**import pandas as pd**

**from sklearn.model\_selection import train\_test\_split**

**from sklearn.preprocessing import LabelEncoder, StandardScaler**

1. import pandas as pd:

* This line imports the pandas library and gives it the alias pd. You’ll use this to handle and manipulate your dataset, such as reading data from a CSV file or performing data cleaning.

2. from sklearn.model\_selection import train\_test\_split:

* This line imports the train\_test\_split function from scikit-learn's model\_selection module. This function helps you divide your dataset into two subsets: one for training the model and one for testing its performance. This is crucial for evaluating how well your model generalizes to new, unseen data.

3. from sklearn.preprocessing import LabelEncoder, StandardScaler:

* LabelEncoder: Converts categorical labels into numerical format. For example, if you have a column with categories like 'yes' and 'no', LabelEncoder can convert them into 1s and 0s.
* StandardScaler: Standardizes features by removing the mean and scaling them to unit variance. This is important for algorithms that are sensitive to the scale of the data, such as gradient descent-based methods.

**# Load dataset**

**data = pd.read\_csv('Telco-Customer-Churn.csv')**

1. pd.read\_csv:

* This is a function from the pandas library used to read a CSV (Comma-Separated Values) file into a DataFrame. A DataFrame is a 2-dimensional labeled data structure with columns of potentially different types, similar to a table or a spreadsheet.

2.'Telco-Customer-Churn.csv':

* This is the filename of the CSV file you want to read. It should be located in the same directory as your script or notebook. If it's in a different directory, you need to provide the full path to the file.

3. data:

* This is the variable where the resulting DataFrame will be stored. After executing this line, data will contain the data read from the CSV file, and you can use it for further analysis or processing.

**# Handle missing values**

**data.bfill(inplace=True)**

1. data:

* This is the DataFrame containing your dataset that you previously loaded from the CSV file.

2.bfill():

* This is a method in the pandas library that stands for "backward fill". It fills missing values by propagating the next valid value backward.
* Backward Fill Missing Values: This method fills the NaN (missing) values in the DataFrame by using the next non-missing value along the column (or row if specified). If a value is missing, it will be replaced by the next valid value found below it in the same column.

3.inplace=True:

* This argument modifies the DataFrame in place, meaning that the changes are applied directly to the DataFrame without needing to assign the result back to data.

**# Encode categorical variables**

**label\_encoders = {}**

**for column in data.select\_dtypes(include=['object']).columns:**

**le = LabelEncoder()**

**data[column] = le.fit\_transform(data[column])**

**label\_encoders[column] = le**

1.label\_encoders = {}

This dictionary will hold the LabelEncoder object for each categorical column. This can be useful if you need to inverse transform the labels back to their original form later.

2.for column in data.select\_dtypes(include=['object']).columns:

data.select\_dtypes(include=['object']) selects columns in the DataFrame that have the data type object, which typically indicates categorical data. The .columns attribute gets the names of these columns.

3.le = LabelEncoder(): Creates a new instance of LabelEncoder.

4.data[column] = le.fit\_transform(data[column]): Fits the LabelEncoder to the categorical data in the column and transforms it into numerical labels, then assigns the transformed data back to the DataFrame.

5.label\_encoders[column] = le: Stores the LabelEncoder instance in the dictionary with the column name as the key.

**# Scale numerical variables**

**scaler = StandardScaler()**

**data[['tenure', 'MonthlyCharges', 'TotalCharges']] = scaler.fit\_transform(data[['tenure', 'MonthlyCharges', 'TotalCharges']])**

1.scaler = StandardScaler()

This creates an instance of the StandardScaler from scikit-learn. StandardScaler standardizes features by removing the mean and scaling to unit variance.

2. data[['tenure', 'MonthlyCharges', 'TotalCharges']] = scaler.fit\_transform(data[['tenure', 'MonthlyCharges', 'TotalCharges']])

data[['tenure', 'MonthlyCharges', 'TotalCharges']]: This selects the specified numerical columns from the DataFrame.

scaler.fit\_transform(data[['tenure', 'MonthlyCharges', 'TotalCharges']]): This method first fits the StandardScaler to the data (computing the mean and standard deviation for scaling) and then transforms the data by applying the scaling.

**# Split the data**

**X = data.drop('Churn', axis=1)**

**y = data['Churn']**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)**

1.X = data.drop('Churn', axis=1)

Features (X): This line creates a new DataFrame X that contains all columns from data except the 'Churn' column, which is the target variable we want to predict.

2. y = data['Churn']

Target Variable (y): This line creates a Series y that contains the values of the 'Churn' column, representing whether each customer has churned or not.

3. X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

Split the Data into Training and Testing Sets:

1. train\_test\_split(X, y, test\_size=0.2, random\_state=42):
   * + This function from sklearn.model\_selection splits the features (X) and target variable (y) into training and testing sets.
     + test\_size=0.2: Specifies that 20% of the data should be used for the test set and 80% for the training set.
     + random\_state=42: Ensures reproducibility of the split by setting a seed for the random number generator. Using the same random\_state value will produce the same split every time.

Result:

* X\_train: Training set features.
* X\_test: Test set features.
* y\_train: Training set target variable.
* y\_test: Test set target variable.

**import matplotlib.pyplot as plt**

**import seaborn as sns**

1.import matplotlib.pyplot as plt

* This imports the pyplot module from the matplotlib library and gives it the alias plt. Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. The pyplot module provides a MATLAB-like interface for making plots and graphs.

2.import seaborn as sns

* This imports the Seaborn library, which is built on top of Matplotlib and provides a high-level interface for drawing attractive and informative statistical graphics. Seaborn makes it easier to create complex visualizations and enhances the visual appeal of plots generated by Matplotlib.

**# Plot distributions of numerical features**

**sns.histplot(data['tenure'], bins=30, kde=True)**

**plt.show()**

1.Plotting a Histogram with KDE:

sns.histplot(data['tenure'], bins=30, kde=True)

sns.histplot: This function from Seaborn creates a histogram, which is a graphical representation of the distribution of numerical data. It allows you to visualize the frequency of different ranges of values.

data['tenure']: This selects the 'tenure' column from the DataFrame, which contains the numerical data you want to plot.

bins=30: This argument specifies the number of bins (intervals) for the histogram. More bins give a more detailed view of the distribution, while fewer bins give a more general view.

kde=True: This argument adds a Kernel Density Estimate (KDE) to the plot, which is a smoothed line that represents the distribution of the data. This helps in understanding the underlying distribution pattern of the data.

2.plt.show()

This function from Matplotlib displays the plot. Without plt.show(), the plot might not appear, especially when running scripts outside of a notebook environment.

**# Plot correlations**

**corr = data.corr()**

**sns.heatmap(corr, annot=True, cmap='coolwarm')**

**plt.show()**

1.Calculate Correlation Matrix:

corr = data.corr()

* data.corr(): This method calculates the correlation matrix for the DataFrame data. The correlation matrix is a table showing correlation coefficients between many variables. Each cell in the table shows the correlation between two variables. The value is between -1 and 1.

**1**: Perfect positive correlation.

**0**: No correlation.

**-1**: Perfect negative correlation.

2.Plot the Heatmap:

sns.heatmap(corr, annot=True, cmap='coolwarm')

* sns.heatmap: This function from Seaborn creates a heatmap to visualize the correlation matrix.
* corr: The correlation matrix to be visualized.
* annot=True: This argument annotates each cell in the heatmap with the correlation coefficient value.
* cmap='coolwarm': This argument sets the colormap of the heatmap. The 'coolwarm' colormap uses blue for negative correlations and red for positive correlations.

3.Display the Plot:

plt.show()

* This function from Matplotlib displays the plot. Without plt.show(), the plot might not appear, especially when running scripts outside of a notebook environment.

**from sklearn.linear\_model import LogisticRegression**

**from sklearn.preprocessing import StandardScaler**

**from sklearn.pipeline import Pipeline**

1.Importing Logistic Regression:

from sklearn.linear\_model import LogisticRegression

* LogisticRegression: This class from scikit-learn is used for logistic regression, a linear model for binary classification tasks. It models the probability that a given input belongs to a particular class.

2.Importing Standard Scaler:

from sklearn.preprocessing import StandardScaler

* StandardScaler: This class from scikit-learn is used to standardize features by removing the mean and scaling to unit variance. This is often a crucial preprocessing step for many machine learning algorithms.

3.Importing Pipeline:

from sklearn.pipeline import Pipeline

* Pipeline: This class from scikit-learn allows you to create a pipeline of sequential data processing and modeling steps. This is useful for bundling preprocessing steps with a machine learning algorithm into a single object.

**# Creating a pipeline to scale the data and apply logistic regression**

**pipeline = Pipeline([**

**('scaler', StandardScaler()), # Step 1: Scale the data**

**('log\_reg', LogisticRegression(max\_iter=200, solver='lbfgs')) # Step 2: Logistic regression with increased max\_iter**

**])**

1.Pipeline Creation:

pipeline = Pipeline([

('scaler', StandardScaler()),

('log\_reg', LogisticRegression(max\_iter=200, solver='lbfgs'))

])

* Pipeline: Combines multiple steps into a single object.
* Steps:
  + ('scaler', StandardScaler()):
    - StandardScaler: This step standardizes the features by removing the mean and scaling to unit variance.
  + ('log\_reg', LogisticRegression(max\_iter=200, solver='lbfgs')):
    - Logistic Regression: This step applies logistic regression for binary classification.
    - max\_iter=200: Increases the maximum number of iterations for the solver to converge. The default is 100, but sometimes more iterations are needed for the algorithm to converge.
    - solver='lbfgs': Specifies the algorithm to use in the optimization problem. 'lbfgs' is an optimization algorithm that's good for small datasets.

**# Fit the model on the training data**

**pipeline.fit(X\_train, y\_train)**

1.Pipeline Fitting:

* pipeline.fit(X\_train, y\_train): This command fits the entire pipeline to the training data. The pipeline object, which includes both preprocessing steps and the logistic regression model, is trained on the input features (X\_train) and the target variable (y\_train).

**# Make predictions on the test data**

**y\_pred = pipeline.predict(X\_test)**

1. Make Predictions:
   * pipeline.predict(X\_test): This command uses the trained pipeline to make predictions on the test data.

Detailed Breakdown:

1. Pipeline Object:
   * The pipeline object has been previously trained using pipeline.fit(X\_train, y\_train), which included scaling the features and fitting the logistic regression model.
2. Predict Method:
   * predict(X\_test): The predict method applies the same preprocessing steps to X\_test (scaling in this case) and then uses the trained logistic regression model to make predictions.
3. Output:
   * y\_pred: The result is an array of predicted class labels (e.g., 0 or 1) for each sample in X\_test.

**# Evaluate the model**

**from sklearn.metrics import classification\_report, confusion\_matrix**

**print(confusion\_matrix(y\_test, y\_pred))**

**print(classification\_report(y\_test, y\_pred))**

1.Import Evaluation Metrics:

from sklearn.metrics import classification\_report, confusion\_matrix

* confusion\_matrix: Computes a confusion matrix to evaluate the performance of a classification model.
* classification\_report: Generates a report that includes precision, recall, F1-score, and support for each class.

2.Confusion Matrix:

print(confusion\_matrix(y\_test, y\_pred))

* confusion\_matrix(y\_test, y\_pred): Computes the confusion matrix comparing the true labels (y\_test) with the predicted labels (y\_pred).
* Output: A 2x2 matrix for binary classification, where:
  + True Positives (TP): Number of correct positive predictions.
  + True Negatives (TN): Number of correct negative predictions.
  + False Positives (FP): Number of incorrect positive predictions.
  + False Negatives (FN): Number of incorrect negative predictions

[[TN FP]

[FN TP]]

3.Classification Report:

print(classification\_report(y\_test, y\_pred))

classification\_report(y\_test, y\_pred): Generates a detailed report showing the precision, recall, F1-score, and support for each class.

1.Precision: Precision is the ratio of correctly predicted positive observations to the total predicted positives.

Precision=TP/TP+FP​

TP (True Positives): Number of correct positive predictions.

FP (False Positives): Number of incorrect positive predictions.

2.Recall: The ratio of correctly predicted positive observations to all observations in the actual class.

Recall=TP/TP+FN

FN (False Negatives): Number of actual positives that were incorrectly predicted as negative.

3.F1-Score: The weighted average of Precision and Recall

F1-Score=2× Precision × Recall / Precision + Recall

4.Support: The number of actual occurrences of the class in the specified dataset

5. Accuracy:Accuracy is the ratio of correctly predicted observations to the total observations. It provides a measure of overall performance of the model.

6. Macro Average:Macro average calculates metrics for each class independently and then takes the average. It treats all classes equally, regardless of their support.

7. Weighted Average: Weighted average calculates metrics for each class and averages them, weighted by the number of true instances for each class. This gives more importance to classes with more instances.

**import matplotlib.pyplot as plt**

**from sklearn.metrics import roc\_auc\_score, roc\_curve**

1. Importing Libraries:
   * matplotlib.pyplot: A popular library for creating visualizations in Python. It provides functions for plotting graphs and charts.
   * roc\_auc\_score: A function from sklearn.metrics that computes the Area Under the Receiver Operating Characteristic Curve (ROC AUC) score, which provides an aggregate measure of performance across all classification thresholds.
   * roc\_curve: A function from sklearn.metrics that computes the Receiver Operating Characteristic (ROC) curve, which plots the true positive rate against the false positive rate at various thresholds.

Detailed Use:

1. ROC Curve:
   * The ROC curve is a graphical representation that shows the trade-off between the true positive rate (sensitivity) and false positive rate at various classification thresholds.
   * It helps in understanding the performance of a binary classifier across different decision thresholds.
2. ROC AUC Score:
   * The ROC AUC score quantifies the overall ability of the model to discriminate between the positive and negative classes. An AUC score of 1 indicates a perfect model, while an AUC score of 0.5 indicates a model with no discrimination ability (similar to random guessing).

**#Roc AUC**

**roc\_auc = roc\_auc\_score(y\_test, model.predict\_proba(X\_test)[:, 1])**

**print(f'ROC AUC: {roc\_auc:.2f}')**

1.Compute ROC AUC Score:

* roc\_auc\_score(y\_test, model.predict\_proba(X\_test)[:, 1]): Calculates the ROC AUC score for the model based on the test data.

2.Print ROC AUC Score:

* print(f'ROC AUC: {roc\_auc:.2f}'): Prints the ROC AUC score formatted to two decimal places.

**# Plot ROC curve**

**fpr, tpr, thresholds = roc\_curve(y\_test, model.predict\_proba(X\_test)[:, 1])**

**plt.plot(fpr, tpr, label=f'Logistic Regression (area = {roc\_auc:.2f})')**

**plt.xlabel('False Positive Rate')**

**plt.ylabel('True Positive Rate')**

**plt.title('ROC Curve')**

**plt.legend()**

**plt.show()**

1.Compute ROC Curve:

* fpr, tpr, thresholds = roc\_curve(y\_test, model.predict\_proba(X\_test)[:, 1]): Calculates the false positive rate (fpr), true positive rate (tpr), and the decision thresholds used to generate these rates.

2.Plot ROC Curve:

* plt.plot(fpr, tpr, label=f'Logistic Regression (area = {roc\_auc:.2f})'): Plots the ROC curve with fpr on the x-axis and tpr on the y-axis. Adds a label showing the ROC AUC score.
* plt.xlabel('False Positive Rate'): Labels the x-axis as "False Positive Rate."
* plt.ylabel('True Positive Rate'): Labels the y-axis as "True Positive Rate."
* plt.title('ROC Curve'): Adds a title to the plot.
* plt.legend(): Displays the legend with the label.
* plt.show(): Displays the plot.