ML Viva

ML_1_Uber_Price_Prediction.py
Python

uber.csv
Spreadsheet

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-dataset Given Above is the Problem Statement for the lab Practical in the subject of machine learning and the solution for the problem along with dataset are attached . explain the program line by line in detail and plot the execution and visualixations.
Given above is the Problem statement for the lab practical in th subject of ML. attached the Code for the implementation Explain the Code Line By line in detail.

Let's go through the code step-by-step to understand how it predicts the Uber fare using Linear Regression and Random Forest Regression:

1. Import Libraries

```
python


  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 r2_score, mean_squared_error
  from geopy.distance import geodesic
```

- These libraries provide tools for data manipulation (`pandas`, `numpy`), visualization (`matplotlib`, `seaborn`), machine learning (`sklearn`), and calculating geographical distances (`geopy`).

2. Load Dataset

```
python


  df = pd.read_csv("./uber.csv")
```

- Loads the dataset from the `uber.csv` file into a DataFrame `df`.

3. Display Dataset Information

```
python


  print("Dataset Information:\n")
  print(df.info())
  print("\nDataset Head:\n")
  print(df.head())
```

- Displays basic information and a preview of the dataset for a quick overview.

4. Data Preprocessing

- **Convert `pickup_datetime` to datetime format**:

  ```
  python


    df['pickup_datetime'] = pd.to_datetime(df['pickup_datetime'], errors='coerce')
  ```

- **Drop rows with missing `pickup_datetime` or `fare_amount`**:

  ```
  python


    df.dropna(subset=['pickup_datetime', 'fare_amount'], inplace=True)
  ```

- **Remove invalid `fare_amount` and `passenger_count` values**:

  ```
  python


    df = df[(df['fare_amount'] > 0) & (df['fare_amount'] < 100)]
    df = df[(df['passenger_count'] > 0) & (df['passenger_count'] <= 6)]
  ```

- **Check and remove invalid latitude and longitude**:

  ```
  python


    invalid_rows = df[(df['pickup_latitude'] < -90) | (df['pickup_latitude'] > 90) | ...]
  ```

5. Calculate Distance between Pickup and Dropoff

```
python


  def calculate_distance(row):
      try:
          return geodesic((row['pickup_latitude'], row['pickup_longitude']),
```

```
                          (row['dropoff_latitude'], row['dropoff_longitude'])).km
df['distance_km'] = df.apply(calculate_distance, axis=1)
```

- Calculates the distance between pickup and drop-off points and adds it as a new column **`distance_km`**.

6. Feature Engineering

- **Extract date and time features**:

```
python


df['pickup_hour'] = df['pickup_datetime'].dt.hour
df['pickup_day'] = df['pickup_datetime'].dt.day
df['pickup_month'] = df['pickup_datetime'].dt.month
df['pickup_year'] = df['pickup_datetime'].dt.year
```

- **Drop unnecessary columns**:

```
python


df.drop(['key', 'pickup_datetime', 'pickup_longitude', 'pickup_latitude', 'dropoff_lo
ngitude', 'dropoff_latitude'], axis=1, inplace=True)
```

7. Identify Outliers

- **Visualize distributions of `fare_amount` and `distance_km`**:

```
python


plt.figure(figsize=(12, 5))
sns.boxplot(df['fare_amount'])
sns.boxplot(df['distance_km'])
```

- **Remove outliers based on z-scores**:

```
python


df = df[(np.abs(stats.zscore(df[['fare_amount', 'distance_km']])) < 3).all(axis=1)]
```

8. Correlation Analysis

```
python


plt.figure(figsize=(10, 6))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm', fmt=".2f")
```

- Displays a correlation heatmap to identify relationships between features.

9. Model Training

- **Define features (`X`) and target variable (`y`)**:

```
python


  X = df.drop('fare_amount', axis=1)
  y = df['fare_amount']
```

- **Split data into training and testing sets**:

```
python


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

- **Train Linear Regression model**:

```
python


  linear_model = LinearRegression()
  linear_model.fit(X_train, y_train)
```

- **Train Random Forest Regression model**:

```
python


  random_forest_model = RandomForestRegressor(n_estimators=100, random_state=42)
  random_forest_model.fit(X_train, y_train)
```

10. Model Evaluation

- **Predict on test set**:

```
python


  y_pred_linear = linear_model.predict(X_test)
  y_pred_rf = random_forest_model.predict(X_test)
```

- **Define evaluation function**:

```
python


  def evaluate_model(y_true, y_pred):
      r2 = r2_score(y_true, y_pred)
```

```
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    return r2, rmse
```

- **Calculate metrics for Linear Regression**:

```
python


  r2_linear, rmse_linear = evaluate_model(y_test, y_pred_linear)
```

- **Calculate metrics for Random Forest Regression**:

```
python


  r2_rf, rmse_rf = evaluate_model(y_test, y_pred_rf)
```

11. Visualization of Results

- **Plot predictions for both models**:

```
python


  plt.scatter(y_test, y_pred_linear, alpha=0.5, color='blue')
  plt.scatter(y_test, y_pred_rf, alpha=0.5, color='green')
```

- **Compare R2 and RMSE in a bar plot**:

```
python


  sns.barplot(x='Model', y='R^2 Score', data=metrics, palette='viridis')
  sns.barplot(x='Model', y='RMSE', data=metrics, palette='viridis')
```

12. Full Solution with Predictions

```
python


  solution= df.copy()
  solution["LinerPred"]=linear_model.predict(df.drop('fare_amount', axis=1))
  solution["RandomForestPred"]=random_forest_model.predict(df.drop('fare_amount', axis=1))
```

- This final code block creates a copy of `df` and adds columns with predictions from both models, creating a comprehensive solution for fare prediction.

This code comprehensively handles Uber fare prediction using linear and random forest regression models, while providing visual insights and comparisons.

Now Predict The Questions which and provide the appropriate answers which are most probably going to asked on this practical in Viva.

Here are some questions that may be asked in a viva based on this Uber fare prediction practical:

1. Explain the process of data pre-processing. Why is it important?

- **Answer**: Data pre-processing involves cleaning the data to remove inaccuracies, missing values, and invalid entries. In this practical, it includes converting the `pickup_datetime` column to datetime format, handling missing values, removing outliers, and ensuring latitude and longitude values are valid. Pre-processing is essential as it ensures the model learns from clean, accurate data, leading to better performance and generalization.

2. Why do we convert the `pickup_datetime` column to datetime format?

- **Answer**: Converting `pickup_datetime` to datetime format allows us to extract useful temporal features, such as hour, day, month, and year. These features help the model learn patterns based on time, which can affect Uber fares due to peak hours or seasonal variations.

3. What are outliers, and how did you handle them in this dataset?

- **Answer**: Outliers are data points that are significantly different from others, often due to anomalies or errors. They can distort the model's understanding of the data. In this dataset, outliers were identified by visualizing box plots of `fare_amount` and `distance_km`, and they were removed based on z-scores, where values beyond three standard deviations from the mean were eliminated.

4. Why did you choose `fare_amount` as the target variable?

- **Answer**: `fare_amount` represents the Uber fare, which we aim to predict based on features like distance, passenger count, and time of pickup. Since it is the dependent variable we want to estimate, it serves as the target variable for both regression models.

5. Explain the importance of correlation analysis. How did you interpret the results?

- **Answer**: Correlation analysis reveals the strength and direction of relationships between variables. In this practical, the heatmap shows how features like `distance_km` correlate with `fare_amount`. A high correlation indicates a strong linear relationship, helping us identify features that may be more impactful for the model's predictions.

6. Why did you use Linear Regression and Random Forest Regression for this task?

- **Answer**: Linear Regression is a simple model that assumes a linear relationship between features and the target. It's good for understanding basic trends. Random Forest, a more complex ensemble model, is better at capturing non-linear relationships and interactions among features, providing potentially higher accuracy.

7. What are R2 and RMSE, and why are they used to evaluate the models?

- **Answer**:
  - **R2 (Coefficient of Determination)**: It measures the proportion of variance in the target variable explained by the features. A higher R2 score indicates a better fit.
  - **RMSE (Root Mean Squared Error)**: It measures the average deviation between predicted and actual values. Lower RMSE indicates more accurate predictions.
- These metrics provide complementary insights—R2 gives an overall fit, while RMSE indicates prediction precision.

8. How does the Random Forest model work, and why might it outperform Linear Regression here?

- **Answer**: Random Forest is an ensemble model that builds multiple decision trees on random subsets of the data and averages their results. It captures non-linear relationships and interactions between features better than Linear Regression. This is useful for predicting fares, as factors like distance and time have non-linear effects on the fare.

9. What is the purpose of the `geodesic` function in this code?

- **Answer**: The `geodesic` function from the `geopy` library calculates the geographical distance between the pickup and drop-off points. This distance is crucial, as it is one of the primary features affecting the fare.

10. Why do we split the data into training and testing sets?

- **Answer**: Splitting data into training and testing sets helps us evaluate the model's performance on unseen data. The model is trained on the training set and tested on the testing set to assess its generalization ability, reducing the risk of overfitting.

11. What does `train_test_split` do, and why is `random_state` set?

- **Answer**: `train_test_split` splits the dataset into training and testing subsets. Setting `random_state` ensures reproducibility, meaning that every time we run the code, the split will be the same, allowing for consistent model evaluation.

12. Explain the significance of hyperparameters like `n_estimators` in Random Forest.

- **Answer**: `n_estimators` specifies the number of trees in the forest. More trees generally improve performance up to a point by reducing variance, though it also increases computational cost. Setting it to 100 balances performance and efficiency.

13. What does `np.abs(stats.zscore(...)) < 3` do in outlier removal?

- **Answer**: This calculates the z-score for each data point, which represents how many standard deviations a value is from the mean. By keeping only values with a z-score less than 3, we filter out extreme outliers beyond three standard deviations.

14. Describe how predictions for fare are visualized in this code.

- **Answer**: Scatter plots compare actual fares (`y_test`) with predicted fares (`y_pred`) for both Linear and Random Forest models. A line representing perfect predictions (actual = predicted) helps visually assess model accuracy. Additionally, bar plots compare R2 and RMSE scores between the two models.

15. How would you improve this model if given more time or data?

- **Answer**: Possible improvements include feature engineering, like adding weather or traffic data for better fare predictions. Hyperparameter tuning, advanced models like Gradient Boosting, or Neural Networks could also enhance accuracy.

These questions cover essential aspects of the practical, from data pre-processing and model selection to evaluation metrics and model improvement, helping to prepare effectively for the viva.

ML_2_Email_Spam_Classification.py
Python

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

Given above is the Problem statement for the lab practical in th subject of ML. attached the Code for the implementation Explain the Code Line By line in detail.

Let's go through this code step-by-step for email spam classification using K-Nearest Neighbors (KNN) and Support Vector Machine (SVM):

1. Import Required Libraries

```python
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.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.decomposition import PCA
```

- These libraries provide tools for data handling (`pandas`, `numpy`), visualization (`matplotlib`, `seaborn`), and machine learning (`sklearn`), with additional tools for scaling (`StandardScaler`), classification (KNN, SVM), and dimensionality reduction (`PCA`).

2. Load the Dataset

```
python


  df = pd.read_csv("emails.csv")
```

- Loads the dataset from `emails.csv` into a DataFrame `df`.

3. Data Exploration

```
python


  print("Dataset Shape:", df.shape)
  print("Dataset Sample:\n", df.head())
  print("Missing values:\n", df.isnull().sum().sum())
```

- Displays the dataset shape, a sample of the data, and checks for any missing values.

4. Data Preprocessing

- **Drop the first column (email identifier)**:

  ```
  python


    df.drop(df.columns[0], axis=1, inplace=True)
  ```

- **Separate features (`X`) and target variable (`y`)**:

  ```
  python


    X = df.iloc[:, :-1]
    y = df.iloc[:, -1]
  ```

- Drops the identifier column and splits data into features and target labels. `X` includes all features, while `y` is the target column indicating if an email is spam (1) or not (0).

5. Train-Test Split

```
python


  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

- Splits the dataset into training and testing sets, with 70% for training and 30% for testing. Setting `random_state=42` ensures consistent results.

6. Feature Scaling

```
python


  scaler = StandardScaler()
  X_train_scaled = scaler.fit_transform(X_train)
  X_test_scaled = scaler.transform(X_test)
```

- Scales features to standardize them, which improves model performance. `fit_transform` is applied to the training set, and `transform` is used on the test set to ensure consistency.

7. Dimensionality Reduction with PCA

```
python


  pca = PCA(n_components=2)
  X_train_pca = pca.fit_transform(X_train_scaled)
  X_test_pca = pca.transform(X_test_scaled)
```

- Reduces the features to 2 principal components for visualizing data distribution, making it easier to observe class separation visually.

8. Visualization of Data Distribution After PCA

```
python


  plt.figure(figsize=(10, 6))
  sns.scatterplot(x=X_train_pca[:, 0], y=X_train_pca[:, 1], hue=y_train, palette="coolwarm",
  s=60)
  plt.title("Data Distribution After PCA")
  plt.xlabel("PCA Component 1")
  plt.ylabel("PCA Component 2")
  plt.show()
```

- Plots the reduced data to observe separation between the classes (spam and non-spam emails).

9. Model Training and Prediction - KNN

```
python


  knn = KNeighborsClassifier(n_neighbors=5)
  knn.fit(X_train_scaled, y_train)
  y_pred_knn = knn.predict(X_test_scaled)
```

- Initializes K-Nearest Neighbors with 5 neighbors, trains it on the scaled training set, and makes predictions on the test set.

10. Model Training and Prediction - SVM

```
python


  svm = SVC(kernel='linear', random_state=42)
  svm.fit(X_train_scaled, y_train)
  y_pred_svm = svm.predict(X_test_scaled)
```

- Initializes an SVM model with a linear kernel, trains it on the scaled training set, and predicts on the test set.

11. Performance Analysis

- **Define a function to display metrics**:

```
python


  def display_metrics(y_true, y_pred, model_name):
      print(f"--- {model_name} ---")
      print("Accuracy:", accuracy_score(y_true, y_pred))
      print("Classification Report:\n", classification_report(y_true, y_pred))
      cm = confusion_matrix(y_true, y_pred)
      plt.figure(figsize=(6, 4))
      sns.heatmap(cm, annot=True, fmt="d", cmap="YlGnBu", cbar=False, xticklabels=['Not
  Spam', 'Spam'], yticklabels=['Not Spam', 'Spam'])
      plt.title(f"{model_name} Confusion Matrix")
      plt.xlabel("Predicted Label")
      plt.ylabel("True Label")
      plt.show()
```

- **Display metrics for KNN and SVM**:

```
python


  display_metrics(y_test, y_pred_knn, "K-Nearest Neighbors")
  display_metrics(y_test, y_pred_svm, "Support Vector Machine")
```

- Calculates accuracy, generates a classification report, and plots the confusion matrix for both models to evaluate precision, recall, F1 score, and model accuracy.

12. Model Comparison Using Bar Chart

```
python


  knn_accuracy = accuracy_score(y_test, y_pred_knn)
  svm_accuracy = accuracy_score(y_test, y_pred_svm)
  model_names = ['K-Nearest Neighbors', 'Support Vector Machine']
  accuracies = [knn_accuracy, svm_accuracy]

  plt.figure(figsize=(8, 5))
  sns.barplot(x=model_names, y=accuracies, palette="viridis")
  plt.title("Comparison of Model Accuracies")
  plt.xlabel("Model")
  plt.ylabel("Accuracy")
  plt.ylim(0, 1)
  plt.show()
```

- Plots a bar chart to visually compare the accuracy of KNN and SVM.

13. Add Predictions to the Original DataFrame

```
python


  X_scaled = scaler.transform(X)
  df['KNN_Prediction'] = knn.predict(X_scaled)
  df['SVM_Prediction'] = svm.predict(X_scaled)
  print("Updated DataFrame with Predictions:")
  print(df.head())
```

- Adds the predictions from both models as new columns in the DataFrame, giving insights into whether each email is classified as spam or not by both KNN and SVM models.

This code provides a comprehensive approach to classifying emails as spam or not, with visual insights, model training, and evaluation of KNN and SVM performance.

Now Predict The Questions which and provide the appropriate answers which are most probably going to asked on this practical in Viva.

Here are some questions likely to be asked during a viva on this practical for email spam classification using K-Nearest Neighbors (KNN) and Support Vector Machine (SVM):

1. What is the purpose of this practical?

- **Answer**: The goal of this practical is to classify emails as spam or not spam using binary classification techniques. K-Nearest Neighbors and Support Vector Machine are used to create classification models, which are then evaluated and compared to determine their effectiveness in detecting spam emails.

2. Why do we need to scale features, and what does `StandardScaler` do?

- **Answer**: Scaling is necessary because KNN and SVM are sensitive to feature magnitudes. Features with larger scales can dominate the distance metrics in KNN or the margin calculations in SVM. `StandardScaler` standardizes features by removing the mean and scaling to unit variance, ensuring all features contribute equally.

3. What are K-Nearest Neighbors and Support Vector Machine algorithms? How do they work?

- **Answer**:
  - **K-Nearest Neighbors (KNN)**: KNN is a lazy learning algorithm that classifies data based on the 'k' nearest neighbors in the feature space. It computes the distance between the test sample and each training sample, and the majority class among the closest `k` neighbors is assigned to the test sample.

- **Support Vector Machine (SVM)**: SVM is a supervised learning algorithm that finds a hyperplane to separate classes by maximizing the margin between data points of different classes. In this practical, a linear kernel is used to create a linear decision boundary between spam and non-spam emails.

4. Why do we use PCA, and how does it help in this practical?

- **Answer**: Principal Component Analysis (PCA) is used for dimensionality reduction. Here, it reduces the features to two principal components, making it easier to visualize data distribution and observe any separation between spam and non-spam emails. PCA also helps in reducing computational load and mitigating noise in data.

5. What does the `train_test_split` function do, and why is it important?

- **Answer**: The `train_test_split` function splits the data into training and testing sets. It is essential to evaluate model performance on unseen data (testing set), ensuring that the model generalizes well and doesn't just memorize the training data, preventing overfitting.

6. What are some advantages and disadvantages of KNN and SVM for spam classification?

- **Answer**:
  - **KNN**:
    - **Advantages**: Simple and intuitive; good when data distribution is relatively even.
    - **Disadvantages**: Computationally expensive for large datasets; sensitive to the choice of `k` and feature scaling.
  - **SVM**:
    - **Advantages**: Effective for high-dimensional data and with a clear margin of separation; works well when classes are linearly separable.
    - **Disadvantages**: Can be less effective if the classes are not linearly separable (though this is improved with kernels); can be memory-intensive.

7. How did you choose the value of `k` for KNN, and what impact does it have?

- **Answer**: The value of `k` was set to 5. The choice of `k` affects the model's performance—small values of `k` make the model sensitive to noise, while large values may smooth out distinctions between classes. In practice, `k` is usually chosen through cross-validation to find the optimal balance.

8. Why is SVM with a linear kernel used here instead of other kernels?

- **Answer**: The linear kernel is used here because it is computationally simpler and works well when there is a linear boundary between classes. For spam classification, a linear separation often performs adequately, and using more complex kernels (like RBF) may not significantly improve accuracy for this dataset.

9. What is a confusion matrix, and how do you interpret it?

- **Answer**: A confusion matrix displays the counts of actual vs. predicted classifications. It has four key values: True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN). By analyzing these values, we can assess a model's accuracy, precision, recall, and F1 score, offering deeper insights into its classification performance.

10. What are accuracy, precision, recall, and F1-score? Why are they important?

- **Answer**:
  - **Accuracy**: The proportion of correctly classified emails.
  - **Precision**: The ratio of true positives to all predicted positives, showing how often a spam prediction is correct.
  - **Recall**: The ratio of true positives to all actual positives, indicating how well the model identifies spam.
  - **F1-score**: The harmonic mean of precision and recall, providing a balanced measure of performance, especially important in cases of imbalanced datasets.
- These metrics are essential for evaluating classification models beyond accuracy, especially in tasks like spam detection where false positives and false negatives carry different consequences.

11. What role does `random_state` play in the train-test split and model training?

- **Answer**: Setting `random_state` ensures reproducibility by controlling the randomness in data splitting and model training. This way, the data splits and model results remain consistent across multiple runs, enabling reliable evaluation.

12. Why do we use classification metrics like `classification_report` instead of just accuracy?

- **Answer**: Accuracy alone does not provide complete information, especially in imbalanced datasets where one class is more prevalent. `classification_report` includes precision, recall, and F1-score, giving a better understanding of how well the model performs on each class, which is crucial in spam detection to minimize false positives (non-spam predicted as spam) and false negatives (spam predicted as non-spam).

13. Explain how you visualized the model's performance.

- **Answer**: We visualized the performance using a confusion matrix heatmap for each model, showing actual vs. predicted classifications for spam and non-spam. Additionally, we used a bar chart to compare the accuracy of KNN and SVM, giving a clear visual representation of each model's effectiveness.

14. How would you improve the models if given more time or data?

- **Answer**: To improve the models, we could:
  - **Tune Hyperparameters**: Use grid search or cross-validation to optimize `k` in KNN and test different SVM kernels.
  - **Feature Engineering**: Add or modify features, such as analyzing email content (e.g., frequency of specific spam-indicative words).
  - **Use Ensemble Models**: Combine predictions from multiple models to increase accuracy.
  - **Balance the Dataset**: If there's an imbalance, consider techniques like SMOTE to balance the spam and non-spam classes.

These questions cover fundamental concepts, technical choices, and evaluation strategies for spam classification, preparing you well for the viva.

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).
Given above is the Problem statement for the lab practical in th subject of ML. attached the Code for the implementation Explain the Code Line By line in detail.

```
# Importing 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.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping

# Step 1: Load the dataset
data = pd.read_csv('/path_to_data/Churn_Modelling.csv')
print("Dataset Shape:", data.shape)
data.head()

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

# Encoding categorical features
le_geography = LabelEncoder()
X['Geography'] = le_geography.fit_transform(X['Geography'])

le_gender = LabelEncoder()
X['Gender'] = le_gender.fit_transform(X['Gender'])

# Step 3: Split 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)

# Step 4: Normalize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
# Step 5: Initialize and build the neural network model
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(X_train.shape[1],)))
model.add(Dropout(0.3))  # Adding dropout to prevent overfitting
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(1, activation='sigmoid'))  # Output layer with sigmoid for binary classification

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

# Model Summary
model.summary()

# Early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Train the model
history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_split=0.2, callbacks=
[early_stopping])

# Step 6: Model Evaluation
# Plot training history
plt.figure(figsize=(14, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

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

# Make predictions
y_pred = (model.predict(X_test) > 0.5).astype("int32")

# Calculate accuracy score
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy Score:", accuracy)

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
```

```
# Plot Confusion Matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=['Not Churn', 'Churn'],
yticklabels=['Not Churn', 'Churn'])
plt.title("Confusion Matrix")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()

# Classification Report
print("Classification Report:\n", classification_report(y_test, y_pred))

# Make predictions
y_pred = (model.predict(X_test) > 0.5).astype("int32")

# Convert predictions back to the original DataFrame's index
# First, create a DataFrame from y_test with matching indices
y_test_df = y_test.reset_index(drop=True)

# Create a DataFrame for predictions
predictions_df = pd.DataFrame(y_pred, columns=["Predicted"], index=y_test_df.index)

# Combine the predictions with the original test set DataFrame (re-indexed)
results_df = X_test.copy()
results_df = pd.DataFrame(scaler.inverse_transform(results_df), columns=X.columns)
results_df['True Label'] = y_test_df
results_df['Predicted Label'] = predictions_df

# Display the updated DataFrame with the true and predicted labels
results_df.head()
```

Let's break down this code line-by-line to understand how it builds and evaluates a neural network classifier for predicting customer churn.

1. Importing Necessary Libraries

```python
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.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
```

- These libraries provide tools for data handling (`pandas`, `numpy`), visualization (`matplotlib`, `seaborn`), data preprocessing (`train_test_split`, `StandardScaler`, `LabelEncoder`), and neural network building and training (`tensorflow.keras`).

2. Load the Dataset

```
python


data = pd.read_csv('/path_to_data/Churn_Modelling.csv')
print("Dataset Shape:", data.shape)
data.head()
```

- Loads the dataset into a DataFrame `data` and prints its shape and the first few rows for an overview.

3. Distinguish Feature and Target Sets

- **Drop Irrelevant Columns**:

```
python


X = data.drop(columns=['RowNumber', 'CustomerId', 'Surname', 'Exited'])
y = data['Exited']
```

- **Encode Categorical Features**:

```
python


le_geography = LabelEncoder()
X['Geography'] = le_geography.fit_transform(X['Geography'])

le_gender = LabelEncoder()
X['Gender'] = le_gender.fit_transform(X['Gender'])
```

- Drops unnecessary columns (`RowNumber`, `CustomerId`, `Surname`), sets `Exited` as the target variable (`y`), and encodes `Geography` and `Gender` into numerical values using `LabelEncoder`.

4. Split Data into Training and Testing Sets

```
python


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

- Splits the data into 80% training and 20% testing sets. Setting `random_state=42` ensures reproducibility of the split.

5. Normalize the Data

```
python


  scaler = StandardScaler()
  X_train = scaler.fit_transform(X_train)
  X_test = scaler.transform(X_test)
```

- Uses `**StandardScaler**` to normalize features so that they all have a mean of 0 and a standard deviation of 1. This helps the neural network converge faster and improve performance.

6. Initialize and Build the Neural Network Model

- **Model Architecture**:

```
python


  model = Sequential()
  model.add(Dense(64, activation='relu', input_shape=(X_train.shape[1],)))
  model.add(Dropout(0.3))
  model.add(Dense(32, activation='relu'))
  model.add(Dropout(0.3))
  model.add(Dense(1, activation='sigmoid'))
```

- **Compile the Model**:

```
python


  model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

- **Model Summary**:

```
python


  model.summary()
```

- Creates a neural network with:
  - **Input Layer**: `**Dense(64)**` with 64 neurons and ReLU activation.
  - **Dropout Layer**: `**Dropout(0.3)**` to prevent overfitting by randomly disabling 30% of neurons.
  - **Hidden Layer**: `**Dense(32)**` with 32 neurons and ReLU activation.
  - **Dropout Layer**: `**Dropout(0.3)**`.
  - **Output Layer**: `**Dense(1)**` with a sigmoid activation for binary classification, outputting probabilities for "Churn" (1) or "Not Churn" (0).
- Compiles the model with `**adam**` optimizer, `**binary_crossentropy**` loss (suitable for binary classification), and `**accuracy**` as the evaluation metric.
- `**model.summary()**` provides an overview of the model architecture.

7. Early Stopping Callback

```
python

early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
```

- Sets up early stopping to monitor validation loss. Training stops if validation loss doesn't improve for 5 epochs, and the model reverts to the best weights.

8. Train the Model

```
python

history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_split=0.2, call
backs=[early_stopping])
```

- Trains the model for up to 50 epochs with a batch size of 32, using 20% of the training data for validation. The `**history**` object records training and validation metrics across epochs.

9. Plot Training History

- **Accuracy and Loss Plots**:

```
python

plt.figure(figsize=(14, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

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

- Plots training and validation accuracy and loss over epochs to visualize the model's performance and assess if it's overfitting or underfitting.

10. Make Predictions

```
python

y_pred = (model.predict(X_test) > 0.5).astype("int32")
```

- Generates predictions on the test set, where predictions greater than 0.5 are classified as 1 (Churn), and the rest as 0 (Not Churn).

11. Calculate Accuracy and Confusion Matrix

- **Accuracy Score**:

```
python


  accuracy = accuracy_score(y_test, y_pred)
  print("Accuracy Score:", accuracy)
```

- **Confusion Matrix**:

```
python


  cm = confusion_matrix(y_test, y_pred)
```

- Prints the accuracy score and calculates the confusion matrix, which shows counts of true positives, false positives, true negatives, and false negatives.

12. Plot Confusion Matrix

```
python


  plt.figure(figsize=(8, 6))
  sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=['Not Churn', 'Churn'], yti
  cklabels=['Not Churn', 'Churn'])
  plt.title("Confusion Matrix")
  plt.xlabel("Predicted Labels")
  plt.ylabel("True Labels")
  plt.show()
```

- Visualizes the confusion matrix with labels for better interpretability of true vs. predicted classifications.

13. Classification Report

```
python


  print("Classification Report:\n", classification_report(y_test, y_pred))
```

- Prints a detailed classification report showing precision, recall, F1 score, and support for each class.

14. Display Predictions in Original DataFrame Format

- **Combine Predictions with Original DataFrame**:

```
python


  y_test_df = y_test.reset_index(drop=True)
  predictions_df = pd.DataFrame(y_pred, columns=["Predicted"], index=y_test_df.index)
  results_df = pd.DataFrame(scaler.inverse_transform(X_test), columns=X.columns)
  results_df['True Label'] = y_test_df
  results_df['Predicted Label'] = predictions_df
  results_df.head()
```

- Creates a final DataFrame that includes the original feature values, true labels, and predicted labels, allowing an overview of model predictions alongside true outcomes.

This code effectively implements a neural network-based classifier for predicting bank customer churn, including data preprocessing, model training, evaluation, and visualization.

Now Predict The Questions which and provide the appropriate answers which are most probably going to asked on this practical in Viva.

Here are some questions that may be asked during a viva based on this practical for customer churn prediction using a neural network:

1. What is the purpose of this practical?

- **Answer**: The purpose of this practical is to build a neural network-based binary classifier to predict if a bank customer will leave (churn) or stay in the next 6 months. Using historical customer data, the model learns patterns that indicate churn, helping the bank make proactive decisions to retain customers.

2. Why is customer churn prediction important for banks?

- **Answer**: Churn prediction helps banks identify customers who are likely to leave, allowing them to intervene with retention strategies. Retaining existing customers is often more cost-effective than acquiring new ones, so accurate churn prediction can improve customer loyalty and reduce financial losses.

3. What are the features and target variables used in this practical?

- **Answer**: The features include attributes like `**CreditScore**`, `**Geography**`, `**Gender**`, `**Age**`, `**Tenure**`, `**Balance**`, and other financial indicators. The target variable is `**Exited**`, where 1 indicates that a customer churned and 0 means they stayed.

4. Why do we drop columns like `**RowNumber**`, `**CustomerId**`, and `**Surname**`?

- **Answer**: These columns do not provide useful information for predicting churn. `**CustomerId**` and `**Surname**` are unique identifiers, which don't contribute to the predictive power of the model, and `**RowNumber**` is just an index. Including them could add noise to the model, potentially decreasing accuracy.

5. Explain the role of `LabelEncoder` in this practical.

- **Answer**: `LabelEncoder` converts categorical features, such as `Geography` and `Gender`, into numerical values. This is necessary because neural networks work with numerical data, not categorical, and encoding helps the model interpret these features properly.

6. Why do we normalize the data, and what is the purpose of `StandardScaler`?

- **Answer**: Normalizing the data ensures that all features have the same scale, which helps the neural network converge faster and avoid bias due to features with large values. `StandardScaler` standardizes each feature by removing the mean and scaling to unit variance, making them comparable.

7. Describe the architecture of the neural network model you used.

- **Answer**: The neural network has:
  - An input layer with 64 neurons and ReLU activation.
  - A dropout layer with a 30% dropout rate to prevent overfitting.
  - A hidden layer with 32 neurons and ReLU activation, followed by another dropout layer.
  - An output layer with 1 neuron and sigmoid activation, producing a probability between 0 and 1 for churn classification.

8. Why is ReLU used as the activation function in hidden layers?

- **Answer**: ReLU (Rectified Linear Unit) is used because it helps mitigate the vanishing gradient problem, allowing the model to learn faster and better. ReLU activation outputs zero for negative values, which adds sparsity, and it is computationally efficient.

9. Why is the output layer activation set to `sigmoid`?

- **Answer**: The sigmoid activation function outputs values between 0 and 1, representing probabilities. This is suitable for binary classification, where values closer to 1 indicate churn (1) and values closer to 0 indicate non-churn (0).

10. What is the purpose of dropout layers, and why set dropout to 0.3?

- **Answer**: Dropout layers help prevent overfitting by randomly setting a fraction (30% in this case) of neurons to zero during training. This encourages the network to learn robust features rather than relying on specific neurons, improving generalization to unseen data.

11. What loss function and optimizer are used in this model? Why?

- **Answer**: The loss function is `binary_crossentropy`, which is suitable for binary classification problems. The optimizer is `adam`, an adaptive optimizer that combines the benefits of `momentum` and `RMSProp`, resulting in faster convergence.

12. Explain early stopping and its role in training the model.

- **Answer**: Early stopping monitors the validation loss and stops training if it does not improve for a set number of epochs (5 in this case). This prevents overfitting by halting training when the model starts to learn noise instead of useful patterns, thus improving generalization.

13. What is a confusion matrix, and what does it tell us about the model's performance?

- **Answer**: A confusion matrix is a table that displays the counts of true positive, false positive, true negative, and false negative predictions. It helps evaluate the model's classification performance, giving insight into its accuracy, precision, recall, and areas for improvement.

14. Why do we plot training and validation accuracy and loss?

- **Answer**: Plotting training and validation accuracy and loss helps visualize the learning process. It allows us to monitor if the model is overfitting (when validation loss diverges from training loss) or underfitting (low accuracy), guiding decisions to fine-tune the model architecture or training process.

15. What is the significance of accuracy, precision, recall, and F1 score in classification?

- **Answer**:
  - **Accuracy**: The proportion of correct predictions over all predictions.
  - **Precision**: The ratio of true positives to the total predicted positives, indicating how many predicted churns are correct.
  - **Recall**: The ratio of true positives to all actual positives, showing how well the model identifies actual churns.
  - **F1 Score**: The harmonic mean of precision and recall, balancing both metrics. It is especially useful when dealing with imbalanced datasets.

16. How would you improve the model if you were to revisit this practical?

- **Answer**: Possible improvements include:
  - **Hyperparameter tuning**: Adjust the number of layers, neurons, dropout rate, and learning rate.
  - **Data augmentation**: Address class imbalance with techniques like SMOTE if churn cases are less common.
  - **Feature engineering**: Create new features or select the most informative ones.
  - **Try other architectures**: Experiment with advanced neural networks, like convolutional or recurrent layers, or ensemble learning for better performance.

17. Why did you choose a batch size of 32 and 50 epochs for training?

- **Answer**: Batch size and epochs are chosen based on trial and error. A batch size of 32 provides a balance between memory efficiency and convergence speed. Training for up to 50 epochs ensures enough learning time, and early stopping is used to halt training once no further improvements occur in validation loss.

18. Why is `accuracy_score` used to evaluate the model's predictions?

- **Answer**: `accuracy_score` provides a simple measure of how many predictions the model got right. Since this is a balanced binary classification task, accuracy is a straightforward and informative metric. However, in cases of class imbalance, precision, recall, and F1 score would be more critical.

19. How does the model handle new data once it is trained?

- **Answer**: The model can be used to predict churn on new data by scaling the new data points with the `StandardScaler` (fitted on the training data) and feeding them into the model. The output probabilities are converted into binary classes to indicate churn or no churn.

20. How would you implement this model in production to assist the bank?

- **Answer**: In production, the model would be deployed as part of a web service or embedded within the bank's CRM software. It would periodically take new customer data, preprocess it, and provide real-time predictions, allowing the bank to monitor and act on high-churn-risk customers effectively.

These questions cover the critical technical, conceptual, and practical aspects of this neural network-based classification task for customer churn prediction, helping you prepare thoroughly for the viva.

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.

Given above is the Problem statement for the lab practical in th subject of ML. attached the Code for the implementation Explain the Code Line By line in detail.

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt

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

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

# Gradient Descent parameters
learning_rate = 0.1  # Step size
n_iterations = 50    # Number of iterations
x_start = 2          # Starting point

# Lists to store x values and corresponding y values for visualization
x_values = [x_start]
y_values = [function(x_start)]

# Gradient Descent Loop
x = x_start
for i in range(n_iterations):
    gradient = derivative(x)          # Compute the gradient at the current point
    x = x - learning_rate * gradient    # Update x value
    y = function(x)                   # Compute y for the updated x
```

```
    # Append updated values for visualization
    x_values.append(x)
    y_values.append(y)

# Plot the function and the path of gradient descent
x_range = np.linspace(-10, 4, 100)
y_range = function(x_range)

plt.figure(figsize=(10, 6))
plt.plot(x_range, y_range, label="y = (x + 3)^2", color="blue")
plt.scatter(x_values, y_values, color="red", label="Gradient Descent Path")
plt.plot(x_values, y_values, color="red", linestyle="--")
plt.title("Gradient Descent to Find Local Minima of y = (x + 3)^2")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()

# Show the plot
plt.show()

# Print final results
print(f"Local minimum occurs at x = {x_values[-1]:.4f}, y = {y_values[-1]:.4f}")
```

Let's go through this code line by line to understand how it implements the gradient descent algorithm to find the local minimum of the function $y = (x + 3)^2$.

1. Import Necessary Libraries

```
python


  import numpy as np
  import matplotlib.pyplot as plt
```

- `**numpy**` is used for numerical computations, and `**matplotlib.pyplot**` is used for plotting the function and the path taken by gradient descent.

2. Define the Function and Its Derivative

- **Define the function**:

```
python


  def function(x):
      return (x + 3) ** 2
```

- This function computes $y = (x + 3)^2$. It takes an input `x` and returns the value of the function at `x`.

- **Define the derivative of the function**:

```python
def derivative(x):
    return 2 * (x + 3)
```

  - This function returns the derivative of $y = (x + 3)^2$, which is $2(x + 3)$. The derivative indicates the slope of the function at any given point `x`, helping determine the direction of descent.

3. Set Gradient Descent Parameters

```python
learning_rate = 0.1  # Step size
n_iterations = 50    # Number of iterations
x_start = 2          # Starting point
```

- **learning_rate**: Determines the step size taken in the direction of the gradient at each iteration.
- **n_iterations**: Specifies the number of gradient descent steps to be taken.
- **x_start**: The initial point from which the gradient descent process begins.

4. Initialize Lists for Visualization

```python
x_values = [x_start]
y_values = [function(x_start)]
```

- **x_values** and **y_values**: Lists to store the `x` values and corresponding `y` values at each step of the gradient descent. These are used to visualize the descent path.

5. Gradient Descent Loop

```python
x = x_start
for i in range(n_iterations):
    gradient = derivative(x)           # Compute the gradient at the current point
    x = x - learning_rate * gradient    # Update x value
    y = function(x)                     # Compute y for the updated x

    # Append updated values for visualization
    x_values.append(x)
    y_values.append(y)
```

- This loop performs the core gradient descent algorithm:
  1. **Compute the gradient** at the current `x` value by calling `derivative(x)`.
  2. **Update** `x` by moving in the direction opposite to the gradient (i.e., `x - learning_rate * gradient`).
  3. **Compute the new `y` value** for the updated `x` and store both `x` and `y` values in their respective lists for visualization.

6. Prepare Data for Plotting the Function and Descent Path

```
python


  x_range = np.linspace(-10, 4, 100)
  y_range = function(x_range)
```

- **x_range**: Creates an array of `x` values ranging from -10 to 4, which is used to plot the function across a broader range.
- **y_range**: Computes `y` values for each point in `x_range`, representing the function $y = (x + 3)^2$.

7. Plot the Function and Gradient Descent Path

```
python


  plt.figure(figsize=(10, 6))
  plt.plot(x_range, y_range, label="y = (x + 3)^2", color="blue")
  plt.scatter(x_values, y_values, color="red", label="Gradient Descent Path")
  plt.plot(x_values, y_values, color="red", linestyle="--")
  plt.title("Gradient Descent to Find Local Minima of y = (x + 3)^2")
  plt.xlabel("x")
  plt.ylabel("y")
  plt.legend()
  plt.show()
```

- **plt.plot(x_range, y_range)**: Plots the function $y = (x + 3)^2$ as a blue curve.
- **plt.scatter(x_values, y_values)** and **plt.plot(x_values, y_values)**: Plots the points and path of gradient descent as red markers and a dashed line, showing each step from the starting point to the local minimum.
- **plt.show()**: Displays the plot.

8. Print Final Results

```
python


  print(f"Local minimum occurs at x = {x_values[-1]:.4f}, y = {y_values[-1]:.4f}")
```

- This prints the final values of `x` and `y`, indicating the approximate location of the local minimum after completing the gradient descent steps.

Explanation of Results

The gradient descent process iteratively moves the `x` value closer to the point where the function $y = (x + 3)^2$ reaches its minimum (which occurs at $x = -3$, where $y = 0$). Starting from $x = 2$, the algorithm updates `x` by stepping in the opposite direction of the gradient, ultimately converging towards this minimum point.

Now Predict The Questions which and provide the appropriate answers which are most probably going to asked on this practical in Viva.

Here are some likely viva questions for this gradient descent practical, along with suitable answers:

1. What is Gradient Descent?

- **Answer**: Gradient Descent is an optimization algorithm used to minimize a function by iteratively moving towards the minimum. It does so by adjusting parameters (like `x` here) in the opposite direction of the gradient, which represents the slope or rate of change of the function.

2. Explain the purpose of Gradient Descent in this practical.

- **Answer**: The purpose is to find the local minimum of the function $y = (x + 3)^2$. Starting from an initial point $x = 2$, the algorithm iteratively updates `x` to minimize the function, converging towards the point where `y` is lowest.

3. What is the role of the learning rate?

- **Answer**: The learning rate determines the size of each step taken in the direction of the gradient. A small learning rate results in smaller steps and slower convergence, while a large learning rate may lead to overshooting the minimum or even divergence.

4. What happens if the learning rate is too high or too low?

- **Answer**:
  - **Too High**: The algorithm may overshoot the minimum, bouncing back and forth without convergence, or diverge entirely.
  - **Too Low**: The algorithm will take very small steps, resulting in slow convergence and potentially getting stuck in local minima.

5. Why do we need the derivative of the function?

- **Answer**: The derivative gives the slope of the function at any point. In gradient descent, this slope (gradient) indicates the direction and rate of change, guiding the algorithm to move `x` in the direction that decreases the function value most effectively.

6. Explain the function $y = (x + 3)^2$ and its derivative.

- **Answer**: The function $y = (x + 3)^2$ is a simple quadratic equation with a minimum at $x = -3$, where $y = 0$. The derivative $\frac{dy}{dx} = 2(x + 3)$ represents the slope of the function at any point. Gradient descent uses this slope to iteratively move `x` towards the minimum.

7. How does the gradient descent algorithm update the value of `x` in each iteration?

- **Answer**: In each iteration, the algorithm calculates the gradient (slope) at the current `x` and updates `x` by moving it in the opposite direction of the gradient. This is done with the formula: $x = x - \text{learning\_rate} \times \text{gradient}$.

8. What are the initial conditions of this algorithm, and why do they matter?

- **Answer**: The initial conditions include the starting point $x = 2$, the learning rate, and the number of iterations. These conditions are crucial because they influence the convergence rate and the ability to reach the minimum. A poor starting point or inappropriate learning rate can prevent the algorithm from finding the minimum.

9. How does the algorithm determine when to stop?

- **Answer**: In this implementation, the algorithm stops after a fixed number of iterations (`n_iterations = 50`). In more advanced versions, stopping criteria may include reaching a sufficiently small gradient (close to zero) or detecting minimal changes in `x` or `y` over iterations.

10. What is the significance of the final `x` and `y` values printed at the end?

- **Answer**: The final values of `x` and `y` represent the approximate location of the local minimum found by the gradient descent algorithm. In this case, they should be close to $x = -3$ and $y = 0$, the actual minimum of the function.

11. Why do we store `x_values` and `y_values` for visualization?

- **Answer**: Storing these values allows us to plot the path taken by gradient descent, visually demonstrating how `x` approaches the local minimum over iterations. This helps in understanding how gradient descent moves step-by-step towards minimizing the function.

12. What is the impact of the number of iterations on the result?

- **Answer**: The number of iterations determines how many steps the algorithm takes toward the minimum. Too few iterations may result in premature stopping, not reaching the minimum. A sufficient number of iterations ensures that the algorithm gets close to the minimum.

13. What would happen if we started from a different initial point for `x`?

- **Answer**: Since $y = (x + 3)^2$ is a convex function with a single minimum, starting from a different point would still lead to the same minimum ($x = -3$), assuming the learning rate is appropriate. The number of iterations required to reach the minimum might vary depending on the starting point.

14. Explain the significance of the plot generated by the code.

- **Answer**: The plot shows the function $y = (x + 3)^2$ and the path of `x` values computed by gradient descent. The red points and dashed line illustrate the step-by-step descent toward the minimum, providing a visual representation of the algorithm's progression.

15. Can this method be used to find the global minimum of any function?

- **Answer**: Gradient descent can find the local minimum of convex functions, which have only one minimum (global minimum). However, for non-convex functions with multiple minima, gradient descent may get stuck in a local minimum and may not find the global minimum without modifications, such as using random restarts or advanced techniques like stochastic gradient descent.

16. What modifications could improve the convergence of gradient descent?

- **Answer**: Convergence can be improved by:
  - **Adaptive Learning Rates**: Using techniques like learning rate decay, momentum, or adaptive optimizers (e.g., Adam).
  - **Dynamic Stopping Criteria**: Stopping when the gradient is close to zero or changes are minimal, instead of using a fixed number of iterations.
  - **Batch Methods**: In larger datasets, using stochastic or mini-batch gradient descent speeds up convergence.

These questions cover the essential theoretical and practical aspects of the gradient descent algorithm, helping you be well-prepared for the viva.

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
Given above is the Problem statement for the lab practical in th subject of ML. attached the Code for the implementation Explain the Code Line By line in detail.

```
# Importing 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.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score
import seaborn as sns
import matplotlib.pyplot as plt

# Step 1: Load the dataset
url = 'https://www.kaggle.com/datasets/abdallamahgoub/diabetes/download'  # Change to actual file path if locally stored
df = pd.read_csv(url)

# Step 2: Data Preprocessing
# Check for any null values
print("Null values:\n", df.isnull().sum())
```

```
# Define feature set (X) and target variable (y)
X = df.drop('Outcome', axis=1)
y = df['Outcome']

# Step 3: Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Step 4: Data Normalization (KNN benefits from scaling)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 5: Implement K-Nearest Neighbors
k = 5  # Number of neighbors
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)

# Step 6: Make Predictions
y_pred = knn.predict(X_test)

# Step 7: Compute Performance Metrics
# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Accuracy
accuracy = accuracy_score(y_test, y_pred)

# Error Rate
error_rate = 1 - accuracy

# Precision
precision = precision_score(y_test, y_pred)

# Recall
recall = recall_score(y_test, y_pred)

# Print results
print("Confusion Matrix:\n", conf_matrix)
print(f"Accuracy: {accuracy:.4f}")
print(f"Error Rate: {error_rate:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")

# Step 8: Visualizations
# Confusion Matrix Heatmap
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=["No Diabetes",
"Diabetes"], yticklabels=["No Diabetes", "Diabetes"])
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
```

```
plt.title("Confusion Matrix")
plt.show()

# Performance Metrics Bar Plot
metrics = ["Accuracy", "Error Rate", "Precision", "Recall"]
values = [accuracy, error_rate, precision, recall]

plt.figure(figsize=(8, 5))
sns.barplot(x=metrics, y=values, palette="viridis")
plt.ylim(0, 1)
plt.title("KNN Performance Metrics")
plt.ylabel("Score")
plt.show()
```

Here's a detailed line-by-line explanation of the code for implementing the K-Nearest Neighbors (KNN) algorithm on the diabetes dataset:

1. Import Necessary Libraries

```
python


  import pandas as pd
  import numpy as np
  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_scor
  e
  import seaborn as sns
  import matplotlib.pyplot as plt
```

- `pandas` and `numpy`: For data handling and numerical operations.
- `train_test_split`: To split data into training and testing sets.
- `StandardScaler`: For normalizing data.
- `KNeighborsClassifier`: The KNN algorithm for classification.
- `confusion_matrix`, `accuracy_score`, `precision_score`, `recall_score`: To evaluate model performance.
- `seaborn` and `matplotlib.pyplot`: For visualizing the confusion matrix and performance metrics.

2. Load the Dataset

```
python


  url = 'https://www.kaggle.com/datasets/abdallamahgoub/diabetes/download'  # Change to actu
  al file path if locally stored
  df = pd.read_csv(url)
```

- **Load data**: Reads the diabetes dataset into a DataFrame `df`. Replace `url` with the actual file path if the dataset is stored locally.

3. Data Preprocessing

- **Check for Null Values**:

```
python

print("Null values:\n", df.isnull().sum())
```

  - Checks for missing values in the dataset. In the diabetes dataset, there should be no null values.

- **Define Feature Set and Target Variable**:

```
python

X = df.drop('Outcome', axis=1)
y = df['Outcome']
```

  - **X**: All columns except `Outcome`, which are the features.
  - **y**: The `Outcome` column, which indicates whether a person has diabetes (1) or not (0).

4. Split the Dataset into Training and Testing Sets

```
python

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

- Splits the dataset into 70% training and 30% testing sets. Setting `random_state=42` ensures reproducibility of the split.

5. Data Normalization

```
python

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

- KNN is sensitive to feature scales, so normalizing the features ensures they are on a similar scale. `StandardScaler` standardizes each feature by removing the mean and scaling to unit variance, resulting in better KNN performance.

6. Implement K-Nearest Neighbors

```
python


  k = 5  # Number of neighbors
  knn = KNeighborsClassifier(n_neighbors=k)
  knn.fit(X_train, y_train)
```

- **k**: Specifies the number of neighbors to consider. Here, `k=5`, meaning the algorithm will consider the 5 closest neighbors to classify a new data point.
- `knn.fit(X_train, y_train)`: Trains the KNN model on the training data.

7. Make Predictions

```
python


  y_pred = knn.predict(X_test)
```

- **y_pred**: Predicted labels for the test data based on the trained KNN model.

8. Compute Performance Metrics

- **Confusion Matrix**:

  ```
  python


    conf_matrix = confusion_matrix(y_test, y_pred)
  ```

  - Calculates the confusion matrix, which shows counts of true positives, false positives, true negatives, and false negatives.

- **Accuracy**:

  ```
  python


    accuracy = accuracy_score(y_test, y_pred)
  ```

  - **Accuracy**: The proportion of correct predictions over all predictions.

- **Error Rate**:

  ```
  python


    error_rate = 1 - accuracy
  ```

  - **Error Rate**: The proportion of incorrect predictions, calculated as `1 - accuracy`.

- **Precision**:

```
python

  precision = precision_score(y_test, y_pred)
```

  - **Precision**: The ratio of true positives to all predicted positives, showing the accuracy of positive predictions (i.e., diabetes).

- **Recall**:

```
python

  recall = recall_score(y_test, y_pred)
```

  - **Recall**: The ratio of true positives to all actual positives, indicating the model's ability to identify all positive cases (diabetes).

- **Print Results**:

```
python

  print("Confusion Matrix:\n", conf_matrix)
  print(f"Accuracy: {accuracy:.4f}")
  print(f"Error Rate: {error_rate:.4f}")
  print(f"Precision: {precision:.4f}")
  print(f"Recall: {recall:.4f}")
```

  - Displays the computed confusion matrix and other performance metrics (accuracy, error rate, precision, and recall).

9. Visualizations

- **Confusion Matrix Heatmap**:

```
python

  plt.figure(figsize=(6, 4))
  sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=["No Diabete
  s", "Diabetes"], yticklabels=["No Diabetes", "Diabetes"])
  plt.xlabel("Predicted Labels")
  plt.ylabel("True Labels")
  plt.title("Confusion Matrix")
  plt.show()
```

  - Plots the confusion matrix as a heatmap for easier visualization, with labels to indicate true and predicted classes for “No Diabetes” and “Diabetes.”

- **Performance Metrics Bar Plot**:

```
python


  metrics = ["Accuracy", "Error Rate", "Precision", "Recall"]
  values = [accuracy, error_rate, precision, recall]

  plt.figure(figsize=(8, 5))
  sns.barplot(x=metrics, y=values, palette="viridis")
  plt.ylim(0, 1)
  plt.title("KNN Performance Metrics")
  plt.ylabel("Score")
  plt.show()
```

- Creates a bar plot to compare different performance metrics (Accuracy, Error Rate, Precision, and Recall) on a scale from 0 to 1. This visualization helps in understanding the model's performance across multiple evaluation metrics.

This code provides a complete implementation of the KNN algorithm for classifying diabetes cases, from data loading and preprocessing to model training, evaluation, and visualization of performance metrics.

Now Predict The Questions which and provide the appropriate answers which are most probably going to asked on this practical in Viva.

Here are some questions that are likely to be asked in a viva based on this practical of implementing K-Nearest Neighbors (KNN) for diabetes classification, along with suitable answers:

1. What is the purpose of this practical?

- **Answer**: The purpose of this practical is to classify individuals as having diabetes or not using the K-Nearest Neighbors (KNN) algorithm. We evaluate the model's performance using metrics like confusion matrix, accuracy, error rate, precision, and recall to understand how well it can predict diabetes.

2. What is K-Nearest Neighbors (KNN), and how does it work?

- **Answer**: K-Nearest Neighbors (KNN) is a supervised classification algorithm that predicts the class of a data point based on the majority class among its `k` nearest neighbors. It calculates the distance between data points and assigns the label that is most common among the closest neighbors.

3. Why did you choose `k = 5` for this implementation?

- **Answer**: `k = 5` is a common choice for KNN as it balances simplicity and accuracy. If `k` is too small, the model can be sensitive to noise, leading to overfitting. Choosing a slightly larger value, like 5, provides a better generalization. Ideally, `k` should be chosen through cross-validation.

4. What is the significance of the `StandardScaler` in this practical?

- **Answer**: `StandardScaler` is used to normalize the feature set by scaling all features to have a mean of 0 and a standard deviation of 1. This is important for KNN because it relies on distance calculations; without scaling, features with larger values could dominate the distance calculation, affecting classification accuracy.

5. Explain the process of splitting the dataset into training and testing sets. Why is it necessary?

- **Answer**: Splitting the dataset allows us to train the model on one portion (training set) and evaluate its performance on an unseen portion (testing set). This simulates how the model would perform on new data, helping to assess its generalization ability and prevent overfitting.

6. What does the confusion matrix tell us?

- **Answer**: The confusion matrix shows the counts of true positives, false positives, true negatives, and false negatives. It provides insight into the model's performance, allowing us to calculate key metrics like accuracy, precision, recall, and error rate, giving a clear picture of where the model makes correct and incorrect predictions.

7. What is accuracy, and how is it calculated in this context?

- **Answer**: Accuracy is the proportion of correct predictions (both positive and negative) out of the total predictions. In this practical, accuracy is calculated as the ratio of true positives and true negatives to the total predictions.

8. Define and explain the error rate.

- **Answer**: Error rate is the proportion of incorrect predictions out of the total predictions. It is calculated as `1 - accuracy`. A lower error rate indicates that the model makes fewer incorrect predictions.

9. What do precision and recall indicate, and why are they important in this context?

- **Answer**:
  - **Precision**: The ratio of true positives to all predicted positives. In this case, it shows how many of the patients predicted to have diabetes actually have diabetes.
  - **Recall**: The ratio of true positives to all actual positives. It indicates how well the model identifies all cases of diabetes.
- Both are crucial in medical diagnosis. High precision ensures that most people predicted to have diabetes actually do, while high recall ensures that most people with diabetes are correctly identified.

10. What are the potential drawbacks of using KNN for this dataset?

- **Answer**: KNN can be computationally intensive for large datasets because it calculates distances to all training samples. It also struggles with imbalanced datasets and may be affected by irrelevant features if they're not properly scaled. Additionally, it's sensitive to the choice of `k` and feature scaling.

11. What does it mean if the recall is high but the precision is low?

- **Answer**: High recall and low precision indicate that the model is good at identifying all positive cases (high recall) but may also incorrectly classify many negative cases as positive, leading to a high number of false positives. In a medical context, this could mean many healthy patients are misclassified as having diabetes.

12. Why is it important to visualize the confusion matrix?

- **Answer**: Visualizing the confusion matrix helps quickly identify how well the model is performing on each class. It provides a clear breakdown of true positives, false positives, true negatives, and false negatives, making it easier to interpret the model's strengths and weaknesses.

13. How would you improve this model if given more time?

- **Answer**:
  - **Optimize `k`**: Perform cross-validation to find the optimal value of `k`.
  - **Feature Engineering**: Analyze feature importance and eliminate irrelevant features to reduce noise.
  - **Use Weighted KNN**: Assign weights to neighbors based on distance, giving closer points more influence.
  - **Try Other Classifiers**: Experiment with models like Logistic Regression, SVM, or Random Forest, which may perform better on this dataset.

14. How would you handle an imbalanced dataset with KNN?

- **Answer**: For imbalanced datasets, we could try:
  - **Weighted KNN**: Assign higher weights to the minority class.
  - **Resampling Techniques**: Use oversampling (e.g., SMOTE) or undersampling to balance classes.
  - **Adjusting Thresholds**: Modify decision thresholds to favor the minority class, improving recall.

15. Explain the distance metric used in KNN and its importance.

- **Answer**: The default distance metric in KNN is Euclidean distance, which calculates the straight-line distance between points. It is crucial because KNN relies on distance to identify neighbors, and the choice of metric can significantly affect performance. Other metrics, like Manhattan or Minkowski distance, may be used depending on the data structure.

16. What is overfitting in KNN, and how can you avoid it?

- **Answer**: Overfitting in KNN occurs when `k` is too small, causing the model to be overly sensitive to noise in the training data. This results in poor generalization. Using a slightly larger `k` (e.g., 5 or 7) helps avoid overfitting, making the model more robust.

17. What are the main advantages and disadvantages of KNN?

- **Answer**:
  - **Advantages**: Simple, intuitive, non-parametric, and effective for smaller datasets.
  - **Disadvantages**: Computationally expensive for large datasets, sensitive to irrelevant features, and performance depends heavily on feature scaling and the choice of `k`.

18. How does KNN handle multiclass classification?

- **Answer**: KNN can handle multiclass classification by assigning the majority class among the nearest neighbors as the predicted class. For example, if the `k` nearest neighbors include multiple classes, the class with the most neighbors is chosen.

19. Why is KNN considered a "lazy" learning algorithm?

- **Answer**: KNN is considered lazy because it doesn't learn a model or decision boundary from the training data. Instead, it stores the training data and performs classification only when a query (test) point is provided. This makes it computationally expensive at prediction time rather than at training time.

20. How would you deploy this model in a real-world scenario?

- **Answer**: In a real-world scenario, the model could be deployed as part of a healthcare application to assist doctors in diagnosing diabetes. It would take patient data as input and provide predictions on diabetes, allowing doctors to prioritize high-risk patients for further testing or monitoring.

These questions cover the key concepts, implementation choices, and evaluation strategies of the KNN algorithm for diabetes classification, helping you be well-prepared for the viva.

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
Given above is the Problem statement for the lab practical in th subject of ML. attached the Code for the implementation Explain the Code Line By line in detail.

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score

# Step 1: Load the dataset
url = 'https://www.kaggle.com/datasets/kyanyoga/sample-sales-data/download'  # Update if stored locally
df = pd.read_csv(url)

# Step 2: Data Preprocessing
# Dropping unnecessary columns for clustering (ORDERNUMBER, ORDERDATE, etc.)
df = df[['QUANTITYORDERED', 'PRICEEACH', 'SALES', 'MONTH_ID', 'YEAR_ID']]

# Check for any null values
print("Null values:\n", df.isnull().sum())

# Scaling the data for K-means
```

```
scaler = StandardScaler()
scaled_data = scaler.fit_transform(df)

# Step 3: Determine the optimal number of clusters using the Elbow Method
inertia = []
K_range = range(1, 11)

for k in K_range:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(scaled_data)
    inertia.append(kmeans.inertia_)

# Plot the Elbow Graph
plt.figure(figsize=(8, 5))
plt.plot(K_range, inertia, marker='o')
plt.xlabel("Number of Clusters (k)")
plt.ylabel("Inertia")
plt.title("Elbow Method for Optimal k")
plt.show()

# Step 4: Apply K-Means with Optimal Number of Clusters (e.g., using k=4 after analyzing elbow
plot)
optimal_k = 4  # Replace based on elbow plot observation
kmeans = KMeans(n_clusters=optimal_k, random_state=42)
df['Cluster'] = kmeans.fit_predict(scaled_data)

# Step 5: Visualizations
# Scatter plot for two features colored by cluster
plt.figure(figsize=(10, 6))
sns.scatterplot(x=df['QUANTITYORDERED'], y=df['SALES'], hue=df['Cluster'], palette='viridis')
plt.title("K-Means Clustering on Sales Data")
plt.xlabel("Quantity Ordered")
plt.ylabel("Sales")
plt.legend(title='Cluster')
plt.show()

# Pair plot of features with cluster color for deeper analysis
sns.pairplot(df, hue='Cluster', palette='viridis', diag_kind='kde', markers='o')
plt.suptitle("Pair Plot of Features with Clusters", y=1.02)
plt.show()

# Optional: Print silhouette score
silhouette_avg = silhouette_score(scaled_data, df['Cluster'])
print(f"Silhouette Score for k={optimal_k}: {silhouette_avg:.4f}")
```

Here's a line-by-line explanation of the code for implementing K-Means clustering on the sales data to determine the optimal number of clusters using the Elbow Method.

1. Import Necessary Libraries

```
python


import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score
```

- `pandas` and `numpy` are used for data manipulation.
- `matplotlib.pyplot` and `seaborn` are for data visualization.
- `KMeans` from `sklearn.cluster` implements the K-Means algorithm.
- `StandardScaler` normalizes the data, and `silhouette_score` evaluates clustering quality.

2. Load the Dataset

```
python


url = 'https://www.kaggle.com/datasets/kyanyoga/sample-sales-data/download'  # Update if s
tored locally
df = pd.read_csv(url)
```

- Loads the sales data from `sales_data_sample.csv` into a DataFrame `df`. Update `url` to a local file path if needed.

3. Data Preprocessing

- **Selecting Relevant Columns**:

  ```
  python


  df = df[['QUANTITYORDERED', 'PRICEEACH', 'SALES', 'MONTH_ID', 'YEAR_ID']]
  ```

  - Drops unnecessary columns, keeping only those relevant for clustering: `QUANTITYORDERED`, `PRICEEACH`, `SALES`, `MONTH_ID`, and `YEAR_ID`.

- **Check for Null Values**:

  ```
  python


  print("Null values:\n", df.isnull().sum())
  ```

  - Checks for missing values in the selected columns, ensuring clean data for clustering.

- **Scale the Data**:

```
python


  scaler = StandardScaler()
  scaled_data = scaler.fit_transform(df)
```

- Uses `StandardScaler` to normalize the features, which helps K-Means perform better by ensuring all features contribute equally to the distance metric.

4. Determine Optimal Number of Clusters using the Elbow Method

- **Calculate Inertia for Different Values of k**:

```
python


  inertia = []
  K_range = range(1, 11)

  for k in K_range:
      kmeans = KMeans(n_clusters=k, random_state=42)
      kmeans.fit(scaled_data)
      inertia.append(kmeans.inertia_)
```

  - **inertia**: List to store the sum of squared distances (inertia) for different numbers of clusters (`k`).
  - **K_range**: Range of `k` values to test (from 1 to 10).
  - For each value of `k`, the K-Means model is fitted on the scaled data, and the inertia (sum of squared distances from each point to its cluster center) is appended to the `inertia` list.

- **Plot the Elbow Graph**:

```
python


  plt.figure(figsize=(8, 5))
  plt.plot(K_range, inertia, marker='o')
  plt.xlabel("Number of Clusters (k)")
  plt.ylabel("Inertia")
  plt.title("Elbow Method for Optimal k")
  plt.show()
```

  - Plots the Elbow Graph, where inertia is plotted against the number of clusters. The "elbow" point, where the inertia reduction slows, suggests an optimal `k`.

5. Apply K-Means with Optimal Number of Clusters

- **Define and Fit K-Means with Optimal `k`**:

```
python


  optimal_k = 4  # Replace based on elbow plot observation
  kmeans = KMeans(n_clusters=optimal_k, random_state=42)
  df['Cluster'] = kmeans.fit_predict(scaled_data)
```

- **optimal_k**: Based on the elbow plot, choose the best `k`. In this example, `k=4` is assumed.
- `kmeans.fit_predict(scaled_data)`: Fits the K-Means model with `optimal_k` clusters and assigns cluster labels to each data point, adding the labels as a new column `Cluster` in `df`.

6. Visualizations

- **Scatter Plot for Two Features Colored by Cluster**:

```
python


  plt.figure(figsize=(10, 6))
  sns.scatterplot(x=df['QUANTITYORDERED'], y=df['SALES'], hue=df['Cluster'], palette='v
  iridis')
  plt.title("K-Means Clustering on Sales Data")
  plt.xlabel("Quantity Ordered")
  plt.ylabel("Sales")
  plt.legend(title='Cluster')
  plt.show()
```

  - Plots a scatter plot with `QUANTITYORDERED` on the x-axis and `SALES` on the y-axis, color-coded by cluster. This visualization shows how the clustering algorithm grouped the data.

- **Pair Plot for Deeper Cluster Analysis**:

```
python


  sns.pairplot(df, hue='Cluster', palette='viridis', diag_kind='kde', markers='o')
  plt.suptitle("Pair Plot of Features with Clusters", y=1.02)
  plt.show()
```

  - Creates a pair plot with all features, with clusters color-coded. The pair plot allows visual inspection of feature relationships and clustering effectiveness across multiple feature combinations.

7. Evaluate Clustering with Silhouette Score (Optional)

```
python


  silhouette_avg = silhouette_score(scaled_data, df['Cluster'])
  print(f"Silhouette Score for k={optimal_k}: {silhouette_avg:.4f}")
```

- **silhouette_score**: Measures the quality of clustering. A higher silhouette score (closer to 1) indicates better-defined clusters.
- This score helps validate the chosen `k` value, with scores closer to 1 indicating better-defined clusters.

This code effectively implements K-Means clustering, including determining the optimal number of clusters using the Elbow Method, assigning cluster labels, and visualizing clustering results.

Now Predict The Questions which and provide the appropriate answers which are most probably going to asked on this practical in Viva.

Here are some potential viva questions for this K-Means clustering practical on the sales dataset, along with suitable answers:

1. What is the purpose of this practical?

- **Answer**: The purpose is to use K-Means clustering to group sales data into clusters based on similar characteristics. By determining optimal clusters with the Elbow Method, we can segment the data for analysis, potentially identifying distinct customer or product behaviors.

2. Explain K-Means clustering.

- **Answer**: K-Means is an unsupervised learning algorithm used for clustering data into `k` groups. It initializes `k` centroids randomly and iteratively assigns data points to the nearest centroid. The centroids are updated based on the mean of assigned points, and the process repeats until convergence, where clusters are stable.

3. What is the Elbow Method, and how does it help determine the number of clusters?

- **Answer**: The Elbow Method helps identify the optimal number of clusters by plotting inertia (within-cluster sum of squares) against the number of clusters (`k`). The "elbow" point, where inertia reduction slows, suggests an ideal `k`, balancing compactness and simplicity of clusters.

4. What is inertia in the context of K-Means?

- **Answer**: Inertia is the sum of squared distances between each data point and its assigned cluster centroid. Lower inertia indicates tighter clusters, but it decreases as `k` increases. Thus, the goal is to choose a `k` that minimizes inertia while avoiding overfitting with too many clusters.

5. Why do we scale the data before applying K-Means?

- **Answer**: K-Means relies on distance calculations, so scaling ensures all features contribute equally to cluster formation. Without scaling, features with larger numerical ranges might dominate the distance calculation, leading to biased clusters.

6. How does K-Means decide where to place the initial centroids?

- **Answer**: By default, K-Means initializes centroids randomly. However, many implementations, like `KMeans` in `sklearn`, use a method called "k-means++" for better initial centroid placement, reducing the likelihood of poor clustering by starting centroids further apart.

7. What is the significance of the silhouette score?

- **Answer**: The silhouette score measures the quality of clustering by comparing the distance of each point to its own cluster (cohesion) with the distance to the nearest other cluster (separation). A higher score (close to 1) indicates well-separated and compact clusters, while scores close to 0 suggest overlapping clusters.

8. What are the main steps in the K-Means algorithm?

- **Answer**:
  1. **Initialize** `k` centroids randomly.
  2. **Assign** each point to the nearest centroid, forming `k` clusters.
  3. **Update** centroids as the mean of assigned points.
  4. Repeat assignment and updating until clusters stabilize or reach a maximum number of iterations.

9. What are some potential drawbacks of K-Means clustering?

- **Answer**: K-Means can:
  - **Converge to a local minimum**, resulting in suboptimal clusters.
  - Be **sensitive to the choice of** `k`, which isn't always straightforward to determine.
  - Perform poorly with **non-spherical clusters** or **imbalanced clusters**.
  - Be affected by **outliers**, as they can skew centroid positions.

10. What happens if we choose too many clusters?

- **Answer**: Too many clusters can lead to overfitting, where each cluster contains very few points, potentially capturing noise in the data instead of meaningful structure. The Elbow Method helps avoid this by suggesting an optimal number of clusters based on diminishing inertia returns.

11. Explain the significance of the Elbow Point in the Elbow Method.

- **Answer**: The Elbow Point is where adding more clusters does not significantly decrease inertia, indicating an optimal `k`. Choosing `k` at the elbow balances tightness within clusters and simplicity by avoiding unnecessary clusters.

12. How would you handle a dataset with non-spherical clusters in K-Means?

- **Answer**: K-Means works best with spherical clusters, so for non-spherical clusters, other methods like **DBSCAN** or **Agglomerative Clustering** may be more appropriate. Additionally, using a **kernelized K-Means** approach can sometimes help capture non-linear boundaries.

13. Why might two runs of K-Means with the same data yield different results?

- **Answer**: K-Means initializes centroids randomly (unless using a fixed random state), leading to different starting points and potentially different final clusters in each run. Setting a fixed `random_state` can ensure reproducibility.

14. How does K-Means differ from hierarchical clustering?

- **Answer**:
  - **K-Means**: A partitioning algorithm that requires specifying `k` in advance and creates `k` clusters by minimizing within-cluster variance.

- **Hierarchical Clustering**: Builds a tree of clusters without needing to specify `k` initially. It can be **agglomerative** (starting with individual points and merging clusters) or **divisive** (starting with all data in one cluster and splitting).

15. What does each cluster represent in this practical?

- **Answer**: Each cluster represents a group of sales data points with similar characteristics, based on features like `QUANTITYORDERED`, `PRICEEACH`, `SALES`, `MONTH_ID`, and `YEAR_ID`. Clusters might reflect sales patterns or customer behaviors.

16. Can K-Means detect outliers?

- **Answer**: K-Means is not designed to detect outliers, but outliers can influence centroid positions. Outliers often have high distances from centroids, which may appear in the form of loosely clustered or isolated points in visualizations.

17. Why is it important to visualize clusters?

- **Answer**: Visualizing clusters helps verify the clustering effectiveness, understand data structure, and interpret the meaning of each cluster. It also allows checking for well-separated clusters, overlapping clusters, and potential outliers.

18. What is the role of `random_state` in K-Means?

- **Answer**: Setting `random_state` ensures reproducibility by controlling the randomness in centroid initialization. This way, running K-Means multiple times yields the same clusters, useful for consistent analysis.

19. How would you explain the difference between `inertia` and `silhouette score`?

- **Answer**:
  - **Inertia**: Measures compactness within clusters by summing squared distances from points to their centroids. Lower inertia indicates more compact clusters.
  - **Silhouette Score**: Measures cohesion within clusters and separation between clusters, providing a score between -1 and 1 to indicate well-defined clusters. Higher silhouette scores indicate better clustering.

20. How would you deploy this clustering model in a real-world application?

- **Answer**: In a real-world setting, this model could be used for customer segmentation, helping a business tailor marketing strategies for different customer clusters. It could also aid in product recommendations by identifying similar purchase patterns across clusters.

These questions cover the main concepts, techniques, and practical considerations of K-Means clustering for sales data analysis, helping to prepare thoroughly for the viva.

ChatGPT can make mistakes. Check important info.