adani | Institute of Digital Technology Management

Submission

of

Machine Learning-Lab Manual

By

Prathmesh B. Talhande (20231043)

PGDM 2023-25

In Partial Fulfillment of the Requirements for the

Post Graduate Diploma in Management (BDA)

At

Adani Institute of Digital Technology Management

(AICTE Approved Two-Year Full-Time Programme)

Plot No. 225, Opp. Maharaj Hotel Lane, Jamiyatpura Road,

S G Highway, PO: Jamiyatpura Gandhinagar - 382423, Gujarat.

1. Data preprocessing

Objective:

1) Finding and handling missing value.

2) Findling outliers and removing outliers.

3) Normalizing data using min max scalar.

4) standardizing data using standard scalar.

Input:

1. Finding and handling missing value.

- Making hypothetical data frame

```
import pandas as pd
import numpy as np

# Create the data dictionary
data = {
    'Product_ID': ['P001', 'P002', 'P003', 'P004', 'P005', 'P006', 'P007', 'P008', 'P009', 'P010'],
    'Production_Output': [500, 520, np.nan, 485, 510, 700, 480, 515, np.nan, 495],
    'Machine_Hours': [25, 26, 24, 28, 30, 35, 22, 27, np.nan, 25],
    'Labor_Cost': [300, 310, 295, 290, np.nan, 600, 280, 305, np.nan, 300],
    'Defect_Rate (%)': [2.5, 3.1, 2.0, np.nan, 4.5, 10.0, 2.9, 3.0, np.nan, 2.6],
    'Energy_Consumption (kWh)': [1500, 1520, 1480, 1450, 1550, 1700, 1490, 1515, np.nan, 1500]
}

# Convert the data dictionary into a pandas DataFrame
df = pd.DataFrame(data)

# Display the DataFrame
df
```

- Null value related Operations

```
# 1. Find Null Values
print("Null values in the dataset:\n", df.isnull().sum())
```

```
 #2. Handle Null Values (mean imputation)
df['Production_Output'] = df['Production_Output'].fillna(df['Production_Output'].mean())
df['Machine_Hours'] = df['Machine_Hours'].fillna(df['Machine_Hours'].mean())
df['Labor_Cost'] = df['Labor_Cost'].fillna(df['Labor_Cost'].mean())
df['Defect_Rate (%)'] = df['Defect_Rate (%)'].fillna(df['Defect_Rate (%)'].mean())
df['Energy_Consumption (kWh)'] = df['Energy_Consumption (kWh)'].fillna(df['Energy_Consumption (kWh)'].mean())
```

```
# Check if any null values remain
print("\nNull values after imputation:\n", df.isnull().sum())
```

2) Findling outliers and removing outliers.

- **Find Outliers using the IQR method**

```
# 3. Find Outliers using the IQR method
def find_outliers_IQR(data, column_name):
    Q1 = data[column_name].quantile(0.25)
    Q3 = data[column_name].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = data[(data[column_name] < lower_bound) | (data[column_name] > upper_bound)]
    return outliers

# Detecting outliers in numeric columns
print("\nOutliers in Production_Output:\n", find_outliers_IQR(df, 'Production_Output'))
print("\nOutliers in Machine_Hours:\n", find_outliers_IQR(df, 'Machine_Hours'))
print("\nOutliers in Labor_Cost:\n", find_outliers_IQR(df, 'Labor_Cost'))
print("\nOutliers in Defect_Rate (%):\n", find_outliers_IQR(df, 'Defect_Rate (%)'))
print("\nOutliers in Energy_Consumption (kWh):\n", find_outliers_IQR(df, 'Energy_Consumption (kWh)'))
```

- **Find and Remove Outliers using the IQR method**

```
# 3. Find and Remove Outliers using the IQR method
def remove_outliers_IQR(data, column_name):
    Q1 = data[column_name].quantile(0.25)
    Q3 = data[column_name].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Filter the data to remove outliers
    data_without_outliers = data[(data[column_name] >= lower_bound) & (data[column_name] <= upper_bound)]
    return data_without_outliers

# Remove outliers for all numeric columns
for column in ['Production_Output', 'Machine_Hours', 'Labor_Cost', 'Defect_Rate (%)', 'Energy_Consumption (kWh)']:
    df = remove_outliers_IQR(df, column)
df
```

3) Normalizing data using min max scalar.

- **Apply Normal Scaler (MinMaxScaler)**

```
from sklearn.preprocessing import MinMaxScaler
# Initialize scalers
normal_scaler = MinMaxScaler()
```

```
# Apply Normal Scaler (MinMaxScaler)
df_normal_scaled = pd.DataFrame(normal_scaler.fit_transform(df_numeric), columns=df_numeric.columns)
print("\nData after MinMaxScaler (Normal Scaling):\n")
df_normal_scaled
```

4) standardizing data using standard scalar.

- **Applying Standard Scaler**

```python
from sklearn.preprocessing import StandardScaler
```

```python
# Initialize scalers
standard_scaler = StandardScaler()
```

```python
#Applying Standard Scaler
df_standard_scaled = pd.DataFrame(standard_scaler.fit_transform(df_numeric), columns=df_numeric.columns)
print("\nData after StandardScaler:\n")
df_standard_scaled
```

Output:

1) Finding and handling missing value.

- **Making hypothetical data frame**

	Product_ID	Production_Output	Machine_Hours	Labor_Cost	Defect_Rate (%)	Energy_Consumption (kWh)
0	P001	500.0	25.0	300.0	2.5	1500.0
1	P002	520.0	26.0	310.0	3.1	1520.0
2	P003	NaN	24.0	295.0	2.0	1480.0
3	P004	485.0	28.0	290.0	NaN	1450.0
4	P005	510.0	30.0	NaN	4.5	1550.0
5	P006	700.0	35.0	600.0	10.0	1700.0
6	P007	480.0	22.0	280.0	2.9	1490.0
7	P008	515.0	27.0	305.0	3.0	1515.0
8	P009	NaN	NaN	NaN	NaN	NaN
9	P010	495.0	25.0	300.0	2.6	1500.0

- **Null value related Operations**

```
Null values in the dataset:
 Product_ID                 0
Production_Output           2
Machine_Hours               1
Labor_Cost                  2
Defect_Rate (%)             2
Energy_Consumption (kWh)    1
dtype: int64
```

```
Null values after imputation:
 Product_ID                  0
Production_Output            0
Machine_Hours                0
Labor_Cost                   0
Defect_Rate (%)              0
Energy_Consumption (kWh)     0
dtype: int64
```

2) Findling outliers and removing outliers.

- **Find Outliers using the IQR method**

```
Outliers in Production_Output:
   Product_ID  Production_Output  Machine_Hours  Labor_Cost  Defect_Rate (%)  \
5       P006              700.0           35.0       600.0             10.0

   Energy_Consumption (kWh)
5                    1700.0

Outliers in Machine_Hours:
   Product_ID  Production_Output  Machine_Hours  Labor_Cost  Defect_Rate (%)  \
5       P006              700.0           35.0       600.0             10.0

   Energy_Consumption (kWh)
5                    1700.0

Outliers in Labor_Cost:
   Product_ID  Production_Output  Machine_Hours  Labor_Cost  Defect_Rate (%)  \
5       P006              700.0           35.0       600.0             10.0

   Energy_Consumption (kWh)
5                    1700.0

Outliers in Defect_Rate (%):
   Product_ID  Production_Output  Machine_Hours  Labor_Cost  Defect_Rate (%)  \
5       P006              700.0           35.0       600.0             10.0

   Energy_Consumption (kWh)
5                    1700.0

Outliers in Energy_Consumption (kWh):
   Product_ID  Production_Output  Machine_Hours  Labor_Cost  Defect_Rate (%)  \
5       P006              700.0           35.0       600.0             10.0

   Energy_Consumption (kWh)
5                    1700.0
```

- **Find and Remove Outliers using the IQR method**

	Product_ID	Production_Output	Machine_Hours	Labor_Cost	Defect_Rate (%)	Energy_Consumption (kWh)
0	P001	500.000	25.0	300.0	2.5	1500.0
1	P002	520.000	26.0	310.0	3.1	1520.0
2	P003	525.625	24.0	295.0	2.0	1480.0
7	P008	515.000	27.0	305.0	3.0	1515.0
9	P010	495.000	25.0	300.0	2.6	1500.0

3) Normalizing data using min max scalar.

- **Apply Normal Scaler (MinMaxScaler)**

```
Data after MinMaxScaler (Normal Scaling):
```

	Production_Output	Machine_Hours	Labor_Cost	Defect_Rate (%)	Energy_Consumption (kWh)
0	0.090909	0.230769	0.062500	0.062500	0.200000
1	0.181818	0.307692	0.093750	0.137500	0.280000
2	0.207386	0.153846	0.046875	0.000000	0.120000
3	0.022727	0.461538	0.031250	0.228125	0.000000
4	0.136364	0.615385	0.171875	0.312500	0.400000
5	1.000000	1.000000	1.000000	1.000000	1.000000
6	0.000000	0.000000	0.000000	0.112500	0.160000
7	0.159091	0.384615	0.078125	0.125000	0.260000
8	0.207386	0.376068	0.171875	0.228125	0.291111
9	0.068182	0.230769	0.062500	0.075000	0.200000

4) standardizing data using standard scalar.

- **Applying Standard Scaler**

Data after StandardScaler:

	Production_Output	Machine_Hours	Labor_Cost	Defect_Rate (%)	Energy_Consumption (kWh)
0	-0.426362	-5.524841e-01	-0.389249	-0.609655	-0.353769
1	-0.093592	-2.599925e-01	-0.278035	-0.333585	-0.043143
2	0.000000	-8.449757e-01	-0.444857	-0.839713	-0.664396
3	-0.675940	3.249907e-01	-0.500464	0.000000	-1.130336
4	-0.259977	9.099738e-01	0.000000	0.310579	0.422797
5	2.901343	2.372432e+00	2.947175	2.841221	2.752497
6	-0.759133	-1.429959e+00	-0.611678	-0.425608	-0.509082
7	-0.176784	3.249907e-02	-0.333642	-0.379596	-0.120799
8	0.000000	-1.039139e-15	0.000000	0.000000	0.000000
9	-0.509555	-5.524841e-01	-0.389249	-0.563643	-0.353769

Result:

1) Finding and handling missing value.

Handling Missing Data with Mean Imputation:
Easy to handle: Ease of Implementation: Mean imputation is straightforward to implement and can be done quickly without complex algorithms.
Bias Introduction:
Potential Bias: Replacing missing values with the mean can introduce bias, especially if the data is not normally distributed. It can skew results toward the mean value and reduce variance.
Loss of Information:
Information Loss: Mean imputation does not take into account the relationship between the feature with missing values and other features. This means that we might lose valuable information that could be used for prediction.
Reduction of Variance: By filling missing values with the mean effectively reduce the variability in the data, which can affect models that rely on variance (like many statistical models).
Conclusion: Handling missing value with normal distributed data mean is effective while for skewed distribution median is effective.

2) Findling outliers and removing outliers.

Improved Model Performance
Reduction of Noise: Outliers can act as noise in the data, distorting the patterns that a model tries to learn. Removing them can lead to a cleaner dataset and better model performance.

Increased Accuracy: By removing outliers, the model may perform better on unseen data, leading to improved predictive accuracy and generalization.

3) Normalizing data using min max scalar.

normalize or **scale** features in a dataset. It transforms the values of numerical features so they fit within a specific range, typically **between 0 and 1**

4) standardizing data using standard scalar.

StandardScaler can significantly enhance the performance and reliability of machine learning models by ensuring that all features contribute equally and appropriately to the modeling process.

2. Statistical Distributions (Gaussian, PDF, CDF)

Objectives:

1) To Calculate Gaussian/Normal distribution.
2) To Perform kernel density estimation (KDE).
3) To Create Q-Q plots to compare data distribution with theoretical normal distribution.

Input:

1) To Calculate Gaussian/Normal distribution.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as stats

# Set the random seed for reproducibility
np.random.seed(42)

# Generate random samples for submarine-related data
# Assuming the data represents the depths (in meters) of submarines
depths = np.random.normal(loc=100, scale=15, size=1000)  # Mean = 100m, Std Dev = 15m

# Calculate mean and standard deviation
mean_depth = np.mean(depths)
std_dev_depth = np.std(depths)

print(f"Mean Depth: {mean_depth:.2f} m")
print(f"Standard Deviation of Depth: {std_dev_depth:.2f} m")

# 1. Plot Normal Distribution
plt.figure(figsize=(8, 6))
x = np.linspace(mean_depth - 4*std_dev_depth, mean_depth + 4*std_dev_depth, 1000)
normal_distribution = stats.norm.pdf(x, mean_depth, std_dev_depth)
plt.plot(x, normal_distribution, color='red', label='Normal Distribution', linewidth=2)

# Configure plot
plt.title('Normal Distribution')
plt.xlabel('Depth (m)')
plt.ylabel('Density')
plt.legend()
plt.grid()
plt.show()
```

2) To Perform kernel density estimation (KDE).

```
# 2. Perform Kernel Density Estimation (KDE)
plt.figure(figsize=(8, 6))
sns.kdeplot(depths, color='blue', label='KDE', fill=True)

# Configure plot
plt.title('Kernel Density Estimation (KDE)')
plt.xlabel('Depth (m)')
plt.ylabel('Density')
plt.legend()
plt.grid()
plt.show()
```

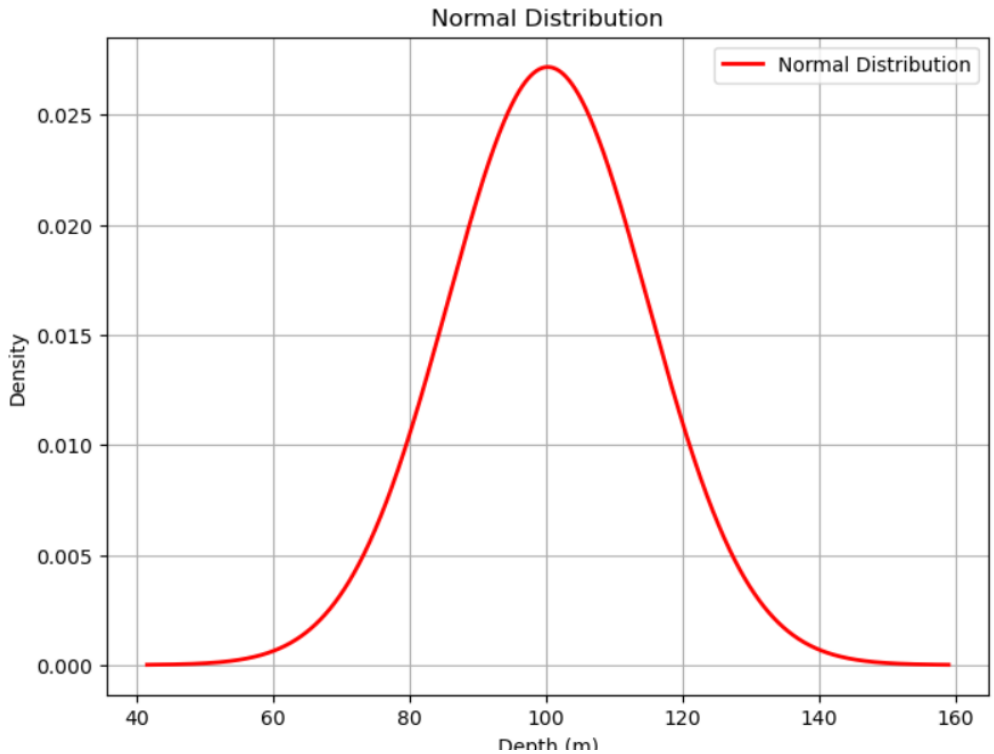
3) To Create Q-Q plots to compare data distribution with theoretical normal distribution.

```
# 3. Create Q-Q plot
plt.figure(figsize=(8, 6))
stats.probplot(depths, dist="norm", plot=plt)
plt.title('Q-Q Plot')
plt.xlabel('Theoretical Quantiles')
plt.ylabel('Sample Quantiles')
plt.grid()
plt.show()
```

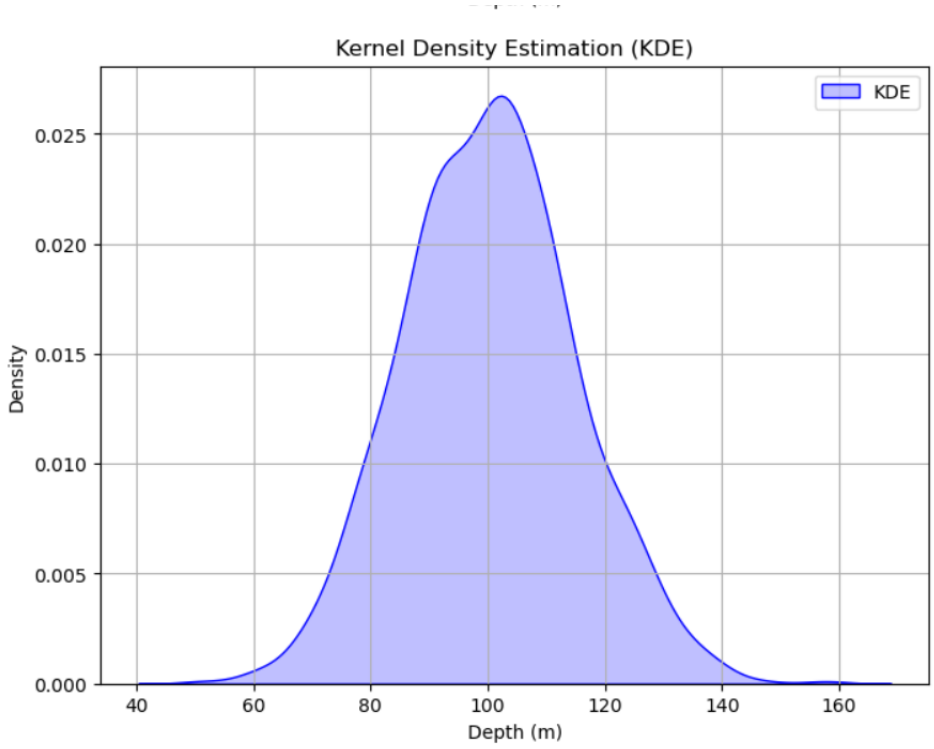
Output:

1) To Calculate Gaussian/Normal distribution.

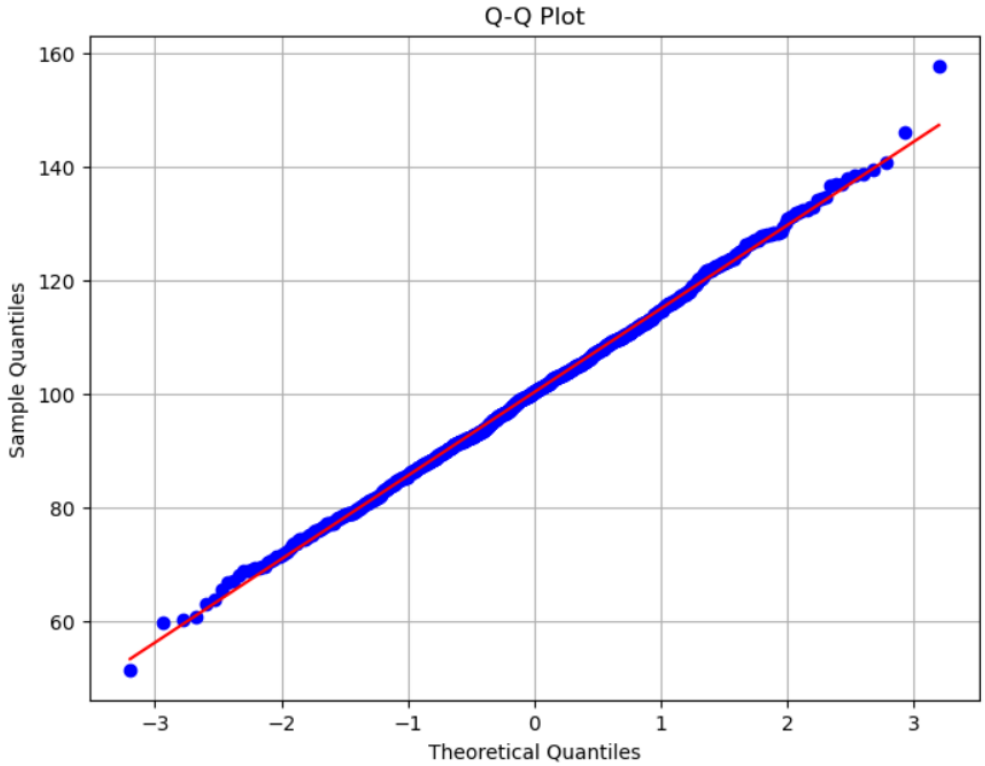
```
Mean Depth: 100.29 m
Standard Deviation of Depth: 14.68 m
```



2) To Perform kernel density estimation (KDE).



3) To Create Q-Q plots to compare data distribution with theoretical normal distribution.



Result:

1) To Calculate Gaussian/Normal distribution.

Normal Distribution Plot

The Normal Distribution plot visualizes the distribution of the data. Since the depths were generated from a normal distribution, the plot should show a bell-shaped curve centered around the mean.

The area under the curve represents the total probability (which equals 1). The graph confirms that the depths follow a normal distribution.

2) To Perform kernel density estimation (KDE).

Kernel Density Estimation (KDE)

The KDE plot provides a smoothed estimate of the probability density function of the depths.

The shape of the KDE should closely resemble the normal distribution curve indicating that the data is normally distributed.

Peaks in the KDE plot reflect where the values are concentrated while the tails show the probability of extreme values.

3) To Create Q-Q plots to compare data distribution with theoretical normal distribution.

Q-Q Plot

The Q-Q (Quantile-Quantile) plot compares the quantiles of the sample data against the quantiles of a theoretical normal distribution.

If the points in the Q-Q plot lie approximately along the diagonal line (45-degree line), it suggests that the sample data follows a normal distribution.

3. Logistic regression

Objective:

To perform classification on the basis of ('mean radius', 'mean texture', 'mean perimeter', 'mean area', 'mean smoothness', 'mean compactness', 'mean concavity', 'mean concave points', 'mean symmetry', 'mean fractal dimension', 'radius error', 'texture error', 'perimeter error', 'area error', 'smoothness error', 'compactness error', 'concavity error', 'concave points error', 'symmetry error', 'fractal dimension error', 'worst radius', 'worst texture', 'worst perimeter', 'worst area', 'worst smoothness', 'worst compactness', 'worst concavity', 'worst concave points', 'worst symmetry', 'worst fractal dimension') whether person have breast cancer (Bening & Malignant) or not.

Input:

- **Building model without L2 and with L2 regularization**

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, classification_report
import matplotlib.pyplot as plt
import seaborn as sns
from pathlib import Path

# Load the data
file_path = Path('C:/Users/roari/Downloads/Logistic regression/breast_cancer_dataset.xlsx')
data = pd.read_excel(file_path)  # Use read_excel since it's an .xlsx file

# Define features (X) and target (y)
X = data.iloc[:, :-1]
y = data.iloc[:, -1]

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the data into training (70%) and testing (30%) sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.3, random_state=42, stratify=y)

# Build logistic regression model without regularization
logreg_no_reg = LogisticRegression(penalty=None, solver='saga', random_state=42)
logreg_no_reg.fit(X_train, y_train)
y_pred_proba_no_reg = logreg_no_reg.predict_proba(X_test)[:, 1]

# Build logistic regression model with L2 regularization
logreg_l2 = LogisticRegression(penalty='l2', solver='liblinear', random_state=42)
logreg_l2.fit(X_train, y_train)
y_pred_proba_l2 = logreg_l2.predict_proba(X_test)[:, 1]
```

```python
# Plotting the predicted probabilities side by side
sns.set(style="whitegrid")
fig, ax = plt.subplots(1, 2, figsize=(15, 6))

# Plot for logistic regression without regularization
ax[0].scatter(range(len(y_pred_proba_no_reg)), y_pred_proba_no_reg, color='blue', marker='o', label='Predicted Probabilities')
ax[0].set_xlabel('Sample Index')
ax[0].set_ylabel('Predicted Probability')
ax[0].set_title('Logistic Regression - No Regularization')
ax[0].axhline(y=0.5, color='red', linestyle='--', label='Threshold at 0.5')
ax[0].legend()

# Plot for logistic regression with L2 regularization
ax[1].scatter(range(len(y_pred_proba_l2)), y_pred_proba_l2, color='blue', marker='o', label='Predicted Probabilities')
ax[1].set_xlabel('Sample Index')
ax[1].set_ylabel('Predicted Probability')
ax[1].set_title('Logistic Regression - L2 Regularization')
ax[1].axhline(y=0.5, color='red', linestyle='--', label='Threshold at 0.5')

# Highlight the points that change position after L2 regularization
changed_indices = np.abs(y_pred_proba_l2 - y_pred_proba_no_reg) > 0.01  # Threshold to identify significant changes
ax[1].scatter(np.where(changed_indices)[0], y_pred_proba_l2[changed_indices], color='orange', edgecolor='black', s=100, label='Changed Points')

ax[1].legend()

plt.tight_layout()
plt.show()
```

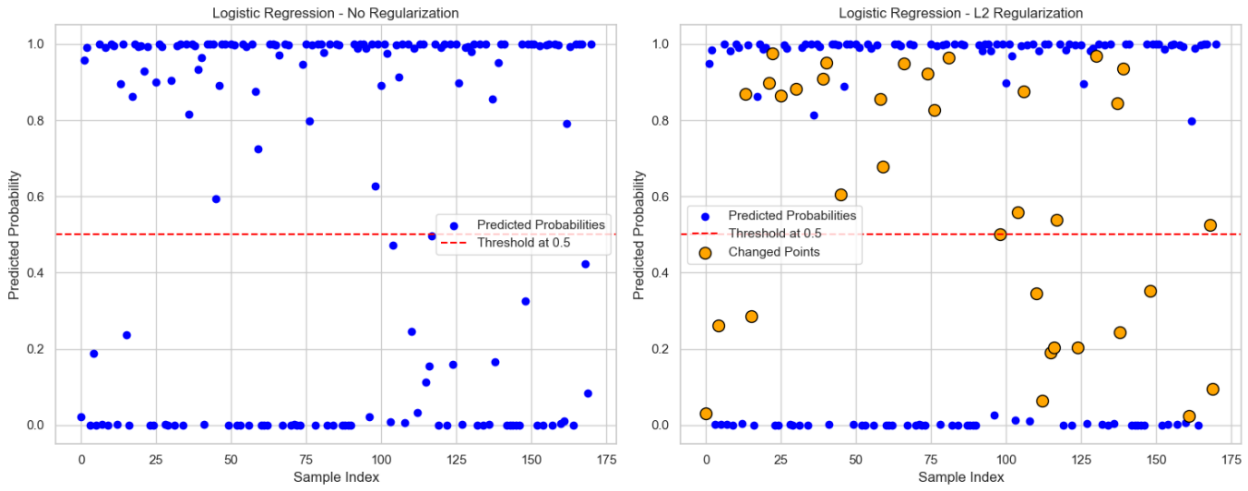
```python
# Output the results
print("Training Accuracy (No Regularization): {:.2f}".format(logreg_no_reg.score(X_train, y_train)))
print("Testing Accuracy (No Regularization): {:.2f}".format(accuracy_score(y_test, logreg_no_reg.predict(X_test))))
print("\nClassification Report (No Regularization):\n", classification_report(y_test, logreg_no_reg.predict(X_test)))

print("Training Accuracy (L2 Regularization): {:.2f}".format(logreg_l2.score(X_train, y_train)))
print("Testing Accuracy (L2 Regularization): {:.2f}".format(accuracy_score(y_test, logreg_l2.predict(X_test))))
print("\nClassification Report (L2 Regularization):\n", classification_report(y_test, logreg_l2.predict(X_test)))
```

- **Confusion matrix comparison**

```python
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Define the confusion matrices
conf_matrix_no_reg = np.array([[60, 4], [4, 103]])
conf_matrix_l2 = np.array([[62, 2], [1, 106]])

# Create a subplot with 1 row and 2 columns
fig, axes = plt.subplots(1, 2, figsize=(14, 7))

# Plot heatmap for confusion matrix without regularization
sns.heatmap(conf_matrix_no_reg, annot=True, fmt='d', cmap='YlGnBu', cbar=True,
            xticklabels=['Benign', 'Malignant'], yticklabels=['Benign', 'Malignant'],
            ax=axes[0], linewidths=0.5, linecolor='gray')
axes[0].set_xlabel('Predicted Label')
axes[0].set_ylabel('True Label')
axes[0].set_title('Confusion Matrix (No Regularization)')
axes[0].set_xticklabels(axes[0].get_xticklabels(), rotation=45, ha='right')
axes[0].set_yticklabels(axes[0].get_yticklabels(), rotation=0)

# Plot heatmap for confusion matrix with L2 regularization
sns.heatmap(conf_matrix_l2, annot=True, fmt='d', cmap='coolwarm', cbar=True,
            xticklabels=['Benign', 'Malignant'], yticklabels=['Benign', 'Malignant'],
            ax=axes[1], linewidths=0.5, linecolor='gray')
axes[1].set_xlabel('Predicted Label')
axes[1].set_ylabel('True Label')
axes[1].set_title('Confusion Matrix (L2 Regularization)')
axes[1].set_xticklabels(axes[1].get_xticklabels(), rotation=45, ha='right')
axes[1].set_yticklabels(axes[1].get_yticklabels(), rotation=0)

# Adjust layout
plt.tight_layout()
plt.show()
```

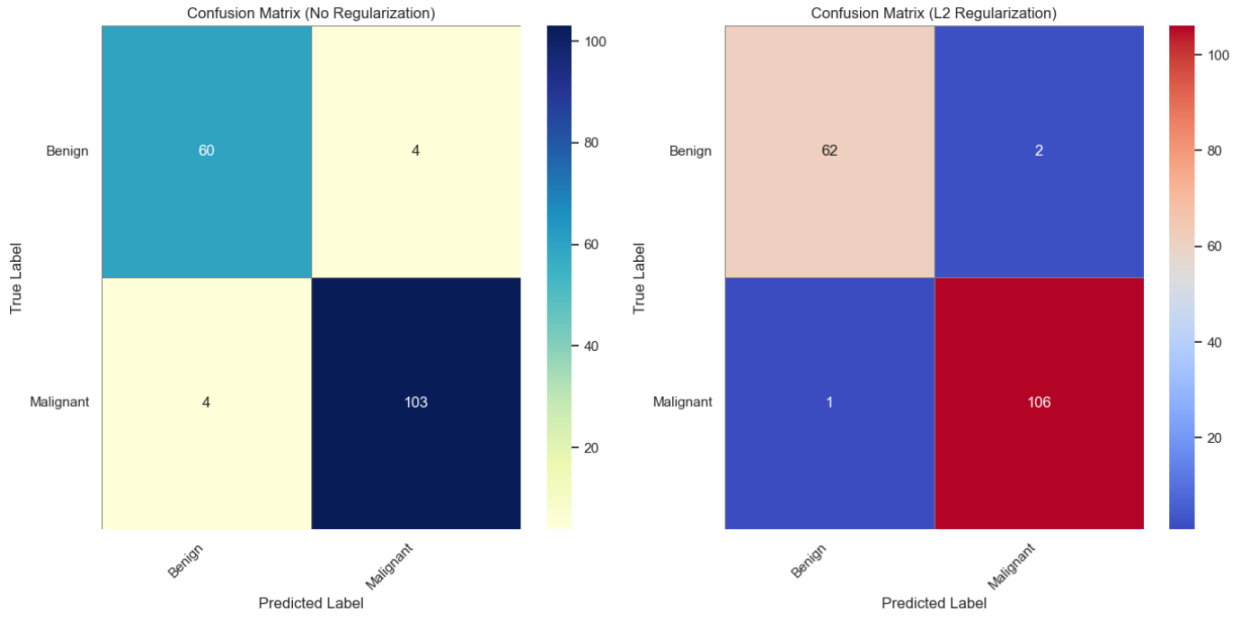
Output:

- **Building model without L2 and with L2 regularization**



```
Training Accuracy (No Regularization): 0.99
Testing Accuracy (No Regularization): 0.96

Classification Report (No Regularization):
              precision    recall  f1-score   support

           0       0.94      0.97      0.95        64
           1       0.98      0.96      0.97       107

    accuracy                           0.96       171
   macro avg       0.96      0.97      0.96       171
weighted avg       0.97      0.96      0.97       171

Training Accuracy (L2 Regularization): 0.99
Testing Accuracy (L2 Regularization): 0.98

Classification Report (L2 Regularization):
              precision    recall  f1-score   support

           0       0.98      0.97      0.98        64
           1       0.98      0.99      0.99       107

    accuracy                           0.98       171
   macro avg       0.98      0.98      0.98       171
weighted avg       0.98      0.98      0.98       171
```

- **Confusion matrix comparison**



Result:

Interpretation:
True Positives (TP): The number of malignant tumors correctly identified as malignant is higher with L2 regularization (106) compared to no regularization (103).
True Negatives (TN): The number of benign tumors correctly identified as benign is also slightly higher with L2 regularization (62) compared to no regularization (60).
False Positives (FP): The number of benign tumors incorrectly identified as malignant is lower with L2 regularization (2) compared to no regularization (4).
False Negatives (FN): The number of malignant tumors incorrectly identified as benign is also lower with L2 regularization (1) compared to no regularization (4).

Conclusion:

Analysis:
False Positives (FP): These can cause unnecessary worry and additional tests for patients. The reduction in FP with L2 regularization suggests that the model is less likely to falsely classify benign tumors as malignant.
False Negatives (FN): These are more dangerous as they can lead to missed diagnoses of cancer, potentially delaying critical treatment. The reduction in FN with L2 regularization indicates that the model is better at identifying malignant tumors.

4. KNN

Objective:

Classification Task:

The primary goal is to predict which drug type a patient is likely to be prescribed based on input features such as age, sodium-to-potassium ratio (Na_to_K), sex, blood pressure categories (BP_LOW, BP_NORMAL), cholesterol level, and other relevant factors.

Input:

- ### Importing data and loading

```
import pandas as pd

# Load the dataset
file_path = r"C:\Users\roari\Downloads\drug.csv"
data = pd.read_csv(file_path)
```

- ### Understand Data

```
data.shape
```

```
data.size
```

```
data.columns.tolist()
```

```
data.dtypes
```

```
# Count of null values in each column
null_count = data.isnull().sum()
print(null_count)
```

- ### Data Visualization and Distribution of data

```
import matplotlib.pyplot as plt
import seaborn as sns

# Plot histogram for 'Age'
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
sns.histplot(data['Age'], bins=10, kde=False, color='blue')
plt.title('Distribution of Age')
plt.xlabel('Age')
plt.ylabel('Frequency')

# Plot histogram for 'Na_to_K'
plt.subplot(1, 2, 2)
sns.histplot(data['Na_to_K'], bins=10, kde=False, color='green')
plt.title('Distribution of Na_to_K')
plt.xlabel('Na_to_K ratio')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```

```
import matplotlib.pyplot as plt
import seaborn as sns

# Function to plot pie chart
def plot_pie(column, title):
    plt.figure(figsize=(6, 6))
    data[column].value_counts().plot.pie(autopct='%1.1f%%', colors=sns.color_palette('pastel'))
    plt.title(title)
    plt.ylabel('')  # Hide the y-label for better visualization
    plt.show()

# Plot pie charts for categorical columns
plot_pie('Sex', 'Proportion of Sex')
plot_pie('BP', 'Proportion of BP Levels')
plot_pie('Cholesterol', 'Proportion of Cholesterol Levels')
plot_pie('Drug_Type', 'Proportion of Drug Types')
```

- **Performing Encoding**

```
import pandas as pd

# Assuming 'data' is your DataFrame loaded from the CSV
# Perform one-hot encoding
data_encoded = pd.get_dummies(data, columns=['Sex', 'BP', 'Cholesterol', 'Drug_Type'], drop_first=True)

# Display the first few rows of the encoded DataFrame
data_encoded.head()
```

```
# Convert True/False to 1/0
data_encoded = data_encoded.astype(int)

# Display the first few rows of the updated DataFrame
data_encoded.head()
```

- **Standardizing Data**

```
from sklearn.preprocessing import StandardScaler

# Initialize the scaler
scaler = StandardScaler()

# Standardize the features
data_encoded[['Age', 'Na_to_K']] = scaler.fit_transform(data_encoded[['Age', 'Na_to_K']])

# Display the first few rows of the standardized DataFrame
data_encoded.head()
```

- **Model Building**

```python
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc
import matplotlib.pyplot as plt

# Assuming data_encoded is your DataFrame after one-hot encoding and standardization

# Split the data into features and target variable
X = data_encoded.drop(columns=['Drug_Type_drugA', 'Drug_Type_drugB', 'Drug_Type_drugC', 'Drug_Type_drugX'])
y = data_encoded[['Drug_Type_drugA', 'Drug_Type_drugB', 'Drug_Type_drugC', 'Drug_Type_drugX']].idxmax(axis=1)

# Initialize the KNN classifier
knn = KNeighborsClassifier()

# Set up the hyperparameter grid
param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan']
}

# Perform Grid Search for hyperparameter tuning with cross-validation
grid_search = GridSearchCV(knn, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X, y)

# Best parameters
best_params = grid_search.best_params_
print(f"Best Parameters: {best_params}")

# Train the model with the best parameters
best_knn = KNeighborsClassifier(**best_params)
```

```python
# Perform K-Fold Cross-Validation
cv_scores = cross_val_score(best_knn, X, y, cv=5)  # 5-fold cross-validation
print(f"Cross-Validation Scores: {cv_scores}")
print(f"Mean Cross-Validation Score: {cv_scores.mean():.2f}")

# Fit the model on the entire dataset
best_knn.fit(X, y)

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

# Evaluate the model on the test set
y_pred = best_knn.predict(X_test)

# Metrics
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=['Drug_Type_drugA', 'Drug_Type_drugB', 'Drug_Type_drugC', 'Drug_Type_drugX']))

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:")
print(conf_matrix)
```

```python
# ROC Curve
y_scores = best_knn.predict_proba(X_test)  # Get predicted probabilities
fpr, tpr, thresholds = roc_curve(y_test, y_scores[:, 1], pos_label='Drug_Type_drugB')  # Example for one class

# Plot ROC Curve
plt.figure(figsize=(10, 6))
plt.plot(fpr, tpr, label='ROC Curve (area = %0.2f)' % auc(fpr, tpr))
plt.plot([0, 1], [0, 1], 'k--')  # Diagonal line
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```

Output:

- **Importing data and loading**

	Age	Sex	BP	Cholesterol	Na_to_K	Drug_Type
0	23	F	HIGH	HIGH	25.355	DrugY
1	47	M	LOW	HIGH	13.093	drugC
2	47	M	LOW	HIGH	10.114	drugC
3	28	F	NORMAL	HIGH	7.798	drugX
4	61	F	LOW	HIGH	18.043	DrugY
...	...	...	...	...	...	...
195	56	F	LOW	HIGH	11.567	drugC
196	16	M	LOW	HIGH	12.006	drugC
197	52	M	NORMAL	HIGH	9.894	drugX
198	23	M	NORMAL	NORMAL	14.020	drugX
199	40	F	LOW	NORMAL	11.349	drugX

200 rows × 6 columns

description of each column's metadata:
Age: Numerical value representing the patient's age.
Sex: Categorical value indicating the patient's gender (M for male, F for female).
BP: Categorical value representing the patient's blood pressure level (LOW, NORMAL, HIGH).
Cholesterol: Categorical value showing the patient's cholesterol level (NORMAL, HIGH).
Na_to_K: Numerical value representing the sodium-to-potassium ratio in the patient's blood.
Drug_Type: Categorical value indicating the type of drug prescribed (DrugY, drugC, drugX, etc.).

- **Understand Data**

```
Finding shape of data
```

```
data.shape
```

```
(200, 6)
```

```
data.size
```

```
1200
```

```
Finding the columns
```

```
data.columns.tolist()
```

```
['Age', 'Sex', 'BP', 'Cholesterol', 'Na_to_K', 'Drug_Type']
```

```
data.dtypes
```

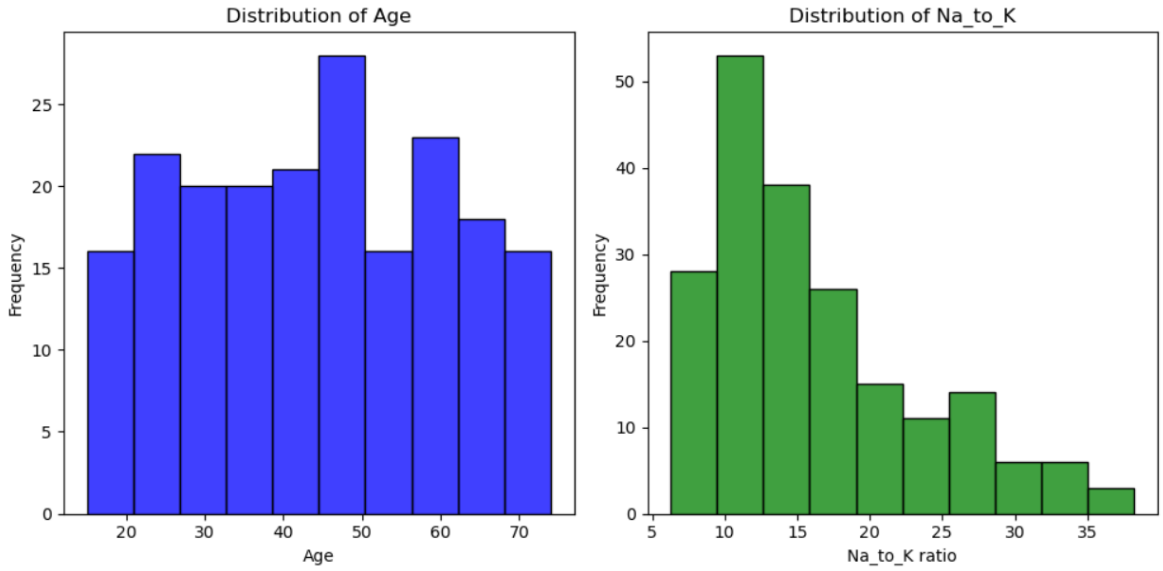
```
Age              int64
Sex             object
BP              object
Cholesterol     object
Na_to_K        float64
Drug_Type       object
dtype: object
```

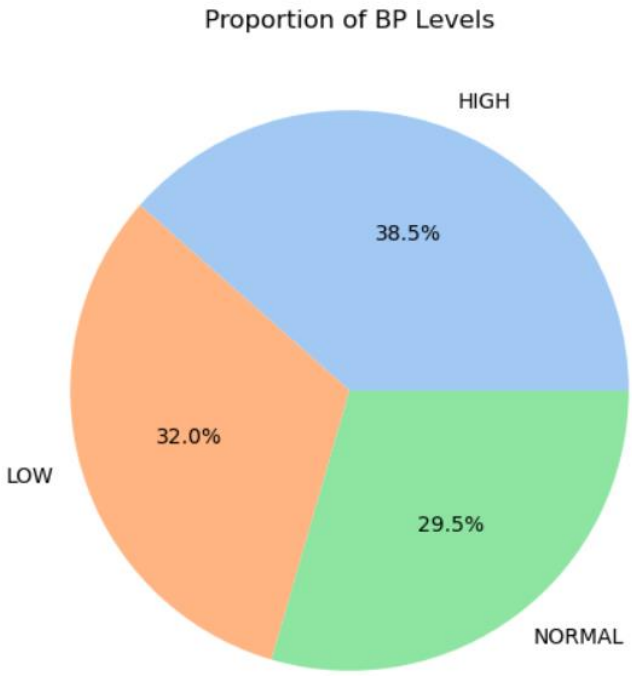
```
Finding null values
```

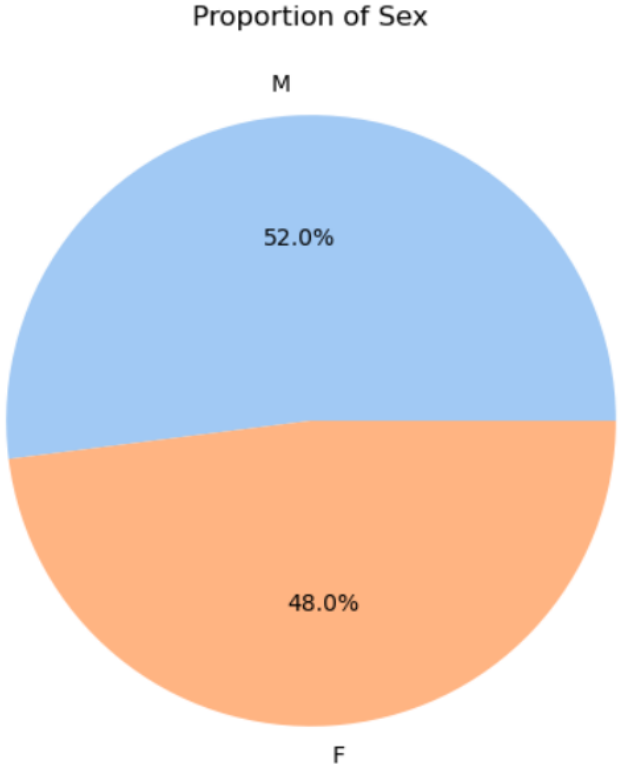
```
# Count of null values in each column
null_count = data.isnull().sum()
print(null_count)
```

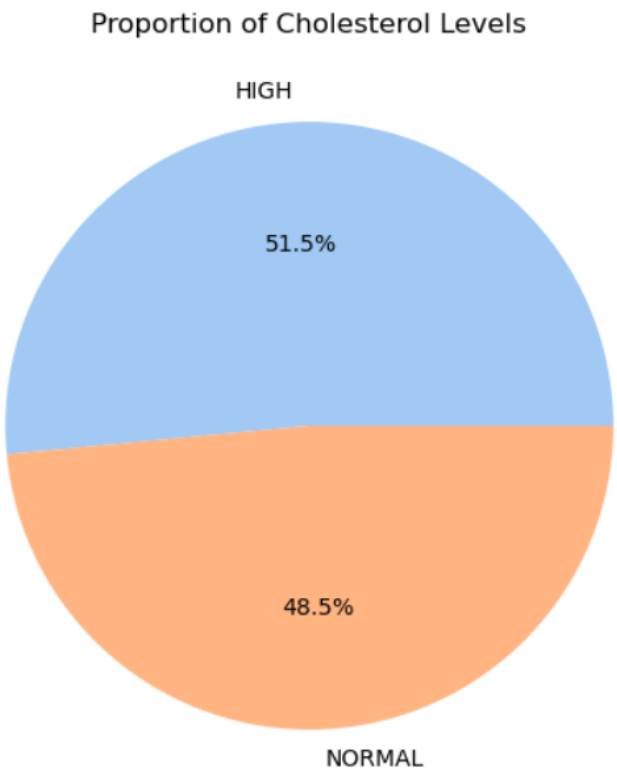
```
Age            0
Sex            0
BP             0
Cholesterol    0
Na_to_K        0
Drug_Type      0
dtype: int64
```

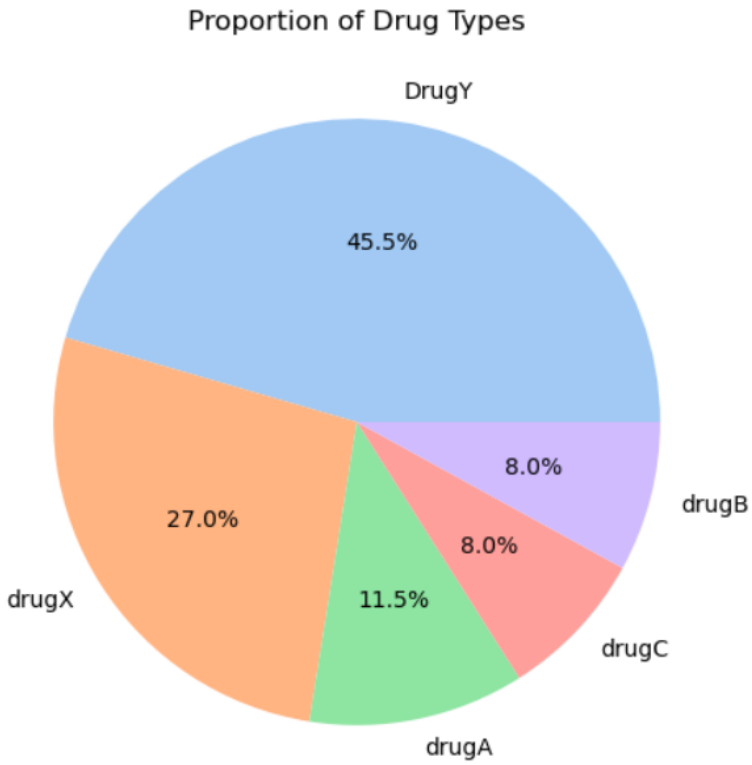
- **Data Visualization and Distribution of data**




Proportion of BP Levels
HIGH
38.5%
LOW
32.0%
29.5%
NORMAL


Proportion of Sex
M
52.0%
48.0%
F





- **Performing Encoding**

	Age	Na_to_K	Sex_M	BP_LOW	BP_NORMAL	Cholesterol_NORMAL	Drug_Type_drugA	Drug_Type_drugB	Drug_Type_drugC	Drug_Type_drugX
0	23	25.355	False	False	False	False	False	False	False	False
1	47	13.093	True	True	False	False	False	False	True	False
2	47	10.114	True	True	False	False	False	False	True	False
3	28	7.798	False	False	True	False	False	False	False	True
4	61	18.043	False	True	False	False	False	False	False	False

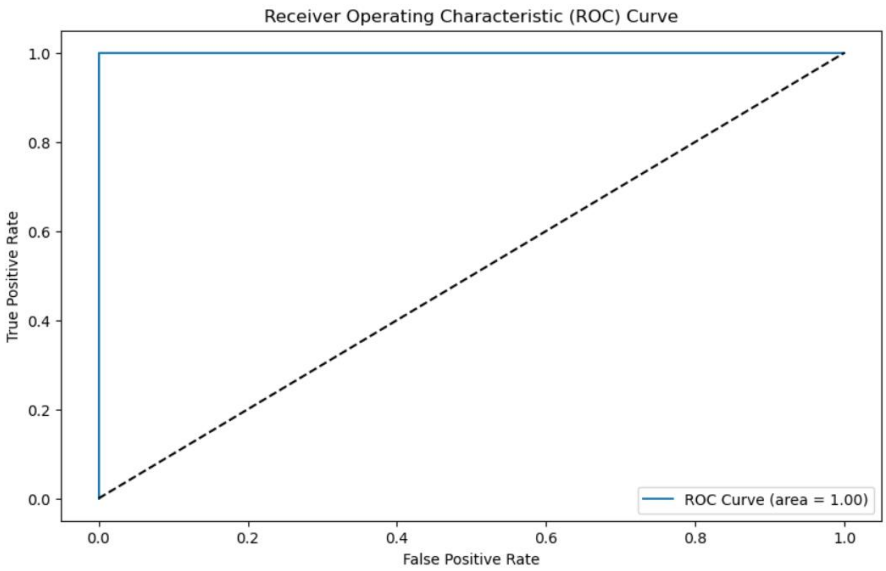
	Age	Na_to_K	Sex_M	BP_LOW	BP_NORMAL	Cholesterol_NORMAL	Drug_Type_drugA	Drug_Type_drugB	Drug_Type_drugC	Drug_Type_drugX
0	23	25	0	0	0	0	0	0	0	0
1	47	13	1	1	0	0	0	0	1	0
2	47	10	1	1	0	0	0	0	1	0
3	28	7	0	0	1	0	0	0	0	1
4	61	18	0	1	0	0	0	0	0	0

- **Standardizing Data**

	Age	Na_to_K	Sex_M	BP_LOW	BP_NORMAL	Cholesterol_NORMAL	Drug_Type_drugA	Drug_Type_drugB	Drug_Type_drugC	Drug_Type_drugX
0	-1.291591	1.311022	0	0	0	0	0	0	0	0
1	0.162699	-0.359957	1	1	0	0	0	0	1	0
2	0.162699	-0.777701	1	1	0	0	0	0	1	0
3	-0.988614	-1.195446	0	0	1	0	0	0	0	1
4	1.011034	0.336285	0	1	0	0	0	0	0	0

Model Building

```
Best Parameters: {'metric': 'euclidean', 'n_neighbors': 3, 'weights': 'distance'}
Cross-Validation Scores: [0.9   0.925 1.    0.9   0.975]
Mean Cross-Validation Score: 0.94

Classification Report:
                  precision    recall  f1-score   support

Drug_Type_drugA       1.00      1.00      1.00        21
Drug_Type_drugB       1.00      1.00      1.00         3
Drug_Type_drugC       1.00      1.00      1.00         5
Drug_Type_drugX       1.00      1.00      1.00        11

       accuracy                           1.00        40
      macro avg       1.00      1.00      1.00        40
   weighted avg       1.00      1.00      1.00        40


Confusion Matrix:
[[21  0  0  0]
 [ 0  3  0  0]
 [ 0  0  5  0]
 [ 0  0  0 11]]
```



Result:

The classification task aimed to predict the type of drug a patient is likely to be prescribed based on features like age, sex, blood pressure, cholesterol levels, and sodium-to-potassium ratio. The best model parameters were found to be {'metric': 'euclidean', 'n_neighbors': 3, 'weights': 'distance'}.

- **Cross-Validation Performance**: The model showed robust performance with cross-validation scores ranging from 0.9 to 1.0, with a mean score of **0.94**.
- **Classification Report**: The model achieved perfect precision, recall, and F1-scores of **1.00** for all drug types (DrugA, DrugB, DrugC, DrugX), indicating flawless classification across all categories in the test data.
- **Accuracy**: The overall accuracy was **100%**, with all predictions matching the true labels in the dataset.
- **Confusion Matrix**: The confusion matrix shows no misclassifications, with all predictions perfectly aligned with the actual drug types.

In conclusion, the model provided highly accurate predictions for the drug prescription classification task, demonstrating its effectiveness with the chosen parameters and dataset.

5. SVM

Objective:

The primary objective of this project was to develop a robust machine learning model using Support Vector Classification (SVC) to accurately predict diabetes in individuals based on clinical features. The study aimed to address the challenge of effectively classifying diabetes status (diabetic vs. non-diabetic) using a limited dataset while optimizing model performance through hyperparameter tuning. By implementing techniques such as one-hot encoding and grid search for hyperparameter optimization, the model achieved a high accuracy of 92.5% with well-balanced precision and recall rates for both classes, demonstrating its potential utility in clinical settings for early diabetes diagnosis and management.

Input:

- **Loading dataset**

```
import pandas as pd

# Load the dataset
file_path = r"C:\Users\roari\Downloads\DiabetesNew\diabetes_prediction.csv"
df = pd.read_csv(file_path)
```

```
df.head()
```

- **Understanding Data**

```
df.shape
```

```
df.size
```

```
df.columns.tolist()
```

```
df.describe()
```

```
df.dtypes
```

```
df.isnull().sum()
```

```
df.info()
```

```
duplicate_count = df.duplicated().sum()
```

```
# Drop duplicate rows
df_no_duplicates = df.drop_duplicates()

# Check the shape after dropping duplicates
print("Shape after dropping duplicates:")
df_no_duplicates.shape
```

- **Data Visualization**

```
import seaborn as sns
import matplotlib.pyplot as plt

# Set the style of Seaborn
sns.set(style="whitegrid")

# Create histograms for numerical features
numerical_features = df.select_dtypes(include=['float64', 'int64']).columns.tolist()

plt.figure(figsize=(15, 10))
for i, feature in enumerate(numerical_features, 1):
    plt.subplot(3, 3, i)  # Adjust the number of rows and columns as needed
    sns.histplot(df[feature], bins=30, kde=True)
    plt.title(f'Distribution of {feature}')
    plt.xlabel(feature)
    plt.ylabel('Frequency')

plt.tight_layout()
plt.show()

# Create histograms for categorical features
categorical_features = df.select_dtypes(include=['object']).columns.tolist()

plt.figure(figsize=(10, 5))
for feature in categorical_features:
    plt.figure(figsize=(8, 4))
    sns.countplot(data=df, x=feature, order=df[feature].value_counts().index)
    plt.title(f'Distribution of {feature}')
    plt.xlabel(feature)
    plt.ylabel('Count')
    plt.xticks(rotation=45)
    plt.show()
```

- **Understanding Target**

```
import seaborn as sns
import matplotlib.pyplot as plt

# Set the aesthetic style of the plots
sns.set(style="whitegrid")

# Histogram for the diabetes distribution
plt.figure(figsize=(10, 5))
sns.histplot(df['diabetes'], bins=2, kde=False)
plt.title('Distribution of Diabetes')
plt.xlabel('Diabetes (0 = No, 1 = Yes)')
plt.ylabel('Frequency')
plt.xticks([0, 1], ['No Diabetes', 'Diabetes'])  # Rename x-ticks
plt.show()

# Pie chart for diabetes distribution
plt.figure(figsize=(8, 8))
diabetes_counts = df['diabetes'].value_counts()
plt.pie(diabetes_counts, labels=diabetes_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Diabetes Distribution')
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.legend(['No Diabetes', 'Diabetes'])  # Adding a legend for clarity
plt.show()
```

- **Taking Balance Data**

```
# Separate the majority and minority classes
majority_class = df[df['diabetes'] == 0]
minority_class = df[df['diabetes'] == 1]

# Undersample the majority class
majority_class_undersampled = majority_class.sample(n=8000, random_state=42)

# Combine the undersampled majority class with the minority class
balanced_df = pd.concat([majority_class_undersampled, minority_class])

# Check the new distribution
new_distribution = balanced_df['diabetes'].value_counts()
new_distribution_percentage = new_distribution / new_distribution.sum() * 100

print("New Diabetes Count and Percentage:")
print(new_distribution)
print(new_distribution_percentage)
```

- **Encoding**

```
# Perform one-hot encoding for the nominal categorical features
encoded_df = pd.get_dummies(balanced_df, columns=['gender', 'smoking_history'], drop_first=True)

# Display the first few rows of the encoded dataframe
encoded_df.head()
```

```
nvert boolean values to integers (0 and 1)
ded_df[encoded_df.columns[encoded_df.columns.str.startswith('gender_')]] = encoded_df[encoded_df.columns[encoded_df.columns.str.startswith('gender_')]].as
ded_df[encoded_df.columns[encoded_df.columns.str.startswith('smoking_history_')]] = encoded_df[encoded_df.columns[encoded_df.columns.str.startswith('smoki

splay the first few rows of the updated encoded dataframe
ded_df.head()
```

- **Model Building:**

```
# Subsampling the data (200 samples from each class)
diabetic_data = encoded_df[encoded_df['diabetes'] == 1].sample(200, random_state=42)
non_diabetic_data = encoded_df[encoded_df['diabetes'] == 0].sample(200, random_state=42)

# Combine the subsamples
reduced_df = pd.concat([diabetic_data, non_diabetic_data])

# Define features and target variable for the reduced dataset
X_reduced = reduced_df.drop('diabetes', axis=1)  # Features
y_reduced = reduced_df['diabetes']               # Target variable

# Split the reduced data into training and testing sets (80% train, 20% test)
X_train_reduced, X_test_reduced, y_train_reduced, y_test_reduced = train_test_split(X_reduced, y_reduced, test_size=0.2, random_state=42)

# Import GridSearchCV
from sklearn.model_selection import GridSearchCV

# Define a parameter grid for SVM
param_grid = {
    'C': [0.1, 1, 10],       # Regularization parameter
    'kernel': ['linear', 'rbf'],  # Kernel type
    'gamma': ['scale', 'auto'],   # Kernel coefficient for 'rbf'
}

# Create a GridSearchCV object with 3-fold cross-validation
grid = GridSearchCV(SVC(probability=True), param_grid, refit=True, verbose=1, cv=3)

# Fit the model on the reduced dataset
grid.fit(X_train_reduced, y_train_reduced)

# Print the best parameters found
print(f'Best Parameters: {grid.best_params_}')
```

```
# Make predictions with the best model
y_pred_reduced = grid.predict(X_test_reduced)

# Calculate accuracy
accuracy_reduced = grid.score(X_test_reduced, y_test_reduced)
print(f'Reduced Test Accuracy: {accuracy_reduced}')

# Generate classification report
report_reduced = classification_report(y_test_reduced, y_pred_reduced, output_dict=True)
print("\nClassification Report (Reduced Dataset):")
print(report_reduced)

# Confusion Matrix for reduced data
conf_matrix_reduced = confusion_matrix(y_test_reduced, y_pred_reduced)
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix_reduced, annot=True, fmt='d', cmap='Blues', xticklabels=['No Diabetes', 'Diabetes'], yticklabels=['No Diabetes', 'Diabetes'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix (Reduced Data)')
plt.show()
```

```
# Calculate ROC Curve for reduced data
y_prob_reduced = grid.predict_proba(X_test_reduced)[:, 1]  # Get probabilities for the positive class
fpr_reduced, tpr_reduced, thresholds_reduced = roc_curve(y_test_reduced, y_prob_reduced)
roc_auc_reduced = roc_auc_score(y_test_reduced, y_prob_reduced)

# Plot ROC Curve for reduced data
plt.figure(figsize=(8, 6))
plt.plot(fpr_reduced, tpr_reduced, color='blue', label='ROC curve (area = %0.2f)' % roc_auc_reduced)
plt.plot([0, 1], [0, 1], color='red', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve (Reduced Data)')
plt.legend(loc='lower right')
plt.show()
```

Output:

- **Loading dataset**

	gender	age	hypertension	heart_disease	smoking_history	bmi	HbA1c_level	blood_glucose_level	diabetes
0	Female	80.0	0	1	never	25.19	6.6	140	0
1	Female	54.0	0	0	No Info	27.32	6.6	80	0
2	Male	28.0	0	0	never	27.32	5.7	158	0
3	Female	36.0	0	0	current	23.45	5.0	155	0
4	Male	76.0	1	1	current	20.14	4.8	155	0

- **Understanding Data**

Finding shape of data

```
df.shape
```

```
(100000, 9)
```

Finding size of the data

```
df.size
```

```
900000
```

Finding column names

```
df.columns.tolist()
```

```
['gender',
 'age',
 'hypertension',
 'heart_disease',
 'smoking_history',
 'bmi',
 'HbA1c_level',
 'blood_glucose_level',
 'diabetes']
```

```
df.describe()
```

	age	hypertension	heart_disease	bmi	HbA1c_level	blood_glucose_level	diabetes
count	100000.000000	100000.00000	100000.000000	100000.000000	100000.000000	100000.000000	100000.000000
mean	41.885856	0.07485	0.039420	27.320767	5.527507	138.058060	0.085000
std	22.516840	0.26315	0.194593	6.636783	1.070672	40.708136	0.278883
min	0.080000	0.00000	0.000000	10.010000	3.500000	80.000000	0.000000
25%	24.000000	0.00000	0.000000	23.630000	4.800000	100.000000	0.000000
50%	43.000000	0.00000	0.000000	27.320000	5.800000	140.000000	0.000000
75%	60.000000	0.00000	0.000000	29.580000	6.200000	159.000000	0.000000
max	80.000000	1.00000	1.000000	95.690000	9.000000	300.000000	1.000000

Finding the data types

```
df.dtypes
```

```
gender                object
age                  float64
hypertension           int64
heart_disease          int64
smoking_history       object
bmi                  float64
HbA1c_level          float64
blood_glucose_level    int64
diabetes               int64
dtype: object
```

Finding the null count

```
df.isnull().sum()
```

```
gender                0
age                   0
hypertension          0
heart_disease         0
smoking_history       0
bmi                   0
HbA1c_level           0
blood_glucose_level   0
diabetes              0
dtype: int64
```

Finding the overall information of dataset

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 9 columns):
 #   Column               Non-Null Count   Dtype  
---  ------               --------------   -----  
 0   gender               100000 non-null  object 
 1   age                  100000 non-null  float64
 2   hypertension         100000 non-null  int64  
 3   heart_disease        100000 non-null  int64  
 4   smoking_history      100000 non-null  object 
 5   bmi                  100000 non-null  float64
 6   HbA1c_level          100000 non-null  float64
 7   blood_glucose_level  100000 non-null  int64  
 8   diabetes             100000 non-null  int64  
dtypes: float64(3), int64(4), object(2)
memory usage: 6.9+ MB
```

Finding the duplicate values

```
duplicate_count = df.duplicated().sum()
```

```
duplicate_count
```

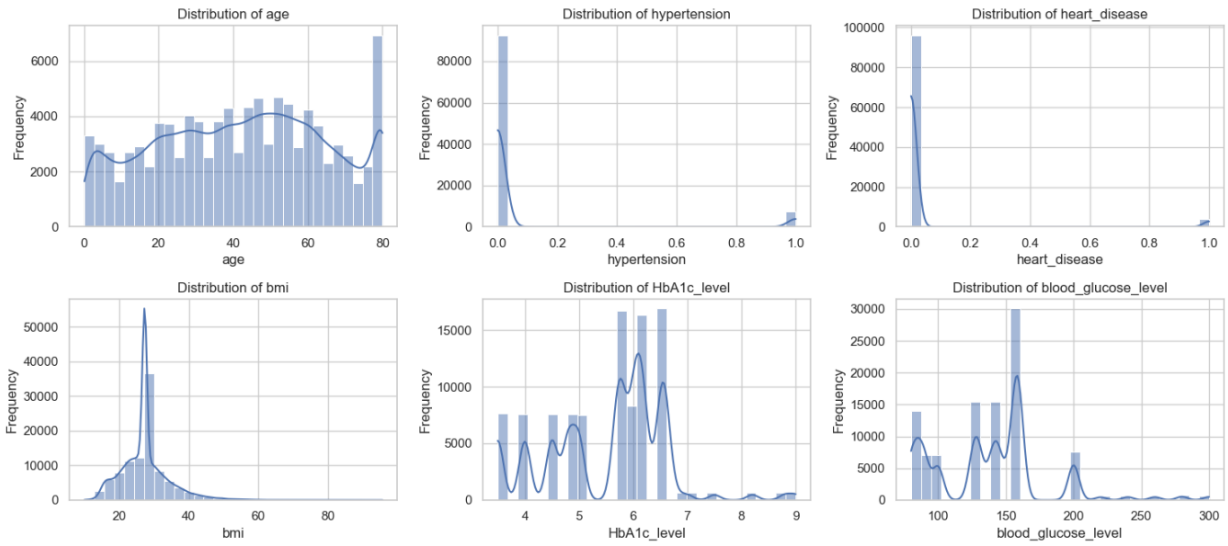
```
3854
```

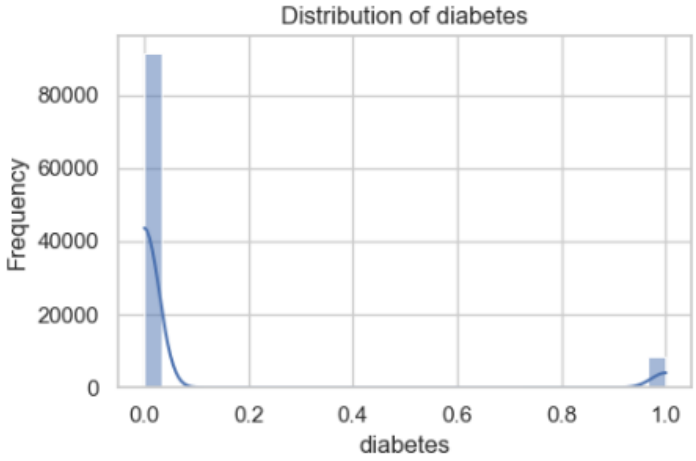
Dropping the duplicate values

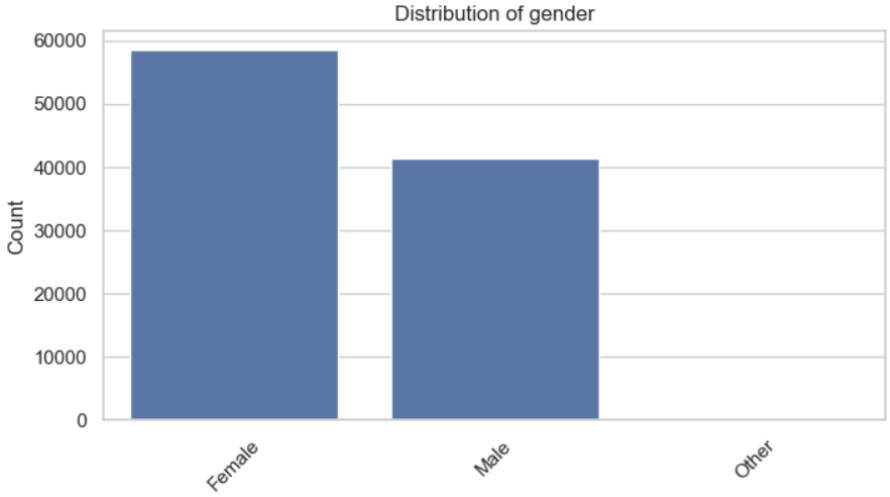
```
# Drop duplicate rows
df_no_duplicates = df.drop_duplicates()

# Check the shape after dropping duplicates
print("Shape after dropping duplicates:")
df_no_duplicates.shape
```

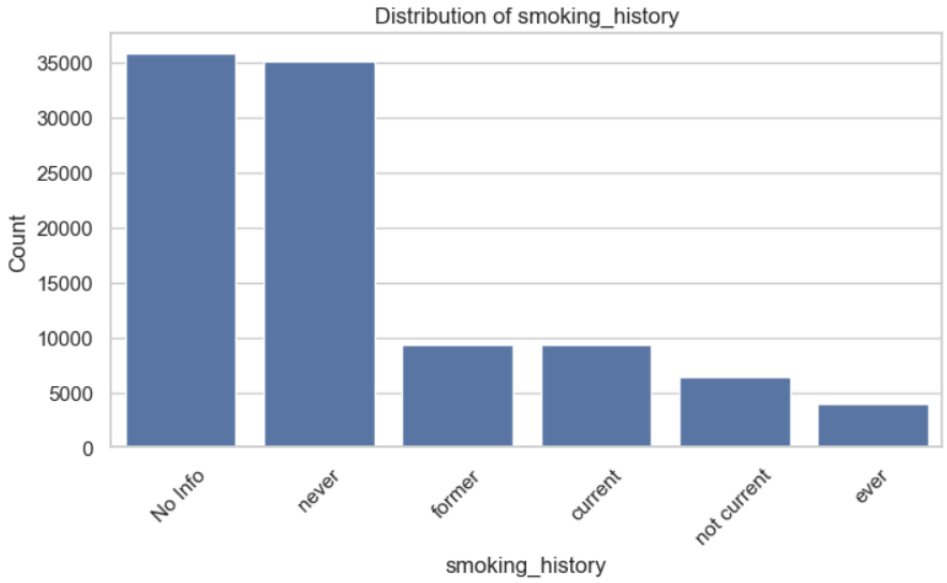
```
Shape after dropping duplicates:
(96146, 9)
```

- **Data Visualization**

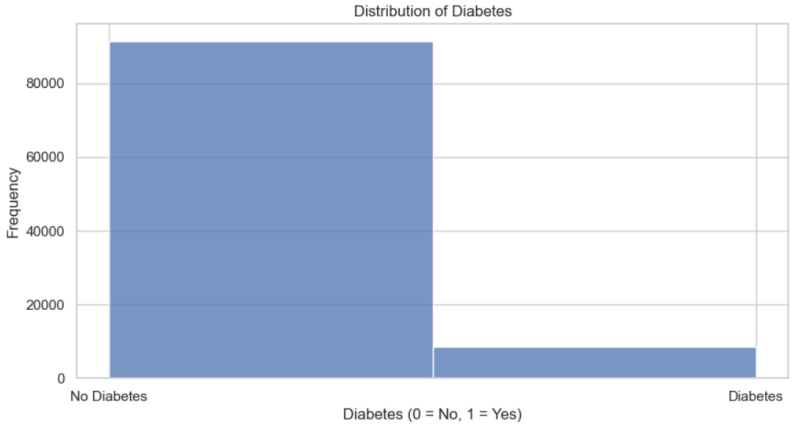

Distribution of age
Frequency
age
Distribution of hypertension
Frequency
hypertension
Distribution of heart_disease
Frequency
heart_disease
Distribution of bmi
Frequency
bmi
Distribution of HbA1c_level
Frequency
HbA1c_level
Distribution of blood_glucose_level
Frequency
blood_glucose_level

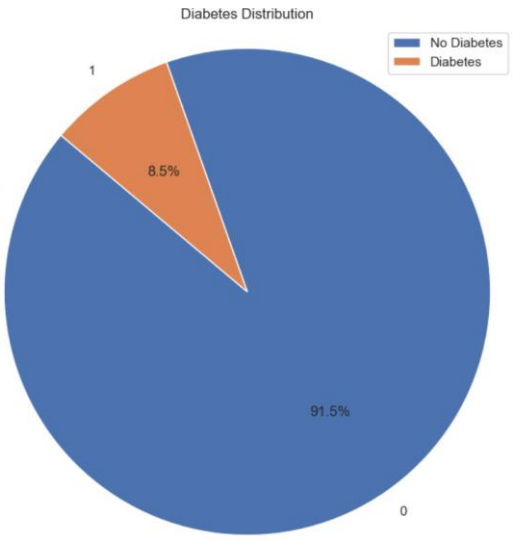

Distribution of diabetes
Frequency
diabetes


Distribution of gender
Count
Female
Male
Other



- **Understanding Target**





- **Taking Balance Data**

```
New Diabetes Count and Percentage:
diabetes
1    8500
0    8000
Name: count, dtype: int64
diabetes
1    51.515152
0    48.484848
Name: count, dtype: float64
```

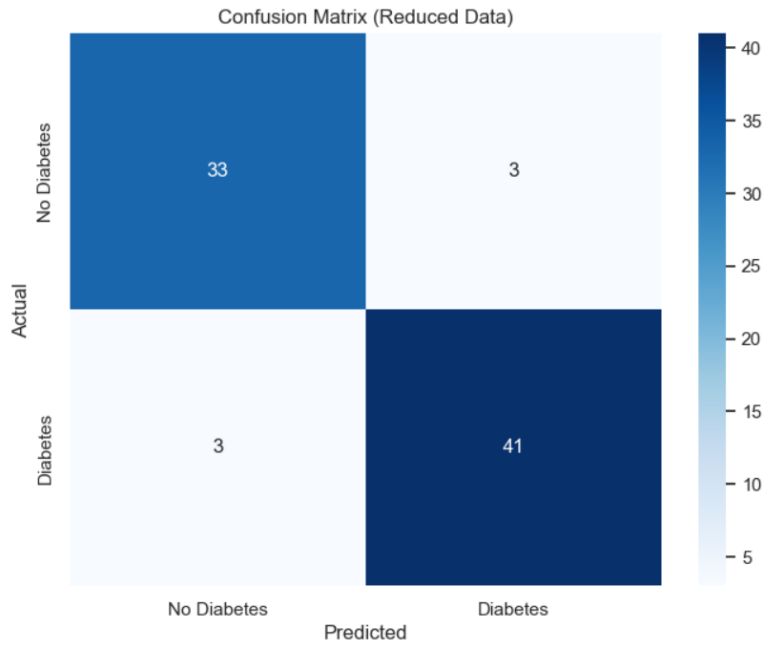
- **Encoding**

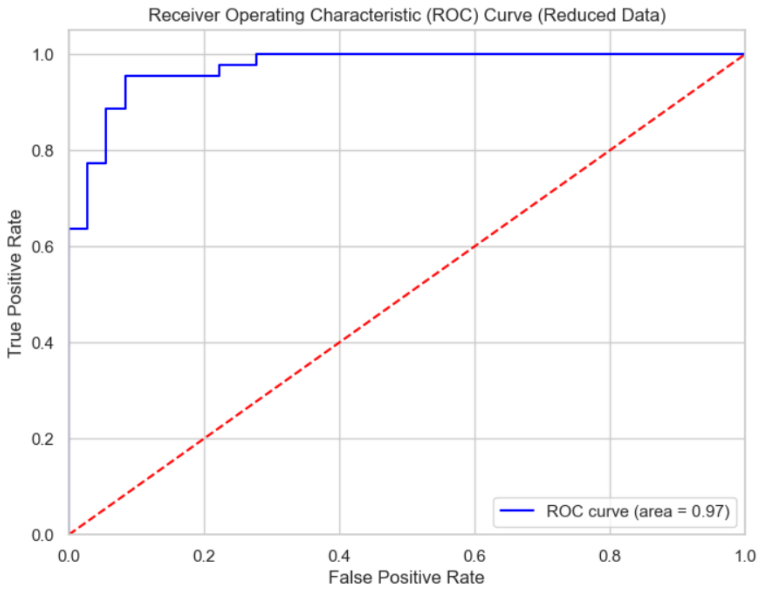
	age	hypertension	heart_disease	bmi	HbA1c_level	blood_glucose_level	diabetes	gender_Male	gender_Other	smoking_history_current	smoking_history_ever
21737	35.0	0	0	27.32	3.5	200	0	False	False	False	False
62807	66.0	0	0	27.32	4.5	145	0	True	False	False	False
38693	32.0	0	0	32.71	6.6	90	0	True	False	False	False
72875	46.0	0	0	32.78	4.0	80	0	True	False	False	False
14817	22.0	0	0	23.51	4.5	158	0	False	False	False	False

	age	hypertension	heart_disease	bmi	HbA1c_level	blood_glucose_level	diabetes	gender_Male	gender_Other	smoking_history_current	smoking_history_ever
21737	35.0	0	0	27.32	3.5	200	0	0	0	0	0
62807	66.0	0	0	27.32	4.5	145	0	1	0	0	0
38693	32.0	0	0	32.71	6.6	90	0	1	0	0	0
72875	46.0	0	0	32.78	4.0	80	0	1	0	0	0
14817	22.0	0	0	23.51	4.5	158	0	0	0	0	0

- **Model Building:**

```
Fitting 3 folds for each of 12 candidates, totalling 36 fits
Best Parameters: {'C': 1, 'gamma': 'scale', 'kernel': 'linear'}
Reduced Test Accuracy: 0.925

Classification Report (Reduced Dataset):
{'0': {'precision': 0.9166666666666666, 'recall': 0.9166666666666666, 'f1-score': 0.9166666666666666, 'support': 36.0}, '1': {'precision': 0.9318181818181818, 'recall': 0.9318181818181818, 'f1-score': 0.9318181818181818, 'support': 44.0}, 'accuracy': 0.925, 'macro avg': {'precision': 0.9242424242424242, 'recall': 0.9242424242424242, 'f1-score': 0.9242424242424242, 'support': 80.0}, 'weighted avg': {'precision': 0.925, 'recall': 0.925, 'f1-score': 0.925, 'support': 80.0}}
```





Result:

1.Best Hyperparameters:

The optimal parameters found through GridSearchCV are:

C: 1 (regularization parameter)

gamma: 'scale' (default behavior for kernel coefficient)

kernel: 'linear' (the model used a linear decision boundary)

This suggests that a linear kernel is sufficient for separating the classes in this reduced dataset.

2. Accuracy:

The model achieved an accuracy of 92.5% on the test set of the reduced dataset (80 samples: 36 non-diabetic and 44 diabetic).
This means that the model correctly predicted 92.5% of all test cases.

3. Precision, Recall, and F1-Score:
Class 0 (No Diabetes):
Precision: 91.67% - Out of all predictions made for the non-diabetic class, 91.67% were correct.
Recall: 91.67% - Out of all actual non-diabetic cases, the model correctly identified 91.67%.
F1-Score: 91.67% - The harmonic mean of precision and recall, indicating a good balance between both.
Class 1 (Diabetes):
Precision: 93.18% - Out of all predictions made for the diabetic class, 93.18% were correct.
Recall: 93.18% - Out of all actual diabetic cases, the model correctly identified 93.18%.
F1-Score: 93.18% - Again, showing a strong balance between precision and recall.

4. Macro Average:
Precision: 92.42%, Recall: 92.42%, F1-Score: 92.42%
These are unweighted averages across both classes, showing that the model performs equally well across both categories.

5. Weighted Average:
Precision: 92.5%, Recall: 92.5%, F1-Score: 92.5%
The weighted averages take the support of each class into account. This confirms that the model's overall performance across all samples is strong.

Conclusion:

The model performs well with a linear kernel and moderate regularization (C = 1). The results show a well-balanced performance between precision and recall for both diabetic and non-diabetic classes.

6. PCA

Objective:

Objective to find k mean on high dimension data vs data whose dimensionality reduced by PCA. Objective is to compare following two

Normal K-Means Clustering:
Directly applies the K-Means algorithm to the original data features.
May struggle to visualize clusters in high-dimensional spaces.
Can be computationally expensive for large datasets with many features.

PCA-based K-Means Clustering:
Applies PCA to reduce the dimensionality of the data before applying K-Means.
Projects the data onto a lower-dimensional space, making it easier to visualize clusters.

Input:

- **Loading Data**

```
import pandas as pd

# Load the dataset from the CSV file
file_path = r"C:\Users\roari\Downloads\customer_kmean_data.csv"
data = pd.read_csv(file_path)

# Display the first few rows of the dataset
data.head()
```

- **Understanding Data**

```
data.shape
```

```
data.size
```

```
print("\nInformation about the dataset:")
data.info()
```

```
print("\nDescriptive statistics:")
data.describe(include='all')
```

```
print("\nCount of null values in each column:")
data.isnull().sum()
```

```
# Get the columns names and their corresponding data types
column_data_types = data.dtypes

# Display column names and data types
print("Column names and their data types:")
column_data_types
```

- **Data visualization, Understanding distribution of data**

```python
import seaborn as sns
import matplotlib.pyplot as plt

# Set the style for the plots
sns.set(style="whitegrid")

# Create a list of numerical columns
numerical_columns = ['age', 'income', 'purchase_frequency', 'spending']

# Create histograms for each numerical column
plt.figure(figsize=(12, 10))  # Set the figure size

for i, column in enumerate(numerical_columns, 1):
    plt.subplot(2, 2, i)  # Create a 2x2 grid of plots
    sns.histplot(data[column], bins=20, kde=True)  # kde=True adds a Kernel Density Estimate line
    plt.title(f'Distribution of {column}')
    plt.xlabel(column)
    plt.ylabel('Frequency')

plt.tight_layout()  # Adjust the layout to prevent overlap
plt.show()
```

```python
import seaborn as sns
import matplotlib.pyplot as plt

# Set the style for the plots
sns.set(style="whitegrid")

# Create a list of categorical columns
categorical_columns = ['gender', 'education']

# Create pie charts for each categorical column
plt.figure(figsize=(12, 10))  # Set the figure size

for i, column in enumerate(categorical_columns, 1):
    plt.subplot(2, 2, i)  # Create a 2x2 grid of plots
    data_counts = data[column].value_counts()  # Count occurrences of each category
    plt.pie(data_counts, labels=data_counts.index, autopct='%1.1f%%', startangle=90)
    plt.title(f'Proportion of {column}')

plt.tight_layout()  # Adjust the layout to prevent overlap
plt.show()
```

- **Encoding**

```python
# Perform one-hot encoding on categorical columns
data_encoded = pd.get_dummies(data, columns=['gender', 'education', 'country'], drop_first=True)

# Display the first few rows of the encoded dataset
data_encoded.head()
```

```python
# Convert only the boolean columns (those that are the result of one-hot encoding) to integers
boolean_cols = data_encoded.select_dtypes(include='bool').columns  # Get boolean columns
data_encoded[boolean_cols] = data_encoded[boolean_cols].astype(int)  # Convert to integers

# Display the first few rows of the modified DataFrame
data_encoded.head()
```

- **Scaling data**

```python
from sklearn.preprocessing import MinMaxScaler

# Initialize the MinMaxScaler
scaler = MinMaxScaler()

# Select the columns to scale (excluding the non-numerical columns like 'name')
numerical_cols = ['age', 'income', 'purchase_frequency', 'spending']  # List of numerical columns

# Fit and transform the data
data_encoded[numerical_cols] = scaler.fit_transform(data_encoded[numerical_cols])

# Display the first few rows of the scaled DataFrame
data_encoded.head()
```

- **Model Building**
  - **Kmean cluster without PCA**

```python
# Import necessary 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.cluster import KMeans

# Assuming data_encoded is already available
# Scale the data using MinMaxScaler
scaler = MinMaxScaler()
numerical_cols = ['age', 'income', 'purchase_frequency', 'spending']
data_encoded[numerical_cols] = scaler.fit_transform(data_encoded[numerical_cols])

# K-Means Clustering
n_clusters = 4  # Optimal number of clusters based on Silhouette Score
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
data_encoded['Cluster'] = kmeans.fit_predict(data_encoded[numerical_cols])

# Plotting the clusters
plt.figure(figsize=(12, 8))
sns.scatterplot(data=data_encoded, x='income', y='spending', hue='Cluster', palette='viridis', s=100, alpha=0.7)
plt.title('K-Means Clustering (7 Clusters) Without PCA')
plt.xlabel('Income (scaled)')
plt.ylabel('Spending (scaled)')
plt.legend(title='Clusters')
plt.grid()
plt.show()
```

- **K mean Cluster with PCA**

```
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans

# Load your dataset (replace with your actual data loading method)
# data_encoded = pd.read_csv('your_data_file.csv') # Example of loading a dataset

# Assuming data_encoded is already available
# Scale the data using MinMaxScaler
scaler = MinMaxScaler()
numerical_cols = ['age', 'income', 'purchase_frequency', 'spending']
data_encoded[numerical_cols] = scaler.fit_transform(data_encoded[numerical_cols])

# Perform PCA to reduce dimensions to 2 for visualization
pca = PCA(n_components=2)
pca_result = pca.fit_transform(data_encoded.drop(columns=['name']))  # Drop the 'name' column

# Create a new DataFrame with PCA results
pca_df = pd.DataFrame(data=pca_result, columns=['PC1', 'PC2'])

# Perform KMeans clustering (choose the number of clusters, e.g., 3)
kmeans = KMeans(n_clusters=3, random_state=42)
pca_df['Cluster'] = kmeans.fit_predict(pca_df)

# Visualize the clusters
plt.figure(figsize=(10, 7))
sns.scatterplot(data=pca_df, x='PC1', y='PC2', hue='Cluster', palette='viridis', s=100, alpha=0.7)
plt.title('PCA of Scaled Data with KMeans Clusters')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(title='Cluster')
plt.grid(True)
plt.show()
```

Output:

- **Loading Data**

	name	age	gender	education	income	country	purchase_frequency	spending
0	Teresa Williams MD	42	Female	High School	53936	Slovenia	0.9	13227.120
1	Christine Myers	49	Female	Master	82468	Aruba	0.6	12674.040
2	Dwayne Moreno	55	Male	Bachelor	56941	Cyprus	0.3	5354.115
3	Amy Norton	24	Female	Bachelor	60651	Palau	0.2	2606.510
4	Tonya Adams	64	Male	Master	81884	Zambia	0.9	18984.780

name: Contains the names of customers as string values.
age: Represents the age of customers as integer values.
gender: Indicates the gender of customers (Male/Female) as string values.
education: Denotes the highest level of education attained by customers as string values.
income: Reflects the annual income of customers as integer values.
country: Lists the country of residence for customers as string values.
purchase_frequency: Measures the frequency of purchases made by customers as float values (ranging from 0 to 1).
spending: Indicates the total spending of customers as float values.

- **Understanding Data**

```
Finding the hape of the data
```

```
data.shape
```

```
(1000, 8)
```

```
Finding the size
```

```
data.size
```

```
8000
```

```
Info of data
```

```
print("\nInformation about the dataset:")
data.info()
```

```
Information about the dataset:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 8 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   name                1000 non-null   object 
 1   age                 1000 non-null   int64  
 2   gender              1000 non-null   object 
 3   education           1000 non-null   object 
 4   income              1000 non-null   int64  
 5   country             1000 non-null   object 
 6   purchase_frequency  1000 non-null   float64
 7   spending            1000 non-null   float64
dtypes: float64(2), int64(2), object(4)
memory usage: 62.6+ KB
```

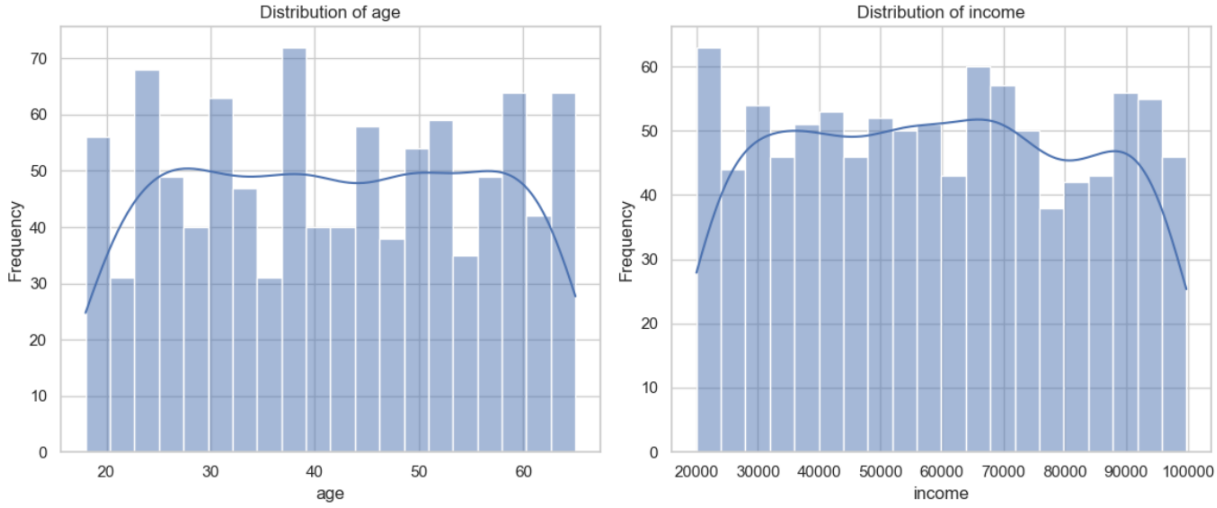
Descriptive statistics:

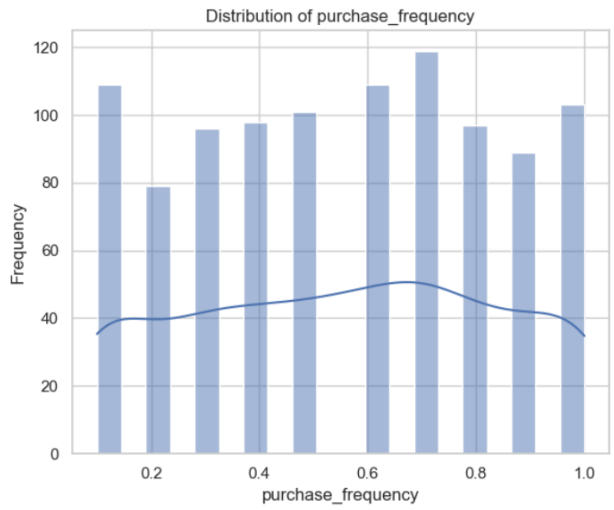
	name	age	gender	education	income	country	purchase_frequency	spending
count	1000	1000.000000	1000	1000	1000.000000	1000	1000.000000	1000.000000
unique	991	NaN	2	4	NaN	239	NaN	NaN
top	Michael Brown	NaN	Male	Bachelor	NaN	Congo	NaN	NaN
freq	3	NaN	501	271	NaN	12	NaN	NaN
mean	NaN	41.754000	NaN	NaN	59277.852000	NaN	0.554600	9613.296835
std	NaN	13.778582	NaN	NaN	23258.377128	NaN	0.284675	5484.707210
min	NaN	18.000000	NaN	NaN	20031.000000	NaN	0.100000	611.985000
25%	NaN	30.000000	NaN	NaN	38825.500000	NaN	0.300000	5020.425000
50%	NaN	42.000000	NaN	NaN	58972.000000	NaN	0.600000	9430.395000
75%	NaN	54.000000	NaN	NaN	79114.000000	NaN	0.800000	13645.507500
max	NaN	65.000000	NaN	NaN	99780.000000	NaN	1.000000	25546.500000

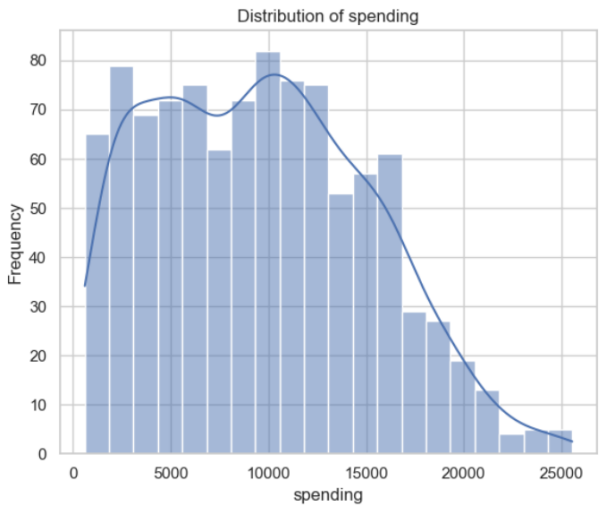
```
Count of null values in each column:

name                  0
age                   0
gender                0
education             0
income                0
country               0
purchase_frequency    0
spending              0
dtype: int64
```

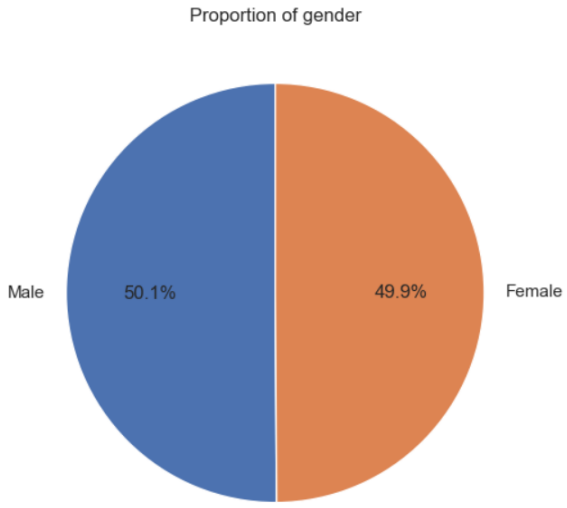
```
Column names and their data types:

name                  object
age                    int64
gender                object
education             object
income                 int64
country               object
purchase_frequency   float64
spending             float64
dtype: object
```

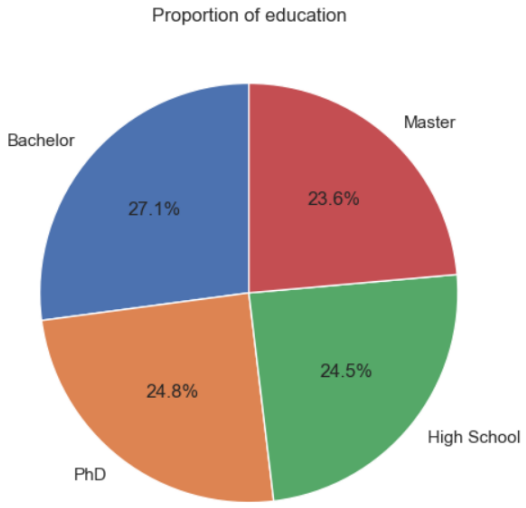
- **Data visualization, Understanding distribution of data**











- **Encoding**

	name	age	income	purchase_frequency	spending	gender_Male	education_High School	education_Master	education_PhD	country_Albania	...	country_Uruguay	coun
0	Teresa Williams MD	42	53936	0.9	13227.120	False	True	False	False	False	...	False	
1	Christine Myers	49	82468	0.6	12674.040	False	False	True	False	False	...	False	
2	Dwayne Moreno	55	56941	0.3	5354.115	True	False	False	False	False	...	False	
3	Amy Norton	24	60651	0.2	2606.510	False	False	False	False	False	...	False	
4	Tonya Adams	64	81884	0.9	18984.780	True	False	True	False	False	...	False	

5 rows × 247 columns

	name	age	income	purchase_frequency	spending	gender_Male	education_High School	education_Master	education_PhD	country_Albania	...	country_Uruguay	coun
0	Teresa Williams MD	42	53936	0.9	13227.120	0	1	0	0	0	...	0	
1	Christine Myers	49	82468	0.6	12674.040	0	0	1	0	0	...	0	
2	Dwayne Moreno	55	56941	0.3	5354.115	1	0	0	0	0	...	0	
3	Amy Norton	24	60651	0.2	2606.510	0	0	0	0	0	...	0	
4	Tonya Adams	64	81884	0.9	18984.780	1	0	1	0	0	...	0	

5 rows × 247 columns

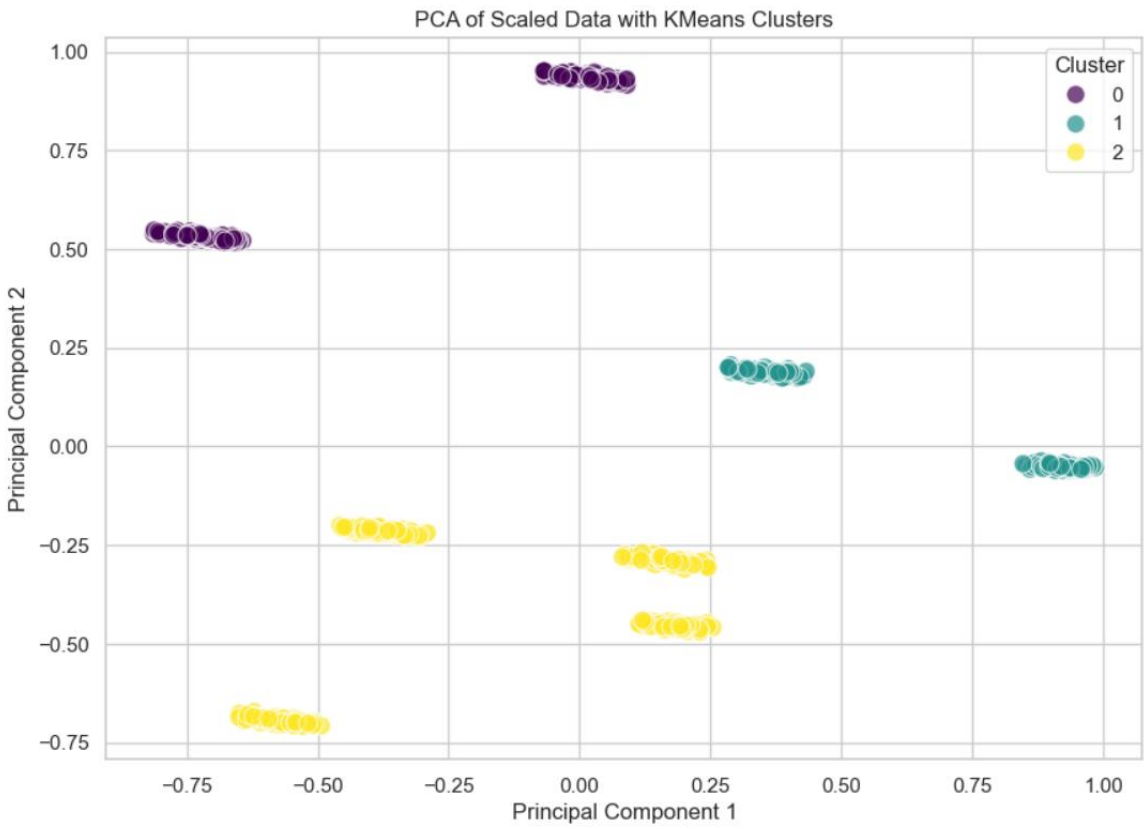
- **Scaling data**

	name	age	income	purchase_frequency	spending	gender_Male	education_High School	education_Master	education_PhD	country_Albania	...	country_Uruguay
0	Teresa Williams MD	0.510638	0.425146	0.888889	0.505931	0	1	0	0	0	...	0
1	Christine Myers	0.659574	0.782919	0.555556	0.483749	0	0	1	0	0	...	0
2	Dwayne Moreno	0.787234	0.462827	0.222222	0.190183	1	0	0	0	0	...	0
3	Amy Norton	0.127660	0.509348	0.111111	0.079991	0	0	0	0	0	...	0
4	Tonya Adams	0.978723	0.775596	0.888889	0.736842	1	0	1	0	0	...	0

5 rows × 247 columns

- **Model Building**
  - **Kmean cluster without PCA**

  - **K mean Cluster with PCA**



Result:

PCA reduced multidimensional data into two dimensional data, while variance is reduced to make efficient clusters.

7. K-means

Objective:

Performing customer groups or clusters for efficient marketing strategies to covert customers from engagement to purchase. Dataset explores which type of cluster should be made and finding appropriate clusters to have balance trade off identifying class label correctly and inertial also will be less.

Input:

- **Loading Data**

```
import pandas as pd  # Import the pandas library for data manipulation and analysis

# Define the path to the CSV file
data_path = "C:\\Users\\roari\\Downloads\\Live.csv"

# Load the CSV file into a DataFrame
df = pd.read_csv(data_path)
```

- **EDA**

```
df.shape
```

```
df.head()
```

```
df.info()
```

```
df.isnull().sum()
```

```
df.drop(['Column1', 'Column2', 'Column3', 'Column4'], axis=1, inplace=True)
```

```
df.info()
```

```
df.describe()
```

- **Exploring on what basis we make clusters for unsupervised label data.**

```
Exploring Status variable
```

```
# view the labels in the variable

df['status_id'].unique()
```

```
# view how many different types of variables are there

len(df['status_id'].unique())
```

```
Exploring Status_published
```

```
# view the labels in the variable

df['status_published'].unique()
```

```
# view how many different types of variables are there

len(df['status_published'].unique())
```

```
Exploring status type
```

```
# view the labels in the variable

df['status_type'].unique()
```

```
# view how many different types of variables are there

len(df['status_type'].unique())
```

- ## Dropping unwanted field

```
Dropping status_id and status_published
```

```
df.drop(['status_id', 'status_published'], axis=1, inplace=True)
```

- ## Defining feature

```
Taking target and status type and other as feature of x
```

```
X = df

y = df['status_type']
```

- ## Encoding

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

X['status_type'] = le.fit_transform(X['status_type'])

y = le.transform(y)
```

- ## Displaying info of x

```
Displaying x with feature in it
```

```
X.info()
```

- ## Scaling the data

```
Scaling data using min max scalar
```

```
cols = X.columns
```

```
from sklearn.preprocessing import MinMaxScaler

ms = MinMaxScaler()

X = ms.fit_transform(X)
```

```
X = pd.DataFrame(X, columns=[cols])
```

```
X.head()
```

- ## Model building

```
Model building - Starting cold start
```

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=2, random_state=0)

kmeans.fit(X)
```

- ## Finding centroid

```
Finding the centroid
```

```
kmeans.cluster_centers_
```

```
array([[9.54921576e-01, 6.46330441e-02, 2.67028654e-02, 2.93171709e-02,
        5.71231462e-02, 4.71007076e-02, 8.18581889e-03, 9.65207685e-03,
        8.04219428e-03, 7.19501847e-03],
       [3.28506857e-01, 3.90710874e-02, 7.54854864e-04, 7.53667113e-04,
        3.85438884e-02, 2.17448568e-03, 2.43721364e-03, 1.20039760e-03,
        2.75348016e-03, 1.45313276e-03]])
```

- ## Finding Inertia

```
kmeans.inertia_
```

```
237.7572640441955
```

- ## Checking Model

```
Checking quality of the model
```

```
labels = kmeans.labels_

# check how many of the samples were correctly labeled
correct_labels = sum(y == labels)

print("Result: %d out of %d samples were correctly labeled." % (correct_labels, y.size))
```

```
print('Accuracy score: {0:0.2f}'. format(correct_labels/float(y.size)))
```

- ## Using Elbow

```
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
cs = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', max_iter = 300, n_init = 10, random_state = 0)
    kmeans.fit(X)
    cs.append(kmeans.inertia_)
plt.plot(range(1, 11), cs)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('CS')
plt.show()
```

- ## Model with cluster 2

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=2,random_state=0)

kmeans.fit(X)

labels = kmeans.labels_

# check how many of the samples were correctly labeled

correct_labels = sum(y == labels)

print("Result: %d out of %d samples were correctly labeled." % (correct_labels, y.size))

print('Accuracy score: {0:0.2f}'. format(correct_labels/float(y.size)))
```

- ## Model with cluster 10

```
kmeans = KMeans(n_clusters=10, random_state=0)

kmeans.fit(X)

# check how many of the samples were correctly labeled
labels = kmeans.labels_

correct_labels = sum(y == labels)
print("Result: %d out of %d samples were correctly labeled." % (correct_labels, y.size))
print('Accuracy score: {0:0.2f}'. format(correct_labels/float(y.size)))
```

- **Model with cluster 20**

```
kmeans = KMeans(n_clusters=20, random_state=0)

kmeans.fit(X)

# check how many of the samples were correctly labeled
labels = kmeans.labels_

correct_labels = sum(y == labels)
print("Result: %d out of %d samples were correctly labeled." % (correct_labels, y.size))
print('Accuracy score: {0:0.2f}'. format(correct_labels/float(y.size)))
```

Output:

- **Loading Data**

	status_id	status_type	status_published	num_reactions	num_comments	num_shares	num_likes	num_loves	num_wows	num_hahas	num_
0	246675545449582_1649696485147474	video	4/22/2018 6:00	529	512	262	432	92	3	1	
1	246675545449582_1649426988507757	photo	4/21/2018 22:45	150	0	0	150	0	0	0	
2	246675545449582_1648730588577397	video	4/21/2018 6:17	227	236	57	204	21	1	1	
3	246675545449582_1648576705259452	photo	4/21/2018 2:29	111	0	0	111	0	0	0	
4	246675545449582_1645700502213739	photo	4/18/2018 3:22	213	0	0	204	9	0	0	

- **EDA**

```
Finding the shape of the data
```

```
df.shape
```

```
(7050, 16)
```

Information about data

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7050 entries, 0 to 7049
Data columns (total 16 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   status_id         7050 non-null   object 
 1   status_type       7050 non-null   object 
 2   status_published  7050 non-null   object 
 3   num_reactions     7050 non-null   int64  
 4   num_comments      7050 non-null   int64  
 5   num_shares        7050 non-null   int64  
 6   num_likes         7050 non-null   int64  
 7   num_loves         7050 non-null   int64  
 8   num_wows          7050 non-null   int64  
 9   num_hahas         7050 non-null   int64  
 10  num_sads          7050 non-null   int64  
 11  num_angrys        7050 non-null   int64  
 12  Column1           0 non-null      float64
 13  Column2           0 non-null      float64
 14  Column3           0 non-null      float64
 15  Column4           0 non-null      float64
dtypes: float64(4), int64(9), object(3)
memory usage: 881.4+ KB
```

Checking null value count

```
df.isnull().sum()
```

```
status_id             0
status_type           0
status_published      0
num_reactions         0
num_comments          0
num_shares            0
num_likes             0
num_loves             0
num_wows              0
num_hahas             0
num_sads              0
num_angrys            0
Column1            7050
Column2            7050
Column3            7050
Column4            7050
dtype: int64
```

```
Droppping redundand rows
```

```
df.drop(['Column1', 'Column2', 'Column3', 'Column4'], axis=1, inplace=True)
```

```
Finding Infor Again
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7050 entries, 0 to 7049
Data columns (total 12 columns):
 #   Column            Non-Null Count  Dtype 
---  ------            --------------  ----- 
 0   status_id         7050 non-null   object
 1   status_type       7050 non-null   object
 2   status_published  7050 non-null   object
 3   num_reactions     7050 non-null   int64 
 4   num_comments      7050 non-null   int64 
 5   num_shares        7050 non-null   int64 
 6   num_likes         7050 non-null   int64 
 7   num_loves         7050 non-null   int64 
 8   num_wows          7050 non-null   int64 
 9   num_hahas         7050 non-null   int64 
 10  num_sads          7050 non-null   int64 
 11  num_angrys        7050 non-null   int64 
dtypes: int64(9), object(3)
memory usage: 661.1+ KB
```

Now, we can see that redundant columns have been removed from the dataset.

We can see that, there are 3 character variables (data type = object) and remaining 9 numerical variables (data type = int64).

```
df.describe()
```

	num_reactions	num_comments	num_shares	num_likes	num_loves	num_wows	num_hahas	num_sads	num_angrys
count	7050.000000	7050.000000	7050.000000	7050.000000	7050.000000	7050.000000	7050.000000	7050.000000	7050.000000
mean	230.117163	224.356028	40.022553	215.043121	12.728652	1.289362	0.696454	0.243688	0.113191
std	462.625309	889.636820	131.599965	449.472357	39.972930	8.719650	3.957183	1.597156	0.726812
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	17.000000	0.000000	0.000000	17.000000	0.000000	0.000000	0.000000	0.000000	0.000000
50%	59.500000	4.000000	0.000000	58.000000	0.000000	0.000000	0.000000	0.000000	0.000000
75%	219.000000	23.000000	4.000000	184.750000	3.000000	0.000000	0.000000	0.000000	0.000000
max	4710.000000	20990.000000	3424.000000	4710.000000	657.000000	278.000000	157.000000	51.000000	31.000000

- **Exploring on what basis we make clusters for unsupervised label data.**

```
Exploring Status variable
```

```
# view the labels in the variable

df['status_id'].unique()
```

```
array(['246675545449582_1649696485147474',
       '246675545449582_1649426988507757',
       '246675545449582_1648730588577397', ...,
       '1050855161656896_1060126464063099',
       '1050855161656896_1058663487542730',
       '1050855161656896_1050858841656528'], dtype=object)
```

```
# view how many different types of variables are there

len(df['status_id'].unique())
```

```
6997
```

```
Exploring Status_published
```

```
# view the labels in the variable

df['status_published'].unique()
```

```
array(['4/22/2018 6:00', '4/21/2018 22:45', '4/21/2018 6:17', ...,
       '9/21/2016 23:03', '9/20/2016 0:43', '9/10/2016 10:30'],
      dtype=object)
```

```
# view how many different types of variables are there

len(df['status_published'].unique())
```

```
6913
```

```
Exploring status type
```

```
# view the labels in the variable

df['status_type'].unique()
```

```
array(['video', 'photo', 'link', 'status'], dtype=object)
```

```
# view how many different types of variables are there

len(df['status_type'].unique())
```

```
4
```

- **Dropping unwanted field**

```
Dropping status_id and status_published
```

```
df.drop(['status_id', 'status_published'], axis=1, inplace=True)
```

```
Finding information of data again
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7050 entries, 0 to 7049
Data columns (total 10 columns):
status_type      7050 non-null object
num_reactions    7050 non-null int64
num_comments     7050 non-null int64
num_shares       7050 non-null int64
num_likes        7050 non-null int64
num_loves        7050 non-null int64
num_wows         7050 non-null int64
num_hahas        7050 non-null int64
num_sads         7050 non-null int64
num_angrys       7050 non-null int64
dtypes: int64(9), object(1)
memory usage: 550.9+ KB
```

- **Defining feature**

```
Taking target and status type and other as feature of x
```

```
X = df

y = df['status_type']
```

- **Encoding &**
- **Displaying info of x**

	status_type	num_reactions	num_comments	num_shares	num_likes	num_loves	num_wows	num_hahas	num_sads	num_angrys
0	3	529	512	262	432	92	3	1	1	0
1	1	150	0	0	150	0	0	0	0	0
2	3	227	236	57	204	21	1	1	0	0
3	1	111	0	0	111	0	0	0	0	0
4	1	213	0	0	204	9	0	0	0	0

- **Scaling the data**

	status_type	num_reactions	num_comments	num_shares	num_likes	num_loves	num_wows	num_hahas	num_sads	num_angrys
0	1.000000	0.112314	0.024393	0.076519	0.091720	0.140030	0.010791	0.006369	0.019608	0.0
1	0.333333	0.031847	0.000000	0.000000	0.031847	0.000000	0.000000	0.000000	0.000000	0.0
2	1.000000	0.048195	0.011243	0.016647	0.043312	0.031963	0.003597	0.006369	0.000000	0.0
3	0.333333	0.023567	0.000000	0.000000	0.023567	0.000000	0.000000	0.000000	0.000000	0.0
4	0.333333	0.045223	0.000000	0.000000	0.043312	0.013699	0.000000	0.000000	0.000000	0.0

- **Model building**

KMeans

```
KMeans(n_clusters=2, random_state=0)
```

- **Finding centroid**

```
array([[9.54921576e-01, 6.46330441e-02, 2.67028654e-02, 2.93171709e-02,
        5.71231462e-02, 4.71007076e-02, 8.18581889e-03, 9.65207685e-03,
        8.04219428e-03, 7.19501847e-03],
       [3.28506857e-01, 3.90710874e-02, 7.54854864e-04, 7.53667113e-04,
        3.85438884e-02, 2.17448568e-03, 2.43721364e-03, 1.20039760e-03,
        2.75348016e-03, 1.45313276e-03]])
```

- **Finding Inertia**

```
kmeans.inertia_
```

```
237.7572640441955
```

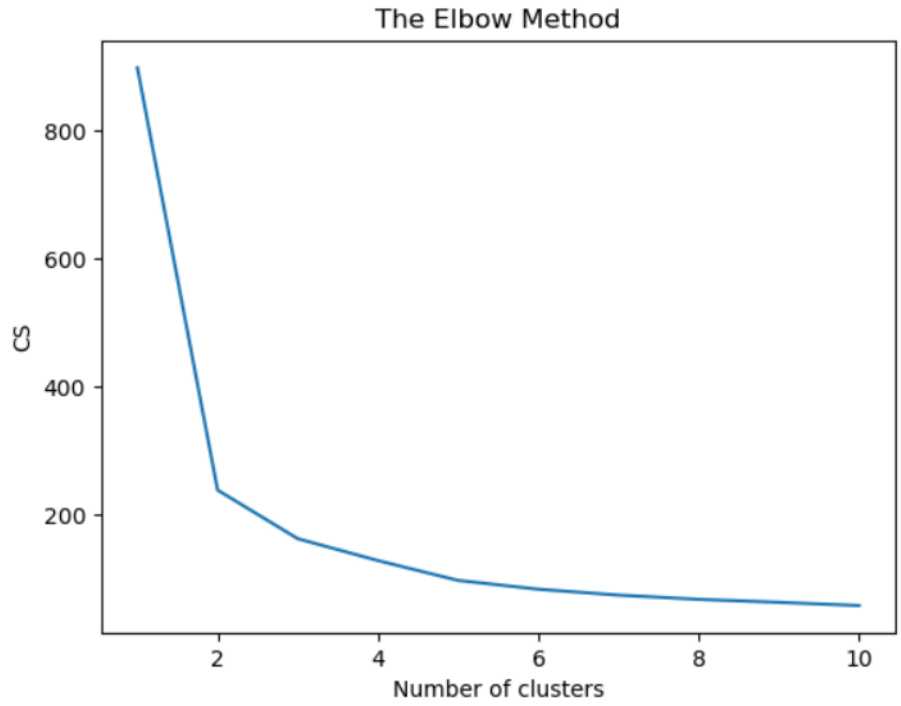
- **Checking Model**

```
Result: 4288 out of 7050 samples were correctly labeled.
```

```
Accuracy score: 0.61
```

We have achieved classification accuracy of 61% by our unsupervised model.

- **Using Elbow**



- By the above plot, we can see that there is a elevation only at k=2.
- Hence k=2 can be considered a good number of the cluster to cluster this data.
- But, we have seen that I have achieved a weak classification accuracy of 61% with k=2.
- I will write the required code with k=2 again for convinience.

- **Model with cluster 2**

```
Result: 4288 out of 7050 samples were correctly labeled.
Accuracy score: 0.61
```

So, our weak unsupervised classification model achieved a very weak classification accuracy of 61%.

I will check the model accuracy with different number of clusters.

- **Model with cluster 10**

```
cluster= 10
```

```
kmeans = KMeans(n_clusters=10, random_state=0)

kmeans.fit(X)

# check how many of the samples were correctly labeled
labels = kmeans.labels_

correct_labels = sum(y == labels)
print("Result: %d out of %d samples were correctly labeled." % (correct_labels, y.size))
print('Accuracy score: {0:0.2f}'. format(correct_labels/float(y.size)))
```

```
Result: 4009 out of 7050 samples were correctly labeled.
Accuracy score: 0.57
```

```
kmeans.inertia_
```

```
59.098055745974435
```

- **Model with cluster 20**

```
kmeans = KMeans(n_clusters=20, random_state=0)

kmeans.fit(X)

# check how many of the samples were correctly labeled
labels = kmeans.labels_

correct_labels = sum(y == labels)
print("Result: %d out of %d samples were correctly labeled." % (correct_labels, y.size))
print('Accuracy score: {0:0.2f}'. format(correct_labels/float(y.size)))
```

```
Result: 3501 out of 7050 samples were correctly labeled.
Accuracy score: 0.50
```

```
kmeans.inertia_
```

```
31.05949981187011
```

Result:

Which model we choose?
In a supervised learning setup (where you care about labels): Accuracy is typically more important if you want the clusters to match predefined labels. In this case, Model 1 (61% accuracy) might be a better option, even though the inertia is higher.

In an unsupervised learning setup (pure clustering): If you are only concerned about how well the data points are grouped together without considering predefined labels, inertia becomes more important. In this case, Model 3 (inertia of 31) would be a better choice, as it indicates tightly packed clusters.

Balancing the two: If you need a balance between good cluster compactness and reasonable alignment with labels, Model 2 (57% accuracy, inertia of 59) might be the best compromise. It's performing moderately well on both metrics.

Conclusion:

If your priority is accuracy (matching predefined labels), go with Model 1 with 2 clusters.

If your priority is compact clusters (minimizing within-cluster distances), go with Model 3 with 20 clusters.
For a balanced trade-off, Model 2 is good option with 10 clusters.

8. Decision tree

Title: Loan Default Prediction Using Machine Learning (Paidoff vs Active or default)

Objective Statement:

The primary objective of this project is to develop a robust machine learning model that accurately predicts the likelihood of loan defaults based on various borrower characteristics and loan parameters. The model aims to enhance decision-making in lending practices by providing insights into the risk associated with loan approvals.

Input:

- **Loading data**

```
import pandas as pd

# Load the dataset
file_path = "C:/Users/roari/Downloads/Credit Risk/credit_risk_dataset.csv"
credit_risk_data = pd.read_csv(file_path)

# Display the first few rows of the dataset
credit_risk_data.head()
```

- **Understanding dataset**

```
credit_risk_data.shape
```

```
credit_risk_data.size
```

```
credit_risk_data.columns
```

```
credit_risk_data.info()
```

```
credit_risk_data.describe()
```

```
credit_risk_data.isnull().sum()
```

- **Finding the distribution of data**

```
import seaborn as sns
import matplotlib.pyplot as plt

# List of numeric columns in your dataset to plot histograms
numeric_columns = [
    'person_age', 'person_income', 'person_emp_length',
    'loan_amnt', 'loan_int_rate', 'loan_percent_income',
    'cb_person_cred_hist_length'
]

# Set the size of the plots
plt.figure(figsize=(15, 10))

# Loop through numeric columns and create histograms
for i, column in enumerate(numeric_columns, 1):
    plt.subplot(3, 3, i)
    sns.histplot(credit_risk_data[column], kde=True, bins=20)
    plt.title(f'Distribution of {column}')
    plt.tight_layout()

# Show the plot
plt.show()
```

- **Filling null based on distribution**

```
# Fill missing values in 'person_emp_length' with the median
credit_risk_data['person_emp_length'].fillna(credit_risk_data['person_emp_length'].median(), inplace=True)

# Fill missing values in 'loan_int_rate' with the mean
credit_risk_data['loan_int_rate'].fillna(credit_risk_data['loan_int_rate'].mean(), inplace=True)

# Verify if missing values are filled
print("Missing values after filling:")
print(credit_risk_data.isnull().sum())
```

- **Proportion of data**

```
import matplotlib.pyplot as plt

# List of categorical columns
categorical_columns = ['person_home_ownership', 'loan_intent', 'loan_grade', 'cb_person_default_on_file']

# Set the size of the plots
plt.figure(figsize=(15, 10))

# Loop through categorical columns and create pie charts
for i, column in enumerate(categorical_columns, 1):
    plt.subplot(2, 2, i)
    data = credit_risk_data[column].value_counts()
    plt.pie(data, labels=data.index, autopct='%1.1f%%', startangle=90, colors=sns.color_palette('pastel'))
    plt.title(f'Proportion of {column}')
    plt.tight_layout()

# Show the plots
plt.show()
```

- **Target Label proportion**

```
import seaborn as sns
import matplotlib.pyplot as plt

# Set the size of the plot
plt.figure(figsize=(8, 6))

# Create a histogram to visualize the proportion of loan_status
sns.histplot(data=credit_risk_data, x='loan_status', stat='probability', discrete=True)

# Add title and labels
plt.title('Proportion of Loan Status', fontsize=16)
plt.xlabel('Loan Status (0 = Paid off, 1 = Active/Defaulted)', fontsize=12)
plt.ylabel('Proportion', fontsize=12)

# Show the plot
plt.show()
```

- **Encoding data**

```
import pandas as pd

# Specify the categorical columns to be one-hot encoded
columns_to_encode = ['person_home_ownership', 'loan_intent', 'loan_grade', 'cb_person_default_on_file']

# Perform one-hot encoding
credit_risk_encoded = pd.get_dummies(credit_risk_data, columns=columns_to_encode, drop_first=True)

# Display the first few rows of the encoded DataFrame
credit_risk_encoded.head()
```

```
# Convert boolean values to integers (True = 1, False = 0)
credit_risk_encoded = credit_risk_encoded.astype(int)

# Display the first few rows of the updated DataFrame
credit_risk_encoded.head()
```

- **Standardizing Data**

```
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Assuming credit_risk_encoded is your DataFrame after one-hot encoding
# Separate features and target variable
X = credit_risk_encoded.drop(columns=['loan_status'])  # Exclude target column
y = credit_risk_encoded['loan_status']  # Target column

# Initialize the StandardScaler
scaler = StandardScaler()

# Fit the scaler to the features and transform them
X_standardized = pd.DataFrame(scaler.fit_transform(X), columns=X.columns)

# Combine standardized features with the target variable
credit_risk_standardized = pd.concat([X_standardized, y.reset_index(drop=True)], axis=1)

# Display the first few rows of the standardized DataFrame
credit_risk_standardized.head()
```

- **Model Building**

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, KFold, GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc, accuracy_score
import matplotlib.pyplot as plt
from imblearn.over_sampling import SMOTE
from sklearn.preprocessing import StandardScaler

# Load your data
# Assuming credit_risk_encoded is your DataFrame after encoding and standardizing
# For demonstration, let's say `credit_risk_encoded` is already defined

# Define your features (X) and target (y)
X = credit_risk_encoded.drop('loan_status', axis=1)  # Drop target variable
y = credit_risk_encoded['loan_status']

# Handle class imbalance using SMOTE
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_resampled, y_resampled, test_size=0.2, random_state=42)

# Create a decision tree classifier
dt_classifier = DecisionTreeClassifier(random_state=42)

# Define the parameters for GridSearchCV
param_grid = {
    'max_depth': [None, 5, 10, 15, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
```

```
# Set up KFold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Perform grid search with cross-validation
grid_search = GridSearchCV(estimator=dt_classifier, param_grid=param_grid,
                           scoring='f1', cv=kf, n_jobs=-1, verbose=1)

grid_search.fit(X_train, y_train)

# Best parameters and model
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_

print(f"Best Parameters: {best_params}")

# Calculate training accuracy
train_accuracy = best_model.score(X_train, y_train)

# Predictions
y_pred = best_model.predict(X_test)

# Test accuracy
test_accuracy = accuracy_score(y_test, y_pred)

print(f"Training Accuracy: {train_accuracy:.4f}")
print(f"Test Accuracy: {test_accuracy:.4f}")

# Classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

```
# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:")
print(conf_matrix)

# ROC Curve
fpr, tpr, thresholds = roc_curve(y_test, best_model.predict_proba(X_test)[:, 1])
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(10, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='red', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid()
plt.show()
```

Output:

- **Loading data**

	person_age	person_income	person_home_ownership	person_emp_length	loan_intent	loan_grade	loan_amnt	loan_int_rate	loan_status	loan_percent_income	cb_per
0	22	59000	RENT	123.0	PERSONAL	D	35000	16.02	1	0.59	
1	21	9600	OWN	5.0	EDUCATION	B	1000	11.14	0	0.10	
2	25	9600	MORTGAGE	1.0	MEDICAL	C	5500	12.87	1	0.57	
3	23	65500	RENT	4.0	MEDICAL	C	35000	15.23	1	0.53	
4	24	54400	RENT	8.0	MEDICAL	C	35000	14.27	1	0.55	

```
person_age: Age of the individual applying for the loan (integer).
person_income: Annual income of the individual (integer).
person_home_ownership: Type of home ownership (categorical: RENT, OWN, MORTGAGE).
person_emp_length: Length of employment in years (float).
loan_intent: Purpose of the loan (categorical: e.g., PERSONAL, MEDICAL, EDUCATION).
loan_grade: Credit grade assigned to the loan (categorical: A, B, C, etc.).
loan_amnt: Amount of the loan (integer).
loan_int_rate: Interest rate on the loan (float).
loan_status: Status of the loan (binary: 0 for paid off, 1 for active or defaulted).
loan_percent_income: Percentage of income used to pay the loan (float).
cb_person_default_on_file: Whether the person has defaulted on a loan before (categorical: Y or N).
cb_person_cred_hist_length: Length of the individual's credit history in years (integer).
```

- **Understanding dataset**

```
Shape of data
```

```
credit_risk_data.shape
```

```
(32581, 12)
```

```
Size of data
```

```
credit_risk_data.size
```

```
390972
```

```
Column name
```

```
credit_risk_data.columns
```

```
Index(['person_age', 'person_income', 'person_home_ownership',
       'person_emp_length', 'loan_intent', 'loan_grade', 'loan_amnt',
       'loan_int_rate', 'loan_status', 'loan_percent_income',
       'cb_person_default_on_file', 'cb_person_cred_hist_length'],
      dtype='object')
```

```
Info of data
```

```
credit_risk_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32581 entries, 0 to 32580
Data columns (total 12 columns):
 #   Column                      Non-Null Count  Dtype  
---  ------                      --------------  -----  
 0   person_age                  32581 non-null  int64  
 1   person_income               32581 non-null  int64  
 2   person_home_ownership       32581 non-null  object 
 3   person_emp_length           31686 non-null  float64
 4   loan_intent                 32581 non-null  object 
 5   loan_grade                  32581 non-null  object 
 6   loan_amnt                   32581 non-null  int64  
 7   loan_int_rate               29465 non-null  float64
 8   loan_status                 32581 non-null  int64  
 9   loan_percent_income         32581 non-null  float64
 10  cb_person_default_on_file   32581 non-null  object 
 11  cb_person_cred_hist_length  32581 non-null  int64  
dtypes: float64(3), int64(5), object(4)
memory usage: 3.0+ MB
```

```
Description of data
```

```
credit_risk_data.describe()
```

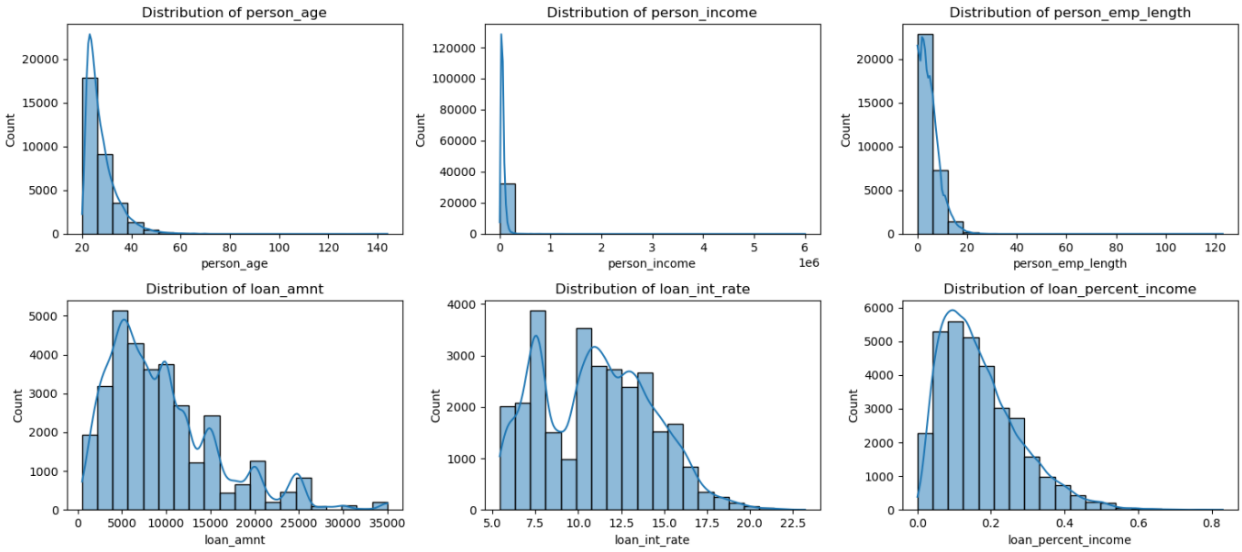
	person_age	person_income	person_emp_length	loan_amnt	loan_int_rate	loan_status	loan_percent_income	cb_person_cred_hist_length
count	32581.000000	3.258100e+04	31686.000000	32581.000000	29465.000000	32581.000000	32581.000000	32581.000000
mean	27.734600	6.607485e+04	4.789686	9589.371106	11.011695	0.218164	0.170203	5.804211
std	6.348078	6.198312e+04	4.142630	6322.086646	3.240459	0.413006	0.106782	4.055001
min	20.000000	4.000000e+03	0.000000	500.000000	5.420000	0.000000	0.000000	2.000000
25%	23.000000	3.850000e+04	2.000000	5000.000000	7.900000	0.000000	0.090000	3.000000
50%	26.000000	5.500000e+04	4.000000	8000.000000	10.990000	0.000000	0.150000	4.000000
75%	30.000000	7.920000e+04	7.000000	12200.000000	13.470000	0.000000	0.230000	8.000000
max	144.000000	6.000000e+06	123.000000	35000.000000	23.220000	1.000000	0.830000	30.000000

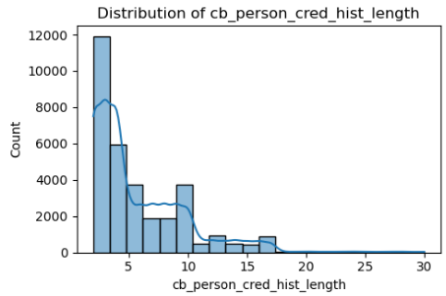
```
Finding null count
```

```
credit_risk_data.isnull().sum()
```

```
person_age                     0
person_income                  0
person_home_ownership          0
person_emp_length            895
loan_intent                    0
loan_grade                     0
loan_amnt                      0
loan_int_rate               3116
loan_status                    0
loan_percent_income            0
cb_person_default_on_file      0
cb_person_cred_hist_length     0
dtype: int64
```

- **Finding the distribution of data**

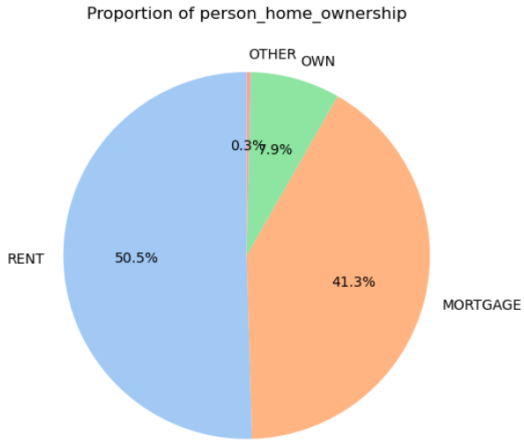


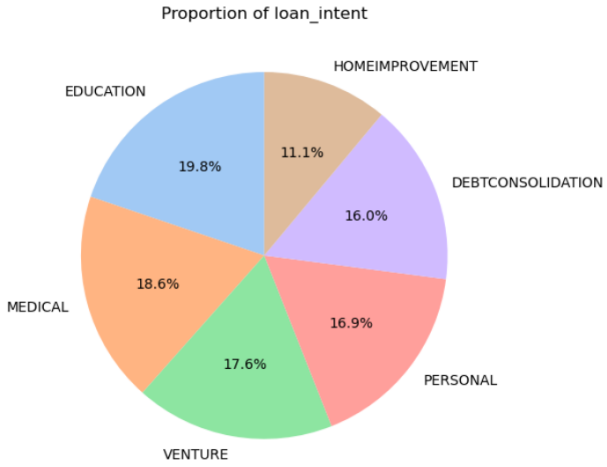


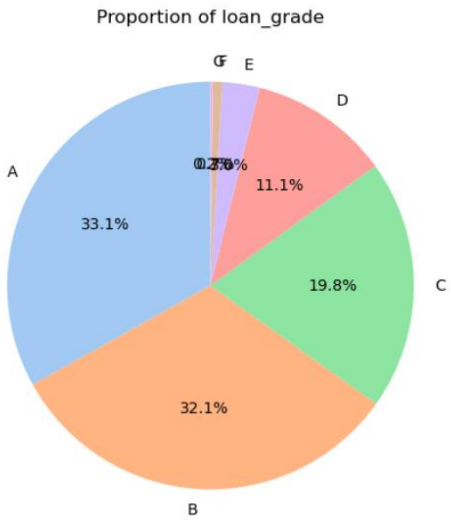
- **Filling null based on distribution**

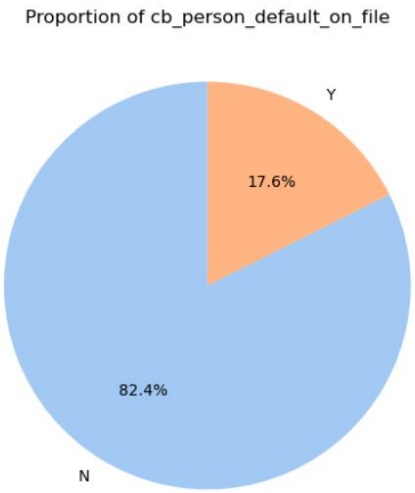
```
Missing values after filling:
person_age                  0
person_income               0
person_home_ownership       0
person_emp_length           0
loan_intent                 0
loan_grade                  0
loan_amnt                   0
loan_int_rate               0
loan_status                 0
loan_percent_income         0
cb_person_default_on_file   0
cb_person_cred_hist_length  0
dtype: int64
```

- **Proportion of data**

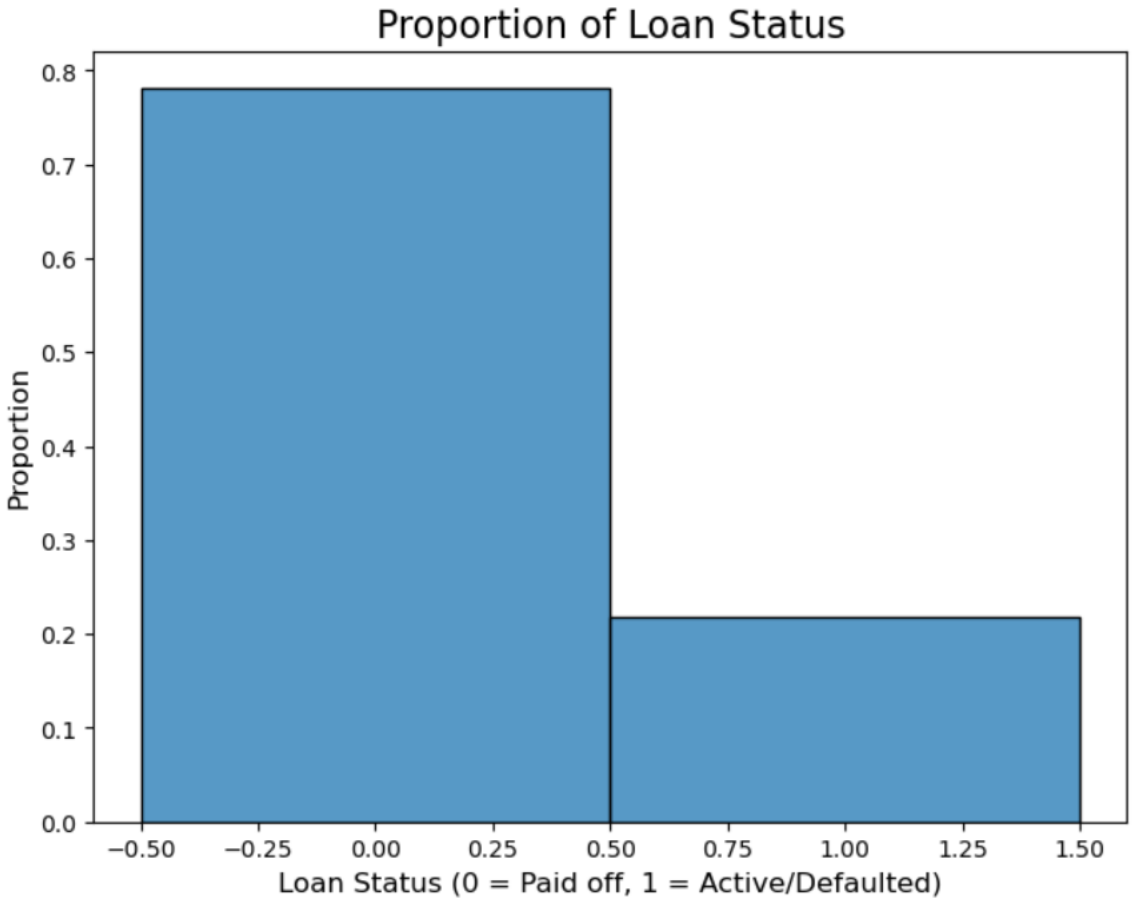








- **Target Label proportion**



- **Encoding data**

age	person_income	person_emp_length	loan_amnt	loan_int_rate	loan_status	loan_percent_income	cb_person_cred_hist_length	person_home_ownership_OTHER	person_
22	59000	123.0	35000	16.02	1	0.59	3	False	
21	9600	5.0	1000	11.14	0	0.10	2	False	
25	9600	1.0	5500	12.87	1	0.57	3	False	
23	65500	4.0	35000	15.23	1	0.53	2	False	
24	54400	8.0	35000	14.27	1	0.55	4	False	

columns

ge	person_income	person_emp_length	loan_amnt	loan_int_rate	loan_status	loan_percent_income	cb_person_cred_hist_length	person_home_ownership_OTHER	person_h
22	59000	123	35000	16	1	0	3	0	
21	9600	5	1000	11	0	0	2	0	
25	9600	1	5500	12	1	0	3	0	
23	65500	4	35000	15	1	0	2	0	
24	54400	8	35000	14	1	0	4	0	

:olumns

- **Standardizing Data**

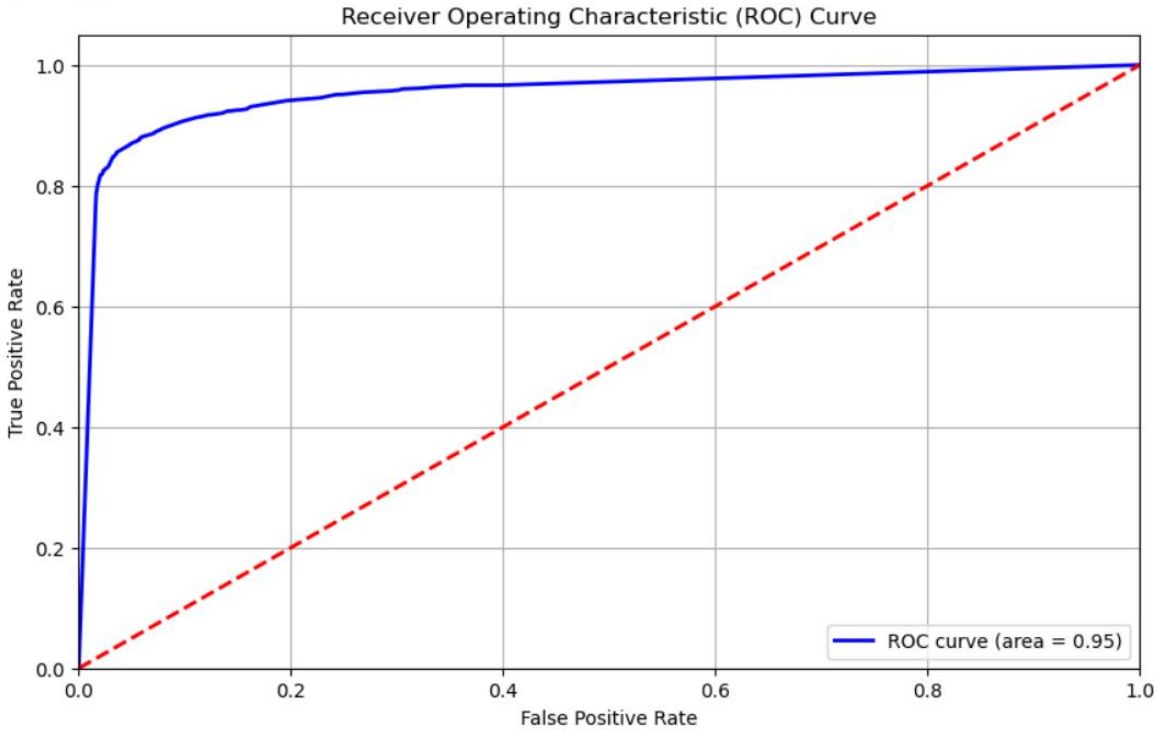
	person_age	person_income	person_emp_length	loan_amnt	loan_int_rate	loan_percent_income	cb_person_cred_hist_length	person_home_ownership_OTHER	person_h
0	-0.903374	-0.114143	28.926614	4.019404	1.769382	0.0	-0.691554	-0.057402	
1	-1.060904	-0.911147	0.056763	-1.358650	0.164538	0.0	-0.938167	-0.057402	
2	-0.430783	-0.911147	-0.921876	-0.646849	0.485507	0.0	-0.691554	-0.057402	
3	-0.745843	-0.009274	-0.187897	4.019404	1.448413	0.0	-0.938167	-0.057402	
4	-0.588313	-0.188358	0.790742	4.019404	1.127445	0.0	-0.444942	-0.057402	

5 rows × 23 columns

- **Model Building**

```
Fitting 5 folds for each of 45 candidates, totalling 225 fits
Best Parameters: {'max_depth': 15, 'min_samples_leaf': 4, 'min_samples_split': 10}
Training Accuracy: 0.9424
Test Accuracy: 0.9079

Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.92      0.91      5094
           1       0.92      0.89      0.91      5096

    accuracy                           0.91     10190
   macro avg       0.91      0.91      0.91     10190
weighted avg       0.91      0.91      0.91     10190


Confusion Matrix:
[[4691  403]
 [ 536 4560]]
```

```
Confusion Matrix:
[[4691  403]
 [ 536 4560]]
```



Result:

1.Best Parameters:
max_depth: 15
min_samples_leaf: 4
min_samples_split: 10
These hyperparameters indicate that the model has been tuned to control overfitting while ensuring that the decision tree is deep enough to capture the complexities of the data.

2. Accuracy:
Training Accuracy: 94.24%
This indicates that the model performs well on the training data, correctly classifying a high percentage of instances.

Test Accuracy: 90.79%
The test accuracy is slightly lower than the training accuracy, which is common as models typically perform better on training data. However, the test accuracy is still high, indicating that the model generalizes well to unseen data.

3. Classification Report:
Class 0 (Loan Not Default):

Precision: 90% - The model is correct 90% of the time when it predicts a loan will not default.
Recall: 92% - It captures 92% of the actual non-defaults, meaning it misses only 8% of true non-default cases.
F1-Score: 91% - The balance between precision and recall is high, indicating a robust performance for this class.

Class 1 (Loan Default):
Precision: 92% - The model correctly identifies 92% of loans that do default when it makes that prediction.
Recall: 89% - It captures 89% of the actual defaults, meaning it misses 11% of true default cases.
F1-Score: 91% - Similar to the first class, the balance between precision and recall for this class is also strong.

4. Overall Performance:
Accuracy: 91% - This indicates that the model is well-balanced in its performance across both classes. It successfully classified paidoff and active or default customers to predict on same kind of data should load is assign or not.

9. Random forest

Objective:

Objectives of the Random Forest Stroke Prediction Model -Early Detection of Strokes:
Objective: To enable early identification of patients at risk of strokes based on various health indicators, allowing for timely medical intervention.
Impact: Early detection can lead to better patient outcomes and reduced mortality rates.

Input:

- **Loading data**

```
import pandas as pd

# Load the data
file_path = r"C:\Users\roari\Downloads\Brain.csv"
data = pd.read_csv(file_path)
```

- **Understanding data**

```
data.shape
```

```
data.size
```

```
data.columns.tolist()
```

```
data.info()
```

```
data.dtypes
```

```
data.describe()
```

```
data.isnull().sum()
```

- **Distribution of data**

```
import seaborn as sns
import matplotlib.pyplot as plt

# Set the aesthetic style of the plots
sns.set(style="whitegrid")

# Create a figure and axis
plt.figure(figsize=(15, 10))

# List of numerical columns to plot
numerical_columns = ['age', 'hypertension', 'heart_disease', 'avg_glucose_level', 'bmi', 'stroke']

# Create histograms for each numerical column
for i, column in enumerate(numerical_columns):
    plt.subplot(3, 2, i + 1)  # Create a 3x2 grid of subplots
    sns.histplot(data[column], kde=True, bins=30)
    plt.title(f'Distribution of {column}')
    plt.xlabel(column)
    plt.ylabel('Frequency')

# Adjust layout
plt.tight_layout()
plt.show()
```

- **Proportion of data**

```
import seaborn as sns
import matplotlib.pyplot as plt

# Set the aesthetic style of the plots
sns.set(style="whitegrid")

# Create a figure
plt.figure(figsize=(15, 10))

# List of categorical columns to plot
categorical_columns = ['gender', 'hypertension', 'heart_disease', 'ever_married',
                       'work_type', 'Residence_type', 'smoking_status', 'stroke']

# Create pie charts for each categorical column
for i, column in enumerate(categorical_columns):
    plt.subplot(3, 3, i + 1)  # Create a 3x3 grid of subplots
    data_counts = data[column].value_counts()
    plt.pie(data_counts, labels=data_counts.index, autopct='%1.1f%%', startangle=140)
    plt.title(f'Proportion of {column}')
    plt.axis('equal')  # Equal aspect ratio ensures that pie chart is circular.

# Adjust layout
plt.tight_layout()
plt.show()
```

- **Finding the Outliers**

```
import pandas as pd

# Assuming 'data' is your DataFrame

# List of numerical columns to check for outliers
numerical_columns = ['age', 'avg_glucose_level', 'bmi', 'hypertension', 'heart_disease', 'stroke']

# Function to identify outliers using the IQR method
def detect_outliers_iqr(df, columns):
    outlier_indices = []
    for column in columns:
        Q1 = df[column].quantile(0.25)
        Q3 = df[column].quantile(0.75)
        IQR = Q3 - Q1
        outlier_step = 1.5 * IQR
        outliers = df[(df[column] < (Q1 - outlier_step)) | (df[column] > (Q3 + outlier_step))]
        outlier_indices.extend(outliers.index.tolist())
    return list(set(outlier_indices))  # Return unique indices of outliers

# Detect outliers
outlier_indices_iqr = detect_outliers_iqr(data, numerical_columns)

# Display the outliers
outliers_iqr = data.loc[outlier_indices_iqr]
print("Outliers detected using IQR method:")
outliers_iqr
```

- **Dropping outliers**

```python
import pandas as pd

# Assuming 'data' is your DataFrame
# Sample data creation (Replace this with your actual DataFrame loading)
# data = pd.read_csv('C:\\Users\\roari\\Downloads\\Brain.csv')

# Define the numerical columns
numerical_columns = ['age', 'avg_glucose_level', 'bmi']

# Function to remove outliers using the IQR method
def remove_outliers_iqr(df, columns):
    df_cleaned = df.copy()  # Create a copy of the DataFrame
    for column in columns:
        Q1 = df_cleaned[column].quantile(0.25)
        Q3 = df_cleaned[column].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        # Remove outliers
        df_cleaned = df_cleaned[(df_cleaned[column] >= lower_bound) & (df_cleaned[column] <= upper_bound)]

    return df_cleaned

# Remove outliers
cleaned_data = remove_outliers_iqr(data, numerical_columns)

# Display the shape of the cleaned DataFrame
print("Original data shape:", data.shape)
print("Cleaned data shape:", cleaned_data.shape)

# Display the cleaned DataFrame
print("Cleaned DataFrame:")
cleaned_data
```

- **Encoding**

```python
import pandas as pd

# Assuming 'data' is your DataFrame
# Sample data creation (Replace this with your actual DataFrame loading)
# data = pd.read_csv('C:\\Users\\roari\\Downloads\\Brain.csv')

# Performing one-hot encoding
data_encoded = pd.get_dummies(data, columns=['gender', 'ever_married', 'work_type', 'Residence_type', 'smoking_status'], drop_first=True)

# Display the shape and the first few rows of the encoded DataFrame
print("Shape of the encoded DataFrame:", data_encoded.shape)
print("First few rows of the encoded DataFrame:")
data_encoded.head()
```

```
Converting to binary (0 and 1)
```

```python
# Transform True/False to 1/0
data_encoded = data_encoded.astype(int)

# Display the transformed DataFrame
print("Transformed DataFrame:")
data_encoded
```

- **Counting stroke distribution**

```python
import pandas as pd

# Assuming 'data_encoded' is your DataFrame after one-hot encoding
# Sample data creation (Replace this with your actual DataFrame loading)
# data_encoded = pd.get_dummies(data, columns=['gender', 'ever_married', 'work_type', 'Residence_type', 'smoking_status'], drop_first=True)

# Calculating the count of stroke occurrences
stroke_counts = data_encoded['stroke'].value_counts()

# Calculating the proportion of each stroke value
stroke_proportion = stroke_counts / stroke_counts.sum()

# Display the results
print("Count of Stroke occurrences:")
print(stroke_counts)
print("\nProportion of Stroke occurrences (0 and 1):")
print(stroke_proportion)
```

- **Balancing the stroke data**

```python
import pandas as pd
from imblearn.over_sampling import SMOTE

# Assuming 'data_encoded' is your DataFrame
# Separate the features and the target variable
X = data_encoded.drop('stroke', axis=1)  # Features
y = data_encoded['stroke']               # Target variable

# Create an instance of SMOTE
smote = SMOTE(random_state=42)

# Fit and resample the data
X_resampled, y_resampled = smote.fit_resample(X, y)

# Combine resampled features and target into a new DataFrame
data_balanced = pd.DataFrame(X_resampled, columns=X.columns)
data_balanced['stroke'] = y_resampled

# Display the new balance of the dataset
stroke_counts_balanced = data_balanced['stroke'].value_counts()
print("Count of Stroke occurrences after balancing:")
print(stroke_counts_balanced)

# Proportions after balancing
stroke_proportion_balanced = stroke_counts_balanced / stroke_counts_balanced.sum()
print("\nProportion of Stroke occurrences after balancing:")
print(stroke_proportion_balanced)
```

- **Model building**

```python
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, GridSearchCV, KFold
from sklearn.metrics import accuracy_score, precision_recall_fscore_support, confusion_matrix, roc_curve, roc_auc_score
import matplotlib.pyplot as plt
import seaborn as sns

# Assuming 'data_balanced' is your balanced DataFrame after SMOTE
# Split the data into features and target variable
X = data_balanced.drop('stroke', axis=1)  # Features
y = data_balanced['stroke']               # Target variable

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

# Define the Random Forest model
rf_model = RandomForestClassifier(random_state=42)

# Define hyperparameter grid for tuning
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2],
    'bootstrap': [True, False]
}

# Create a GridSearchCV object
grid_search = GridSearchCV(estimator=rf_model, param_grid=param_grid,
                           cv=5, n_jobs=-1, verbose=2, scoring='f1')

# Fit GridSearchCV
grid_search.fit(X_train, y_train)
```

```python
# Best parameters from GridSearchCV
print("Best parameters found: ", grid_search.best_params_)

# Train the model with the best parameters
best_rf_model = grid_search.best_estimator_

# Predictions
y_pred = best_rf_model.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Train Accuracy: {:.2f}%".format(grid_search.best_score_ * 100))
print("Test Accuracy: {:.2f}%".format(accuracy * 100))

# Calculate Precision, Recall, F1 Score, and Support
precision, recall, f1, support = precision_recall_fscore_support(y_test, y_pred)
print("\nPrecision: ", precision)
print("Recall: ", recall)
print("F1 Score: ", f1)
print("Support: ", support)

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=['No Stroke', 'Stroke'],
            yticklabels=['No Stroke', 'Stroke'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()
```

```
# ROC Curve
y_probs = best_rf_model.predict_proba(X_test)[:, 1]  # Probabilities for the positive class
fpr, tpr, thresholds = roc_curve(y_test, y_probs)
roc_auc = roc_auc_score(y_test, y_probs)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', label='ROC Curve (area = {:.2f})'.format(roc_auc))
plt.plot([0, 1], [0, 1], color='red', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend()
plt.show()
```

Output:

- **Loading data**

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	Male	67.0	0	1	Yes	Private	Urban	228.69	36.6	formerly smoked	1
1	Male	80.0	0	1	Yes	Private	Rural	105.92	32.5	never smoked	1
2	Female	49.0	0	0	Yes	Private	Urban	171.23	34.4	smokes	1
3	Female	79.0	1	0	Yes	Self-employed	Rural	174.12	24.0	never smoked	1
4	Male	81.0	0	0	Yes	Private	Urban	186.21	29.0	formerly smoked	1
...	...	...	...	...	...	...	...	...	...	...	...
4976	Male	41.0	0	0	No	Private	Rural	70.15	29.8	formerly smoked	0
4977	Male	40.0	0	0	Yes	Private	Urban	191.15	31.1	smokes	0
4978	Female	45.0	1	0	Yes	Govt_job	Rural	95.02	31.8	smokes	0
4979	Male	40.0	0	0	Yes	Private	Rural	83.94	30.0	smokes	0
4980	Female	80.0	1	0	Yes	Private	Urban	83.75	29.1	never smoked	0

4981 rows × 11 columns

- **Understanding data**

```
gender: The gender of the individual (Male or Female).
age: The age of the individual in years.
hypertension: Indicates whether the individual has hypertension (1 for Yes, 0 for No).
heart_disease: Indicates whether the individual has heart disease (1 for Yes, 0 for No).
ever_married: Indicates if the individual has ever been married (Yes or No).
work_type: The type of work the individual is engaged in (e.g., Private, Self-employed, Govt_job).
Residence_type: Indicates the type of residence (Urban or Rural).
avg_glucose_level: The average glucose level in the individual's blood.
bmi: The Body Mass Index (BMI) of the individual, a measure of body fat based on height and weight.
smoking_status: The smoking status of the individual (e.g., smokes, never smoked, formerly smoked).
stroke: Indicates whether the individual has had a stroke (1 for Yes, 0 for No).
```

Shape of the data

```
data.shape
```

```
(4981, 11)
```

Size of data

```
data.size
```

```
54791
```

Column name

```
data.columns.tolist()
```

```
['gender',
 'age',
 'hypertension',
 'heart_disease',
 'ever_married',
 'work_type',
 'Residence_type',
 'avg_glucose_level',
 'bmi',
 'smoking_status',
 'stroke']
```

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4981 entries, 0 to 4980
Data columns (total 11 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   gender             4981 non-null   object 
 1   age                4981 non-null   float64
 2   hypertension       4981 non-null   int64  
 3   heart_disease      4981 non-null   int64  
 4   ever_married       4981 non-null   object 
 5   work_type          4981 non-null   object 
 6   Residence_type     4981 non-null   object 
 7   avg_glucose_level  4981 non-null   float64
 8   bmi                4981 non-null   float64
 9   smoking_status     4981 non-null   object 
 10  stroke             4981 non-null   int64  
dtypes: float64(3), int64(3), object(5)
memory usage: 428.2+ KB
```

Data type

```
data.dtypes
```

```
gender              object
age                float64
hypertension         int64
heart_disease        int64
ever_married        object
work_type           object
Residence_type      object
avg_glucose_level  float64
bmi                float64
smoking_status      object
stroke               int64
dtype: object
```

```
data.describe()
```

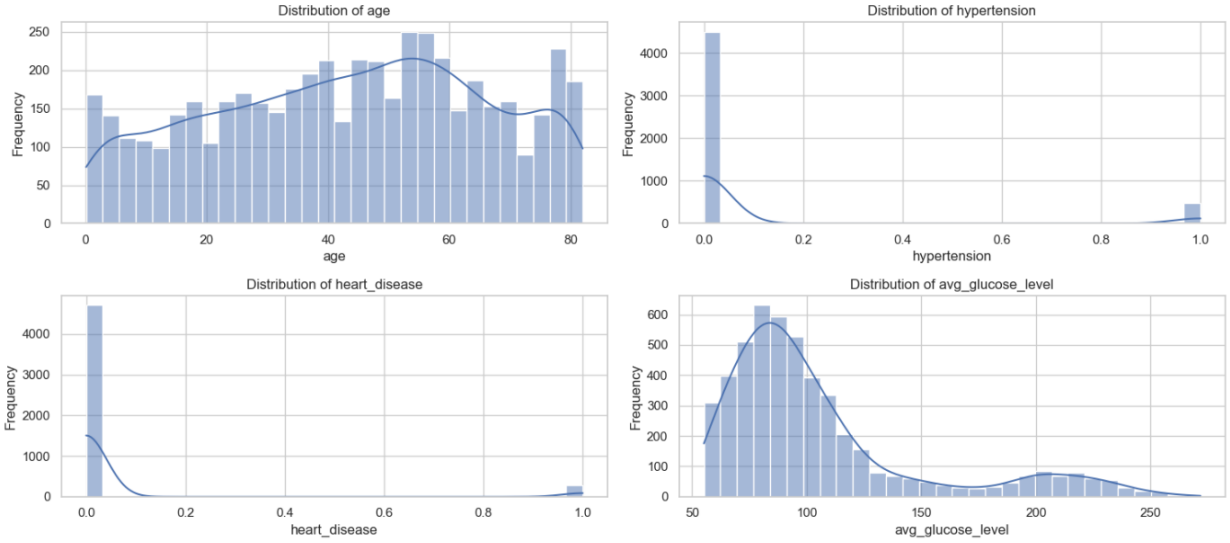
	age	hypertension	heart_disease	avg_glucose_level	bmi	stroke
count	4981.000000	4981.000000	4981.000000	4981.000000	4981.000000	4981.000000
mean	43.419859	0.096165	0.055210	105.943562	28.498173	0.049789
std	22.662755	0.294848	0.228412	45.075373	6.790464	0.217531
min	0.080000	0.000000	0.000000	55.120000	14.000000	0.000000
25%	25.000000	0.000000	0.000000	77.230000	23.700000	0.000000
50%	45.000000	0.000000	0.000000	91.850000	28.100000	0.000000
75%	61.000000	0.000000	0.000000	113.860000	32.600000	0.000000
max	82.000000	1.000000	1.000000	271.740000	48.900000	1.000000

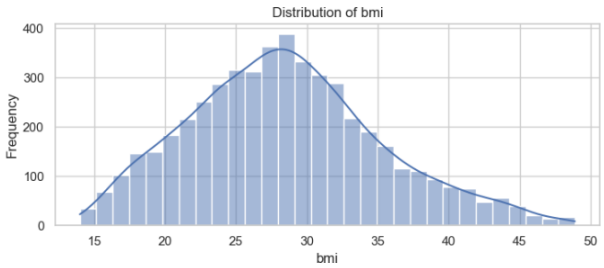
```
Null values count
```

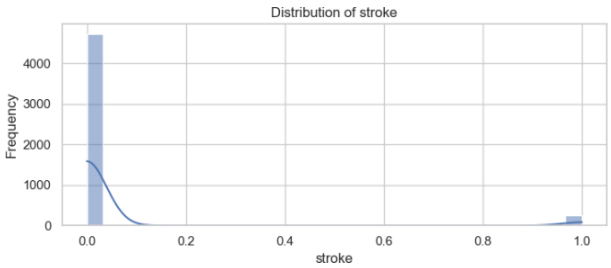
```
data.isnull().sum()
```

```
gender              0
age                 0
hypertension        0
heart_disease       0
ever_married        0
work_type           0
Residence_type      0
avg_glucose_level   0
bmi                 0
smoking_status      0
stroke              0
dtype: int64
```

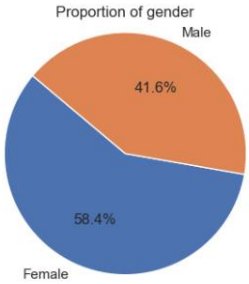
- **Distribution of data**

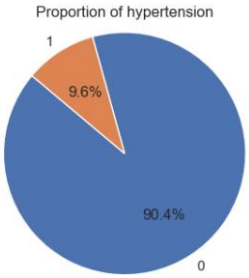


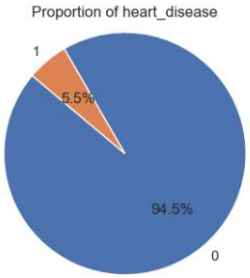


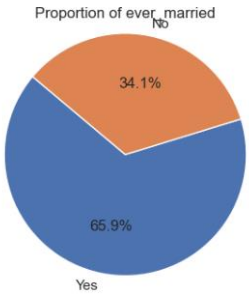


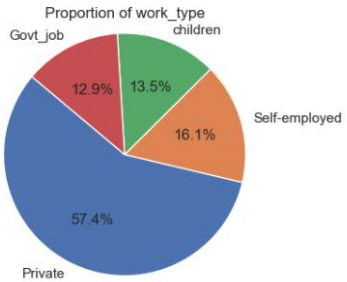
- **Proportion of data**

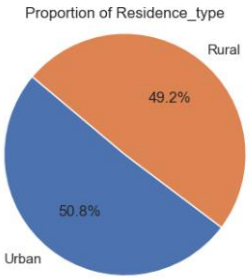


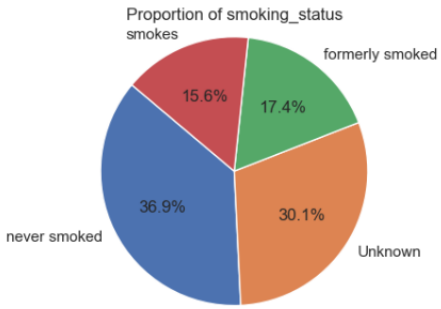


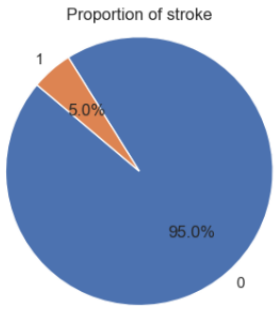












- **Finding the Outliers**

Outliers detected using IQR method:

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	Male	67.0	0	1	Yes	Private	Urban	228.69	36.6	formerly smoked	1
1	Male	80.0	0	1	Yes	Private	Rural	105.92	32.5	never smoked	1
2	Female	49.0	0	0	Yes	Private	Urban	171.23	34.4	smokes	1
3	Female	79.0	1	0	Yes	Self-employed	Rural	174.12	24.0	never smoked	1
4	Male	81.0	0	0	Yes	Private	Urban	186.21	29.0	formerly smoked	1
...	...	...	...	...	...	...	...	...	...	...	...
2035	Female	80.0	0	1	Yes	Self-employed	Rural	103.06	28.8	never smoked	0
4077	Female	81.0	0	1	Yes	Govt_job	Urban	90.11	28.6	never smoked	0
162	Female	74.0	0	0	Yes	Self-employed	Urban	74.96	26.6	never smoked	1
2040	Male	78.0	1	0	Yes	Self-employed	Rural	75.19	27.6	never smoked	0
4092	Female	71.0	1	1	Yes	Private	Rural	221.24	24.2	Unknown	0

1211 rows × 11 columns

- **Dropping outliers**

```
Original data shape: (4981, 11)
Cleaned data shape: (4337, 11)
Cleaned DataFrame:
```

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
1	Male	80.0	0	1	Yes	Private	Rural	105.92	32.5	never smoked	1
5	Male	74.0	1	1	Yes	Private	Rural	70.09	27.4	never smoked	1
6	Female	69.0	0	0	No	Private	Urban	94.39	22.8	never smoked	1
7	Female	78.0	0	0	Yes	Private	Urban	58.57	24.2	Unknown	1
8	Female	81.0	1	0	Yes	Private	Rural	80.43	29.7	never smoked	1
...	...	...	...	...	...	...	...	...	...	...	...
4974	Male	58.0	0	0	Yes	Govt_job	Urban	84.94	30.2	never smoked	0
4976	Male	41.0	0	0	No	Private	Rural	70.15	29.8	formerly smoked	0
4978	Female	45.0	1	0	Yes	Govt_job	Rural	95.02	31.8	smokes	0
4979	Male	40.0	0	0	Yes	Private	Rural	83.94	30.0	smokes	0
4980	Female	80.0	1	0	Yes	Private	Urban	83.75	29.1	never smoked	0

4337 rows × 11 columns

- **Encoding**

	age	hypertension	heart_disease	avg_glucose_level	bmi	stroke	gender_Male	ever_married_Yes	work_type_Private	work_type_Self-employed	work_type_children	Residence_
0	67.0	0	1	228.69	36.6	1	True	True	True	False	False	
1	80.0	0	1	105.92	32.5	1	True	True	True	False	False	
2	49.0	0	0	171.23	34.4	1	False	True	True	False	False	
3	79.0	1	0	174.12	24.0	1	False	True	False	True	False	
4	81.0	0	0	186.21	29.0	1	True	True	True	False	False	

	age	hypertension	heart_disease	avg_glucose_level	bmi	stroke	gender_Male	ever_married_Yes	work_type_Private	work_type_Self-employed	work_type_children	Residen
0	67	0	1	228	36	1	1	1	1	0	0	
1	80	0	1	105	32	1	1	1	1	0	0	
2	49	0	0	171	34	1	0	1	1	0	0	
3	79	1	0	174	24	1	0	1	0	1	0	
4	81	0	0	186	29	1	1	1	1	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	
4976	41	0	0	70	29	0	1	0	1	0	0	
4977	40	0	0	191	31	0	1	1	1	0	0	
4978	45	1	0	95	31	0	0	1	0	0	0	
4979	40	0	0	83	30	0	1	1	1	0	0	
4980	80	1	0	83	29	0	0	1	1	0	0	

4981 rows × 15 columns

- **Counting stroke distribution**

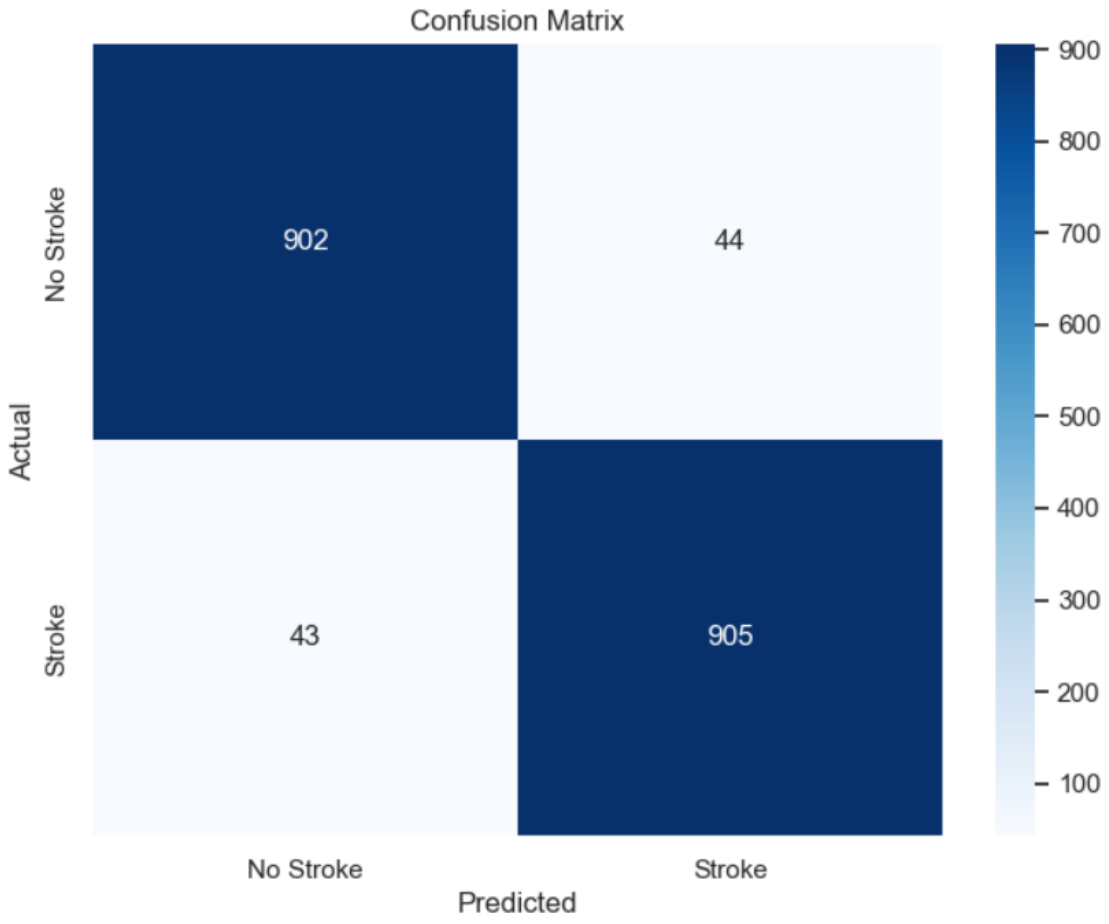
```
Count of Stroke occurrences:
stroke
0    4733
1     248
Name: count, dtype: int64

Proportion of Stroke occurrences (0 and 1):
stroke
0    0.950211
1    0.049789
Name: count, dtype: float64
```

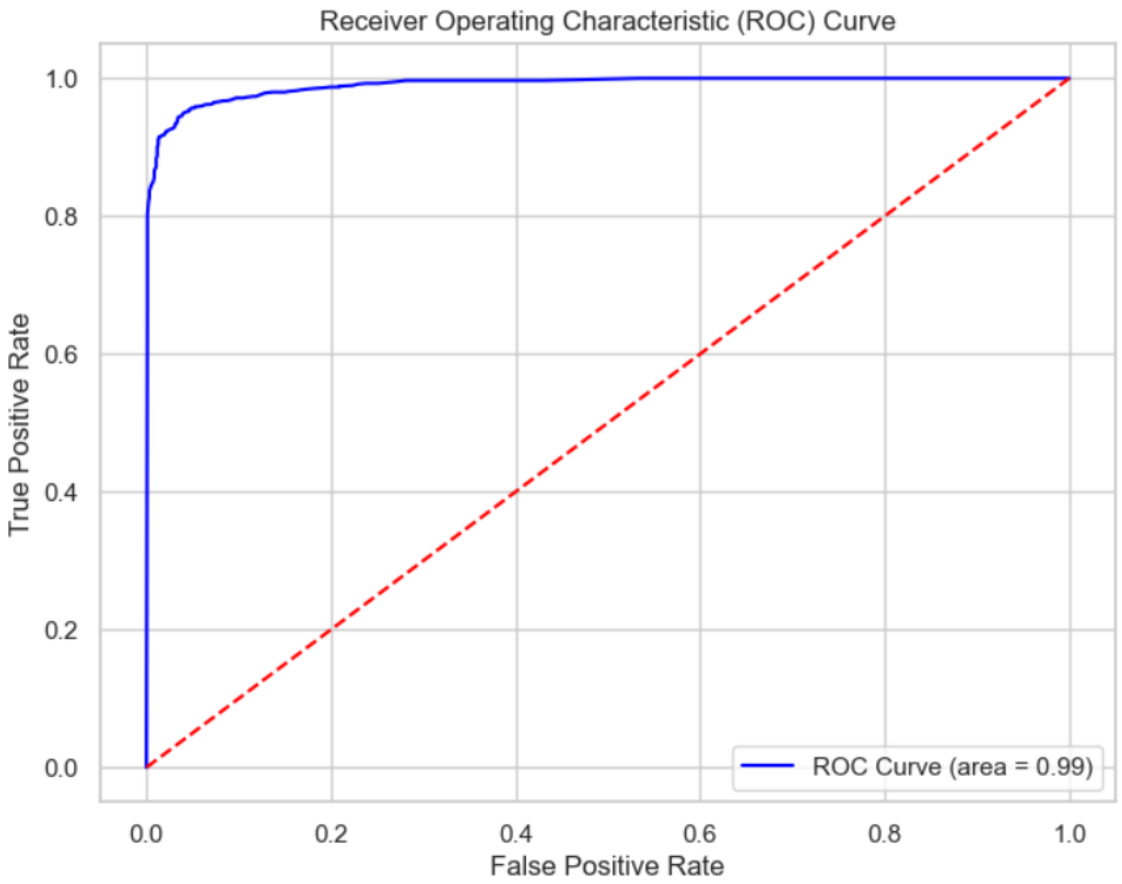
- **Balancing the stroke data**

```
Count of Stroke occurrences after balancing:
stroke
1    4733
0    4733
Name: count, dtype: int64

Proportion of Stroke occurrences after balancing:
stroke
1    0.5
0    0.5
Name: count, dtype: float64
```

- **Model building**

```
Fitting 5 folds for each of 48 candidates, totalling 240 fits
Best parameters found:  {'bootstrap': False, 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
Train Accuracy: 95.12%
Test Accuracy: 95.41%

Precision:  [0.95449735 0.95363541]
Recall:  [0.95348837 0.95464135]
F1 Score:  [0.9539926  0.95413811]
Support:  [946 948]
```


Confusion Matrix
902
44
43
905
No Stroke
Stroke
Actual
No Stroke
Stroke
Predicted
900
800
700
600
500
400
300
200
100



Result:

Model Performance Summary

Train Accuracy: 95.12%

This indicates that the model correctly predicts 95.12% of the training data, suggesting it has learned the patterns in the training dataset well.

Test Accuracy: 95.41%

This indicates that the model performs well on unseen data, with 95.41% accuracy on the test set. This is a good sign of generalization, as it suggests the model is not overfitting the training data.

Precision:

Precision for Class 0 (No Stroke): 95.45%

Precision for Class 1 (Stroke): 95.36%

Precision indicates the percentage of true positive predictions out of all positive predictions. High precision for both classes shows that when the model predicts a stroke or no stroke, it is likely to be correct.

Recall:

Recall for Class 0: 95.35%
Recall for Class 1: 95.46%
Recall measures the percentage of actual positives that were correctly identified. High recall for both classes indicates that the model is effective at capturing most of the true cases (both strokes and non-strokes).

F1 Score:
F1 Score for Class 0: 95.40%
F1 Score for Class 1: 95.41%
The F1 score is the harmonic mean of precision and recall. High F1 scores for both classes indicate a balanced performance, suggesting that the model maintains a good trade-off between precision and recall.

Support:
Support for Class 0: 946
Support for Class 1: 948
Support refers to the number of actual occurrences of each class in the specified dataset. This indicates a relatively balanced dataset with approximately equal numbers of strokes and non-strokes.

Conclusion:

Overall Model Effectiveness: The model shows high accuracy, precision, recall, and F1 scores, indicating it is highly effective at predicting stroke occurrences while maintaining a low rate of false predictions.

10. Gradient boosting classifier

Objective:

Finding is there is maintenance failure or not

Input:

- **Loading data and renaming**

```
import pandas as pd

# Load the data
file_path = r"C:\Users\roari\Downloads\Machine Learning\failures.csv"
data = pd.read_csv(file_path)

# Original column names
original_columns = data.columns

# Simplified column names
simplified_columns = {
    'UDI': 'UDI',
    'Product ID': 'Product_ID',
    'Type': 'Type',
    'Air temperature [K]': 'Air_Temp_K',
    'Process temperature [K]': 'Process_Temp_K',
    'Rotational speed [rpm]': 'Rotational_Speed_rpm',
    'Torque [Nm]': 'Torque_Nm',
    'Tool wear [min]': 'Tool_Wear_min',
    'Target': 'Target',
    'Failure Type': 'Failure_Type'
}

# Rename the columns in the DataFrame
data.rename(columns=simplified_columns, inplace=True)
```

```
# Print the meanings of each column
meanings = {
    'UDI': 'Unique Device Identifier',
    'Product_ID': 'Identifier for the product',
    'Type': 'Type of the product',
    'Air_Temp_K': 'Air temperature in Kelvin',
    'Process_Temp_K': 'Process temperature in Kelvin',
    'Rotational_Speed_rpm': 'Speed of the rotating component in revolutions per minute',
    'Torque_Nm': 'Torque applied in Newton meters',
    'Tool_Wear_min': 'Minutes the tool has been in use',
    'Target': 'Target variable indicating the success criteria',
    'Failure_Type': 'Type of failure (if any) occurred'
}

# Output the new DataFrame and column meanings
print("Renamed Columns:")
print(data.head())  # Show the first few rows of the renamed DataFrame
print("\nMeanings of Each Column:")
for col, meaning in meanings.items():
    print(f"{col}: {meaning}")
```

- **Understanding data**

```
# Get the shape of the DataFrame
shape = data.shape
shape
```

```
# Get the column names
columns = data.columns.tolist()
columns
```

```
# Get DataFrame info
info = data.info()
info
```

```
# Get descriptive statistics
description = data.describe()
description
```

```
# Get data types of each column
data_types = data.dtypes
data_types
```

```
# Count of null values in each column
null_counts = data.isnull().sum()
null_counts
```

- **Dropping unwanted column**

```
# Drop the specified columns
data_dropped = data.drop(columns=['UDI', 'Product_ID'])

print("*" * 125)

# Display the first few rows of the modified DataFrame
print("DataFrame after dropping 'UDI' and 'Product_ID':")
print(data_dropped.head())

print("*" * 125)

# Optionally, check the shape and remaining columns
print("\nNew Shape of DataFrame:", data_dropped.shape)
print("\nRemaining Columns:", data_dropped.columns.tolist())
```

- **Distribution of features**

```
import seaborn as sns
import matplotlib.pyplot as plt

# Set the style of seaborn
sns.set(style="whitegrid")

# List of numeric columns to plot
numeric_columns = ['Air_Temp_K', 'Process_Temp_K', 'Rotational_Speed_rpm',
                   'Torque_Nm', 'Tool_Wear_min', 'Target']

# Create a histogram for each numeric column
plt.figure(figsize=(15, 10))
for i, column in enumerate(numeric_columns, 1):
    plt.subplot(2, 3, i)  # Create a subplot for each column
    sns.histplot(data_dropped[column], bins=30, kde=True)  # kde=True adds a density curve
    plt.title(f'Distribution of {column}')
    plt.xlabel(column)
    plt.ylabel('Frequency')

# Adjust layout and show the plots
plt.tight_layout()
plt.show()
```

- **Proportion of failure type**

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import warnings

# Suppress FutureWarnings
warnings.simplefilter(action='ignore', category=FutureWarning)

# Set the style of seaborn
sns.set(style="whitegrid")

# Create a figure with larger size
plt.figure(figsize=(10, 12))

# Distribution of Type
plt.subplot(2, 1, 1)  # 2 rows, 1 column, 1st subplot
sns.countplot(data=data_dropped, x='Type', palette='viridis')
plt.title('Distribution of Type')
plt.xlabel('Type')
plt.ylabel('Count')

# Distribution of Failure Type
plt.subplot(2, 1, 2)  # 2 rows, 1 column, 2nd subplot
sns.countplot(data=data_dropped, x='Failure_Type', palette='viridis')
plt.title('Distribution of Failure Type')
plt.xlabel('Failure Type')
plt.ylabel('Count')

# Adjust layout and show the plots
plt.tight_layout()
plt.show()
```

- **Identifying the duplicate rows**

```
# Identify duplicate rows
duplicates = data_dropped[data_dropped.duplicated(keep=False)]

# Display the duplicate rows
print("Duplicate Rows in the DataFrame:")
print(duplicates)

# Count of duplicate rows
num_duplicates = duplicates.shape[0]
print("\nNumber of Duplicate Rows:", num_duplicates)
```

- **Understanding distribution of target**

```
# Import necessary libraries
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import warnings

# Suppress FutureWarnings
warnings.simplefilter(action='ignore', category=FutureWarning)

# Set the style of seaborn
sns.set(style="whitegrid")

# Create a figure for the histogram
plt.figure(figsize=(10, 6))

# Plot the distribution of the Target column
sns.countplot(data=data_dropped, x='Target', palette='viridis')
plt.title('Distribution of Target Column')
plt.xlabel('Target')
plt.ylabel('Count')

# Show the plot
plt.tight_layout()
plt.show()

# Text format for distribution of Target column
target_distribution = data_dropped['Target'].value_counts().reset_index()
target_distribution.columns = ['Target', 'Count']

# Display the text format
print("Distribution of Target Column:")
print(target_distribution)
```

- **Encoding and standardizing data**

```
# Import necessary libraries
import pandas as pd
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer

# Assuming 'data' is your DataFrame

# Separate features and target
X = data.drop('Target', axis=1)
y = data['Target']

# Define which columns are categorical and numerical
categorical_cols = ['Type', 'Failure_Type']
numerical_cols = ['Air_Temp_K', 'Process_Temp_K', 'Rotational_Speed_rpm', 'Torque_Nm', 'Tool_Wear_min']

# Create a preprocessor that will handle both scaling and one-hot encoding
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_cols),  # Standardize numerical columns
        ('cat', OneHotEncoder(), categorical_cols)  # One-hot encode categorical columns
    ])

# Apply transformations to the data
X_processed = preprocessor.fit_transform(X)

# Convert the result back to a DataFrame with feature names
X_processed_df = pd.DataFrame(X_processed, columns=preprocessor.get_feature_names_out())

# Check the head of the transformed data
print(X_processed_df.head())
```

- **Model Building**

```python
# Gradient Boosting
from sklearn.ensemble import GradientBoostingClassifier

# Initialize and train the model
gb_model = GradientBoostingClassifier()
gb_model.fit(X_train, y_train)

# Predictions
y_train_pred_gb = gb_model.predict(X_train)
y_test_pred_gb = gb_model.predict(X_test)

# Evaluate model
gb_accuracy_train = accuracy_score(y_train, y_train_pred_gb)
gb_accuracy_test = accuracy_score(y_test, y_test_pred_gb)
gb_report = classification_report(y_test, y_test_pred_gb, output_dict=True)
gb_cm = confusion_matrix(y_test, y_test_pred_gb)
y_prob_gb = gb_model.predict_proba(X_test)[:, 1]
gb_fpr, gb_tpr, _ = roc_curve(y_test, y_prob_gb)
gb_roc_auc = roc_auc_score(y_test, y_prob_gb)

# Display results
print("\nGradient Boosting Results:")
print(f"Training Accuracy: {gb_accuracy_train:.4f}")
print(f"Testing Accuracy: {gb_accuracy_test:.4f}")
print("Classification Report:\n", gb_report)
print("Confusion Matrix:\n", gb_cm)
print(f"ROC AUC: {gb_roc_auc:.4f}")
```

```python
# Plot ROC Curve
plt.figure(figsize=(8, 6))
plt.plot(gb_fpr, gb_tpr, label=f'ROC Curve (area = {gb_roc_auc:.2f})')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Gradient Boosting')
plt.legend(loc="lower right")
plt.show()
```

Output:

- **Loading data and renaming**

```
Renamed Columns:
   UDI Product_ID Type  Air_Temp_K  Process_Temp_K  Rotational_Speed_rpm  \
0    1     M14860    M       298.1           308.6                  1551
1    2     L47181    L       298.2           308.7                  1408
2    3     L47182    L       298.1           308.5                  1498
3    4     L47183    L       298.2           308.6                  1433
4    5     L47184    L       298.2           308.7                  1408

   Torque_Nm  Tool_Wear_min  Target Failure_Type
0       42.8              0       0   No Failure
1       46.3              3       0   No Failure
2       49.4              5       0   No Failure
3       39.5              7       0   No Failure
4       40.0              9       0   No Failure

Meanings of Each Column:
UDI: Unique Device Identifier
Product_ID: Identifier for the product
Type: Type of the product
Air_Temp_K: Air temperature in Kelvin
Process_Temp_K: Process temperature in Kelvin
Rotational_Speed_rpm: Speed of the rotating component in revolutions per minute
Torque_Nm: Torque applied in Newton meters
Tool_Wear_min: Minutes the tool has been in use
Target: Target variable indicating the success criteria
Failure_Type: Type of failure (if any) occurred
```

- **Understanding data**

```
data.head()
```

	UDI	Product_ID	Type	Air_Temp_K	Process_Temp_K	Rotational_Speed_rpm	Torque_Nm	Tool_Wear_min	Target	Failure_Type
0	1	M14860	M	298.1	308.6	1551	42.8	0	0	No Failure
1	2	L47181	L	298.2	308.7	1408	46.3	3	0	No Failure
2	3	L47182	L	298.1	308.5	1498	49.4	5	0	No Failure
3	4	L47183	L	298.2	308.6	1433	39.5	7	0	No Failure
4	5	L47184	L	298.2	308.7	1408	40.0	9	0	No Failure

```
# Get the shape of the DataFrame
shape = data.shape
shape
```

```
(10000, 10)
```

```
# Get the column names
columns = data.columns.tolist()
columns
```

```
['UDI',
 'Product_ID',
 'Type',
 'Air_Temp_K',
 'Process_Temp_K',
 'Rotational_Speed_rpm',
 'Torque_Nm',
 'Tool_Wear_min',
 'Target',
 'Failure_Type']
```

```
# Get DataFrame info
info = data.info()
info
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 10 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   UDI                   10000 non-null  int64  
 1   Product_ID            10000 non-null  object 
 2   Type                  10000 non-null  object 
 3   Air_Temp_K            10000 non-null  float64
 4   Process_Temp_K        10000 non-null  float64
 5   Rotational_Speed_rpm  10000 non-null  int64  
 6   Torque_Nm             10000 non-null  float64
 7   Tool_Wear_min         10000 non-null  int64  
 8   Target                10000 non-null  int64  
 9   Failure_Type          10000 non-null  object 
dtypes: float64(3), int64(4), object(3)
memory usage: 781.4+ KB
```

```
# Get descriptive statistics
description = data.describe()
description
```

	UDI	Air_Temp_K	Process_Temp_K	Rotational_Speed_rpm	Torque_Nm	Tool_Wear_min	Target
count	10000.00000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000
mean	5000.50000	300.004930	310.005560	1538.776100	39.986910	107.951000	0.033900
std	2886.89568	2.000259	1.483734	179.284096	9.968934	63.654147	0.180981
min	1.00000	295.300000	305.700000	1168.000000	3.800000	0.000000	0.000000
25%	2500.75000	298.300000	308.800000	1423.000000	33.200000	53.000000	0.000000
50%	5000.50000	300.100000	310.100000	1503.000000	40.100000	108.000000	0.000000
75%	7500.25000	301.500000	311.100000	1612.000000	46.800000	162.000000	0.000000
max	10000.00000	304.500000	313.800000	2886.000000	76.600000	253.000000	1.000000

```
# Get data types of each column
data_types = data.dtypes
data_types
```

```
UDI                        int64
Product_ID                object
Type                      object
Air_Temp_K               float64
Process_Temp_K           float64
Rotational_Speed_rpm       int64
Torque_Nm                float64
Tool_Wear_min              int64
Target                     int64
Failure_Type              object
dtype: object
```

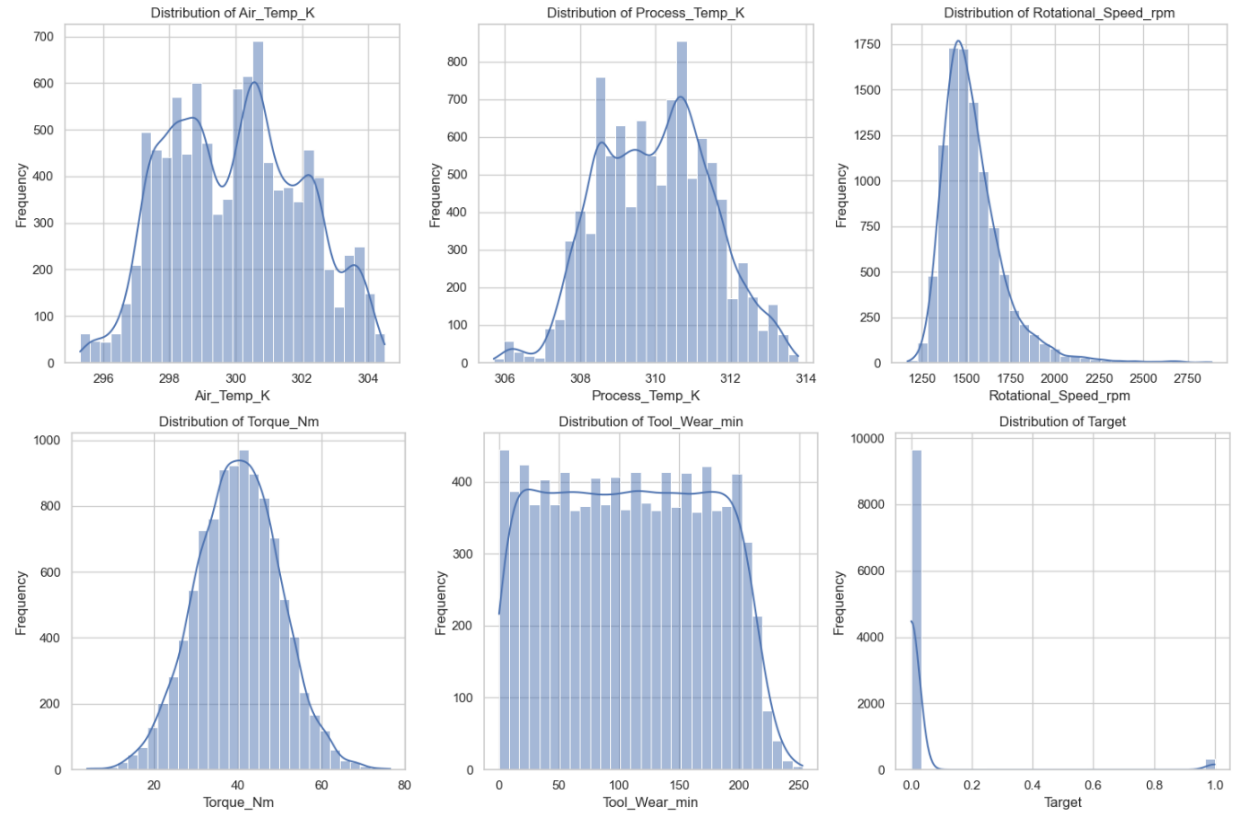
```
# Count of null values in each column
null_counts = data.isnull().sum()
null_counts
```

```
UDI                        0
Product_ID                 0
Type                       0
Air_Temp_K                 0
Process_Temp_K             0
Rotational_Speed_rpm       0
Torque_Nm                  0
Tool_Wear_min              0
Target                     0
Failure_Type               0
dtype: int64
```

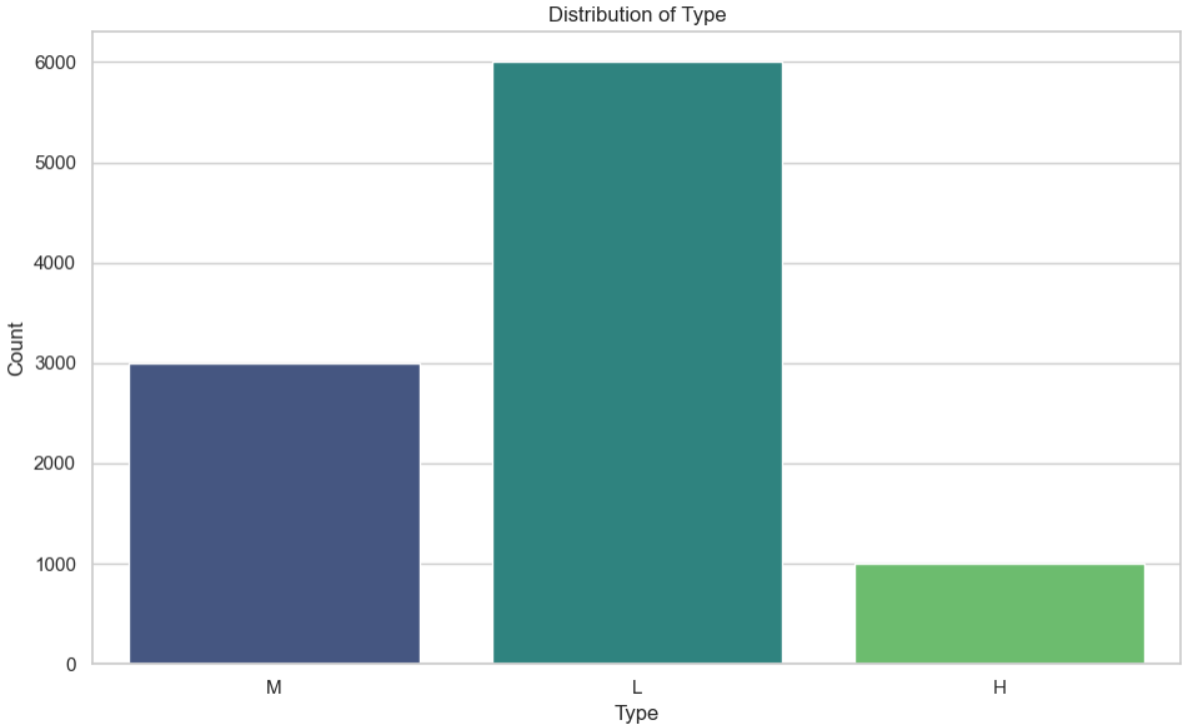
- **Dropping unwanted column**

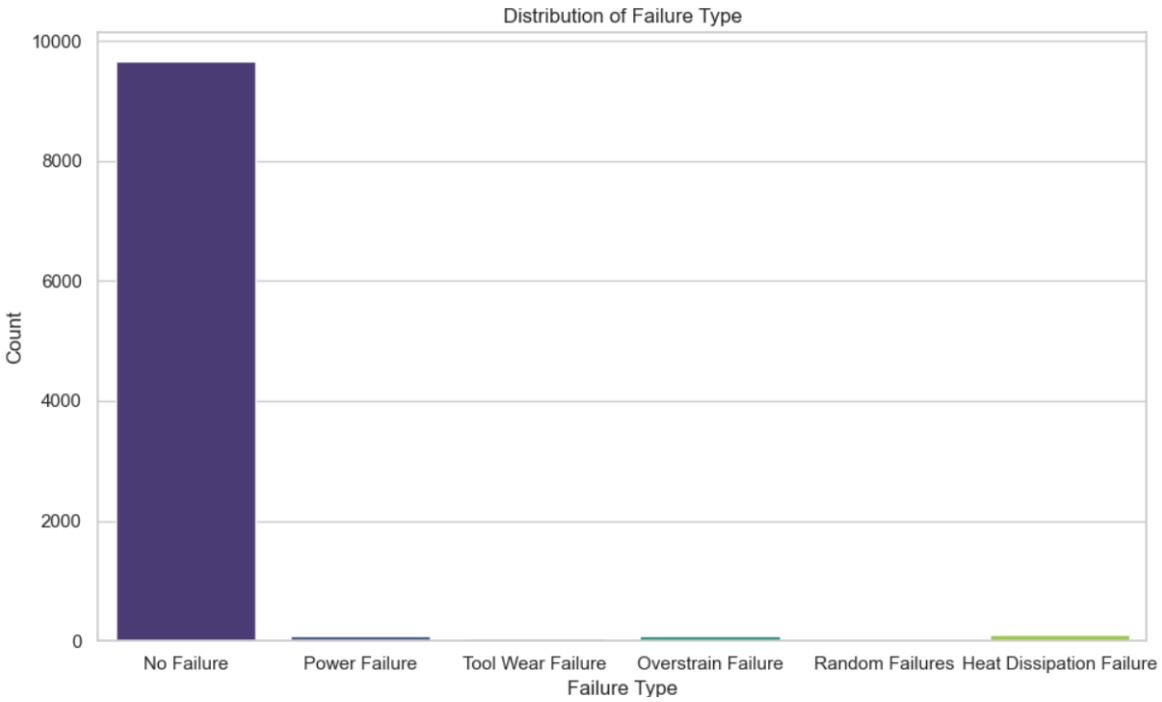
```
**********************************************************************************************************************
DataFrame after dropping 'UDI' and 'Product_ID':
  Type  Air_Temp_K  Process_Temp_K  Rotational_Speed_rpm  Torque_Nm  \
0    M       298.1           308.6                  1551       42.8
1    L       298.2           308.7                  1408       46.3
2    L       298.1           308.5                  1498       49.4
3    L       298.2           308.6                  1433       39.5
4    L       298.2           308.7                  1408       40.0

   Tool_Wear_min  Target Failure_Type
0              0       0   No Failure
1              3       0   No Failure
2              5       0   No Failure
3              7       0   No Failure
4              9       0   No Failure
**********************************************************************************************************************

New Shape of DataFrame: (10000, 8)

Remaining Columns: ['Type', 'Air_Temp_K', 'Process_Temp_K', 'Rotational_Speed_rpm', 'Torque_Nm', 'Tool_Wear_min', 'Target', 'Failure_Type']
```

- **Distribution of features**



- **Proportion of failure type**

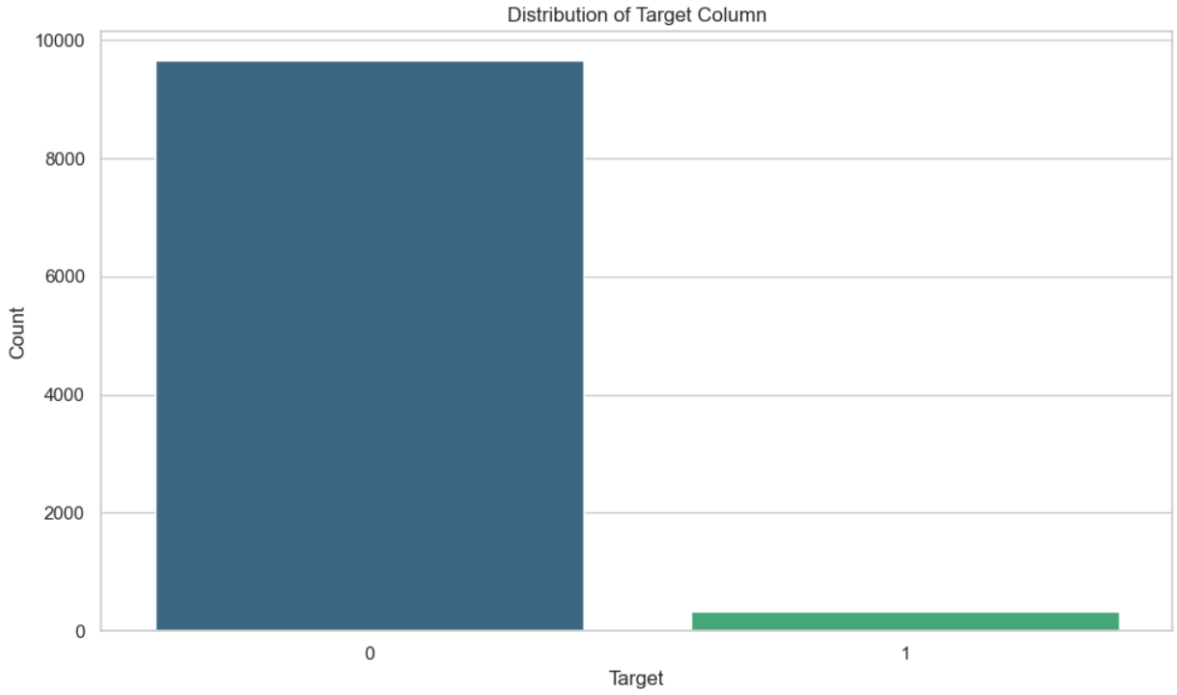




- **Identifying the duplicate rows**

```
Duplicate Rows in the DataFrame:
Empty DataFrame
Columns: [Type, Air_Temp_K, Process_Temp_K, Rotational_Speed_rpm, Torque_Nm, Tool_Wear_min, Target, Failure_Type]
Index: []

Number of Duplicate Rows: 0
```

- **Understanding distribution of target**



- **Encoding and standardizing data**

```
   num__Air_Temp_K  num__Process_Temp_K  num__Rotational_Speed_rpm  \
0        -0.952389            -0.947360                   0.068185
1        -0.902393            -0.879959                  -0.729472
2        -0.952389            -1.014761                  -0.227450
3        -0.902393            -0.947360                  -0.590021
4        -0.902393            -0.879959                  -0.729472

   num__Torque_Nm  num__Tool_Wear_min  cat__Type_H  cat__Type_L  cat__Type_M  \
0        0.282200           -1.695984          0.0          0.0          1.0
1        0.633308           -1.648852          0.0          1.0          0.0
2        0.944290           -1.617430          0.0          1.0          0.0
3       -0.048845           -1.586009          0.0          1.0          0.0
4        0.001313           -1.554588          0.0          1.0          0.0

   cat__Failure_Type_Heat Dissipation Failure  cat__Failure_Type_No Failure  \
0                                         0.0                          1.0
1                                         0.0                          1.0
2                                         0.0                          1.0
3                                         0.0                          1.0
4                                         0.0                          1.0

   cat__Failure_Type_Overstrain Failure  cat__Failure_Type_Power Failure  \
0                                   0.0                             0.0
1                                   0.0                             0.0
2                                   0.0                             0.0
3                                   0.0                             0.0
4                                   0.0                             0.0

   cat__Failure_Type_Random Failures  cat__Failure_Type_Tool Wear Failure
0                                0.0                                 0.0
1                                0.0                                 0.0
2                                0.0                                 0.0
3                                0.0                                 0.0
4                                0.0                                 0.0
```

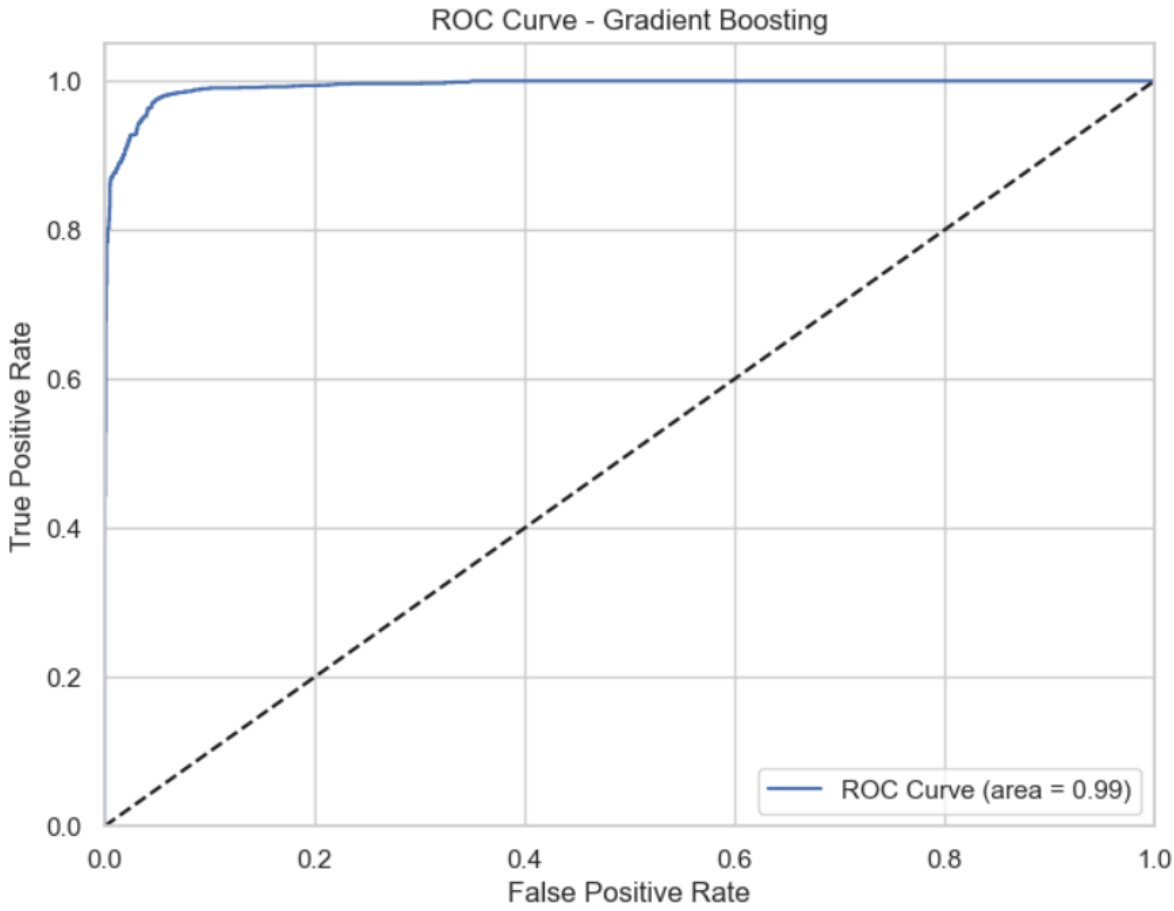
	num__Air_Temp_K	num__Process_Temp_K	num__Rotational_Speed_rpm	num__Torque_Nm	num__Tool_Wear_min	cat__Type_H	cat__Type_L	cat__Type_M	cat__Failure Dissipa
0	-0.952389	-0.947360	0.068185	0.282200	-1.695984	0.0	0.0	1.0	
1	-0.902393	-0.879959	-0.729472	0.633308	-1.648852	0.0	1.0	0.0	
2	-0.952389	-1.014761	-0.227450	0.944290	-1.617430	0.0	1.0	0.0	
3	-0.902393	-0.947360	-0.590021	-0.048845	-1.586009	0.0	1.0	0.0	
4	-0.902393	-0.879959	-0.729472	0.001313	-1.554588	0.0	1.0	0.0	
...	...	...	...	...	...	...	...	...	
9995	-0.602417	-1.082162	0.363820	-1.052012	-1.476034	0.0	0.0	1.0	
9996	-0.552421	-1.082162	0.520005	-0.821283	-1.428902	1.0	0.0	0.0	
9997	-0.502425	-0.947360	0.592519	-0.660777	-1.350349	0.0	0.0	1.0	
9998	-0.502425	-0.879959	-0.729472	0.854005	-1.303217	1.0	0.0	0.0	
9999	-0.502425	-0.879959	-0.216294	0.021376	-1.224663	0.0	0.0	1.0	

10000 rows × 14 columns

- **Model Building**

```
Gradient Boosting Results:
Training Accuracy: 0.9667
Testing Accuracy: 0.9625
Classification Report:
 {'0': {'precision': 0.9740328563857975, 'recall': 0.9503619441571872, 'f1-score': 0.9620518188955771, 'support': 1934.0}, '1': {'precision': 0.951466127
4014156, 'recall': 0.9746245468669084, 'f1-score': 0.9629061140956766, 'support': 1931.0}, 'accuracy': 0.96248382923674, 'macro avg': {'precision': 0.962
7494918936066, 'recall': 0.9624932455120478, 'f1-score': 0.9624789664956268, 'support': 3865.0}, 'weighted avg': {'precision': 0.9627582500031735, 'recal
l': 0.96248382923674, 'f1-score': 0.9624786349450966, 'support': 3865.0}}
Confusion Matrix:
 [[1838   96]
 [  49 1882]]
ROC AUC: 0.9931
```

```
Confusion Matrix:
 [[1838   96]
 [  49 1882]]
ROC AUC: 0.9931
```



Result:

Thus gradient boosting classifier successfully identified is the there any maintenance failure or by 96% and on feature in data.