

Uber Data Preprocessing



```
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
import warnings
warnings.filterwarnings('ignore')

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

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

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

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

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

print(df[['pickup_latitude', 'pickup_longitude', 'dropoff_latitude', 'dropoff_longitude']].isnull().sum())

invalid_rows = df[
    (df['pickup_latitude'] < -90) | (df['pickup_latitude'] > 90) |
    (df['pickup_longitude'] < -180) | (df['pickup_longitude'] > 180) |
    (df['dropoff_latitude'] < -90) | (df['dropoff_latitude'] > 90) |
    (df['dropoff_longitude'] < -180) | (df['dropoff_longitude'] > 180)
]
print(invalid_rows)

lat_min, lat_max = -90.0, 90.0
lon_min, lon_max = -90.0, 90.0

df['pickup_longitude'] = df['pickup_longitude'].apply(lambda x: x if lon_min <= x <= lon_max else
df['pickup_longitude'].median())
df['pickup_latitude'] = df['pickup_latitude'].apply(lambda x: x if lat_min <= x <= lat_max else
df['pickup_latitude'].median())
df['dropoff_longitude'] = df['dropoff_longitude'].apply(lambda x: x if lon_min <= x <= lon_max else
```

```

df['dropoff_longitude'].median()
df['dropoff_latitude'] = df['dropoff_latitude'].apply(lambda x: x if lat_min <= x <= lat_max else
df['dropoff_latitude'].median())

df

def calculate_distance(row):
    try:
        return geodesic(
            (row['pickup_latitude'], row['pickup_longitude']),
            (row['dropoff_latitude'], row['dropoff_longitude'])
        ).km
    except ValueError as e:
        print(f"Error calculating distance for row {row.name}: {e}")
        return None # or 0, depending on how you want to handle errors

df['distance_km'] = df.apply(calculate_distance, axis=1)

df = df[df['distance_km'] > 0]
df = df[df['distance_km'] < 100]

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

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

df

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
sns.boxplot(df['fare_amount'])
plt.title('Boxplot of Fare Amount')
plt.subplot(1, 2, 2)
sns.boxplot(df['distance_km'])
plt.title('Boxplot of Distance (km)')
plt.show()

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

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
sns.boxplot(df['fare_amount'])
plt.title('Boxplot of Fare Amount')
plt.subplot(1, 2, 2)
sns.boxplot(df['distance_km'])
plt.title('Boxplot of Distance (km)')
plt.show()

```

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

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

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

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

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

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

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

r2_linear, rmse_linear = evaluate_model(y_test, y_pred_linear)
print("Linear Regression:")
print(f"R^2 Score: {r2_linear:.4f}")
print(f"RMSE: {rmse_linear:.4f}")

r2_rf, rmse_rf = evaluate_model(y_test, y_pred_rf)
print("\nRandom Forest Regression:")
print(f"R^2 Score: {r2_rf:.4f}")
print(f"RMSE: {rmse_rf:.4f}")

plt.figure(figsize=(14, 6))
# Linear Regression Predictions
plt.subplot(1, 2, 1)
plt.scatter(y_test, y_pred_linear, alpha=0.5, color='blue')
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--')
plt.xlabel("Actual Fare")
plt.ylabel("Predicted Fare")
plt.title("Linear Regression: Actual vs Predicted Fares")

# Random Forest Predictions
plt.subplot(1, 2, 2)
plt.scatter(y_test, y_pred_rf, alpha=0.5, color='green')
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--')
plt.xlabel("Actual Fare")
plt.ylabel("Predicted Fare")
plt.title("Random Forest: Actual vs Predicted Fares")
```

```
plt.show()

metrics = pd.DataFrame({
    'Model': ['Linear Regression', 'Random Forest'],
    'R^2 Score': [r2_linear, r2_rf],
    'RMSE': [rmse_linear, rmse_rf]
})

plt.figure(figsize=(14, 6))
plt.subplot(1, 2, 1)
sns.barplot(x='Model', y='R^2 Score', data=metrics, palette='viridis')
plt.title("R^2 Score Comparison")

plt.subplot(1, 2, 2)
sns.barplot(x='Model', y='RMSE', data=metrics, palette='viridis')
plt.title("RMSE Comparison")
plt.show()

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))

solution

explain this code line by line code by code in detail of the assignment 1
```



Let's go through the code step-by-step. This code performs data preprocessing, exploratory data analysis, feature engineering, and predictive modeling on a dataset containing Uber fare details.

Step 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
import warnings
warnings.filterwarnings('ignore')
```

- `pandas`: For data manipulation and analysis.
- `numpy`: For numerical operations.

- ``matplotlib.pyplot`` and ``seaborn``: For data visualization.
- ``train_test_split``: To split the dataset into training and testing sets.
- ``LinearRegression`` and ``RandomForestRegressor``: Algorithms for predictive modeling.
- ``r2_score`` and ``mean_squared_error``: Metrics for model evaluation.
- ``geodesic``: Used to calculate the geographic distance between pickup and drop-off locations.
- ``warnings``: Suppresses warnings to keep the output clean.

Step 2: Load the Dataset and Inspect

python

```
df = pd.read_csv("./uber.csv")
print("Dataset Information:\n")
print(df.info())
print("\nDataset Head:\n")
print(df.head())
```

- ``pd.read_csv``: Loads the Uber dataset.
- ``df.info()``: Provides an overview of the dataset, including data types and missing values.
- ``df.head()``: Displays the first few rows to give a preview of the data.

Step 3: Convert Pickup Datetime Column and Handle Missing Values

python

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

- ``pd.to_datetime``: Converts ``pickup_datetime`` to a datetime object. ``errors='coerce'`` converts invalid parsing to NaT (Not a Time).
- ``dropna``: Removes rows where ``pickup_datetime`` or ``fare_amount`` has missing values.

Step 4: Filter Rows Based on Fare Amount and Passenger Count

python

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

- The data is filtered to only keep rows where ``fare_amount`` is between 0 and 100, and ``passenger_count`` is between 1 and 6.

Step 5: Check for Null Values in Location Columns

python

```
print(df[['pickup_latitude', 'pickup_longitude', 'dropoff_latitude', 'dropoff_longitude']].isnull().sum())
```

- Checks for any missing values in the columns for pickup and drop-off latitude and longitude.

Step 6: Identify Invalid Geographic Coordinates

python

```
invalid_rows = df[
    (df['pickup_latitude'] < -90) | (df['pickup_latitude'] > 90) |
    (df['pickup_longitude'] < -180) | (df['pickup_longitude'] > 180) |
    (df['dropoff_latitude'] < -90) | (df['dropoff_latitude'] > 90) |
    (df['dropoff_longitude'] < -180) | (df['dropoff_longitude'] > 180)
]
print(invalid_rows)
```

- Identifies rows where the latitude and longitude values are outside valid ranges (-90 to 90 for latitude, -180 to 180 for longitude).

Step 7: Replace Invalid Coordinates with Median Values

python

```
lat_min, lat_max = -90.0, 90.0
lon_min, lon_max = -180.0, 180.0

df['pickup_longitude'] = df['pickup_longitude'].apply(lambda x: x if lon_min <= x <= lon_max else df['pickup_longitude'].median())
df['pickup_latitude'] = df['pickup_latitude'].apply(lambda x: x if lat_min <= x <= lat_max else df['pickup_latitude'].median())
df['dropoff_longitude'] = df['dropoff_longitude'].apply(lambda x: x if lon_min <= x <= lon_max else df['dropoff_longitude'].median())
df['dropoff_latitude'] = df['dropoff_latitude'].apply(lambda x: x if lat_min <= x <= lat_max else df['dropoff_latitude'].median())
```

- For each latitude and longitude column, if values fall outside the valid range, they're replaced with the median of that column.

Step 8: Calculate Distance Between Pickup and Drop-Off Points

python

```
def calculate_distance(row):
    try:
        return geodesic(
            (row['pickup_latitude'], row['pickup_longitude']),
            (row['dropoff_latitude'], row['dropoff_longitude'])
        ).km
    except ValueError as e:
        print(f"Error calculating distance for row {row.name}: {e}")
        return None

df['distance_km'] = df.apply(calculate_distance, axis=1)
```

- This function calculates the distance in kilometers between pickup and drop-off locations.
- If any issue arises (like invalid coordinates), it returns `None`.

Step 9: Filter Rows Based on Distance

python

```
df = df[df['distance_km'] > 0]
df = df[df['distance_km'] < 100]
```

- Only rows where the calculated distance is between 0 and 100 km are kept.

Step 10: 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
```

- Extracts hour, day, month, and year from `pickup_datetime` for feature engineering.

Step 11: Drop Unneeded Columns

python

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

- Removes columns that are no longer needed for modeling.

Step 12: Boxplot Visualization

python

```
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
sns.boxplot(df['fare_amount'])
plt.title('Boxplot of Fare Amount')
plt.subplot(1, 2, 2)
sns.boxplot(df['distance_km'])
plt.title('Boxplot of Distance (km)')
plt.show()
```

- Visualizes the distributions of `fare_amount` and `distance_km` to check for outliers.

Step 13: Remove Outliers

python

```
from scipy import stats
```

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

- Removes rows where `fare_amount` or `distance_km` has a z-score greater than 3, assuming these are outliers.

Step 14: Correlation Heatmap

python

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

- Shows a correlation heatmap for all features.

Step 15: Split Data into Features and Target, Train-Test Split

python

```
X = df.drop('fare_amount', axis=1)
y = df['fare_amount']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

- Splits the dataset into features `X` (excluding `fare_amount`) and target `y` (`fare_amount`).
- Splits data into training (80%) and testing (20%) sets.

Step 16: Model Training

python

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

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

- Trains two models: Linear Regression and Random Forest.

Step 17: Make Predictions and Evaluate Models

python

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

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
```



```
r2_linear, rmse_linear = evaluate_model(y_test, y_pred_linear)
print("Linear Regression:")
print(f"R^2 Score: {r2_linear:.4f}")
print(f"RMSE: {rmse_linear:.4f}")

r2_rf, rmse_rf = evaluate_model(y_test, y_pred_rf)
print("\nRandom Forest Regression:")
print(f"R^2 Score: {r2_rf:.4f}")
print(f"RMSE: {rmse_rf:.4f}")
```

- Makes predictions and evaluates each model's R^2 and RMSE.

Step 18: Visualize Model Predictions

python

```
plt.figure(figsize=(14, 6))
# Linear Regression Predictions
plt.subplot(1, 2, 1)
plt.scatter(y_test, y_pred_linear, alpha=0.5, color='blue')
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--')
plt.xlabel("Actual Fare")
plt.ylabel("Predicted Fare")
plt.title("Linear Regression: Actual vs Predicted Fares")

# Random Forest Predictions
plt.subplot(1, 2, 2)
plt.scatter(y_test, y_pred_rf, alpha=0.5, color='green')
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--')
plt.xlabel("Actual Fare")
plt.ylabel("Predicted Fare")
plt.title("Random Forest: Actual vs Predicted Fares")

plt.show()
```

- Creates scatter plots to show the relationship between actual and predicted fares.

Step 19: Visualize Model Performance

python

```
metrics = pd.DataFrame({
    'Model': ['Linear Regression', 'Random Forest'],
    'R^2 Score': [r2_linear, r2_rf],
    'RMSE': [rmse_linear, rmse_rf]
})

plt.figure(figsize=(14, 6))
plt.subplot(1, 2, 1)
sns.barplot(x='Model', y='R^2 Score', data=metrics, palette='viridis')
plt.title("R^2 Score Comparison")

plt.subplot(1, 2, 2)
sns.barplot(x='Model', y='RMSE', data=metrics, palette='viridis')
plt.title("RMSE Comparison")
plt.show()
```

- Compares the R^2 and RMSE scores of both models with bar plots.

Step 20: Add Predictions to Data

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))
solution
```

- Creates a copy of the cleaned dataset and adds two new columns with predictions from both models.



```
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
import warnings
warnings.filterwarnings('ignore')

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

print("Dataset Shape:", df.shape)
print("Dataset Sample:\n", df.head())

"""Check for missing values"""

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

"""# Step 4: Data Preprocessing

Dropping the first column as it is just an email identifier
"""

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

"""Separate features and target variable"""

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

"""# Step 5: Train-Test Split"""

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

```
"""# Step 6: Feature Scaling"""
```

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

```
"""# Step 7: Dimensionality Reduction for Visualization
```

```
Reduce dimensions to 2 for plotting using PCA
"""
```

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

```
"""Visualizing the data distribution after PCA"""
```

```
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()
```

```
"""# Step 8: Model Training and Evaluation - KNN"""
```

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

```
"""# Step 9: Model Training and Evaluation - SVM"""
```

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

```
"""# Step 10: Performance Analysis
```

```
Function to display performance metrics
"""
```

```
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"""

display_metrics(y_test, y_pred_knn, "K-Nearest Neighbors")

"""Display metrics for SVM"""

display_metrics(y_test, y_pred_svm, "Support Vector Machine")

"""# Step 11: Compare Model Performance Using a Bar Chart"""

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()

"""# Step 12: Add Predictions to the Original DataFrame

Compute predictions on the entire dataset (scaled)
"""

X_scaled = scaler.transform(X) # Scaling the full feature set for consistency
df["KNN_Prediction"] = knn.predict(X_scaled)
df["SVM_Prediction"] = svm.predict(X_scaled)

"""Display the updated DataFrame with predictions"""

df

now explain this line by line code by code
```



Here's an in-depth explanation of each part of the code you provided. This code performs data preprocessing, visualization, dimensionality reduction, and model training using K-Nearest Neighbors (KNN) and Support Vector Machine (SVM) classifiers on an email dataset to classify emails as spam or not spam.

Step 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.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
import warnings
warnings.filterwarnings('ignore')
```

- **pandas:** Used for data manipulation.
- **numpy:** Provides support for numerical operations.
- **matplotlib.pyplot** and **seaborn:** Used for data visualization.
- **train_test_split:** Splits the dataset into training and testing sets.
- **StandardScaler:** Standardizes features by removing the mean and scaling to unit variance.
- **KNeighborsClassifier** and **SVC:** Classification models for K-Nearest Neighbors and Support Vector Machine, respectively.
- **accuracy_score**, **classification_report**, and **confusion_matrix:** Evaluation metrics.
- **PCA:** Reduces data to two dimensions for easier visualization.
- **warnings.filterwarnings:** Suppresses warnings to keep the output clean.

Step 2: Load the Dataset

python

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

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

- **pd.read_csv:** Loads the dataset from a CSV file.
- **df.shape:** Shows the dataset's dimensions (number of rows and columns).
- **df.head():** Displays the first few rows of the dataset.

Step 3: Check for Missing Values

python

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

- **df.isnull().sum().sum():** Checks for any missing values in the dataset by summing up all null values in each column.

Step 4: Data Preprocessing - Drop Identifier Column

python

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

- Drops the first column, which is an email identifier, as it is not needed for modeling.

Step 5: Separate Features and Target Variable

python

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

- **X**: Contains all columns except the last one (features).
- **y**: Contains only the last column, which is the target variable (whether the email is spam or not).

Step 6: Split Data 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)
```

- **train_test_split**: Splits `X` and `y` into training and testing sets. The test set size is 30% of the data, and `random_state=42` ensures reproducibility.

Step 7: Feature Scaling

python

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

- **StandardScaler**: Standardizes features by centering to mean 0 and scaling to variance 1.
- **fit_transform**: Fits the scaler on the training data and applies the transformation.
- **transform**: Scales the testing data with the same parameters used for training data to maintain consistency.

Step 8: Dimensionality Reduction Using PCA

python

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

- **PCA(n_components=2):** Reduces features to 2 principal components for easier visualization.
- **fit_transform:** Computes the PCA on training data and applies the transformation.
- **transform:** Applies the PCA transformation on testing data.

Step 9: Visualize 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()
```

- **sns.scatterplot:** Plots a scatter plot with PCA components as axes and colors based on the target variable `y_train`.
- **hue=y_train:** Uses target values to color points for visualizing class distribution.

Step 10: Train and Evaluate K-Nearest Neighbors Model

python

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

- **KNeighborsClassifier:** Initializes the KNN classifier with `n_neighbors=5`.
- **fit:** Trains the model on scaled training data.
- **predict:** Makes predictions on the scaled test data.

Step 11: Train and Evaluate Support Vector Machine Model

python

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

- **SVC(kernel='linear'):** Initializes the SVM classifier with a linear kernel.
- **fit:** Trains the SVM model on scaled training data.
- **predict:** Predicts class labels on the scaled test data.

Step 12: Performance Analysis - Define Metrics Display Function

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 Spa
m', 'Spam'], yticklabels=['Not Spam', 'Spam'])
plt.title(f"{model_name} Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```

- **display_metrics:** Custom function to display model metrics.
 - Prints accuracy, classification report, and confusion matrix as a heatmap.

Step 13: Display Metrics for K-Nearest Neighbors

python

```
display_metrics(y_test, y_pred_knn, "K-Nearest Neighbors")
```

- Calls `display_metrics` for KNN model predictions, printing metrics and displaying the confusion matrix.

Step 14: Display Metrics for Support Vector Machine

python

```
display_metrics(y_test, y_pred_svm, "Support Vector Machine")
```

- Calls `display_metrics` for SVM model predictions, printing metrics and displaying the confusion matrix.

Step 15: Compare Model Accuracies with 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()
```

- Computes accuracy scores for both models.
- **sns.barplot:** Creates a bar chart comparing model accuracies.

Step 16: Add Predictions to the Original DataFrame


```
python
```

```
X_scaled = scaler.transform(X) # Scaling the full feature set for consistency
df['KNN_Prediction'] = knn.predict(X_scaled)
df['SVM_Prediction'] = svm.predict(X_scaled)
```

- Scales the entire feature set `X` with the `StandardScaler`.
- **df['KNN_Prediction']** and **df['SVM_Prediction']**: Adds columns to the original DataFrame containing predictions from KNN and SVM models on the entire dataset.

Step 17: Display Updated DataFrame with Predictions

```
python
```

```
df
```

- Displays the DataFrame, which now includes the predictions from both models.



```
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
import warnings
warnings.filterwarnings('ignore')
```

```
"""# Step 1: Load the dataset"""
```

```
data = pd.read_csv('./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()

explain code by code line by line

```



Let's break down the code line-by-line. This code performs data preprocessing, encoding, scaling, and builds a neural network to predict customer churn based on various customer attributes. It also evaluates model performance and adds the predictions back into a DataFrame for analysis.

Step 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.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
import warnings
warnings.filterwarnings('ignore')

```

- **pandas** and **numpy**: Libraries for data manipulation and numerical operations.
- **matplotlib.pyplot** and **seaborn**: Used for data visualization.
- **train_test_split**: Splits the dataset into training and testing sets.
- **StandardScaler** and **LabelEncoder**: Used for scaling features and encoding categorical variables.
- **accuracy_score**, **confusion_matrix**, and **classification_report**: Evaluation metrics for the model.
- **Sequential**, **Dense**, **Dropout**: Keras classes for building a neural network.
- **EarlyStopping**: Keras callback to stop training when validation loss stops improving.
- **warnings.filterwarnings**: Suppresses warnings for a cleaner output.

Step 2: Load the Dataset

python

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

- **pd.read_csv**: Loads the dataset from a CSV file.
- **data.shape**: Displays the dataset's dimensions (rows, columns).
- **data.head()**: Displays the first few rows to get an overview of the data.

Step 3: Prepare Features and Target Variable

python

```
X = data.drop(columns=['RowNumber', 'CustomerId', 'Surname', 'Exited'])
y = data['Exited'] # 'Exited' column is the target
```

- **X**: Feature set, excluding irrelevant columns (`RowNumber`, `CustomerId`, `Surname`) and the target column (`Exited`).
- **y**: Target variable, which is the `Exited` column indicating whether a customer has churned.

Step 4: 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'])
```

- **LabelEncoder**: Encodes categorical columns into numeric values.
 - **Geography**: Transforms different locations into numeric codes.
 - **Gender**: Converts 'Male' and 'Female' into 0 and 1, respectively.

Step 5: Split Data into Training and Test Sets

python

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

- **train_test_split**: Splits the data into 80% training and 20% testing sets. `random_state=42` ensures reproducibility.

Step 6: Scale the Features

python

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

- **StandardScaler:** Scales features to have zero mean and unit variance.
 - **fit_transform:** Fits the scaler on training data and scales it.
 - **transform:** Scales test data using the same scaling as training data.

Step 7: Build the Neural Network Model

python

```
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
```

- **Sequential:** Initializes a sequential neural network model.
- **Dense(64, activation='relu', input_shape=(X_train.shape[1],)):** First layer with 64 neurons, ReLU activation, and input shape matching the feature count.
- **Dropout(0.3):** Dropout layer that randomly drops 30% of neurons to prevent overfitting.
- **Dense(32, activation='relu'):** Second layer with 32 neurons and ReLU activation.
- **Dense(1, activation='sigmoid'):** Output layer with 1 neuron and sigmoid activation, suited for binary classification.

Step 8: Compile the Model

python

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

- **compile:** Configures the model for training.
 - **optimizer='adam':** Uses the Adam optimization algorithm.
 - **loss='binary_crossentropy':** Binary cross-entropy loss function for binary classification.
 - **metrics=['accuracy']:** Evaluates the model using accuracy.

Step 9: Model Summary

python

```
model.summary()
```

- **model.summary():** Prints a summary of the model's architecture, layer names, output shapes, and parameter counts.

Step 10: Early Stopping Callback

python

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

- **EarlyStopping:** Stops training when validation loss does not improve for 5 epochs (`patience=5`), restoring the best weights observed during training.

Step 11: Train the Model

python

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

- **model.fit:** Trains the model with the following settings:
 - **epochs=50:** Max number of training epochs.
 - **batch_size=32:** Number of samples per batch.
 - **validation_split=0.2:** 20% of training data is used for validation.
 - **callbacks=[early_stopping]:** Stops training early if validation loss doesn't improve.

Step 12: Plot Training History

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()
```

- **plt.figure:** Creates a new figure with subplots.
 - **Subplot 1:** Plots training and validation accuracy over epochs.
 - **Subplot 2:** Plots training and validation loss over epochs.

Step 13: Make Predictions

python

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

- **model.predict(X_test)**: Predicts probabilities for each sample in the test set.
- **(> 0.5).astype("int32")**: Converts probabilities to binary predictions (0 or 1) based on a threshold of 0.5.

Step 14: Calculate Accuracy Score

python

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

- **accuracy_score**: Computes the accuracy of the model by comparing predicted labels with true labels.

Step 15: Confusion Matrix

python

```
cm = confusion_matrix(y_test, y_pred)
```

- **confusion_matrix**: Computes the confusion matrix to evaluate model performance.

Step 16: Plot Confusion Matrix

python

```
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()
```

- **sns.heatmap**: Visualizes the confusion matrix with annotations.

Step 17: Classification Report

python

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

- **classification_report**: Generates a report with precision, recall, and F1-score for each class.

Step 18: Re-index the Target Labels

```
python

y_test_df = y_test.reset_index(drop=True)
```

- **reset_index:** Resets the index of `y_test` to align it with the predictions for easy combination.

Step 19: Create a DataFrame for Predictions

```
python

predictions_df = pd.DataFrame(y_pred, columns=["Predicted"], index=y_test_df.index)
```

- **pd.DataFrame:** Creates a DataFrame for predictions, matching indices with `y_test_df`.

Step 20: Combine Predictions with Test Set

```
python

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
```

- **scaler.inverse_transform:** Inverse scales `X_test` to its original scale for better interpretability.
- **results_df:** Combines feature values, true labels, and predicted labels.

Step 21: Display Results

```
python

results_df.head()
```

- **head():** Shows the first few rows of the DataFrame with features, true labels, and predicted labels for easy review.



```
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}")

explain this code by code line by line

```



This code implements and visualizes gradient descent on the function $y = (x + 3)^2$. It finds the local minimum of the function by iteratively adjusting x in the direction that minimizes y , based on the derivative (gradient) of the function.

Step 1: Import Libraries

python

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

- **numpy:** Used for numerical operations, like creating ranges of numbers.
- **matplotlib.pyplot:** Used for plotting and visualizing the function and gradient descent path.

Step 2: Define the Function and Its Derivative

python

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

- **function(x):** Defines $f(x) = (x + 3)^2$, which is the function we want to minimize.
 - For any input x , it returns the squared value of $x + 3$.

python

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

- **derivative(x):** Defines the derivative of $f(x)$, which is $f'(x) = 2(x + 3)$.
 - This derivative helps compute the gradient, giving the direction and rate of change at a given x .

Step 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:** Controls the step size of each update. A smaller rate results in slower convergence but greater precision.
- **n_iterations:** The number of iterations (steps) to perform.

- **x_start**: The initial value of x , where gradient descent starts.

Step 4: Prepare Lists for Visualization

python

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

- **x_values** and **y_values**: Lists to store x and y values at each step for visualization.
 - They start with the initial x (2) and corresponding y value (`function(x_start)`).

Step 5: Perform Gradient Descent

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)
```

- **x = x_start**: Initializes x for the gradient descent loop.
- **for i in range(n_iterations)**: Loops for the specified number of iterations.
 - **gradient = derivative(x)**: Computes the gradient (rate of change) of $f(x)$ at the current x .
 - **x = x - learning_rate * gradient**: Updates x by moving it in the opposite direction of the gradient. The step size is scaled by the learning rate.
 - **y = function(x)**: Calculates y for the updated x , which represents the current value of the function.
 - **x_values.append(x)** and **y_values.append(y)**: Adds the current x and y values to their respective lists for plotting the descent path.

Step 6: Plot the Function and Gradient Descent Path

python

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

- **x_range**: Creates a range of x values from -10 to 4 to plot the function.
- **y_range = function(x_range)**: Calculates y values for each x in `x_range` using the `function` definition.

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.figure:** Creates a new figure for the plot with a defined size.
- **plt.plot(x_range, y_range, ...):** Plots the function $y = (x + 3)^2$ over `x_range`.
- **plt.scatter(x_values, y_values, ...):** Plots points for each (x, y) value during gradient descent in red.
- **plt.plot(x_values, y_values, ...):** Connects the descent path points with a dashed red line.
- **plt.title, plt.xlabel, plt.ylabel:** Adds a title and axis labels.
- **plt.legend():** Displays a legend for the plot.
- **plt.show():** Renders the plot.

Step 7: Print Final Results

python

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

- Prints the coordinates of the final point in the gradient descent path, which is the approximate local minimum.
 - **x_values[-1]** and **y_values[-1]:** The last values in `x_values` and `y_values`, representing the coordinates of the minimum found by gradient descent.

ChatGPT can make mistakes. Check important info.