Q.1

Predict the price of the Uber ride from a given pickup point to the agreed drop-off location. Perform following tasks:

1. Pre-process the dataset.
2. Identify outliers.
3. Check the correlation.
4. Implement linear regression and random forest regression models.
5. Evaluate the models and compare their respective scores like R2, RMSE, etc.

Dataset link: https://www.kaggle.com/datasets/yasserh/uber-fares-datase

[5]:
```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
from scipy import stats

# Load the dataset
df = pd.read_csv('uber.csv')
```

[7]:
```
# Display the first few rows of the dataset
df.head()
```

[7]:
```
   Unnamed: 0                            key  fare_amount  \
0    24238194    2015-05-07 19:52:06.0000003          7.5
1    27835199    2009-07-17 20:04:56.0000002          7.7
2    44984355   2009-08-24 21:45:00.00000061         12.9
3    25894730    2009-06-26 08:22:21.0000001          5.3
4    17610152  2014-08-28 17:47:00.000000188         16.0

           pickup_datetime  pickup_longitude  pickup_latitude  \
0  2015-05-07 19:52:06 UTC        -73.999817        40.738354
1  2009-07-17 20:04:56 UTC        -73.994355        40.728225
2  2009-08-24 21:45:00 UTC        -74.005043        40.740770
3  2009-06-26 08:22:21 UTC        -73.976124        40.790844
4  2014-08-28 17:47:00 UTC        -73.925023        40.744085

   dropoff_longitude  dropoff_latitude  passenger_count
0         -73.999512         40.723217                1
1         -73.994710         40.750325                1
2         -73.962565         40.772647                1
3         -73.965316         40.803349                3
4         -73.973082         40.761247                5
```

[9]:
```
# Convert pickup_datetime to datetime format
df['pickup_datetime'] = pd.to_datetime(df['pickup_datetime'].str.replace('␣
↪UTC', ''), format='%Y-%m-%d %H:%M:%S')
```

[11]:
```
# Feature engineering: Extracting date and time features
df['hour'] = df['pickup_datetime'].dt.hour
df['day_of_week'] = df['pickup_datetime'].dt.dayofweek
df['day_of_month'] = df['pickup_datetime'].dt.day
df['month'] = df['pickup_datetime'].dt.month
df['year'] = df['pickup_datetime'].dt.year
```

[13]:
```
# Drop the key and datetime columns as they are not needed for modeling
df = df.drop(['key', 'pickup_datetime'], axis=1)
```

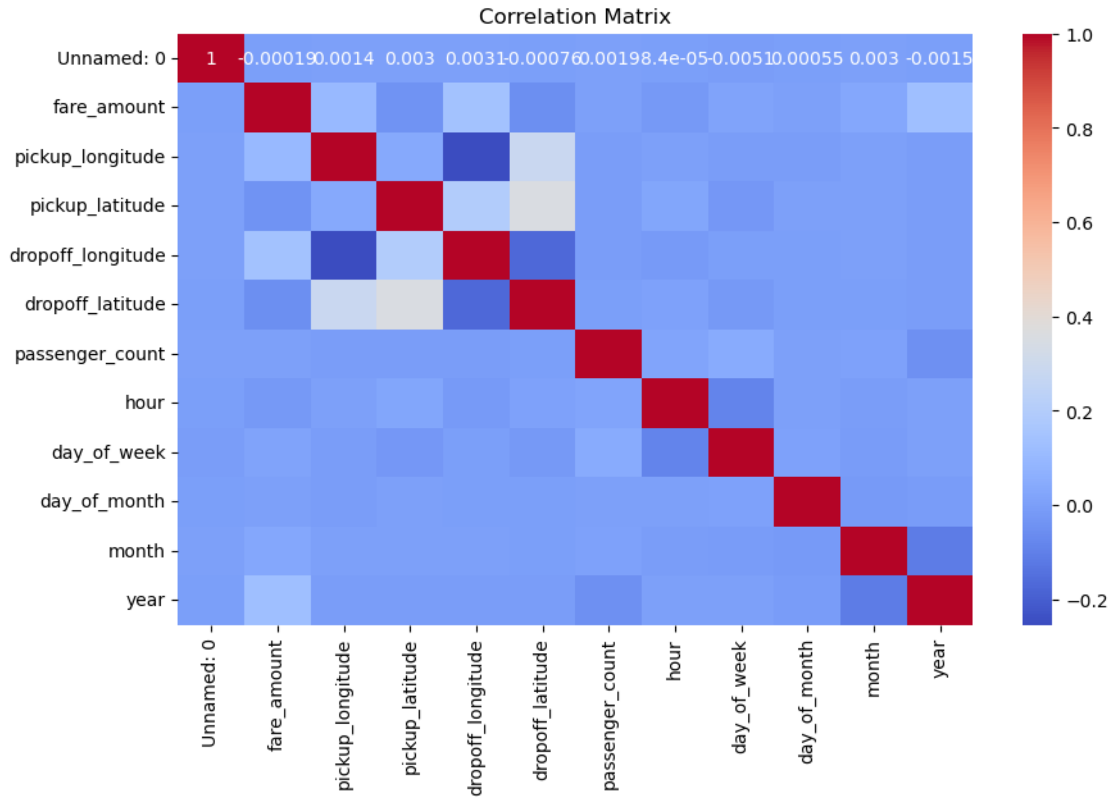
[15]:
```
# Handle missing values if any
df.isnull().sum()
```

[15]:
```
Unnamed: 0           0
fare_amount          0
pickup_longitude     0
pickup_latitude      0
dropoff_longitude    1
dropoff_latitude     1
passenger_count      0
hour                 0
day_of_week          0
day_of_month         0
month                0
year                 0
dtype: int64
```

[17]:
```
# Drop rows with missing values (if any)
df = df.dropna()
```

[19]:
```
# Outlier detection and removal using z-score
z_scores = np.abs(stats.zscore(df[['fare_amount', 'pickup_longitude',␣
↪'pickup_latitude', 'dropoff_longitude', 'dropoff_latitude',␣
↪'passenger_count']]))
df = df[(z_scores < 3).all(axis=1)]
```

[21]:
```
# Correlation matrix
plt.figure(figsize=(10, 6))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```



[23]:
```
# Split the data into features (X) and target (y)
X = df.drop('fare_amount', axis=1)
y = df['fare_amount']
```

[25]:
```
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

[27]:
```
# Linear Regression Model
lr_model = LinearRegression()
lr_model.fit(X_train, y_train)
```

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

[29]:
```
# Random Forest Regressor Model
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
```

[29]:
```
RandomForestRegressor(random_state=42)
```

[30]:
```
# Predictions
y_pred_lr = lr_model.predict(X_test)
y_pred_rf = rf_model.predict(X_test)
```

[31]:
```
# Evaluate models using R² and RMSE
def evaluate_model(y_test, y_pred):
    r2 = r2_score(y_test, y_pred)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    return r2, rmse
```

[32]:
```
# Linear Regression Evaluation
r2_lr, rmse_lr = evaluate_model(y_test, y_pred_lr)
print(f"Linear Regression R²: {r2_lr}, RMSE: {rmse_lr}")
```

```
Linear Regression R²: 0.08288363316197933, RMSE: 6.178183909580108
```

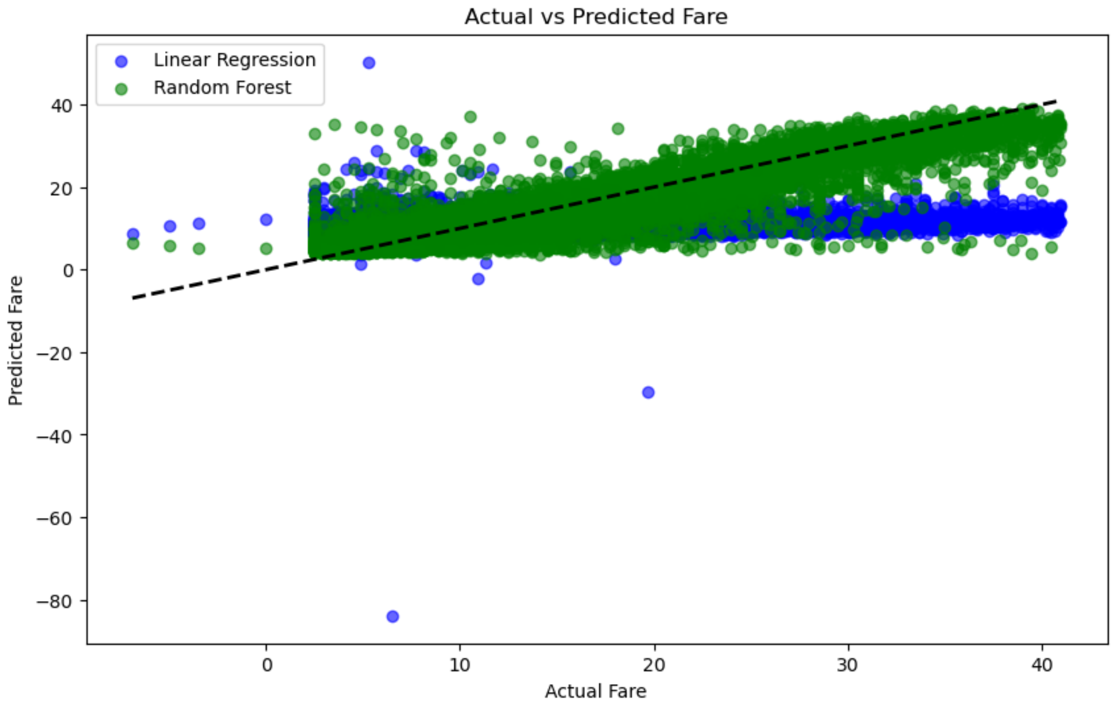
[33]:
```
# Random Forest Evaluation
r2_rf, rmse_rf = evaluate_model(y_test, y_pred_rf)
print(f"Random Forest R²: {r2_rf}, RMSE: {rmse_rf}")
```

```
Random Forest R²: 0.8468259662442992, RMSE: 2.524882573017952
```

[34]:
```
# Compare models
print("\nModel Comparison:")
print(f"Linear Regression - R²: {r2_lr}, RMSE: {rmse_lr}")
print(f"Random Forest - R²: {r2_rf}, RMSE: {rmse_rf}")
```

```

Model Comparison:
Linear Regression - R²: 0.08288363316197933, RMSE: 6.178183909580108
Random Forest - R²: 0.8468259662442992, RMSE: 2.524882573017952
```

[35]:
```
# Plot predictions
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred_lr, label='Linear Regression', color='blue', alpha=0.
  ↪6)
plt.scatter(y_test, y_pred_rf, label='Random Forest', color='green', alpha=0.6)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--',␣
  ↪lw=2)
plt.xlabel('Actual Fare')
plt.ylabel('Predicted Fare')
plt.legend()
plt.title('Actual vs Predicted Fare')
plt.show()
```


Actual vs Predicted Fare
Linear Regression
Random Forest
Predicted Fare
Actual Fare
40
20
0
−20
−40
−60
−80
0
10
20
30
40

[ ]:

Q.2

Classify the email using the binary classification method. Email Spam detection has two states:
a) Normal State – Not Spam,
b) Abnormal State – Spam.

Use K-Nearest Neighbors and Support Vector Machine for classification. Analyze their performance

Dataset link: The emails.csv dataset on the Kaggle
https://www.kaggle.com/datasets/balaka18/email-spam-classification-dataset-csv

[2]:
```
# Import necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report,␣
↪confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset
# Make sure the dataset is available in the correct path
emails_df = pd.read_csv('emails.csv')
```

[4]:
```
# Split the data into features and target
X = emails_df.drop(columns=['Email No.', 'Prediction'])  # Drop unnecessary␣
↪columns
y = emails_df['Prediction']
```

[6]:
```
# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
↪random_state=42)
```

[8]:
```
# Feature scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

[10]:
```
# K-Nearest Neighbors (KNN) model
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
```

[10]:
```
KNeighborsClassifier()
```

[12]:
```
# Predicting with KNN
y_pred_knn = knn.predict(X_test)
```

[14]:
```
# Support Vector Machine (SVM) model
svm = SVC(kernel='linear')
svm.fit(X_train, y_train)
```

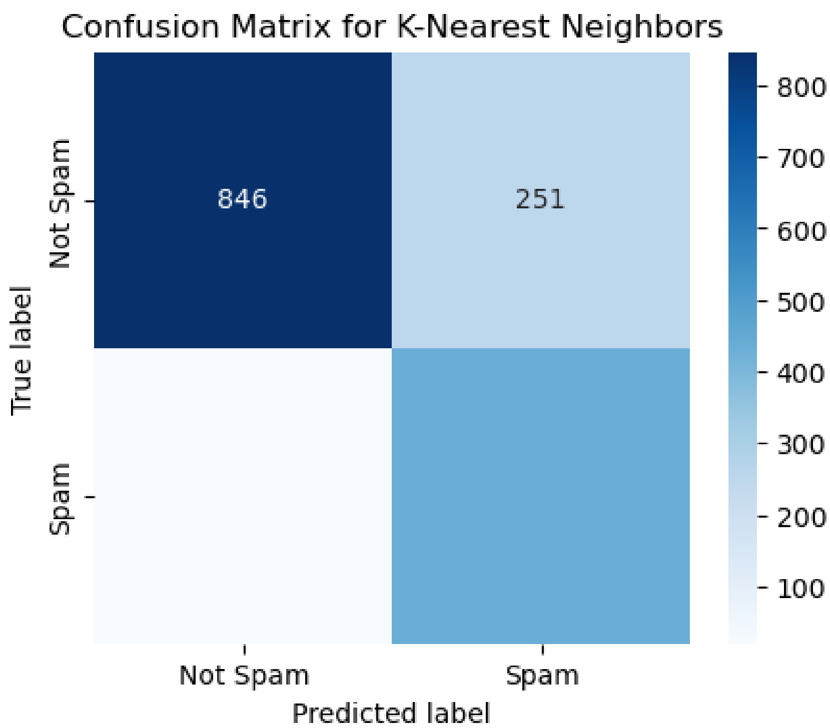
[14]:
```
SVC(kernel='linear')
```

[15]:
```
# Predicting with SVM
y_pred_svm = svm.predict(X_test)
```

[17]:
```
# Evaluate performance of both models
def evaluate_model(y_test, y_pred, model_name):
    print(f"Performance of {model_name}:\n")
    print("Accuracy:", accuracy_score(y_test, y_pred))
    print("Classification Report:\n", classification_report(y_test, y_pred))
    print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))

    # Plot confusion matrix
    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(5, 4))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Spam',␣
↪'Spam'], yticklabels=['Not Spam', 'Spam'])
    plt.title(f"Confusion Matrix for {model_name}")
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()
```

[20]:
```
# Evaluate KNN model
evaluate_model(y_test, y_pred_knn, "K-Nearest Neighbors")
```

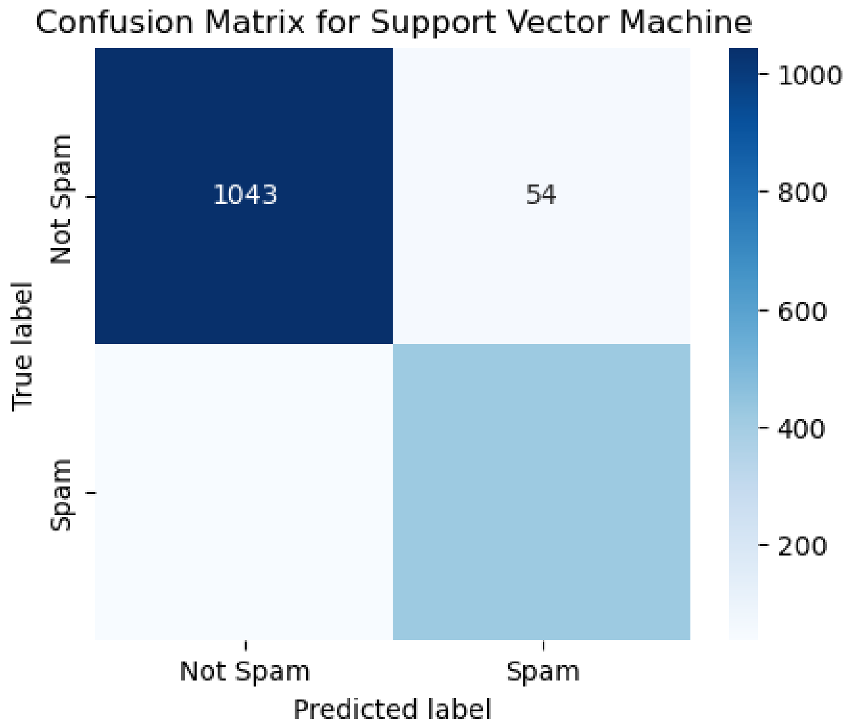
```
Performance of K-Nearest Neighbors:

Accuracy: 0.8253865979381443
Classification Report:
               precision    recall  f1-score   support

           0       0.98      0.77      0.86      1097
           1       0.63      0.96      0.76       455

    accuracy                           0.83      1552
   macro avg       0.81      0.86      0.81      1552
weighted avg       0.88      0.83      0.83      1552

Confusion Matrix:
 [[846 251]
 [ 20 435]]
```



[22]:

```python
# Evaluate SVM model
evaluate_model(y_test, y_pred_svm, "Support Vector Machine")
```

```
Performance of Support Vector Machine:

Accuracy: 0.9400773195876289
Classification Report:
              precision    recall  f1-score   support

           0       0.96      0.95      0.96      1097
           1       0.89      0.91      0.90       455

    accuracy                           0.94      1552
   macro avg       0.92      0.93      0.93      1552
weighted avg       0.94      0.94      0.94      1552

Confusion Matrix:
 [[1043   54]
 [  39  416]]
```



[24]:
```
# Compare accuracy scores
print("K-Nearest Neighbors Accuracy:", accuracy_score(y_test, y_pred_knn))
print("Support Vector Machine Accuracy:", accuracy_score(y_test, y_pred_svm))
```

```
K-Nearest Neighbors Accuracy: 0.8253865979381443
Support Vector Machine Accuracy: 0.9400773195876289
```

[ ]:
```

```

Q.3

Given a bank customer, build a neural network-based classifier that can determine whether they will leave or not in the next 6 months.

Dataset Description: The case study is from an open-source dataset from Kaggle. The dataset contains 10,000 sample points with 14 distinct features such as CustomerId, CreditScore, Geography, Gender, Age, Tenure, Balance, etc.

Link to the Kaggle project: https://www.kaggle.com/barelydedicated/bank-customer-churn-modeling

Perform following steps:
1. Read the dataset.
2. Distinguish the feature and target set and divide the data set into training and test sets.
3. Normalize the train and test data.
4. Initialize and build the model. Identify the points of improvement and implement the same.
5. Print the accuracy score and confusion matrix (5 points).

[9]:
```
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical

# Step 1: Read the dataset
data = pd.read_csv('Churn_Modelling.csv')

# Step 2: Distinguish the feature and target set
X = data.drop(columns=['RowNumber', 'CustomerId', 'Surname', 'Exited'])
y = data['Exited']

# One-hot encode categorical variables
X = pd.get_dummies(X, drop_first=True)

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

```python
# Step 3: Normalize the train and test data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 4: Initialize and build the model
model = Sequential()
model.add(Dense(32, activation='relu', input_shape=(X_train.shape[1],)))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))  # Binary classification

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

# Fit the model to the training data
model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=1)

# Step 5: Evaluate the model
y_pred = (model.predict(X_test) > 0.5).astype("int32")  # Convert probabilities␣
 ↪to binary values

# Calculate accuracy score
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy Score: {accuracy:.4f}")

# Generate confusion matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)
```

```
Epoch 1/50
```

C:\Users\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:87:

```
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
250/250              1s 703us/step -
accuracy: 0.7817 - loss: 0.5122
Epoch 2/50
250/250              0s 649us/step -
accuracy: 0.8185 - loss: 0.4259
Epoch 3/50
250/250              0s 730us/step -
accuracy: 0.8309 - loss: 0.4101
Epoch 4/50
```

```
250/250              0s 646us/step -
accuracy: 0.8720 - loss: 0.3126
Epoch 21/50
250/250              0s 696us/step -
accuracy: 0.8653 - loss: 0.3205
Epoch 22/50
250/250              0s 657us/step -
accuracy: 0.8661 - loss: 0.3226
Epoch 23/50
250/250              0s 659us/step -
accuracy: 0.8621 - loss: 0.3245
Epoch 24/50
250/250              0s 818us/step -
accuracy: 0.8733 - loss: 0.3026
Epoch 25/50
250/250              0s 857us/step -
accuracy: 0.8728 - loss: 0.3095
Epoch 26/50
250/250              0s 718us/step -
accuracy: 0.8660 - loss: 0.3163
Epoch 27/50
250/250              0s 627us/step -
accuracy: 0.8683 - loss: 0.3176
Epoch 28/50
250/250              0s 655us/step -
accuracy: 0.8690 - loss: 0.3177
Epoch 29/50
250/250              0s 623us/step -
accuracy: 0.8672 - loss: 0.3182
Epoch 30/50
250/250              0s 683us/step -
accuracy: 0.8642 - loss: 0.3164
Epoch 31/50
250/250              0s 676us/step -
accuracy: 0.8658 - loss: 0.3144
Epoch 32/50
250/250              0s 779us/step -
accuracy: 0.8709 - loss: 0.3047
Epoch 33/50
250/250              0s 697us/step -
accuracy: 0.8689 - loss: 0.3112
Epoch 34/50
250/250              0s 667us/step -
accuracy: 0.8694 - loss: 0.3084
Epoch 35/50
250/250              0s 703us/step -
accuracy: 0.8762 - loss: 0.3019
Epoch 36/50
```

```
250/250              0s 724us/step -
accuracy: 0.8669 - loss: 0.3111
Epoch 37/50
250/250              0s 738us/step -
accuracy: 0.8779 - loss: 0.2965
Epoch 38/50
250/250              0s 705us/step -
accuracy: 0.8701 - loss: 0.3056
Epoch 39/50
250/250              0s 662us/step -
accuracy: 0.8771 - loss: 0.2979
Epoch 40/50
250/250              0s 655us/step -
accuracy: 0.8691 - loss: 0.3061
Epoch 41/50
250/250              0s 699us/step -
accuracy: 0.8765 - loss: 0.2988
Epoch 42/50
250/250              0s 729us/step -
accuracy: 0.8724 - loss: 0.3032
Epoch 43/50
250/250              0s 705us/step -
accuracy: 0.8731 - loss: 0.3151
Epoch 44/50
250/250              0s 696us/step -
accuracy: 0.8780 - loss: 0.3002
Epoch 45/50
250/250              0s 739us/step -
accuracy: 0.8757 - loss: 0.2980
Epoch 46/50
250/250              0s 819us/step -
accuracy: 0.8742 - loss: 0.3036
Epoch 47/50
250/250              0s 887us/step -
accuracy: 0.8741 - loss: 0.3022
Epoch 48/50
250/250              0s 842us/step -
accuracy: 0.8756 - loss: 0.3024
Epoch 49/50
250/250              0s 784us/step -
accuracy: 0.8703 - loss: 0.3098
Epoch 50/50
250/250              0s 830us/step -
accuracy: 0.8793 - loss: 0.2916
63/63            0s 1ms/step
Accuracy Score: 0.8620
Confusion Matrix:
[[1529   78]
 [ 198  195]]
```

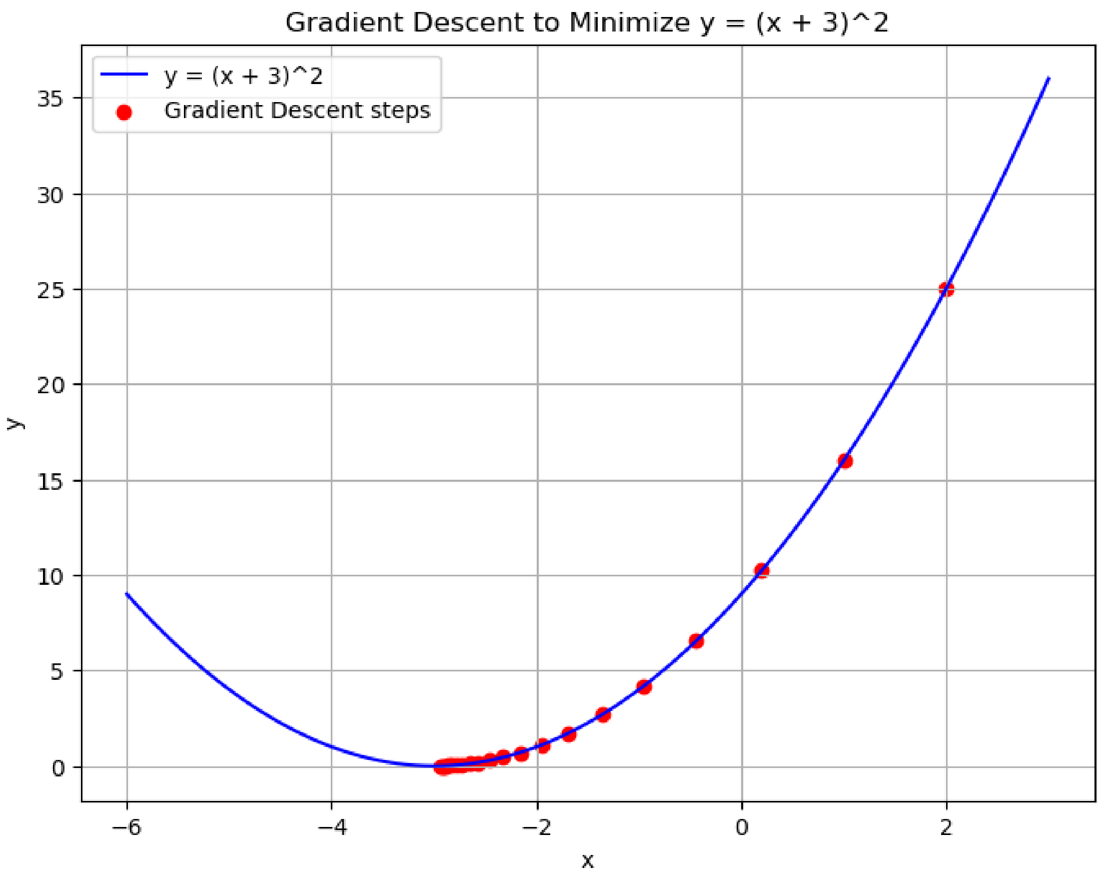
Q.4

Implement Gradient Descent Algorithm to find the local minima of a function.
For example, find the local minima of the function y=(x+3)² starting from the point x=2.

[1]:
```
# Gradient Descent Algorithm to find the local minima of y = (x+3)^2

import numpy as np
import matplotlib.pyplot as plt

# Define the function and its derivative
def function(x):
    return (x + 3) ** 2

def gradient(x):
    return 2 * (x + 3)

# Gradient Descent function
def gradient_descent(starting_x, learning_rate, num_iterations):
    x = starting_x
    x_values = [x]  # To store values for plotting
    y_values = [function(x)]

    for i in range(num_iterations):
        grad = gradient(x)
        x = x - learning_rate * grad

        # Store the values
        x_values.append(x)
        y_values.append(function(x))

        print(f"Iteration {i+1}: x = {x}, f(x) = {function(x)}")

    return x, x_values, y_values

# Parameters
starting_x = 2  # Starting point
learning_rate = 0.1  # Learning rate
num_iterations = 20  # Number of iterations

# Run Gradient Descent
```

```
final_x, x_values, y_values = gradient_descent(starting_x, learning_rate,␣
 ↪num_iterations)

# Plot the results
plt.figure(figsize=(8, 6))
x_range = np.linspace(-6, 3, 100)
y_range = function(x_range)
plt.plot(x_range, y_range, label='y = (x + 3)^2', color='blue')
plt.scatter(x_values, y_values, color='red', label='Gradient Descent steps')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Gradient Descent to Minimize y = (x + 3)^2')
plt.legend()
plt.grid(True)
plt.show()
```

```
Iteration 1: x = 1.0, f(x) = 16.0
Iteration 2: x = 0.19999999999999996, f(x) = 10.240000000000002
Iteration 3: x = -0.44000000000000017, f(x) = 6.553599999999998
Iteration 4: x = -0.9520000000000001, f(x) = 4.194304
Iteration 5: x = -1.3616000000000001, f(x) = 2.6843545599999996
Iteration 6: x = -1.6892800000000001, f(x) = 1.7179869183999996
Iteration 7: x = -1.951424, f(x) = 1.099511627776
Iteration 8: x = -2.1611392, f(x) = 0.7036874417766399
Iteration 9: x = -2.32891136, f(x) = 0.4503599627370493
Iteration 10: x = -2.463129088, f(x) = 0.28823037615171165
Iteration 11: x = -2.5705032704, f(x) = 0.1844674407370954
Iteration 12: x = -2.6564026163200003, f(x) = 0.11805916207174093
Iteration 13: x = -2.725122093056, f(x) = 0.07555786372591429
Iteration 14: x = -2.7800976744448, f(x) = 0.04835703278458515
Iteration 15: x = -2.82407813955584, f(x) = 0.030948500982134555
Iteration 16: x = -2.8592625116446717, f(x) = 0.019807040628566166
Iteration 17: x = -2.8874100093157375, f(x) = 0.012676506002282305
Iteration 18: x = -2.90992800745259, f(x) = 0.008112963841460692
Iteration 19: x = -2.927942405962072, f(x) = 0.005192296858534868
Iteration 20: x = -2.9423539247696575, f(x) = 0.0033230699894623056
```


Gradient Descent to Minimize y = (x + 3)^2
y = (x + 3)^2
Gradient Descent steps
35
30
25
20
15
10
5
0
y
−6
−4
−2
0
2
x

[ ]:

Q.5

Implement K-Nearest Neighbors algorithm on diabetes.csv dataset. Compute confusion matrix, accuracy, error rate, precision and recall on the given dataset.

Dataset link : https://www.kaggle.com/datasets/abdallamahgoub/diabetes

[1]:
```
# Import necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score,␣
↪recall_score

# Load the dataset
df = pd.read_csv('diabetes.csv')

# Split the dataset into features (X) and target (y)
X = df.drop('Outcome', axis=1)
y = df['Outcome']

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

# Standardize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initialize the KNN classifier with k=5
knn = KNeighborsClassifier(n_neighbors=5)

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

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

# Compute the confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Compute accuracy, error rate, precision, and recall
```

```python
accuracy = accuracy_score(y_test, y_pred)
error_rate = 1 - accuracy
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)

# Print the results
print(f"Confusion Matrix:\n{conf_matrix}")
print(f"Accuracy: {accuracy * 100:.2f}%")
print(f"Error Rate: {error_rate * 100:.2f}%")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
```

```
Confusion Matrix:
[[79 20]
 [27 28]]
Accuracy: 69.48%
Error Rate: 30.52%
Precision: 0.58
Recall: 0.51
```

[ ]:

```

```

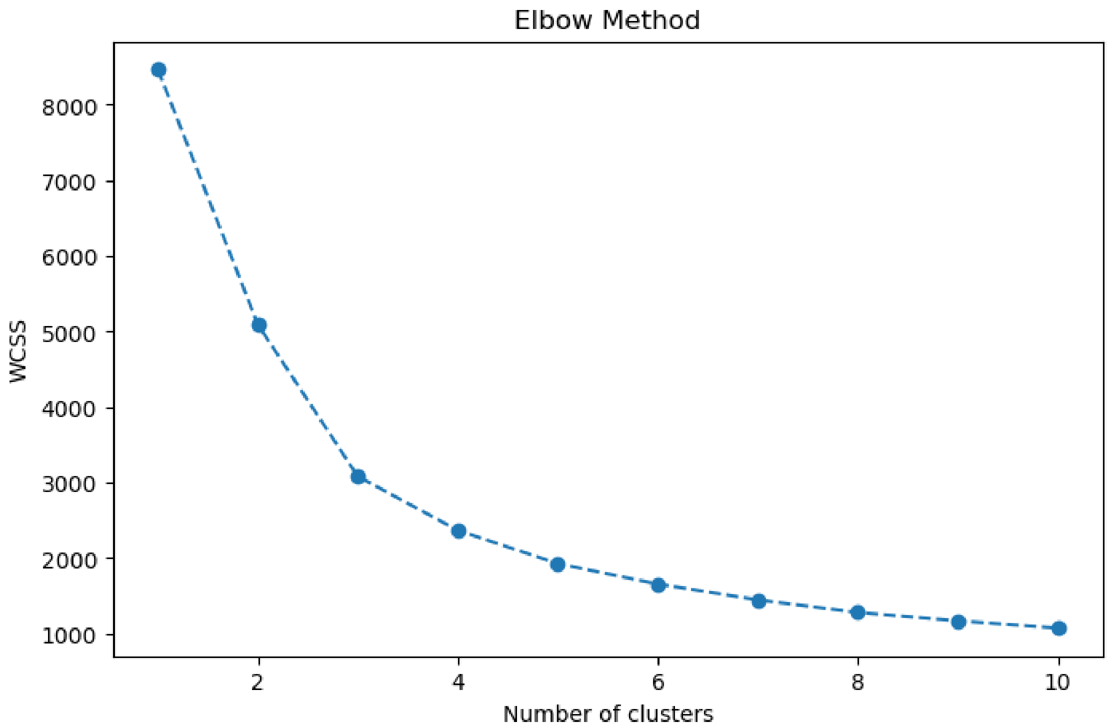
Q.6

Implement K-Means clustering/ hierarchical clustering on sales_data_sample.csv dataset. Determine the number of clusters using the elbow method.

Dataset link : https://www.kaggle.com/datasets/kyanyoga/sample-sales-data

[17]:
```
# Importing necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

# Load the dataset with a specified encoding to avoid the UnicodeDecodeError
df = pd.read_csv('sales_data_sample.csv', encoding='ISO-8859-1')

# Select relevant features for clustering
# We will use 'QUANTITYORDERED', 'PRICEEACH', and 'SALES'
features = df[['QUANTITYORDERED', 'PRICEEACH', 'SALES']]

# Scale the data
scaler = StandardScaler()
scaled_features = scaler.fit_transform(features)

# Elbow method to find the optimal number of clusters
wcss = []  # List to store the within-cluster sum of squares

# Trying different values of k (number of clusters)
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', random_state=42)
    kmeans.fit(scaled_features)
    wcss.append(kmeans.inertia_)  # Inertia: Sum of squared distances of␣
↪samples to their closest cluster center

# Plotting the elbow graph
plt.figure(figsize=(8,5))
plt.plot(range(1, 11), wcss, marker='o', linestyle='--')
plt.title('Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()

# Based on the elbow plot, choose the optimal number of clusters (let's say 3)
optimal_clusters = 3
```

```
# Perform K-Means clustering
kmeans = KMeans(n_clusters=optimal_clusters, init='k-means++', random_state=42)
df['Cluster'] = kmeans.fit_predict(scaled_features)
# Print the resulting clusters
print(df[['ORDERNUMBER', 'QUANTITYORDERED', 'PRICEEACH', 'SALES', 'Cluster']])
```




```
      ORDERNUMBER  QUANTITYORDERED  PRICEEACH    SALES  Cluster
0           10107               30      95.70  2871.00        1
1           10121               34      81.35  2765.90        1
2           10134               41      94.74  3884.34        2
3           10145               45      83.26  3746.70        2
4           10159               49     100.00  5205.27        2
…             …                …        …       …       …
2818        10350               20     100.00  2244.40        1
2819        10373               29     100.00  3978.51        1
2820        10386               43     100.00  5417.57        2
2821        10397               34      62.24  2116.16        0
2822        10414               47      65.52  3079.44        0

[2823 rows x 5 columns]
```

[ ]: