as gradient descent. Such methods update the learner so as to make it better fit the training data with each iteration.

SoftMax is particularly suited for multi-class classification problems, as it provides a clear and normalized probability distribution across all possible classes.

CODE:

```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Standardize the features
scaler = StandardScaler()
```

```
X = scaler.fit_transform(X)

# Add a bias term (column of ones) to the data
X = np.c_[np.ones(X.shape[0]), X]

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
def softmax(logits):
    exp_logits = np.exp(logits - np.max(logits, axis=1, keepdims=True))
    return exp_logits / np.sum(exp_logits, axis=1, keepdims=True)

def compute_loss_and_gradients(X, y, theta):
    logits = X.dot(theta)
    y_proba = softmax(logits)
    m = X.shape[0]
    entropy_loss = -np.mean(np.log(y_proba[np.arange(m), y]))
    gradients = (1/m) * X.T.dot(y_proba - np.eye(np.max(y) + 1)[y])
    return entropy_loss, gradients

def predict(X, theta):
    logits = X.dot(theta)
    return np.argmax(softmax(logits), axis=1)


def softmax_regression(X_train, y_train, X_val, y_val, learning_rate=0.01,
n_epochs=1000, tol=1e-4, patience=5):
    n_inputs = X_train.shape[1]
```

```
    n_outputs = np.max(y_train) + 1
    theta = np.random.randn(n_inputs, n_outputs)

    best_loss = np.inf
    epochs_without_improvement = 0

    for epoch in range(n_epochs):
        loss, gradients = compute_loss_and_gradients(X_train, y_train, theta)
        theta = theta - learning_rate * gradients

        val_loss, _ = compute_loss_and_gradients(X_val, y_val, theta)

        if val_loss < best_loss - tol:
            best_loss = val_loss
            epochs_without_improvement = 0
        else:
            epochs_without_improvement += 1

        if epochs_without_improvement >= patience:
            print(f"Early stopping at epoch {epoch}")
            break

    return theta

# Split the training data into training and validation sets
X_train_split, X_val_split, y_train_split, y_val_split =
train_test_split(X_train, y_train, test_size=0.2, random_state=42)
```

 # Train the model

theta = softmax_regression(X_train_split, y_train_split, X_val_split, y_val_split)

```
Test accuracy: 93.33%
```

Practical No 8

Aim: Implement MLP for Classification of Handwritten digits(MNIST datasets)

Theory:

The **MNIST database** (Modified National Institute of Standards and Technology database) is a large collection of handwritten digits. It has a training set of 60,000 examples, and a test set of 10,000 examples.

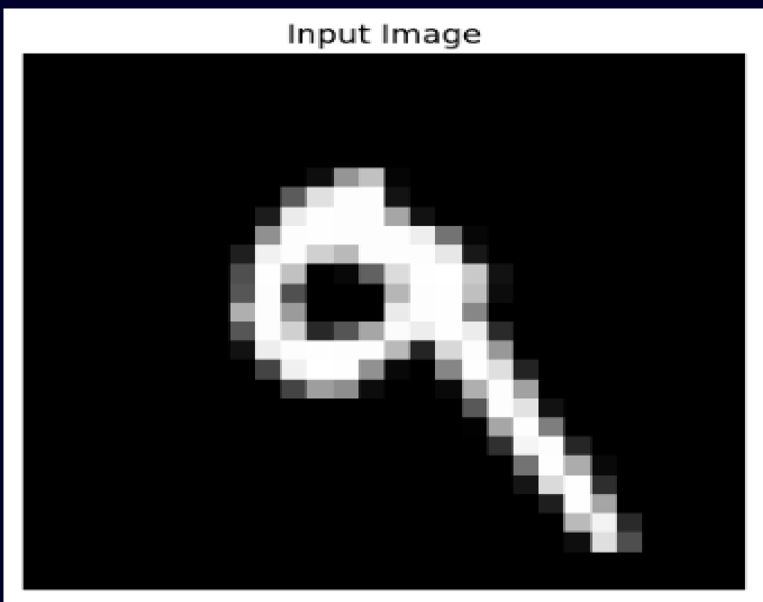
CODE:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
import matplotlib.pyplot as plt

# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
print(x_train.shape)
x_train = x_train / 255.0
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)

# Define the CNN model architecture
model = Sequential([
   Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
```

```
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model using the training data
model.fit(x_train, y_train, epochs=5)

# Choose a single image from the test set
index = 0  # Replace with the index of the image you want to use
single_image = x_test[index]
input_image = np.expand_dims(single_image, axis=0)

# Get the predicted probabilities for the single image
predicted_probabilities = model.predict(input_image)

# Display the input image
plt.imshow(single_image, cmap='gray')
plt.title('Input Image')
```

```
plt.axis('off')
plt.show()

# Display the predicted probabilities
print("Predicted Probabilities:", predicted_probabilities)

# Get the predicted class (index with highest probability)
predicted_class = np.argmax(predicted_probabilities)
print("Predicted Class:", predicted_class)
```

```
(60000, 28, 28)
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 30s 13ms/step - accuracy: 0.9015 - loss: 0.3057
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 25s 13ms/step - accuracy: 0.9855 - loss: 0.0459
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 25s 14ms/step - accuracy: 0.9911 - loss: 0.0286
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 26s 14ms/step - accuracy: 0.9941 - loss: 0.0201
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 25s 13ms/step - accuracy: 0.9954 - loss: 0.0145
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 98ms/step
```




```
Predicted Probabilities: [[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
Predicted Class: 9
```

Practical No 9

Aim: Classification of Image of clothing using Tensorflow(Fashion MNIST dataset)

Theory:

TensorFlow is a free and open-source software library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks. It was developed by the Google Brain team for Google's internal use in research and production

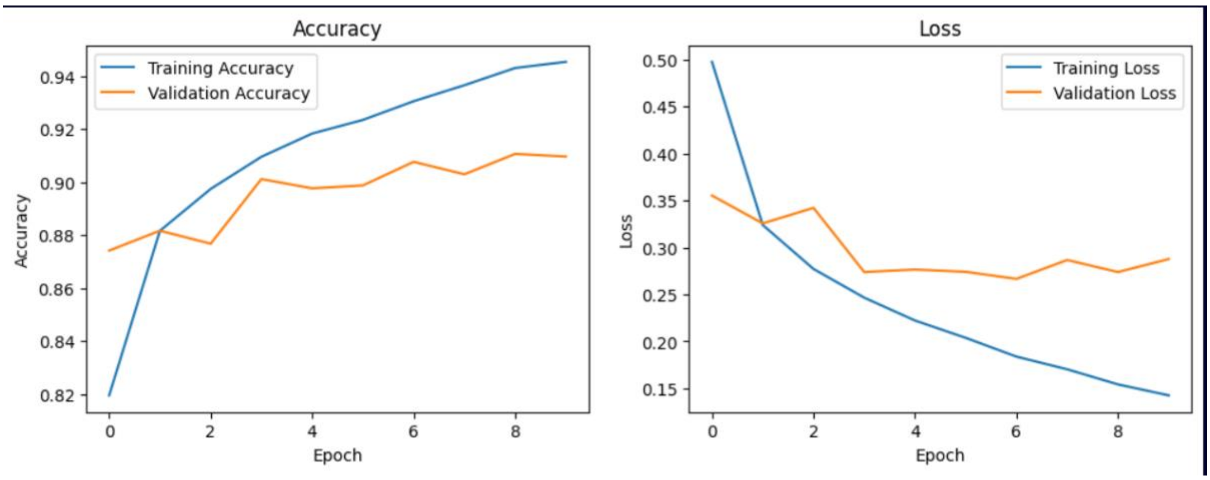
CODE:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

# Load the Fashion MNIST dataset
(train_images, train_labels), (test_images, test_labels) =
datasets.fashion_mnist.load_data()

# Normalize the images to a range of 0 to 1
train_images, test_images = train_images / 255.0, test_images / 255.0



model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
```

```
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
# Reshape the data to include the channel dimension
train_images = train_images.reshape((train_images.shape[0], 28, 28, 1))
test_images = test_images.reshape((test_images.shape[0], 28, 28, 1))

# Train the model
history = model.fit(train_images, train_labels, epochs=10,
validation_data=(test_images, test_labels))
```

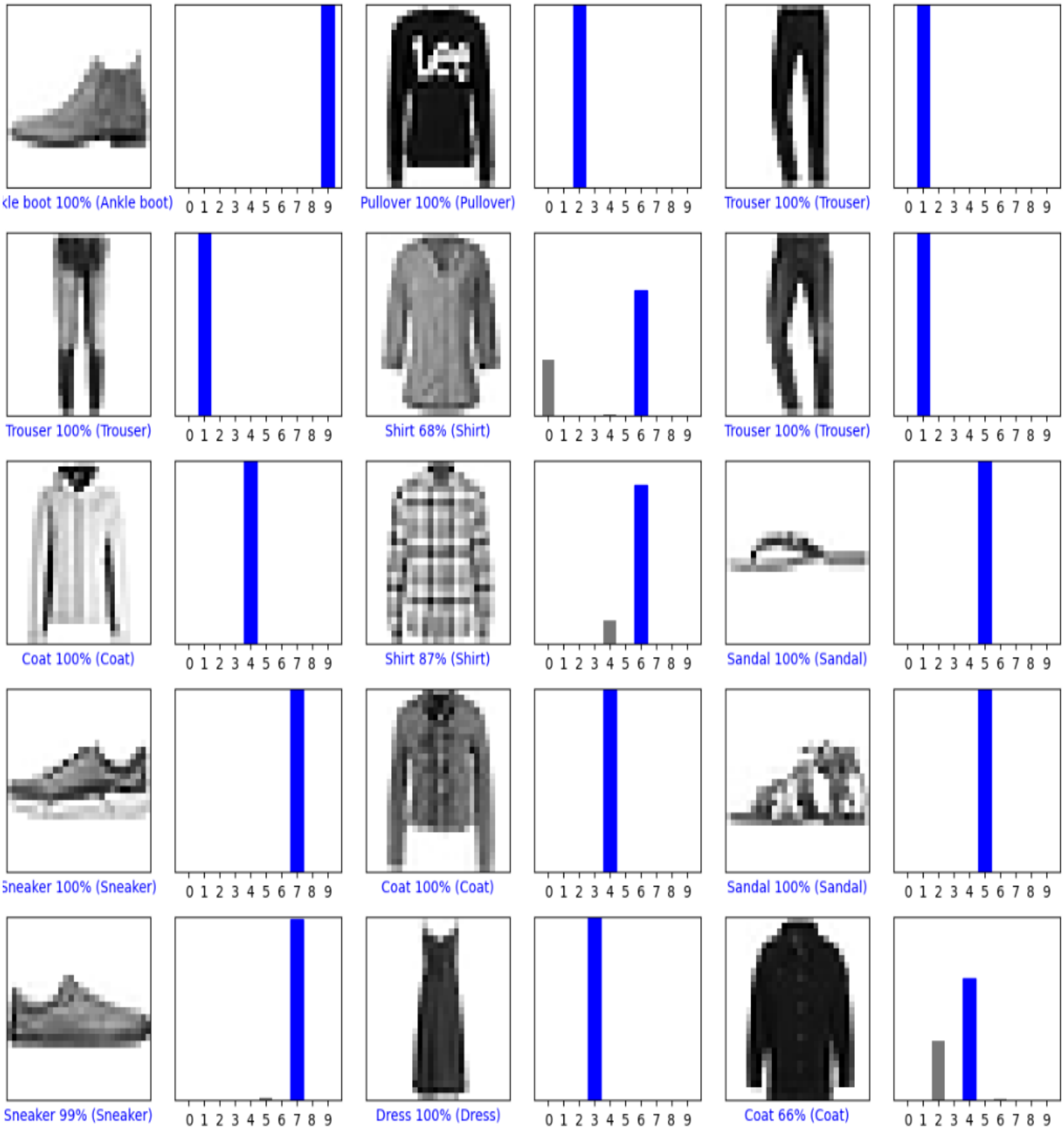
```
Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 27s 13ms/step - accuracy: 0.7506 - loss: 0.6901 - val_accuracy: 0.8742 - val_loss: 0.3552
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 40s 12ms/step - accuracy: 0.8773 - loss: 0.3322 - val_accuracy: 0.8817 - val_loss: 0.3256
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 41s 13ms/step - accuracy: 0.8981 - loss: 0.2773 - val_accuracy: 0.8768 - val_loss: 0.3421
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 24s 13ms/step - accuracy: 0.9087 - loss: 0.2479 - val_accuracy: 0.9012 - val_loss: 0.2738
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 24s 13ms/step - accuracy: 0.9193 - loss: 0.2200 - val_accuracy: 0.8977 - val_loss: 0.2765
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 42s 13ms/step - accuracy: 0.9253 - loss: 0.1996 - val_accuracy: 0.8988 - val_loss: 0.2741
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 24s 13ms/step - accuracy: 0.9322 - loss: 0.1795 - val_accuracy: 0.9077 - val_loss: 0.2666
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 25s 13ms/step - accuracy: 0.9386 - loss: 0.1655 - val_accuracy: 0.9030 - val_loss: 0.2866
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 41s 13ms/step - accuracy: 0.9449 - loss: 0.1507 - val_accuracy: 0.9107 - val_loss: 0.2738
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 27s 15ms/step - accuracy: 0.9476 - loss: 0.1382 - val_accuracy: 0.9097 - val_loss: 0.2876
```

```
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Accuracy')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss')

plt.show()
```



```
import numpy as np
predictions = model.predict(test_images)

# Define a function to plot the images and predictions
def plot_image(predictions_array, true_label, img):
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])

    plt.imshow(img, cmap=plt.cm.binary)

    predicted_label = np.argmax(predictions_array)
    if predicted_label == true_label:
        color = 'blue'
    else:
        color = 'red'

    plt.xlabel(f"{class_names[predicted_label]}
{100*np.max(predictions_array):2.0f}% ({class_names[true_label]})",
color=color)

def plot_value_array(predictions_array, true_label):
    plt.grid(False)
    plt.xticks(range(10))
    plt.yticks([])
    thisplot = plt.bar(range(10), predictions_array, color="#777777")
    plt.ylim([0, 1])
```

```
    predicted_label = np.argmax(predictions_array)

    thisplot[predicted_label].set_color('red')
    thisplot[true_label].set_color('blue')

# Define the class names
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# Plot the first 15 test images, their predicted labels, and the true labels
# Color correct predictions in blue and incorrect predictions in red
num_rows = 5
num_cols = 3
num_images = num_rows * num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
    plt.subplot(num_rows, 2*num_cols, 2*i+1)
    plot_image(predictions[i], test_labels[i], test_images[i].reshape(28, 28))
    plt.subplot(num_rows, 2*num_cols, 2*i+2)
    plot_value_array(predictions[i], test_labels[i])
plt.tight_layout()
plt.show()
```


kle boot 100% (Ankle boot)
0 1 2 3 4 5 6 7 8 9
Pullover 100% (Pullover)
Trouser 100% (Trouser)
Trouser 100% (Trouser)
Shirt 68% (Shirt)
Trouser 100% (Trouser)
Coat 100% (Coat)
Shirt 87% (Shirt)
Sandal 100% (Sandal)
Sneaker 100% (Sneaker)
Coat 100% (Coat)
Sandal 100% (Sandal)
Sneaker 99% (Sneaker)
Dress 100% (Dress)
Coat 66% (Coat)

Practical No 10

Aim: Implement Regression to predict fuel efficiency using TensorFlow (Auto MPG dataset).

Theory:

Regression is a statistical method used in finance, investing, and other disciplines that attempts to determine the strength and character of the relationship between a dependent variable and one or more independent variables. Linear regression is the most common form of this technique.

TensorFlow is a free and open-source software library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks. It was developed by the Google Brain team for Google's internal use in research and production

CODE:

```
import pandas as pd

import numpy as np

import tensorflow as tf

from tensorflow.keras import layers, models

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

import matplotlib.pyplot as plt



# Load the dataset

url = "auto-mpg.csv"  # Replace with your CSV file path

column_names = ['MPG', 'Cylinders', 'Displacement', 'Horsepower',
'Weight', 'Acceleration', 'Model Year', 'Origin']
```

dataset = pd.read_csv(url, names=column_names, na_values='?', comment='\t', sep=',', skipinitialspace=True)

dataset.info()

```
<class 'pandas.core.frame.DataFrame'>
Index: 399 entries, mpg to 31
Data columns (total 8 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   MPG           399 non-null    object
 1   Cylinders     399 non-null    object
 2   Displacement  393 non-null    object
 3   Horsepower    399 non-null    object
 4   Weight        399 non-null    object
 5   Acceleration  399 non-null    object
 6   Model Year    399 non-null    object
 7   Origin        399 non-null    object
dtypes: object(8)
memory usage: 28.1+ KB
```

dataset.head()

	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model Year	Origin
mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
18	8	307	130	3504	12	70	1	chevrolet chevelle malibu
15	8	350	165	3693	11.5	70	1	buick skylark 320
18	8	318	150	3436	11	70	1	plymouth satellite
16	8	304	150	3433	12	70	1	amc rebel sst

dataset.describe()

```
dataset.describe()
```

	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model Year	Origin
count	399	399	393	399	399	399	399	399
unique	6	83	94	352	96	14	4	306
top	4	97	150	1985	14.5	73	1	ford pinto
freq	204	21	22	4	23	40	249	6

```
# Drop rows with missing values
dataset = dataset.dropna()

# Convert columns to appropriate numeric data types
for column in ['MPG', 'Cylinders', 'Displacement', 'Horsepower', 'Weight', 'Acceleration', 'Model Year']:
    dataset[column] = pd.to_numeric(dataset[column], errors='coerce')

# Check for NaN values in the dataset
print("NaN values before dropping: \n", dataset.isnull().sum())

# Drop any rows with NaN values
dataset = dataset.dropna()

# Check again for NaN values to confirm
print("NaN values after dropping: \n", dataset.isnull().sum())

# Convert 'Origin' to string for one-hot encoding
dataset['Origin'] = dataset['Origin'].astype(str)
# Convert categorical 'Origin' column to one-hot encoding
dataset = pd.get_dummies(dataset, columns=['Origin'], prefix='', prefix_sep='')
```

```
NaN values before dropping:
 MPG            1
Cylinders       1
Displacement    1
Horsepower      1
Weight          1
Acceleration    1
Model Year      1
Origin          0
dtype: int64
NaN values after dropping:
 MPG            0
Cylinders       0
Displacement    0
Horsepower      0
Weight          0
Acceleration    0
Model Year      0
Origin          0
dtype: int64
```

 # Separate features and labels

train_features = train_dataset.copy()

test_features = test_dataset.copy()

train_labels = train_features.pop('MPG')

test_labels = test_features.pop('MPG')

 # Check for NaN values in the dataset

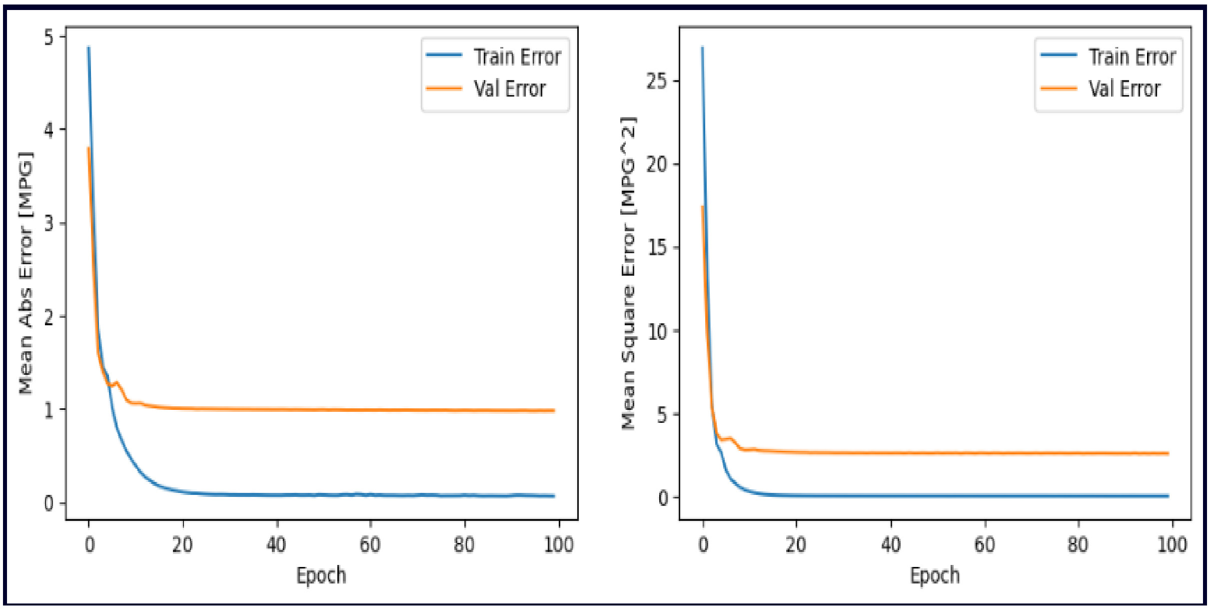
assert not train_features.isnull().any().any(), "There are NaN values in the training features"

assert not test_features.isnull().any().any(), "There are NaN values in the test features"

assert not train_labels.isnull().any(), "There are NaN values in the training labels"

```
assert not test_labels.isnull().any(), "There are NaN values in the test labels"

# Normalize the features
scaler = StandardScaler()
train_features = scaler.fit_transform(train_features)
test_features = scaler.transform(test_features)

def build_model():
    model = models.Sequential([
        layers.Dense(64, activation='relu',
input_shape=[train_features.shape[1]]),
        layers.Dense(64, activation='relu'),
        layers.Dense(1)
    ])
    return model

model = build_model()

model.compile(optimizer='adam',
              loss='mse',
              metrics=['mae', 'mse'])
history = model.fit(train_features, train_labels,
                    epochs=100, validation_split=0.2, verbose=0)

# Plot training history
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
```

```
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.xlabel('Epoch')
plt.ylabel('Mean Abs Error [MPG]')
plt.plot(hist['epoch'], hist['mae'], label='Train Error')
plt.plot(hist['epoch'], hist['val_mae'], label='Val Error')
plt.legend()

plt.subplot(1, 2, 2)
plt.xlabel('Epoch')
plt.ylabel('Mean Square Error [MPG^2]')
plt.plot(hist['epoch'], hist['mse'], label='Train Error')
plt.plot(hist['epoch'], hist['val_mse'], label='Val Error')
plt.legend()

plt.show()
```



test_loss, test_mae, test_mse = model.evaluate(test_features, test_labels, verbose=2)

print(f'\nTest MAE: {test_mae:.2f} MPG')

```
3/3 - 0s - loss: 5.7426 - mae: 1.9889 - mse: 5.7426 - 52ms/epoch - 17ms/step

Test MAE: 1.99 MPG
```

test_predictions = model.predict(test_features).flatten()

plt.scatter(test_labels, test_predictions)

plt.xlabel('True Values [MPG]')

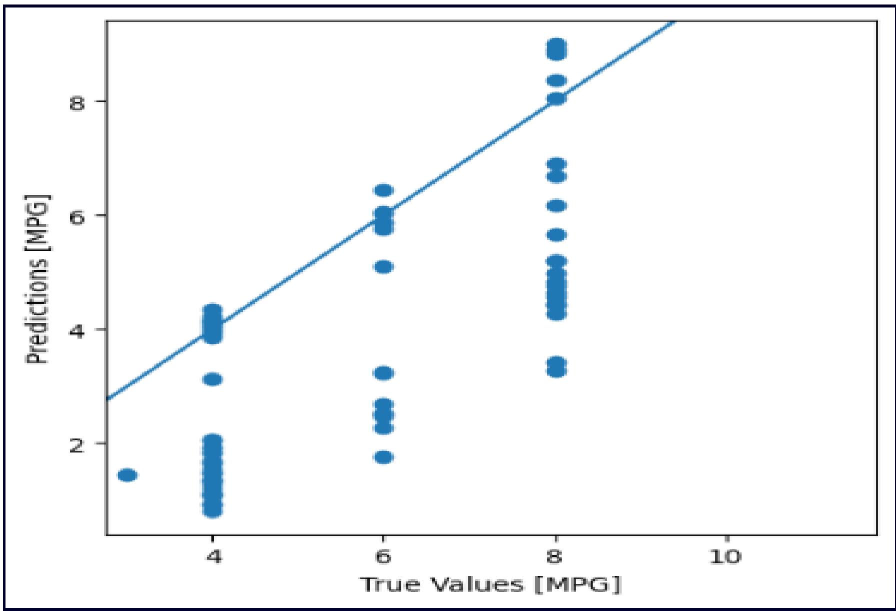
plt.ylabel('Predictions [MPG]')

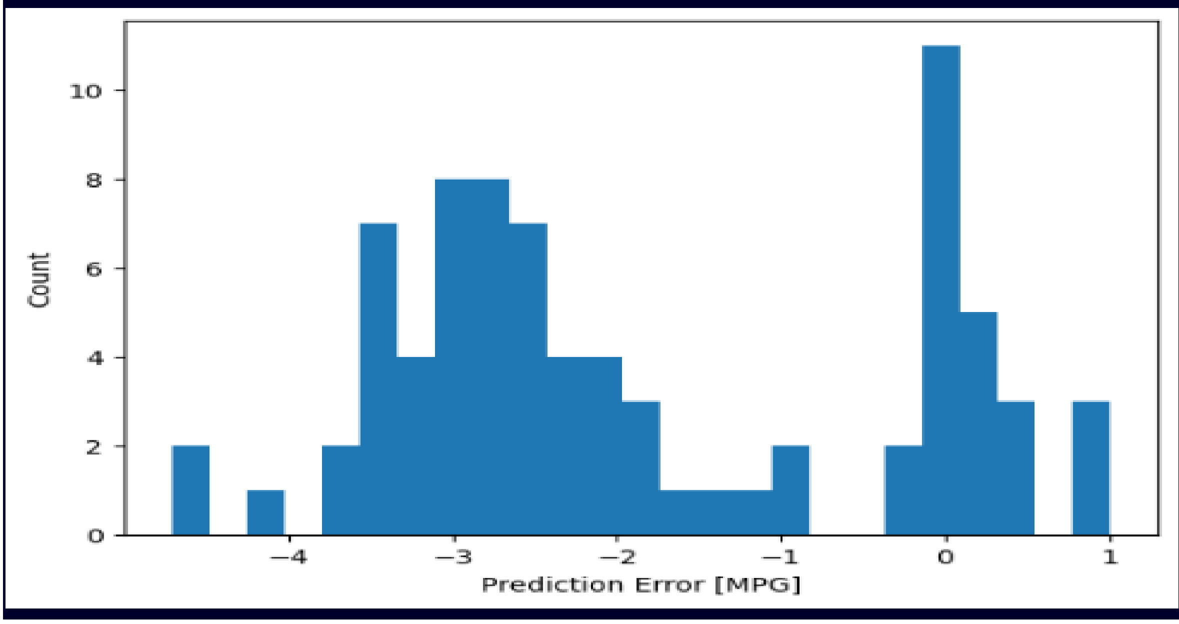
plt.axis('equal')

plt.axis('square')

plt.plot([-100, 100], [-100, 100])

plt.show()


Predictions [MPG]
True Values [MPG]
2
4
6
8
4
6
8
10


Count
Prediction Error [MPG]
0
2
4
6
8
10
−4
−3
−2
−1
0
1

```
error = test_predictions - test_labels
plt.hist(error, bins=25)
plt.xlabel('Prediction Error [MPG]')
plt.ylabel('Count')
plt.show()
```

```
Predicted vs Actual MPG values:
Predicted: 1.33, Actual: 4.00
Predicted: 1.69, Actual: 4.00
Predicted: 4.12, Actual: 4.00
Predicted: 1.36, Actual: 4.00
Predicted: 1.94, Actual: 4.00
Predicted: 4.14, Actual: 4.00
Predicted: 8.92, Actual: 8.00
Predicted: 0.91, Actual: 4.00
Predicted: 2.69, Actual: 6.00
Predicted: 1.09, Actual: 4.00
Predicted: 6.17, Actual: 8.00
Predicted: 5.75, Actual: 6.00
Predicted: 4.98, Actual: 8.00
Predicted: 4.17, Actual: 4.00
Predicted: 2.53, Actual: 6.00
Predicted: 1.36, Actual: 4.00
Predicted: 6.02, Actual: 6.00
Predicted: 4.09, Actual: 4.00
Predicted: 1.85, Actual: 4.00
Predicted: 1.47, Actual: 4.00
Predicted: 6.06, Actual: 6.00
Predicted: 0.94, Actual: 4.00
Predicted: 0.93, Actual: 4.00
...
Predicted: 1.92, Actual: 4.00
Predicted: 4.77, Actual: 8.00
Predicted: 4.29, Actual: 8.00
Predicted: 4.06, Actual: 4.00
```