Data Science Lab

Student PRN:123M1H041

Student Name:DARSHAN SHASHIKANT PATHAK

Submission Date:05/07/2024

Advanced Data Science Lab - Practical Assignment: 4

Part 1: Assignment Based on Supervised Learning - Regression and Classification Algorithm

a) Linear Regression to Predict House Prices

In this task, we'll build a linear regression model to predict house prices using the "California Housing" dataset, which includes various features like median income, housing median age, total rooms, and more. The goal is to implement a complete supervised learning process: data preprocessing, train/test split, training the model, and finally, evaluating the model.

Step 1: **Data Preprocessing**

The first step in any machine learning task is data preprocessing. Preprocessing ensures that the dataset is clean, well-formatted, and ready for model training. Here's a step-by-step breakdown of preprocessing for the California Housing dataset.

1. **Load the Dataset**
   We load the dataset, which contains features like `median_income`, `housing_median_age`, `total_rooms`, and other factors that influence house prices.

In [1]:
```
import pandas as pd
```

```python
# Load the California Housing dataset from the URL
housing_data = pd.read_csv("housing.csv")

# Inspect the first few rows to understand the structure
housing_data.head()
```

Out[1]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	hou
0	-122.23	37.88	41.0	880.0	129.0	322.0	
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	
2	-122.24	37.85	52.0	1467.0	190.0	496.0	
3	-122.25	37.85	52.0	1274.0	235.0	558.0	
4	-122.25	37.85	52.0	1627.0	280.0	565.0	

In [2]:
```python
housing_data = housing_data.drop("ocean_proximity",axis = 1)
```

2. **Handle Missing Values**
   Missing values can lead to inaccurate predictions, so we inspect the dataset for any missing or null values. If we find any, we either fill in those values or drop the rows/columns, depending on the scenario.

In [3]:
```python
# Check for missing values
print(housing_data.isnull().sum())

# Drop missing values (if any)
housing_data = housing_data.dropna()
```

```
longitude               0
latitude                0
housing_median_age      0
total_rooms             0
total_bedrooms        207
population              0
households              0
median_income           0
median_house_value      0
dtype: int64
```

In [4]:
```python
print(housing_data.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 20433 entries, 0 to 20639
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   longitude           20433 non-null  float64
 1   latitude            20433 non-null  float64
 2   housing_median_age  20433 non-null  float64
 3   total_rooms         20433 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20433 non-null  float64
 6   households          20433 non-null  float64
 7   median_income       20433 non-null  float64
 8   median_house_value  20433 non-null  float64
dtypes: float64(9)
memory usage: 1.6 MB
None
```

3. **Feature Scaling/Normalization**
   Since the features have different units (e.g., `total_rooms` and `median_income`), we apply feature scaling using `StandardScaler` from `sklearn`. Feature scaling ensures that all features are on the same scale, which is essential for linear regression models.

In [5]:
```python
housing_data.describe()
```

Out[5]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms
count	20433.000000	20433.000000	20433.000000	20433.000000	20433.000000
mean	-119.570689	35.633221	28.633094	2636.504233	537.870553
std	2.003578	2.136348	12.591805	2185.269567	421.385070
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1450.000000	296.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000
75%	-118.010000	37.720000	37.000000	3143.000000	647.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000

In [6]:
```python
from sklearn.preprocessing import StandardScaler

# Separate features and target variable (median_house_value)
X = housing_data.drop('median_house_value', axis=1)
y = housing_data['median_house_value']

# Normalize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Step 2: **Train/Test Split**

After preprocessing the data, we split it into training and testing sets. Typically, we allocate 80% of the data for training and 20% for testing. This allows us to train the model on the majority of the data and test it on unseen data to evaluate its performance.

In [7]:
```
from sklearn.model_selection import train_test_split

# Split the data into 80% training and 20% testing
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, ran
```

Step 3: **Linear Regression Model**

Once we have the training and testing sets, we can implement the linear regression model using `LinearRegression` from `sklearn`. This model tries to fit a linear relationship between the features (X) and the target variable (house prices).

1. **Initialize the Model**
   We first initialize the linear regression model.

In [8]:
```
from sklearn.linear_model import LinearRegression

# Initialize the Linear Regression model
model = LinearRegression()
```

2. **Train the Model**
   Next, we train the model on the training data. The model learns the relationships between the features and the target (house prices) during this step.

In [9]:
```
# Train the model on the training data
model.fit(X_train, y_train)
```

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

3. **Make Predictions**
   After the model is trained, we make predictions on the test data. These predictions represent the house prices for the unseen test data.

In [10]:
```
# Make predictions on the test set
y_pred = model.predict(X_test)
```

Step 4: **Model Evaluation**

To determine the accuracy and performance of the model, we evaluate it using several metrics. The most common metrics for regression models are:

1. **Mean Squared Error (MSE)**: Measures the average squared difference between the actual and predicted values.
2. **Root Mean Squared Error (RMSE)**: The square root of MSE, providing an interpretable measure of the error in the same units as the target variable.
3. **R-squared (R^2)**: Represents the proportion of variance in the target variable that can be explained by the features.

In [11]:
```
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

# Calculate Mean Squared Error
mse = mean_squared_error(y_test, y_pred)

# Calculate Root Mean Squared Error
rmse = np.sqrt(mse)

# Calculate R-squared
r_squared = r2_score(y_test, y_pred)

print(f"MSE: {mse}")
print(f"RMSE: {rmse}")
print(f"R-squared: {r_squared}")
```

```
MSE: 4921881237.628147
RMSE: 70156.12045736385
R-squared: 0.6400865688993735
```

b) Linear Regression Model to Predict Car Prices

In this task, we'll build a linear regression model to predict car prices based on features such as engine size, horsepower, curb weight, and more. This involves following a similar process as in part (a), but with a different dataset.

Step 1: Data Preprocessing

1. **Load the Dataset**
   We load the "Automobile" dataset, which contains features related to cars such as `engine_size`, `horsepower`, `curb_weight`, and the target variable `price`.

In [12]:
```
# Load the Automobile dataset
auto_data = pd.read_csv("Automobile_data.csv")

# Inspect the first few rows to understand the structure
auto_data.head()
```

Out[12]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front
3	2	164	audi	gas	std	four	sedan	fwd	front
4	2	164	audi	gas	std	four	sedan	4wd	front

5 rows × 26 columns

2. **Handle Missing Values**
   As with the housing dataset, we inspect for and handle missing values. Dropping rows or filling missing values ensures that our data is clean.

In [13]:

```
# Handle missing values
auto_data = auto_data.dropna()
```

3. **Feature Scaling/Normalization**
   We scale the features like `engine_size`, `horsepower`, and `curb_weight` to ensure they are on the same scale.

In [14]:

```
from sklearn.preprocessing import MinMaxScaler, StandardScaler
import numpy as np

# Replace '?' with NaN for proper handling of missing values
auto_data.replace('?', np.nan, inplace=True)

# Convert columns to numeric where applicable
numeric_cols = ['normalized-losses', 'bore', 'stroke', 'horsepower', 'peak-rpm', 'p
for col in numeric_cols:
    auto_data[col] = pd.to_numeric(auto_data[col], errors='coerce')

# Fill missing values with the mean of the column
for col in numeric_cols:
    auto_data[col].fillna(auto_data[col].mean(), inplace=True)

# Select numerical columns for scaling
num_cols = auto_data.select_dtypes(include=['float64', 'int64']).columns

# Apply Min-Max Scaling
min_max_scaler = MinMaxScaler()
auto_data_minmax_scaled = auto_data.copy()
auto_data_minmax_scaled[num_cols] = min_max_scaler.fit_transform(auto_data[num_cols
```

```
# Apply Z-score Normalization
standard_scaler = StandardScaler()
auto_data_standard_scaled = auto_data.copy()
auto_data_standard_scaled[num_cols] = standard_scaler.fit_transform(auto_data[num_c

# Display the first few rows of the scaled data
print("Min-Max Scaled Data:")
print(auto_data_minmax_scaled.head())

print("\
Z-score Normalized Data:")
print(auto_data_standard_scaled.head())
```

```
Min-Max Scaled Data:
   symboling  normalized-losses         make fuel-type aspiration  \
0        1.0           0.298429  alfa-romero       gas        std   
1        1.0           0.298429  alfa-romero       gas        std   
2        0.6           0.298429  alfa-romero       gas        std   
3        0.8           0.518325         audi       gas        std   
4        0.8           0.518325         audi       gas        std   

  num-of-doors   body-style drive-wheels engine-location  wheel-base  ...  \
0          two  convertible          rwd           front    0.058309  ...   
1          two  convertible          rwd           front    0.058309  ...   
2          two    hatchback          rwd           front    0.230321  ...   
3         four        sedan          fwd           front    0.384840  ...   
4         four        sedan          4wd           front    0.373178  ...   

   engine-size  fuel-system      bore    stroke compression-ratio horsepower  \
0     0.260377         mpfi  0.664286  0.290476            0.1250   0.262500   
1     0.260377         mpfi  0.664286  0.290476            0.1250   0.262500   
2     0.343396         mpfi  0.100000  0.666667            0.1250   0.441667   
3     0.181132         mpfi  0.464286  0.633333            0.1875   0.225000   
4     0.283019         mpfi  0.464286  0.633333            0.0625   0.279167   

   peak-rpm  city-mpg  highway-mpg     price  
0  0.346939  0.222222     0.289474  0.207959  
1  0.346939  0.222222     0.289474  0.282558  
2  0.346939  0.166667     0.263158  0.282558  
3  0.551020  0.305556     0.368421  0.219254  
4  0.551020  0.138889     0.157895  0.306142  

[5 rows x 26 columns]
Z-score Normalized Data:
   symboling  normalized-losses         make fuel-type aspiration  \
0   1.743470           0.000000  alfa-romero       gas        std   
1   1.743470           0.000000  alfa-romero       gas        std   
2   0.133509           0.000000  alfa-romero       gas        std   
3   0.938490           1.328961         audi       gas        std   
4   0.938490           1.328961         audi       gas        std   

  num-of-doors   body-style drive-wheels engine-location  wheel-base  ...  \
0          two  convertible          rwd           front   -1.690772  ...   
1          two  convertible          rwd           front   -1.690772  ...   
2          two    hatchback          rwd           front   -0.708596  ...   
3         four        sedan          fwd           front    0.173698  ...   
4         four        sedan          4wd           front    0.107110  ...   

   engine-size  fuel-system      bore    stroke compression-ratio horsepower  \
0     0.074449         mpfi  0.519089 -1.839404         -0.288349   0.171065   
1     0.074449         mpfi  0.519089 -1.839404         -0.288349   0.171065   
2     0.604046         mpfi -2.404862  0.685920         -0.288349   1.261807   
3    -0.431076         mpfi -0.517248  0.462157         -0.035973  -0.057230   
4     0.218885         mpfi -0.517248  0.462157         -0.540725   0.272529   

   peak-rpm  city-mpg  highway-mpg     price  
0 -0.263484 -0.646553    -0.546059  0.036674  
1 -0.263484 -0.646553    -0.546059  0.419498  
2 -0.263484 -0.953012    -0.691627  0.419498  
```

```
3  0.787346 -0.186865    -0.109354  0.094639
4  0.787346 -1.106241    -1.273900  0.540524

[5 rows x 26 columns]
```

Step 2: Train/Test Split

We split the automobile dataset into training and testing sets.

In [15]:
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Select features and target variable
features = auto_data_standard_scaled.drop(columns=['price', 'make', 'fuel-type', 'a
target = auto_data_standard_scaled['price']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2
```

Step 3: Linear Regression Model

1. **Initialize and Train the Model**

   We use `LinearRegression` to train the model on the training data.

In [16]:
```
# Initialize and train the linear regression model
model = LinearRegression()
model.fit(X_train, y_train)
```

Out[16]:
```
▾ LinearRegression
LinearRegression()
```

2. **Make Predictions**

   We use the trained model to predict car prices for the test data.

In [17]:
```
# Make predictions on the test set
y_pred = model.predict(X_test)
```

Step 4: Model Evaluation

We evaluate the model using the same metrics as before: MSE, RMSE, and R-squared.

In [18]:
```
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

# Calculate Mean Squared Error
mse = mean_squared_error(y_test, y_pred)

# Calculate R-squared
r_squared_auto = r2_score(y_test, y_pred)
```

```python
# Print the evaluation metrics
print('Mean Squared Error:', mse)
print('R-squared:', r_squared_auto)

print(f"MSE (Auto): {mse}")
print(f"R-squared (Auto): {r_squared_auto}")
```

```
Mean Squared Error: 0.28235460210155405
R-squared: 0.7768763598184034
MSE (Auto): 0.28235460210155405
R-squared (Auto): 0.7768763598184034
```

2) Assignments Based on Supervised Learning - Regression and Classification Algorithms

This assignment involves implementing both a logistic regression model and a decision tree classifier for binary classification tasks. Each task involves data preprocessing, model training, and evaluation.

a) Implement a Logistic Regression Model to Predict Diabetes

In this part, we'll use the **Pima Indians Diabetes Dataset** to predict whether or not a person has diabetes based on health features such as pregnancies, glucose levels, blood pressure, and more. The steps include data preprocessing, train/test splitting, model training using logistic regression, and model evaluation.

Step 1: Data Preprocessing

1. **Load the Dataset**
   The dataset includes features like Pregnancies, Glucose, BloodPressure, SkinThickness, etc., which will be used to predict the target variable, `Outcome` (1 for diabetes, 0 for no diabetes).

In [19]:
```python
import pandas as pd

# Load the dataset from the provided URL
diabetes_data = pd.read_csv("diabetes.csv")

# Inspect the first few rows
print(diabetes_data.head())
```

```
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            6      148             72             35        0  33.6
1            1       85             66             29        0  26.6
2            8      183             64              0        0  23.3
3            1       89             66             23       94  28.1
4            0      137             40             35      168  43.1

   DiabetesPedigreeFunction  Age  Outcome
0                     0.627   50        1
1                     0.351   31        0
2                     0.672   32        1
3                     0.167   21        0
4                     2.288   33        1
```

2. **Handle Missing Values**
   Check for any missing values in the dataset. For simplicity, let's assume there are no missing values. If there were, we could use imputation techniques to fill them in.

In [20]:
```
# Check for missing values
print(diabetes_data.isnull().sum())
```

```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

3. **Convert Categorical Features**
   In this dataset, there are no categorical features to encode, but if there were, we would use one-hot encoding or label encoding.

Step 2: Train/Test Split

We split the data into 80% for training and 20% for testing. The `Outcome` column is our target variable (whether the person has diabetes or not).

In [21]:
```
from sklearn.model_selection import train_test_split

# Separate features and target variable
X = diabetes_data.drop('Outcome', axis=1)
y = diabetes_data['Outcome']

# Split the data into 80% training and 20% testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

Step 3: Logistic Regression Model

1. **Initialize the Logistic Regression Model**
   Logistic regression is used for binary classification tasks, such as predicting whether someone has diabetes.

In [22]:
```python
from sklearn.linear_model import LogisticRegression

# Initialize the Logistic Regression model
model = LogisticRegression(max_iter=1000)
```

2. **Train the Model**
   We train the logistic regression model on the training data.

In [23]:
```python
# Train the model
model.fit(X_train, y_train)
```

Out[23]:
```
▾        LogisticRegression
LogisticRegression(max_iter=1000)
```

3. **Make Predictions**
   After training, we make predictions on the test set to evaluate how well the model generalizes to unseen data.

In [24]:
```python
# Make predictions on the test set
y_pred = model.predict(X_test)
```

Step 4: **Model Evaluation**

We evaluate the performance of the logistic regression model using metrics such as **accuracy**, **precision**, **recall**, and **F1-score**.

In [25]:
```python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Calculate 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)

# Display the results
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1-Score: {f1}")
```

```
Accuracy: 0.7467532467532467
Precision: 0.6379310344827587
Recall: 0.6727272727272727
F1-Score: 0.6548672566371682
```

b) Build a Decision Tree Classifier for Titanic Survival Prediction

In this task, we'll use the **Titanic Dataset** to predict whether a person survived or not based on features like `Pclass` (passenger class), `Age`, `Sex`, and others. We will follow similar steps as before, but this time using a decision tree classifier.

Step 1: Data Preprocessing

1. **Load the Dataset**
   The Titanic dataset includes both categorical and numerical features. We'll use features like `Pclass`, `Age`, `Sex`, `SibSp`, `Parch`, and `Fare` to predict whether the person survived (`Survived` column).

In [26]:
```
# Load the dataset from the provided URL
titanic_data = pd.read_csv("titanic.csv")

# Inspect the first few rows
titanic_data.head()
```

Out[26]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.250C
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.925C
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.100C
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.050C

2. **Handle Missing Values**
   In the Titanic dataset, the `Age` column has missing values, which we'll fill using the

median age. We'll also drop the `Cabin` column since it has too many missing values.

In [27]:
```
# Fill missing values in the 'Age' column
titanic_data['Age'].fillna(titanic_data['Age'].median(), inplace=True)

# Drop the 'Cabin' column (too many missing values)
titanic_data.drop('Cabin', axis=1, inplace=True)

# Drop other rows with missing values (e.g., Embarked)
titanic_data.dropna(inplace=True)

# Drop Name     column
titanic_data = titanic_data.drop("Name", axis=1)
titanic_data = titanic_data.drop("Ticket", axis=1)
```

3. **Convert Categorical Features**

   We'll convert categorical features like `Sex` and `Embarked` into numerical values using one-hot encoding.

In [28]:
```
# Convert 'Sex' column to numerical values
titanic_data['Sex'] = titanic_data['Sex'].map({'male': 0, 'female': 1})

# One-hot encode the 'Embarked' column
titanic_data = pd.get_dummies(titanic_data, columns=['Embarked'], drop_first=True)
```

Step 2: Train/Test Split

We split the data into 80% training and 20% testing, with `Survived` as the target variable.

In [29]:
```
# Separate features and target variable
X_titanic = titanic_data.drop('Survived', axis=1)
y_titanic = titanic_data['Survived']

# Split the data into 80% training and 20% testing
X_train_titanic, X_test_titanic, y_train_titanic, y_test_titanic = train_test_split
```

Step 3: Decision Tree Model

1. **Initialize the Decision Tree Classifier**

   A decision tree classifier is a tree-like model where decisions are made at each node, leading to a classification at the leaf nodes.

In [30]:
```
from sklearn.tree import DecisionTreeClassifier

# Initialize the Decision Tree model
dt_model = DecisionTreeClassifier(random_state=42)
```

2. **Train the Model**
   We train the decision tree on the training data.

In [31]:
```
# Train the model
dt_model.fit(X_train_titanic, y_train_titanic)
```

Out[31]:
```
DecisionTreeClassifier

DecisionTreeClassifier(random_state=42)
```

3. **Make Predictions**
   After training, we use the model to make predictions on the test set.

In [32]:
```
# Make predictions on the test set
y_pred_titanic = dt_model.predict(X_test_titanic)
```

Step 4: Model Evaluation

We evaluate the decision tree model using the same metrics: accuracy, precision, recall, and F1-score.

In [33]:
```
# Calculate evaluation metrics
accuracy_titanic = accuracy_score(y_test_titanic, y_pred_titanic)
precision_titanic = precision_score(y_test_titanic, y_pred_titanic)
recall_titanic = recall_score(y_test_titanic, y_pred_titanic)
f1_titanic = f1_score(y_test_titanic, y_pred_titanic)

# Display the results
print(f"Accuracy: {accuracy_titanic}")
print(f"Precision: {precision_titanic}")
print(f"Recall: {recall_titanic}")
print(f"F1-Score: {f1_titanic}")
```

```
Accuracy: 0.7303370786516854
Precision: 0.6329113924050633
Recall: 0.7246376811594203
F1-Score: 0.6756756756756758
```

Conclusion

In these tasks, we implemented two classification models:

1. **Logistic Regression** to predict diabetes using health-related features.
2. **Decision Tree Classifier** to predict survival on the Titanic based on passenger information.

Both models were evaluated using standard classification metrics like accuracy, precision, recall, and F1-score, allowing us to measure how well the models performed on their respective tasks.