ML Practical-

1. Predict the price of the Uber ride from a given pickup point to the agreed drop-off location. Perform following tasks:
1. Pre-process the dataset.
2. Identify outliers.
3. Check the correlation.
4. Implement linear regression and random forest regression models.
5. Evaluate the models and compare their respective scores like R2, RMSE, etc.
Dataset link: https://www.kaggle.com/datasets/yasserh/uber-fares-dataset

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from math import radians, sin, cos, asin, sqrt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
# Step 1: Load Data
df = pd.read_csv("uber.csv")

# Preprocessing
df = df.drop(["Unnamed: 0", "key"], axis=1)
df["pickup_datetime"] = pd.to_datetime(df["pickup_datetime"])
df = df.dropna()

# Function to calculate distance (haversine formula)
def distance_transform(lon1, lat1, lon2, lat2):
    d = []
    for i in range(len(lon1)):
        lon1_r, lat1_r, lon2_r, lat2_r = map(radians, [lon1[i], lat1[i], lon2[i], lat2[i]])
        dlon = lon2_r - lon1_r
        dlat = lat2_r - lat1_r
        a = sin(dlat/2)**2 + cos(lat1_r) * cos(lat2_r) * sin(dlon/2)**2
        c = 2 * asin(sqrt(a)) * 6371
        d.append(c)
    return d

df["distance_km"] = distance_transform(df.pickup_longitude.values,
                                       df.pickup_latitude.values,
                                       df.dropoff_longitude.values,
                                       df.dropoff_latitude.values)

# Extract datetime features
df["pickup_hr"] = df.pickup_datetime.dt.hour
df["day"] = df.pickup_datetime.dt.day
df["month"] = df.pickup_datetime.dt.month
df["year"] = df.pickup_datetime.dt.year
df["day_of_week"] = df.pickup_datetime.dt.dayofweek
# Step 2: Remove Outliers
df = df[(df.fare_amount > 0) & (df.fare_amount < 100)]
df = df[(df.distance_km > 0) & (df.distance_km < 60)]
df = df[(df.passenger_count > 0) & (df.passenger_count <= 6)]
```

```
# Step 3: Correlation
plt.figure(figsize=(10,6))
sns.heatmap(df.corr(), annot=True)
plt.title("Correlation Heatmap")
plt.show()

# Feature Selection
X = df[["distance_km", "passenger_count", "pickup_hr", "day", "month", "year", "day_of_week"]]
y = df["fare_amount"]

# Scale
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Step 4: Train Models
lr = LinearRegression()
lr.fit(X_train, y_train)

rfr = RandomForestRegressor()
rfr.fit(X_train, y_train)

# Step 5: Predictions
y_pred_lr = lr.predict(X_test)
y_pred_rf = rfr.predict(X_test)

# Evaluation Function
def evaluate(model_name, y_test, y_pred):
    print(f"\n📌 {model_name} Performance:")
    print("MAE:", mean_absolute_error(y_test, y_pred))
    print("MSE:", mean_squared_error(y_test, y_pred))
    print("RMSE:", np.sqrt(mean_squared_error(y_test, y_pred)))
    print("R² Score:", r2_score(y_test, y_pred))

evaluate("Linear Regression", y_test, y_pred_lr)
evaluate("Random Forest Regressor", y_test, y_pred_rf)
```

Explanation-

The first step in your project is **importing libraries**, which are essential for data handling, analysis, visualization, and modeling. `pandas` is a library designed for **data manipulation and analysis**; it provides the `DataFrame` structure, which is a tabular representation of data with rows and columns, allowing easy filtering, indexing, and computation. `numpy` is a library for **numerical computations**, offering high-performance arrays and mathematical functions, which are more efficient than standard Python lists. Visualization is handled using `matplotlib.pyplot` and `seaborn`. `matplotlib` is a **low-level plotting library** that provides complete control over figures, while `seaborn` is a **high-level statistical visualization library** built on matplotlib that simplifies complex plots and integrates well with pandas data structures. Additionally, mathematical functions like `radians`, `sin`, `cos`, `asin`, and `sqrt` from the `math` module are used to perform trigonometric and square root calculations, essential for computing distances between two geographical points. From `sklearn`, modules like `train_test_split` for splitting data,

`StandardScaler` for feature scaling, regression models like `LinearRegression` and `RandomForestRegressor`, and evaluation metrics such as `mean_absolute_error`, `mean_squared_error`, and `r2_score` are imported to handle the machine learning workflow.

The **next step is loading the dataset** using `pd.read_csv("uber.csv")`. This function reads the CSV file into a pandas DataFrame, which is suitable for tabular data analysis. Following this, preprocessing is done by dropping irrelevant columns using `df.drop(["Unnamed: 0", "key"], axis=1)`. Removing unnecessary features is part of **data cleaning**, which ensures that the model is not distracted by irrelevant information. The `pickup_datetime` column is converted to a pandas datetime object using `pd.to_datetime()`, enabling extraction of useful temporal features like hour, day, month, year, and day of the week. Any missing values in the dataset are removed with `df.dropna()`, which is important because machine learning models cannot process null values directly.

One of the most crucial steps is **feature engineering**, specifically calculating the distance between pickup and dropoff points using the **Haversine formula**. This formula calculates the shortest distance over the Earth's surface between two latitude-longitude points. It first converts the coordinates from degrees to radians because trigonometric functions in Python work with radians. The formula computes `a = sin²(Δlat/2) + cos(lat1) * cos(lat2) * sin²(Δlon/2)` and then `c = 2 * asin(sqrt(a)) * R`, where R is the Earth's radius (6371 km). This returns the distance in kilometers. Calculating the distance is essential because the fare is highly dependent on how far the Uber trip travels. This derived feature is stored in a new column `distance_km`.

After creating distance, additional **datetime-based features** are extracted to capture temporal patterns: `pickup_hr` (hour of the day) captures peak vs. off-peak trends, `day`, `month`, and `year` can reveal seasonal trends or year-on-year fare changes, and `day_of_week` can distinguish weekday and weekend behavior. Feature engineering like this allows the model to understand relationships between time and fare patterns more effectively.

Next is **outlier removal**, which is critical because extreme values can distort model performance. Here, only rows with `fare_amount` between 0 and 100 are kept to avoid unrealistic fares, `distance_km` between 0 and 60 km to ignore extremely long trips, and `passenger_count` between 1 and 6 to remove invalid entries. Outlier removal ensures that models, especially sensitive ones like linear regression, are not skewed by extreme values.

Before modeling, a **correlation analysis** is performed using a heatmap from seaborn (`sns.heatmap`). Correlation measures the **linear relationship** between two variables, ranging from -1 to +1. A value near +1 indicates a strong positive relationship, -1 indicates a strong negative relationship, and 0 indicates no linear relationship. This step helps identify which features are most closely related to the target (`fare_amount`) and ensures only relevant features are selected.

For **feature selection**, columns like `distance_km`, `passenger_count`, `pickup_hr`, `day`, `month`, `year`, and `day_of_week` are chosen as independent variables (`X`), and `fare_amount` is the dependent variable (`y`). These features are selected based on their relevance to fare prediction.

Next, **feature scaling** is applied using `StandardScaler`, which standardizes features by removing the mean and scaling to unit variance: `X_scaled = (X - mean)/std`. Scaling ensures that features with larger ranges (like distance) do not dominate features with smaller ranges (like passenger count), which improves model performance and convergence, particularly for algorithms like linear regression.

The dataset is then split into **training and testing sets** using `train_test_split` with a 70-30 split. The training set is used to teach the model, while the testing set evaluates performance on unseen data. This is fundamental to prevent **overfitting**, where a model learns training data too well but performs poorly on new data.

Model training is done with two algorithms. First, **Linear Regression**, which assumes a linear relationship between features and the target, estimates coefficients (β values) that minimize the sum of squared errors between predicted and actual fares. Second, **Random Forest Regressor** is an ensemble model consisting of multiple decision trees. Each tree predicts fare based on a random subset of data and features, and the final prediction is the average of all trees. Random forests can capture **nonlinear relationships** and are less sensitive to outliers compared to linear regression.

Predictions are generated on the test set using the trained models. Finally, models are evaluated with metrics: **MAE**measures average absolute error, **MSE** measures average squared error (penalizing large deviations), **RMSE** is the square root of MSE to maintain original units, and **R^2 score** indicates how much variance in the target is explained by the features. Higher R^2 and lower errors indicate better model performance.

In summary, this pipeline covers **data preprocessing, feature engineering, outlier handling, feature selection, scaling, train-test splitting, model training, and evaluation**. Linear regression provides a simple, interpretable baseline, while random forest captures complex, nonlinear patterns, making the pipeline robust for predicting Uber fares. Every step is designed to ensure the model is accurate, reliable, and generalizable.

2. 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

Ans.

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import warnings
warnings.filterwarnings('ignore')
# Load dataset
df = pd.read_csv("sales_data_sample.csv", encoding="latin")
print(df.head())
# Select numeric features useful for clustering
data = df[['QUANTITYORDERED', 'PRICEEACH', 'SALES', 'QTR_ID', 'MONTH_ID', 'YEAR_ID']]

# Scale data
```

```
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data)
# Elbow Method to determine optimal number of clusters
wcss = []
for i in range(1, 11):
    model = KMeans(n_clusters=i, init='k-means++', random_state=42)
    model.fit(scaled_data)
    wcss.append(model.inertia_)

plt.figure(figsize=(6,4))
plt.plot(range(1, 11), wcss, marker='o')
plt.title("Elbow Method")
plt.xlabel("Number of Clusters (k)")
plt.ylabel("WCSS")
plt.show()



# Choose k = 4 (based on elbow bend)
k = 4
kmeans = KMeans(n_clusters=k, init='k-means++', random_state=42)
clusters = kmeans.fit_predict(scaled_data)
# Add cluster column back to dataset
df['Cluster'] = clusters
print(df[['QUANTITYORDERED', 'PRICEEACH', 'SALES', 'Cluster']].head())

# Visualize clusters with two main numeric features
plt.figure(figsize=(6,4))
plt.scatter(df['SALES'], df['QUANTITYORDERED'], c=df['Cluster'])
plt.xlabel("Sales")
plt.ylabel("Quantity Ordered")
plt.title("K-Means Clustering Result")
plt.show()
#for hierachical clustering
import scipy.cluster.hierarchy as sch
from sklearn.cluster import AgglomerativeClustering

# Dendrogram
plt.figure(figsize=(6,4))
sch.dendrogram(sch.linkage(scaled_data, method='ward'))
plt.title("Dendrogram")
plt.xlabel("Samples")
plt.ylabel("Distance")
plt.show()

# Hierarchical Clustering Model
hc = AgglomerativeClustering(n_clusters=4, metric='euclidean', linkage='ward')
df['HC_Cluster'] = hc.fit_predict(scaled_data)

print(df[['SALES', 'QUANTITYORDERED', 'HC_Cluster']].head())
```

Explanation-

The workflow begins with **importing libraries** essential for data analysis and clustering. `pandas` is used for **data handling**, reading CSV files, and manipulating tabular data. `matplotlib.pyplot` is for **data visualization**, allowing plotting of graphs to understand patterns or clustering results. `sklearn.cluster` provides clustering algorithms such as `KMeans` for partition-based clustering and `AgglomerativeClustering` for hierarchical

clustering. `StandardScaler` from `sklearn.preprocessing` standardizes numerical features, which is crucial for clustering because distance-based algorithms are sensitive to the scale of features. Finally, `warnings.filterwarnings('ignore')` is used to suppress warnings that may clutter the output.

Next, the dataset is **loaded** using `pd.read_csv("sales_data_sample.csv", encoding="latin")`. Encoding `latin`ensures compatibility with special characters. Using `df.head()`, the first few rows are displayed to verify that the data has loaded correctly. This step is part of **exploratory data analysis (EDA)** to understand the dataset's structure.

Before clustering, only **numerical features relevant for clustering** are selected: `QUANTITYORDERED`, `PRICEEACH`, `SALES`, `QTR_ID`, `MONTH_ID`, and `YEAR_ID`. Clustering algorithms require numerical input because they rely on computing distances between data points.

Since K-Means and hierarchical clustering are **distance-based algorithms**, features must be on a comparable scale. Standardization is done using `StandardScaler`, which **centers the data around zero and scales it to unit variance**. The formula is:

Xscaled=X−mean (X)std(X)

Xscaled =std(X)X−mean(X)

This ensures that no single feature dominates the clustering process due to its magnitude.

The **Elbow Method** is then used to determine the optimal number of clusters (`k`) for K-Means. In K-Means, each data point is assigned to the nearest cluster centroid, and the **Within-Cluster Sum of Squares (WCSS)** measures the total squared distance between points and their cluster centroid. By fitting K-Means for `k = 1` to `10` and plotting WCSS against `k`, we observe an "elbow" in the curve, which indicates the point beyond which adding more clusters does not significantly reduce WCSS. In this case, `k = 4` is chosen based on the elbow bend.

After selecting `k`, **K-Means clustering** is performed using `KMeans(n_clusters=k, init='k-means++', random_state=42)`. The `init='k-means++'` parameter improves convergence by carefully initializing centroids. The model is trained on the scaled data, and `fit_predict` assigns each sample to a cluster. The resulting cluster labels are added to the original DataFrame as a new column `Cluster`. This allows analysis of cluster-specific patterns. For example, observing `SALES` vs. `QUANTITYORDERED` per cluster helps understand customer or product segments.

To **visualize K-Means clusters**, a scatter plot of `SALES` vs. `QUANTITYORDERED` is created, coloring points according to their assigned cluster. This visual representation shows how data points group together in feature space, revealing patterns that can guide business decisions, such as targeting high-sales, high-quantity segments.

The code also demonstrates **Hierarchical Clustering**, which differs from K-Means because it does not require predefining the number of clusters. It builds a hierarchy of clusters by either agglomerating (merging) or dividing points based on distances. Here, `scipy.cluster.hierarchy.dendrogram` is used to plot a **dendrogram**, a tree-like diagram that shows the sequence of cluster merges and the distances at which merges occur. The **Ward linkage method** minimizes the variance within clusters at each merge step. By examining the dendrogram, the number of clusters can be determined visually.

Once the desired number of clusters (`n_clusters=4`) is decided, **Agglomerative Hierarchical Clustering** is applied using `AgglomerativeClustering`. Each sample starts as its own cluster, and iteratively, the closest clusters are merged based on Euclidean distance and Ward linkage. The cluster labels are added to the DataFrame as `HC_Cluster`, allowing direct comparison with K-Means clusters.

In summary, the workflow demonstrates two key **unsupervised machine learning techniques**. K-Means is a partition-based algorithm optimal for large datasets where cluster shapes are convex, while Hierarchical Clustering provides a flexible, dendrogram-based approach to explore natural cluster hierarchies. Feature scaling, WCSS evaluation (Elbow Method), and visualization are integral concepts to ensure clusters are meaningful, interpretable, and actionable. The final

output allows segmentation of sales data based on numerical patterns, which is useful in customer profiling, inventory management, and marketing strategies.

3. 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

Ans.

```
# Import libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics
from sklearn.utils import resample
# Load dataset
df = pd.read_csv("diabetes.csv")
print(df.head())
print(df.info())

# Upsample minority class to balance dataset
negative_data = df[df["Outcome"] == 0]
positive_data = df[df["Outcome"] == 1]

positive_upsample = resample(positive_data,
                             replace=True,
                             n_samples=int(0.9 * len(negative_data)),
                             random_state=42)

new_df = pd.concat([negative_data, positive_upsample])
new_df = new_df.sample(frac=1, random_state=42)  # Shuffle
print(new_df["Outcome"].value_counts())

# Split features and target
X = new_df.drop("Outcome", axis=1)
y = new_df["Outcome"]

# Scale features
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

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

# Find optimal K (simplified)
k_values = list(range(1, 20, 2))
accuracy_values = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    accuracy_values.append(metrics.accuracy_score(y_test, y_pred))
```

```
# Plot accuracy vs K
plt.figure(figsize=(8,5))
plt.plot(k_values, accuracy_values, marker='o')
plt.xlabel("K value")
plt.ylabel("Accuracy")
plt.title("KNN Accuracy for different K values")
plt.show()

# Select best K
optimal_k = k_values[np.argmax(accuracy_values)]
print("Optimal K:", optimal_k)

# Train KNN with optimal K
knn_model = KNeighborsClassifier(n_neighbors=optimal_k)
knn_model.fit(X_train, y_train)
y_pred = knn_model.predict(X_test)

# Evaluate model
print("\nConfusion Matrix:\n")
cm = metrics.confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()

print("\nClassification Report:\n")
print(metrics.classification_report(y_test, y_pred))

accuracy = metrics.accuracy_score(y_test, y_pred)
error_rate = 1 - accuracy
precision = metrics.precision_score(y_test, y_pred)
recall = metrics.recall_score(y_test, y_pred)

print(f"Accuracy: {accuracy:.2f}")
print(f"Error Rate: {error_rate:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")

# ROC Curve
y_prob = knn_model.predict_proba(X_test)[:,1]
fpr, tpr, _ = metrics.roc_curve(y_test, y_prob)
roc_auc = metrics.roc_auc_score(y_test, y_prob)

plt.figure(figsize=(8,6))
plt.plot(fpr, tpr, label=f"AUC = {roc_auc:.2f}")
plt.plot([0,1], [0,1], 'k--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve - KNN")
plt.legend()
plt.show()
```

The workflow begins by **importing essential libraries** for data handling, visualization, preprocessing, modeling, and evaluation. `pandas` is used for reading CSV files and handling tabular data efficiently using DataFrames. `numpy` provides array operations and numerical computations. `matplotlib.pyplot` and `seaborn` are used for **data visualization**, allowing plotting of graphs such as accuracy curves, confusion matrices, and ROC curves. From

`sklearn.preprocessing`, `MinMaxScaler` is imported to **normalize feature values** between 0 and 1, which is critical for distance-based algorithms like K-Nearest Neighbors (KNN). `train_test_split` from `sklearn.model_selection` is used to divide the dataset into training and testing subsets to evaluate the model's generalization. `KNeighborsClassifier` implements the **KNN algorithm**, a non-parametric classification method that assigns a class to a sample based on the majority class of its K nearest neighbors in the feature space. `sklearn.metrics` provides performance evaluation metrics like accuracy, precision, recall, ROC-AUC, and the confusion matrix. Finally, `sklearn.utils.resample` is used for **resampling** to handle class imbalance.

The dataset is loaded using `pd.read_csv("diabetes.csv")`. Displaying the first few rows with `df.head()` and checking the structure with `df.info()` ensures the data has loaded correctly and shows the data types, non-null counts, and potential preprocessing requirements. In this dataset, the target variable `Outcome` is binary (0 = no diabetes, 1 = diabetes), which is typical in medical diagnostic datasets.

Because the dataset may be **imbalanced** (more negative cases than positive cases), the code applies **upsampling** to the minority class. Here, all positive cases (`Outcome = 1`) are resampled with replacement to create a larger sample size equal to 90% of the negative class (`Outcome = 0`). This balances the classes and prevents the KNN classifier from being biased toward the majority class. The positive and negative samples are concatenated and shuffled to create a balanced dataset for modeling.

Next, the **features and target variable are separated**. The features `X` consist of all columns except `Outcome`, and the target `y` is the `Outcome` column. Because KNN relies on **distance metrics** (typically Euclidean distance), features are scaled using `MinMaxScaler`. This **normalizes each feature to a range [0,1]**, ensuring that features with larger numerical ranges, like glucose levels or age, do not dominate smaller-scale features, such as BMI or pregnancies, during distance calculations.

The dataset is then split into **training and testing sets** using `train_test_split`, reserving 20% for testing. This ensures that the model is trained on one subset and evaluated on unseen data to check its **generalization capability**.

To find the **optimal value of K**, the code tests different odd values between 1 and 19. The KNN algorithm works by **assigning a class based on the majority vote of its K nearest neighbors**. Too small a K (e.g., 1) may lead to **overfitting**, as the prediction is highly sensitive to noise. Too large a K may lead to **underfitting**, smoothing over important patterns. For each K, the model is trained and evaluated on the test set, and accuracy is stored. Plotting **accuracy vs. K** allows visual identification of the K with the highest predictive performance, which is selected as the optimal K.

With the optimal K, the **final KNN model** is trained on the scaled training data. Predictions are made on the test set, and evaluation is performed using several metrics. The **confusion matrix** is plotted using seaborn, showing the number of true positives, true negatives, false positives, and false negatives. The **classification report** provides precision (correct positive predictions / total predicted positives), recall (correct positive predictions / total actual positives), F1-score (harmonic mean of precision and recall), and support (number of samples per class). These metrics are essential in **medical datasets**, where minimizing false negatives (missing a diabetes case) is critical.

Overall **accuracy** is calculated as the fraction of correct predictions, and the **error rate** is `1 - accuracy`. Precision and recall are explicitly printed to evaluate **model reliability and sensitivity**.

Finally, the **ROC curve** is plotted using the predicted probabilities of the positive class. The **ROC curve (Receiver Operating Characteristic)** visualizes the trade-off between true positive rate (sensitivity) and false positive rate (1-specificity) across different thresholds. The **AUC (Area Under the Curve)** quantifies the overall ability of the model to discriminate between classes, with values closer to 1 indicating excellent classification performance.

In summary, this pipeline implements a **K-Nearest Neighbors classification model** on a diabetes dataset with imbalanced classes. It includes **data preprocessing, feature scaling, class balancing via upsampling, train-test splitting, hyperparameter tuning for K, and comprehensive model evaluation** using accuracy, precision, recall, confusion matrix, and ROC-AUC. Each step ensures that the model is robust, fair, and interpretable, which is particularly important for healthcare applications where misclassification can have significant consequences.

4. 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).

Ans.

```
# Step 1: Import Libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Step 2: Read the dataset
df = pd.read_csv("Churn_Modelling.csv")  # replace with your path if needed

# Step 3: Prepare feature set X and target y
# Drop irrelevant columns (CustomerId, RowNumber, Surname)
X = df.drop(['RowNumber', 'CustomerId', 'Surname', 'Exited'], axis=1)
y = df['Exited']  # target: 1 = left, 0 = stayed


# Step 4: Encode categorical features
```

```
# Geography and Gender are categorical
X = pd.get_dummies(X, columns=['Geography', 'Gender'], drop_first=True)

# Step 5: Split dataset into training and testing
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Step 6: Normalize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 7: Initialize and build the model
mlp = MLPClassifier(
    hidden_layer_sizes=(50, 25),  # two hidden layers
    activation='relu',
    solver='adam',
    max_iter=500,
    random_state=42
)
mlp.fit(X_train, y_train)

# Step 8: Make predictions
y_pred = mlp.predict(X_test)

# Step 9: Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
report = classification_report(y_test, y_pred)

print("Accuracy Score:", accuracy)
print("\nConfusion Matrix:\n", conf_matrix)
print("\nClassification Report:\n", report)
```

Explanation

The workflow begins by **importing the necessary libraries**. `pandas` is used for handling tabular datasets efficiently, including reading CSV files and manipulating columns and rows. From `sklearn.model_selection`, `train_test_split`is imported to divide the dataset into **training and testing subsets**, which is essential to evaluate model performance on unseen data and prevent overfitting. `StandardScaler` from `sklearn.preprocessing` is used for **feature scaling**, which standardizes features to have zero mean and unit variance—this is particularly important for neural networks, as inputs with widely varying ranges can slow down convergence. `LabelEncoder` and `pd.get_dummies` handle **categorical data encoding**, converting non-numeric features into numerical representations suitable for modeling. The `MLPClassifier`from `sklearn.neural_network` is a **multi-layer perceptron (MLP)**, a type of feedforward artificial neural network used for classification tasks. Finally, metrics such as `accuracy_score`, `confusion_matrix`, and `classification_report` are imported to evaluate the model's performance comprehensively.

The dataset is loaded using `pd.read_csv("Churn_Modelling.csv")`, which reads the customer churn data into a pandas DataFrame. Displaying the head of the dataset helps understand the structure and types of features available. In this dataset, the target variable `Exited` indicates whether a customer left (1) or stayed (0). To prepare the feature matrix `X` and target vector `y`,

irrelevant columns like `CustomerId`, `RowNumber`, and `Surname` are dropped because they do not contain predictive information. This step ensures the model focuses only on meaningful features.

Categorical features such as `Geography` and `Gender` are encoded using **one-hot encoding** via `pd.get_dummies()`, which converts each category into a binary column while avoiding multicollinearity by dropping the first category (`drop_first=True`). This transformation allows neural networks to process categorical inputs as numerical vectors.

The dataset is then split into **training and testing sets** using an 80-20 split. The training set is used to **fit the MLP model**, while the testing set evaluates how well the model generalizes to new, unseen customers. Before training, the features are **standardized** with `StandardScaler`, transforming all features to a standard scale. This is crucial for MLP networks because large variations in feature ranges can cause the gradient descent optimization to converge slowly or get stuck.

An **MLPClassifier** is initialized with two hidden layers containing 50 and 25 neurons respectively, using the **ReLU activation function** (`relu`) which introduces nonlinearity and allows the network to model complex relationships. The solver `adam` is a stochastic optimization method that efficiently updates weights during training. `max_iter=500` sets the maximum number of training epochs, and `random_state` ensures reproducibility. The model is trained using `fit()` on the scaled training data, where weights and biases are adjusted to minimize classification error using backpropagation.

Predictions are generated on the test set using `predict()`. Model evaluation is performed using multiple metrics. **Accuracy** measures the proportion of correctly classified customers. The **confusion matrix** shows true positives, true negatives, false positives, and false negatives, giving insight into the types of errors made. The **classification report**provides precision (correct positive predictions / total predicted positives), recall (correct positive predictions / total actual positives), F1-score (harmonic mean of precision and recall), and support (number of samples per class). These metrics are particularly important in churn prediction, where identifying potential churners accurately (high recall) is critical for business interventions.

In summary, this workflow demonstrates a **supervised classification pipeline** for predicting customer churn using a neural network. It includes **data preprocessing, feature selection, categorical encoding, feature scaling, train-test splitting, model training, prediction, and comprehensive evaluation**. The MLPClassifier captures nonlinear patterns in customer behavior, making it more powerful than linear models for churn prediction, while evaluation metrics ensure the model's reliability for practical business decision-making.

5. 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

Ans.

```
# Step 1: Import 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.svm import SVC
from sklearn.utils import resample
from sklearn.metrics import classification_report, accuracy_score
import plotly.express as px
from tqdm import tqdm
# Step 2: Read Dataset
df = pd.read_csv("emails.csv")  # Replace with your path
df = df.drop("Email No.", axis=1)  # Drop irrelevant column

# Step 3: Check class distribution
print("Original class distribution:\n", df["Prediction"].value_counts())

# Step 4: Handle Class Imbalance (Upsample Spam)
spam = df[df["Prediction"] == 1]
ham = df[df["Prediction"] == 0]

spam_upsampled = resample(
    spam,
    replace=True,
    n_samples=int(0.8*len(ham)),  # 80% of ham
    random_state=42
)

df_balanced = pd.concat([ham, spam_upsampled])
df_balanced = df_balanced.sample(frac=1, random_state=42)  # Shuffle
print("Balanced class distribution:\n", df_balanced["Prediction"].value_counts())

# Step 5: Split Features and Target
X = df_balanced.drop("Prediction", axis=1)
y = df_balanced["Prediction"]  # 1D array for sklearn

# Step 6: Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)


# Step 7: Feature Scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
# Step 8: KNN Classifier with Elbow Method
k_values = list(range(1, 30, 2))
accuracy_values = []

for k in tqdm(k_values):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    accuracy_values.append(acc)

# Visualize K vs Accuracy
fig = px.line(x=k_values, y=accuracy_values, title="K Value vs Accuracy (KNN)")
fig.update_layout(xaxis_title="K Value", yaxis_title="Accuracy")
fig.show()

# Choose optimal K
optimal_k = k_values[np.argmax(accuracy_values)]
print(f"Optimal K: {optimal_k} with Accuracy: {max(accuracy_values):.4f}")


# Train final KNN model
knn_model = KNeighborsClassifier(n_neighbors=optimal_k)
knn_model.fit(X_train, y_train)
y_pred_knn = knn_model.predict(X_test)

# Step 9: SVM Classifier
svm_model = SVC(kernel='rbf', random_state=42)
svm_model.fit(X_train, y_train)
y_pred_svm = svm_model.predict(X_test)


# Step 10: Performance Evaluation
print("\n--- KNN Classification Report ---")
print(classification_report(y_test, y_pred_knn))
print("KNN Accuracy:", accuracy_score(y_test, y_pred_knn))

print("\n--- SVM Classification Report ---")
print(classification_report(y_test, y_pred_svm))
print("SVM Accuracy:", accuracy_score(y_test, y_pred_svm))
```

Explanation-

The workflow begins with **importing essential libraries**. `pandas` is used for reading and manipulating tabular data, allowing easy handling of rows, columns, and data types. `numpy` provides numerical computation support for array operations. `train_test_split` from `sklearn.model_selection` divides the dataset into training and testing subsets, enabling evaluation of model performance on unseen data and preventing overfitting. `StandardScaler` from `sklearn.preprocessing` standardizes feature values so that each has zero mean and unit variance, which is crucial for distance-based algorithms like KNN and algorithms sensitive to feature scales like SVM. `KNeighborsClassifier`implements the **K-Nearest Neighbors algorithm**, a non-parametric classifier that assigns a class label to a sample based on the majority vote of its nearest K neighbors in feature space. `SVC` implements the **Support Vector Machine algorithm**, which finds an optimal hyperplane that separates classes in a high-dimensional space and can capture nonlinear boundaries using kernels like RBF. `resample` is used to handle **class**

imbalance by upsampling the minority class, and `classification_report` and `accuracy_score` provide evaluation metrics. `plotly.express` is used to visualize the relationship between hyperparameter K and accuracy, and `tqdm` gives a progress bar for iterations.

The dataset is loaded using `pd.read_csv("emails.csv")`, and the irrelevant column `Email No.` is dropped because it does not contain predictive information. Initial exploration of the **class distribution** reveals an imbalance between spam (`1`) and ham (`0`) emails, which is common in real-world email datasets. Imbalanced datasets can cause classifiers to be biased toward the majority class, reducing the detection rate of the minority class. To address this, the minority class (spam) is **upsampled** using `resample` to create a larger sample equal to 80% of the majority class. The resulting balanced dataset is shuffled to randomize the order, ensuring fair training.

The features (`X`) and target (`y`) are separated. `X` contains all input variables, while `y` is the `Prediction` column representing spam (1) or ham (0). The dataset is then split into **training and testing sets** with an 80-20 ratio, using stratification to preserve the proportion of spam and ham in both sets. This ensures that both training and testing subsets are representative of the overall dataset.

Feature scaling is applied using `StandardScaler`, which standardizes each feature to zero mean and unit variance. This is essential because KNN calculates distances between points, and features with larger magnitudes would dominate smaller ones, potentially biasing the model. SVM also benefits from scaling as it relies on distances in the kernel space.

For the **KNN classifier**, a hyperparameter search is conducted to determine the optimal number of neighbors (K). KNN is a **distance-based classifier** where each sample is classified based on the majority label among its K nearest neighbors. Small K values can lead to overfitting (sensitive to noise), while large K values can cause underfitting (oversmoothing). The accuracy of KNN is evaluated for odd K values from 1 to 29, and the results are visualized with an interactive plot to identify the K with the highest accuracy. The final KNN model is trained with the optimal K and predictions are made on the test set.

The **SVM classifier** with an RBF kernel is trained in parallel. SVM is a **supervised learning algorithm** that finds an optimal hyperplane separating two classes. The RBF kernel allows the model to capture nonlinear relationships in the data, making it suitable for complex patterns in email features. Predictions are generated on the test set for comparison.

Finally, the **performance of both classifiers** is evaluated using `classification_report` and `accuracy_score`. The classification report provides precision (correct positive predictions / total predicted positives), recall (correct positive predictions / total actual positives), F1-score (harmonic mean of precision and recall), and support (number of samples per class). These metrics are particularly important in spam detection because false negatives (spam classified as ham) can have serious consequences. Accuracy provides an overall measure of correct predictions.

In summary, this pipeline demonstrates a **comprehensive supervised machine learning workflow for spam detection**. It includes **data preprocessing, handling class imbalance via upsampling, feature scaling, train-test splitting, hyperparameter tuning for KNN, training and comparing KNN and SVM classifiers, and detailed model evaluation**. KNN offers a simple, interpretable distance-based approach, while SVM captures nonlinear decision boundaries. Handling class imbalance ensures fair detection of spam, making the workflow robust and practical for real-world email filtering systems.