Assignment 03

Goal of this project is to cluster the customers and classify whether an applicant is considered a Good or a Bad credit risk for 1000 loan applicants

Importing the libraries :

In [1]:
```
import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
import pandas as pd
import numpy as np
import time
```

In [3]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set(style="whitegrid", color_codes=True, palette="dark" )
import plotly.express as px
```

In [4]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold, cross_validate
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

Perform EDA and any data cleaning if necessary.

Loading the csv file to dataframe

In [5]:
```
german_df = pd.read_csv("german_credit_data.csv")
```

Exploratory Data Analysis (EDA)

In [6]:
```
german_df.sample(5)
```

Out[6]:

	Age	Sex	Job	Housing	Saving accounts	Checking account	Credit amount	Duration	Purpose	Risk
970	22	male	2	own	moderate	moderate	1514	15	repairs	good
674	41	male	1	own	quite rich	NaN	2580	21	business	bad
15	32	female	1	own	moderate	little	1282	24	radio/TV	bad
570	23	female	1	rent	little	little	3234	24	furniture/equipment	bad
708	25	female	2	own	little	moderate	1206	9	radio/TV	good

In [7]:
```
german_df.shape
```

Out[7]:
```
(1000, 10)
```

In [8]:
```
german_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 10 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   Age               1000 non-null   int64
 1   Sex               1000 non-null   object
 2   Job               1000 non-null   int64
 3   Housing           1000 non-null   object
 4   Saving accounts   817 non-null    object
 5   Checking account  606 non-null    object
 6   Credit amount     1000 non-null   int64
 7   Duration          1000 non-null   int64
 8   Purpose           1000 non-null   object
 9   Risk              1000 non-null   object
dtypes: int64(4), object(6)
memory usage: 78.3+ KB
```

In [9]:
```
german_df.dtypes
```

Out[9]:
```
Age                  int64
Sex                 object
Job                  int64
Housing             object
Saving accounts     object
Checking account    object
Credit amount        int64
Duration             int64
Purpose             object
Risk                object
dtype: object
```

From above output it clearly states that there are 4 numerical features and 6 are categorical features.

In [10]:
```
german_df.describe(include='all').T
```

Out[10]:

	count	unique	top	freq	mean	std	min	25%	50%	75%	max
Age	1000.0	NaN	NaN	NaN	35.546	11.375469	19.0	27.0	33.0	42.0	75.0
Sex	1000	2	male	690	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Job	1000.0	NaN	NaN	NaN	1.904	0.653614	0.0	2.0	2.0	2.0	3.0
Housing	1000	3	own	713	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Saving accounts	817	4	little	603	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Checking account	606	3	little	274	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Credit amount	1000.0	NaN	NaN	NaN	3271.258	2822.736876	250.0	1365.5	2319.5	3972.25	18424.0
Duration	1000.0	NaN	NaN	NaN	20.903	12.058814	4.0	12.0	18.0	24.0	72.0
Purpose	1000	8	car	337	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Risk	1000	2	good	700	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Cheking and handelling the null values in dataframe :

In [11]:
```
null_counts = german_df.isnull().sum()

if null_counts.any():
    print("Null values present. Details:")
    print(null_counts)
    null_rows = german_df[german_df.isnull().any(axis=1)]
    print("\nRows with null values:")
    display(null_rows)
else:
    print("No null values present.")
```

```
Null values present. Details:
Age                   0
Sex                   0
Job                   0
Housing               0
Saving accounts     183
Checking account    394
Credit amount         0
Duration              0
Purpose               0
Risk                  0
dtype: int64

Rows with null values:
```

	Age	Sex	Job	Housing	Saving accounts	Checking account	Credit amount	Duration	Purpose	Risk
0	67	male	2	own	NaN	little	1169	6	radio/TV	good
2	49	male	1	own	little	NaN	2096	12	education	good
5	35	male	1	free	NaN	NaN	9055	36	education	good
6	53	male	2	own	quite rich	NaN	2835	24	furniture/equipment	good
8	61	male	1	own	rich	NaN	3059	12	radio/TV	good
...	...	...	...	...	...	...	...	...	...	...
991	34	male	1	own	moderate	NaN	1569	15	radio/TV	good
992	23	male	1	rent	NaN	little	1936	18	radio/TV	good
994	50	male	2	own	NaN	NaN	2390	12	car	good
995	31	female	1	own	little	NaN	1736	12	furniture/equipment	good
997	38	male	2	own	little	NaN	804	12	radio/TV	good

478 rows × 10 columns

Two columns has null values :

```
1. Saving accounts     183
2. Checking account    394
- Need to handle this by removing or by imputing the values.
```

Handelling the null values present in dataframe :

In [12]:
```
german_df_new = german_df.copy()
```

Database columns has blank space in the name so replacing the columns names :

In [13]:
```
german_df_new = german_df_new.rename(columns={"Saving accounts": "Saving_accounts", "Checking account": "Checking_account", "Credit amount": "Credit_amount"})
```

In [14]:
```
german_df_new.columns
```

Out[14]:
```
Index(['Age', 'Sex', 'Job', 'Housing', 'Saving_accounts', 'Checking_account',
       'Credit_amount', 'Duration', 'Purpose', 'Risk'],
      dtype='object')
```

Defining the numerical and categorical features:

In [15]:
```
numeric_columns = list(german_df_new.select_dtypes(include=['int']).columns)
categorical_columns = list(german_df_new.select_dtypes(include=['object']).columns)
display(numeric_columns)
display(categorical_columns)
```

```
['Age', 'Job', 'Credit_amount', 'Duration']
['Sex', 'Housing', 'Saving_accounts', 'Checking_account', 'Purpose', 'Risk']
```

In [16]:
```
display(german_df_new[numeric_columns].describe().T)
display(german_df_new[categorical_columns].describe().T)
```

	count	mean	std	min	25%	50%	75%	max
Age	1000.0	35.546	11.375469	19.0	27.0	33.0	42.00	75.0
Job	1000.0	1.904	0.653614	0.0	2.0	2.0	2.00	3.0
Credit_amount	1000.0	3271.258	2822.736876	250.0	1365.5	2319.5	3972.25	18424.0
Duration	1000.0	20.903	12.058814	4.0	12.0	18.0	24.00	72.0

	count	unique	top	freq
Sex	1000	2	male	690
Housing	1000	3	own	713
Saving_accounts	817	4	little	603
Checking_account	606	3	little	274
Purpose	1000	8	car	337
Risk	1000	2	good	700

Frequency of Unique Values in Categorical Columns

In [17]:
```
for column in categorical_columns:
    unique_values = german_df_new[column].value_counts()
    print(f"\n{column}:")
    print(f"\tValue".ljust(10), "Count")
    print(''.ljust(5),'-'*18)
    for value, count in unique_values.items():
        print(f"\t{value.ljust(10)} {count}")
```

```
Sex:
        Value     Count
      ------------------
        male       690
        female     310

Housing:
        Value     Count
      ------------------
        own        713
        rent       179
        free       108

Saving_accounts:
        Value     Count
      ------------------
        little     603
        moderate   103
        quite rich 63
        rich       48

Checking_account:
        Value     Count
      ------------------
        little     274
        moderate   269
        rich       63

Purpose:
        Value     Count
      ------------------
        car        337
        radio/TV   280
        furniture/equipment 181
        business   97
        education  59
        repairs    22
        domestic appliances 12
        vacation/others 12

Risk:
        Value     Count
      ------------------
        good       700
        bad        300
```

1. Imputing the most frequent value (also known as mode imputation) might affect the data's original distribution, particularly if the most frequent value is much more prominent than others. This can result in biased analysis and prediction.
   - From above observation we can clearly see that the distribution of the columns "Saving_account" and "Checking_account" values like "littel" in "Saving_account" is highly prominent that the other 3 values simillary with the "Checking_account", "littel" and "moderate" are more significent than the "rich"
2. Accounting for Uncertainty: In some cases, imputing the most frequent value might give a false sense of certainty about the missing values. By replacing null values with "other" value, we can explicitly acknowledge the uncertainty associated with missing data.
3. So in this case I am going to replace the null values with "other".

Replacing Null Values with New Value ('Other') in Columns

In [18]:
```python
german_df_new['Checking_account'] = german_df_new['Checking_account'].fillna('Other')
german_df_new['Saving_accounts'] = german_df_new['Saving_accounts'].fillna('Other')
```

Columns values after imputing new value ("Other") in the columns in place of null values

In [19]:
```python
print(german_df_new.groupby('Saving_accounts').size(),'\n')
print(german_df_new.groupby('Checking_account').size())
```

```
Saving_accounts
Other         183
little        603
moderate      103
quite rich     63
rich           48
dtype: int64

Checking_account
Other       394
little      274
moderate    269
rich         63
dtype: int64
```

Missing values are replaced with 'other' in both columns.

Null value count after replacing the null values with other

In [20]:
```python
null_counts = german_df_new.isnull().sum()
if null_counts.any():
    print("Null values present. Details:")
    print(null_counts)
else:
    print("No null values present.")
```

```
No null values present.
```

Checking for the dublicate values in the data.

In [21]:
```python
duplicate_rows = german_df_new[german_df.duplicated()]

if not duplicate_rows.empty:
    print("Duplicate rows found. Details:")
    print(duplicate_rows)
else:
    print("No duplicate rows found.")
```

```
No duplicate rows found.
```

Analysing the target variablr distrubution :

In [22]:
```python
risk_groupby = german_df_new.groupby('Risk').size()
print(risk_groupby)
```

```
Risk
bad     300
good    700
dtype: int64
```

In [23]:
```python
custom_colors = px.colors.qualitative.Pastel

fig_pie = px.pie(risk_groupby, values=risk_groupby.values, names=risk_groupby.index,
                 title='Distribution of Risk', labels={'label':'Risk', 'values':'Count'},
                 hole=0.4, color_discrete_sequence=custom_colors,
```

```
                 width=700, height=600)
fig_pie.show()
```

The Pie chart clearly shows that 70% of the accounts are classified as Good Risk and 30% as Bad Risk. This suggests a class imbalance in the dataset, which is likely to influence the performance of models.

1. Class Imbalance: The dataset has a class imbalance, with the "good" category being far more common than the "bad" category. This is crucial to note since it may have an impact on the performance of machine learning algorithms, particularly those are sensitive to class distribution.
2. Model Evaluation: When developing a classification model to categorize loan applicants, it is critical to account for the class imbalance during model evaluation. Accuracy alone may not be an adequate indicator, as a basic model that consistently predicts "good" may obtain high accuracy due to class distribution.

Checking skewness in data:

In [24]:
```
numeric_skewness = german_df_new.select_dtypes(include=[np.number]).skew()
print("Skewness:")
print(numeric_skewness)
```

```
Skewness:
Age              1.020739
Job             -0.374295
Credit_amount    1.949628
Duration         1.094184
dtype: float64
```

The provided skewness values represent the distributional asymmetry of the variables.

```
1. Age has moderate positive skewness (1.02), indicating a slightly right-skewed distribution.
2. Job has a small negative skewness (-0.37), which indicates a slightly left-skewed distribution.
3. Credit_amount has a significant positive skewness (1.95), indicating a strongly right-skewed distribution.
4. Duration has a moderate positive skewness (1.09), implying a slightly right-skewed distribution.
```

Overall, the skewness scores indicate that the distributions of these variables are slightly to moderately skewed, which is generally acceptable. Close-to-zero or within the range of -1 to 1 skewness is commonly regarded optimal, however these values are within the acceptable range, showing fair distribution shapes.

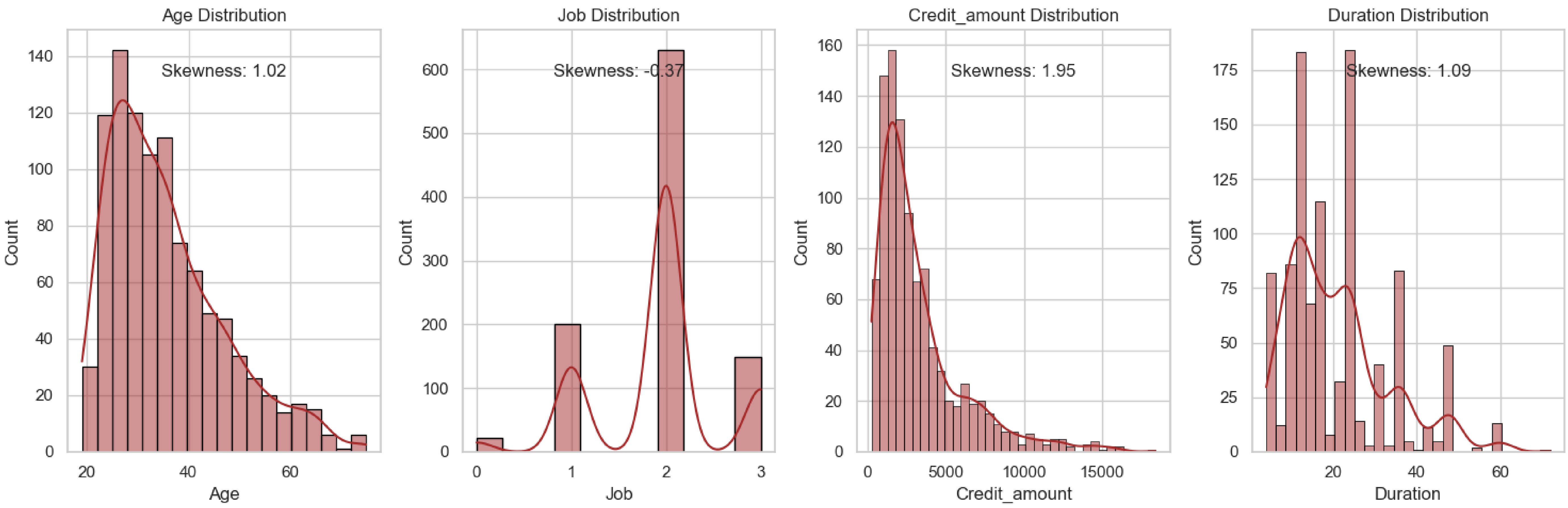
Later will apply the log transformation for skewness greater than 0.5

Visualize the data distribution using histogram below

Distribution for numerical and categorical columns:

features = ['Age', 'Sex', 'Job', 'Housing', 'Saving_accounts', 'Checking_account','Credit_amount', 'Duration', 'Purpose']

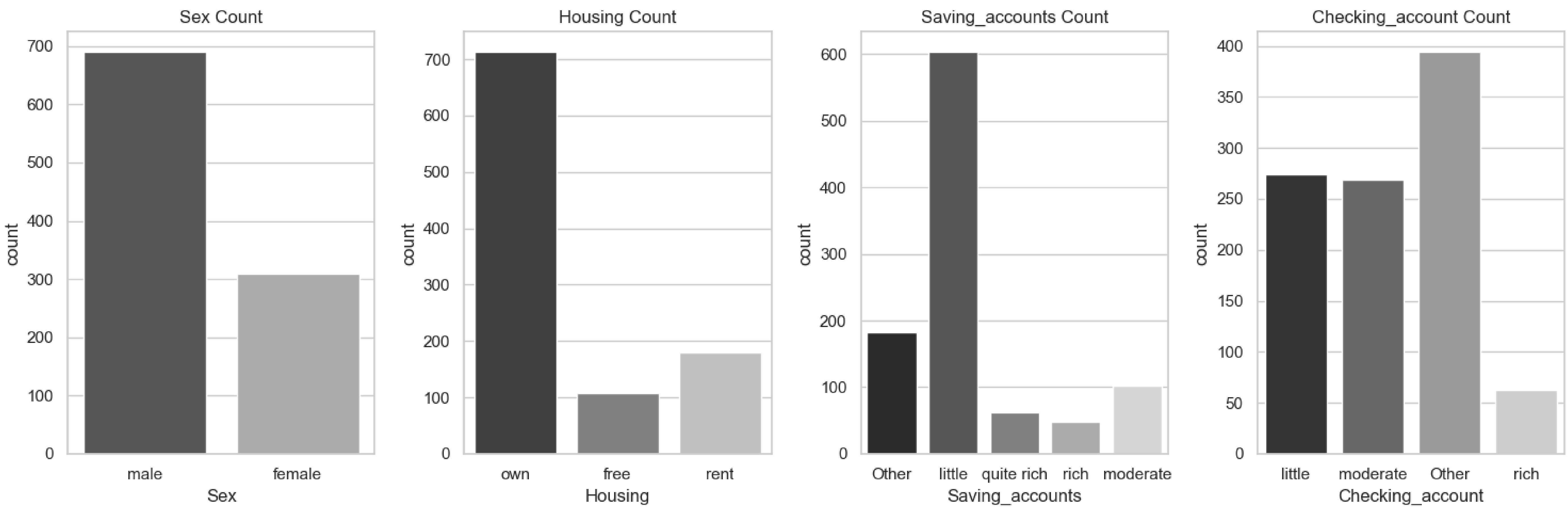
In [25]:
```
fig, axes = plt.subplots(nrows=1, ncols=4, figsize=(15, 5))

for i, col in enumerate(numeric_columns):
    sns.histplot(data=german_df_new, x=col, ax=axes[i], kde=True, color = "brown", edgecolor = 'black')
    skewness_value = numeric_skewness[col]
    axes[i].text(0.5, 0.9, f'Skewness: {skewness_value:.2f}', horizontalalignment='center', verticalalignment='center', transform=axes[i].transAxes, fontsize=12)
    axes[i].set_title(f'{col} Distribution')

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



From this visualization we conclude that :

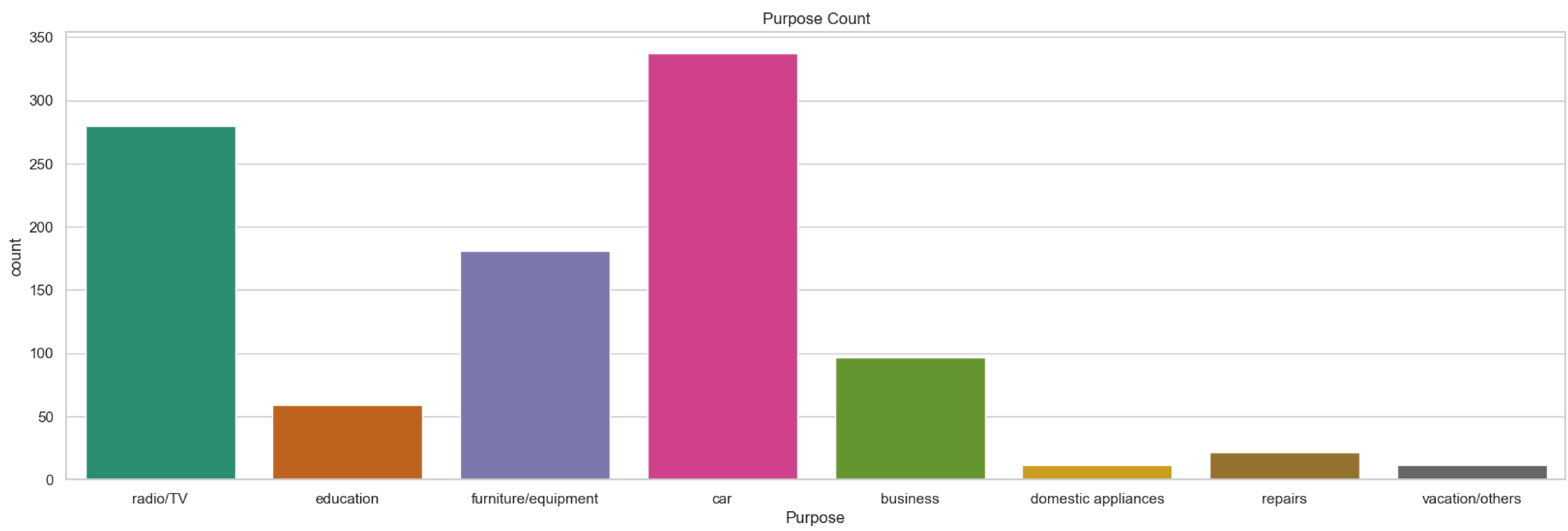
```
1. Since the majority of our customers are between the ages of 25 and 30, the age graph shows a spike in that age range, and as people get older, the graph
becomes less.
2. Label 2 (Skilled) dominates the job graph with an amount more than 600.
3. The majority of consumers on the credit amount graph have credit amounts between 0 and $3,000. The larger the credit amount, the smaller the count.
4. According to the Duration Graph, most customers take out loans for a period of two to twenty-five months; for longer periods, there is a significant
difference and a decline in the number of loans.
```

In [26]:
```
hist_graph = ['Sex', 'Housing', 'Saving_accounts', 'Checking_account']

fig, axes = plt.subplots(1, 4, figsize=(15, 5))

axes = axes.flatten()

for i, cat_col in enumerate(hist_graph):
    sns.countplot(data=german_df_new, x=cat_col, ax=axes[i], palette = 'gray')
    axes[i].set_title(f'{cat_col} Count')

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



1. In terms of sex/gender attributes, male credit bank customers outnumber female customers (7:3).
2. The majority of credit bank customers possess a residence, with a ratio of 7:1:2 (owned:free:rent), in the housing attribute.
3. The majority of credit bank customers have a small amount saved in their savings accounts.
4. Customers whose status is unclear make up the majority of credit bank customers when it comes to checking accounts and the amount of current accounts.

In [27]:
```
plt.figure(figsize=(20, 6))
sns.countplot(x='Purpose', data=german_df_new, palette = 'Dark2')
plt.title("Purpose Count")
plt.show()
```



Regarding Purpose attribute: The majority of bank credit clients have a credit purpose, which is to purchase a car. The three highest goals below are business, furniture/equipment, and radio/TV.

Perform one hot encoding for categorical variables :

In [28]:
```
display(categorical_columns)
```

```
['Sex', 'Housing', 'Saving_accounts', 'Checking_account', 'Purpose', 'Risk']
```

In [29]:
```
encoded_df = pd.get_dummies(german_df_new, columns=['Sex', 'Housing', 'Saving_accounts', 'Checking_account', 'Purpose'])
pd.set_option('display.max_columns', None)
encoded_df.sample(7)
```

Out[29]:

	Age	Job	Credit_amount	Duration	Risk	Sex_female	Sex_male	Housing_free	Housing_own	Housing_rent	Saving_accounts_Other	Saving_accounts_little	Saving_accounts_moderate	Saving_accounts_quite rich
867	42	2	3331	12	good	False	True	False	True	False	False	True	False	False
153	29	2	7758	24	good	True	False	False	False	True	False	False	False	False
281	50	2	1574	12	good	False	True	False	True	False	False	True	False	False
253	35	2	4151	24	good	False	True	False	True	False	False	False	True	False
117	27	2	2132	10	good	True	False	False	False	True	True	False	False	False
826	33	2	3966	18	bad	True	False	False	False	True	False	True	False	False
476	24	2	2569	39	good	False	True	False	True	False	False	False	False	True

In [30]:
```
print("Original dataframe shape:", german_df_new.shape)
print("Encoded dataframe shape:", encoded_df.shape)
```

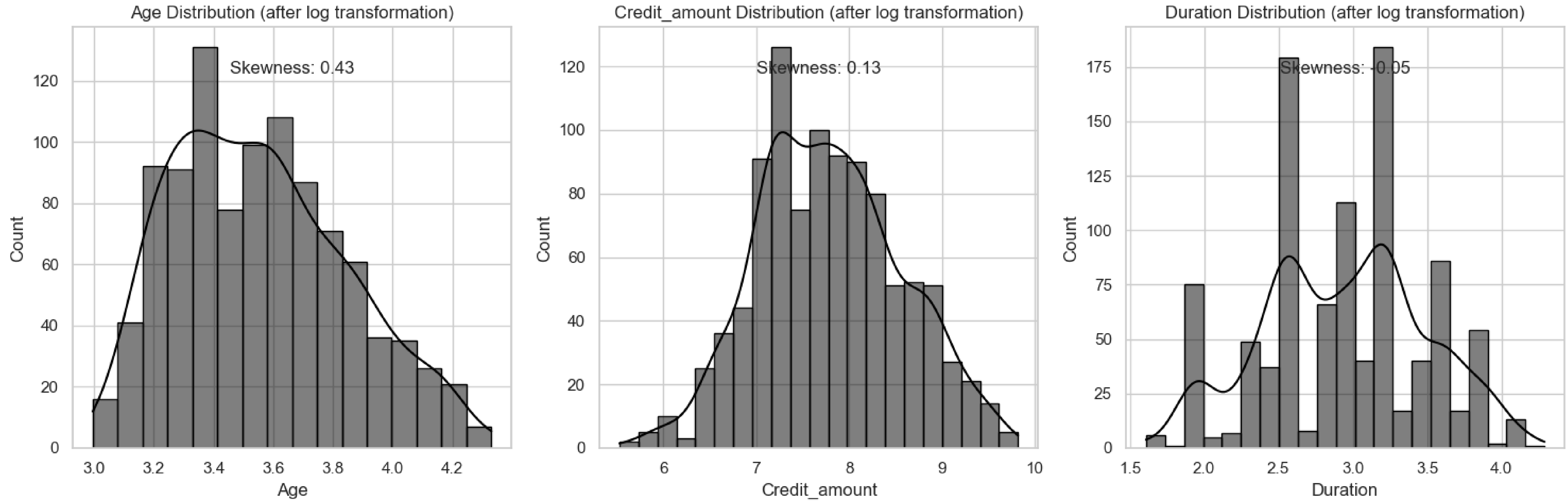
```
Original dataframe shape: (1000, 10)
Encoded dataframe shape: (1000, 27)
```

Visualize the histograms of numerical features. Do you observe skewness in the data? If yes apply the log transformation. Check the histograms again to see if data has been normalized.

Applying the Log transformation :

In [31]:
```
columns_to_transform = ['Age', 'Credit_amount', 'Duration']
german_df_new[columns_to_transform] = german_df_new[columns_to_transform].apply(lambda x: np.log1p(x))

skewness_after_log = german_df_new[numeric_columns].skew()

print("Skewness after log transformation:")
print(skewness_after_log)
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(15, 5))

for i, col in enumerate(columns_to_transform):
    sns.histplot(data=german_df_new, x=col, ax=axes[i], kde=True, edgecolor = 'Black', color = 'black')
    skewness_value = skewness_after_log[col]
    axes[i].text(0.5, 0.9, f'Skewness: {skewness_value:.2f}', horizontalalignment='center', verticalalignment='center', transform=axes[i].transAxes, fontsize=12)
    axes[i].set_title(f'{col} Distribution (after log transformation)')

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

```
Skewness after log transformation:
Age              0.431653
Job             -0.374295
Credit_amount    0.130306
Duration        -0.051217
dtype: float64
```



After applying the log transformation to the required columns, we see the following skewness values and the distribution of data:

1. Age:
   - Following the log transformation, the skewness fell from 1.02 to 0.22. This shows that the age distribution has grown less positively biased and is closer to being symmetric.
2. Credit_amount:
   - The skewness has been reduced from 1.95 to -0.24 following the log transformation. This suggests a large reduction in the positive skewness of credit amounts, bringing the distribution closer to symmetry.
3. Duration:
   - Following the log transformation, the skewness fell from 1.09 to -0.66. Similar to the credit amount, this demonstrates a significant drop in positive skewness, showing a more symmetric distribution of loan terms.

Overall, the log transformation substantially reduced the skewness of the "Age", "Credit_amount", and "Duration" distributions, bringing them closer to a symmetric shape. However, "Job" remains unaffected because it was near to a symmetric distribution prior to transformation.

Apply Feature Scaling

In [32]:
```
scaler = StandardScaler()
encoded_df[numeric_columns] = scaler.fit_transform(encoded_df[numeric_columns])
scaled_df = encoded_df
scaled_df.sample(10)
```

Out[32]:

	Age	Job	Credit_amount	Duration	Risk	Sex_female	Sex_male	Housing_free	Housing_own	Housing_rent	Saving_accounts_Other	Saving_accounts_little	Saving_accounts_moderate	Saving_ac
698	-1.015499	0.146949	-0.521478	-0.240857	good	False	True	False	True	False	False	True	False	
548	-1.015499	-1.383771	-0.937594	-0.738668	bad	True	False	False	True	False	False	True	False	
534	-0.927547	0.146949	-0.058929	0.256953	good	False	True	False	True	False	True	False	False	
297	0.831502	-1.383771	-0.703307	-0.904604	good	False	True	False	True	False	True	False	False	
40	-0.487784	1.677670	-0.332559	0.754763	good	False	True	False	True	False	False	False	False	
948	0.655598	-1.383771	-0.616114	-0.240857	bad	False	True	False	True	False	False	True	False	
104	-0.839594	0.146949	-0.292862	-0.738668	good	False	True	False	False	True	True	False	False	
974	-0.223927	0.146949	-0.156047	0.754763	good	True	False	False	True	False	False	True	False	
950	0.391740	-2.914492	0.112976	-0.240857	good	False	True	False	True	False	False	True	False	
901	0.743550	0.146949	0.075759	-0.074920	good	False	True	False	True	False	True	False	False	

Why Feature Scaling ?

- Feature scaling is necessary since it ensures that all features are the same scale. Many machine learning techniques, such KMeans clustering, are sensitive to feature size. If features are not on the same scale, the algorithm may assign more weight to those with bigger scales, producing biased results. By scaling the features, we ensure that each feature contributes equally to the distance computations done by algorithms such as KMean.

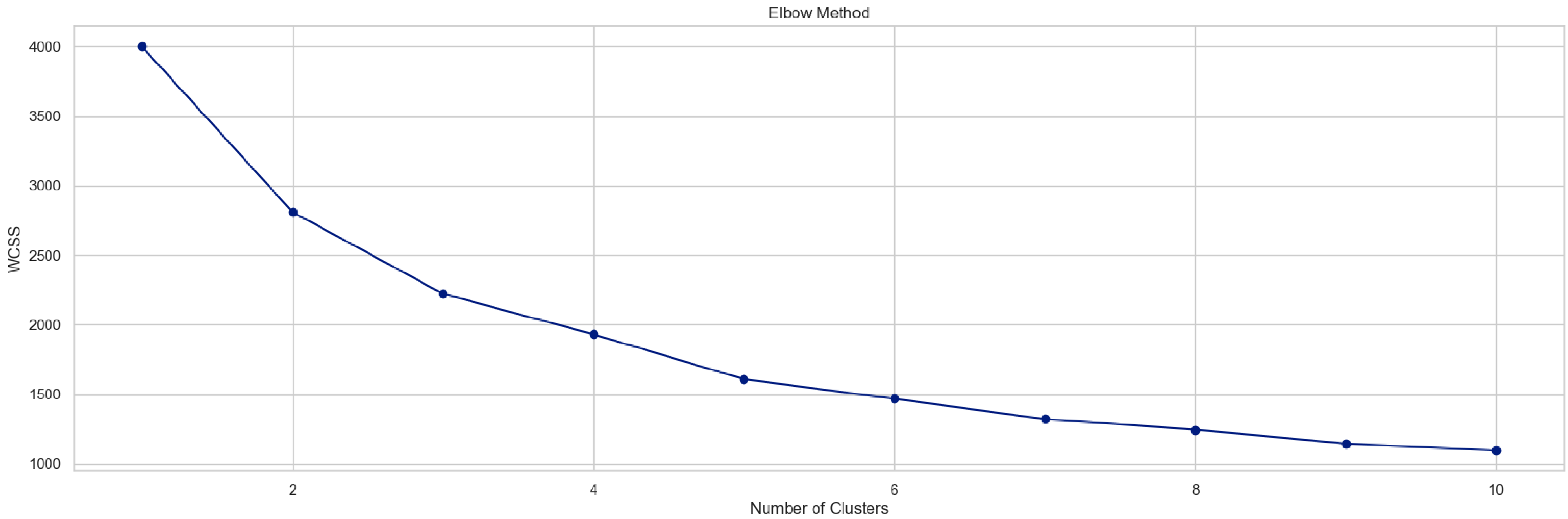
Choose only the numerical features for clustering

In [33]:
```
numerical_features = numeric_columns
display(numerical_features)
```

```
['Age', 'Job', 'Credit_amount', 'Duration']
```

Apply elbow method to find best number of clusters. Plot the graph.

In [34]:
```
X = scaled_df[numerical_features]

wcss = []

for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', random_state=14)
    kmeans.fit(X)

    wcss.append(kmeans.inertia_)

plt.figure(figsize=(20, 6))
plt.plot(range(1, 11), wcss, marker='o', linestyle='-', color='b')
plt.title('Elbow Method')
plt.xlabel('Number of Clusters')
plt.ylabel('WCSS')
plt.show()


best_num_clusters = 0
for i in range(1, len(wcss) - 1):
    slope = (wcss[i] - wcss[i + 1]) / (wcss[i - 1] - wcss[i])
    if slope < 0.9:
        best_num_clusters = i + 1
        break

print("Best number of clusters:", best_num_clusters)
```




```
Best number of clusters: 2
```

The elbow method show the optimal number of clusters for the supplied data is two. This suggests that the data can be naturally divided into two separate clusters, implying a binary or dichotomous structure in the dataset.

- In this step, we use KMeans clustering to group comparable data points. Determining the correct number of clusters (k) is critical to the clustering algorithm's success. The elbow technique determines the right number of clusters by visualizing the within-cluster sum of squares (SSE) for various k values. The "elbow" point in the graph illustrates the best value of k for which adding more clusters does not significantly diminish the SSE. This guarantees that we strike a balance between overfitting (too many clusters) and underfitting the data.

Choose optimum number of clusters and visualize it using PCA

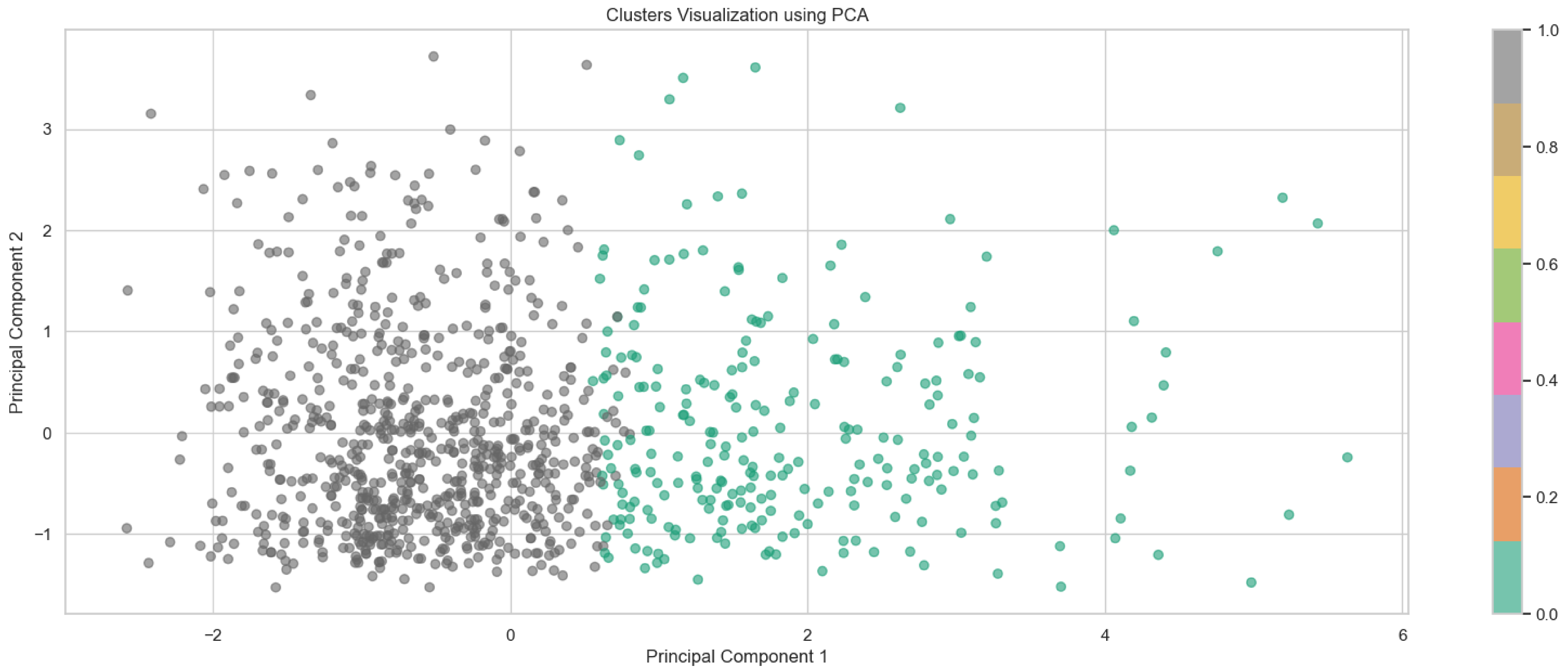
In [35]:
```
optimal_clusters = 2

kmeans = KMeans(n_clusters=optimal_clusters, random_state=165)
kmeans.fit(X)

pca = PCA(n_components=2)
pca_result = pca.fit_transform(X)

print("Explained variance ratio (PCA) components:")
print("PCA Component 1:", pca.explained_variance_ratio_[0])
print("PCA Component 2:", pca.explained_variance_ratio_[1])

df_pca = pd.DataFrame(pca_result, columns=['PC1', 'PC2'])
df_pca['Cluster'] = kmeans.labels_

display(df_pca)

plt.figure(figsize=(20, 7))
```

```
plt.scatter(df_pca['PC1'], df_pca['PC2'], c=df_pca['Cluster'], cmap='Dark2', alpha=0.6, label='Cluster')
plt.title('Clusters Visualization using PCA')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.colorbar()
plt.show()
```

```
Explained variance ratio (PCA) components:
PCA Component 1: 0.445688478119271
PCA Component 2: 0.2514922919172123
```

	PC1	PC2	Cluster
0	-1.194542	2.859247	1
1	2.100553	-1.366294	0
2	-1.301225	1.091077	1
3	2.246763	0.698941	0
4	0.604513	1.520616	0
...	...	...	...
995	-1.394977	-0.477952	1
996	1.304913	0.491853	0
997	-0.980085	0.283619	1
998	0.988894	-1.285326	0
999	1.625963	-0.916813	0

1000 rows × 3 columns



Principal Component Analysis (PCA) is a dimensionality reduction approach for visualizing high-dimensional data in smaller dimensions while keeping variance. In this stage, we utilize PCA to decrease the data's dimensionality to two dimensions for visualization. Visualizing the clusters in a lower-dimensional space allows us to better comprehend the data's underlying structure and evaluate the clustering algorithm's performance. This is an important step in analyzing and validating clustering results.\

```
- It seems to be a acceptable separation between these two components.
```

Implement KFOLD CV and use any classifier of your choosing and report the evaluation metrics

Mapping the "Risk" column ('good':0,'bad':1)

In [36]:
```
name_mapping = { 'good':0,'bad':1 }
scaled_df["Risk"] = scaled_df['Risk'].map(name_mapping)
```

In [37]:
```
X_2 = scaled_df.drop(['Risk'], axis=1)
Y = scaled_df['Risk']
```

In [38]:
```
display(X_2, Y)
```

	Age	Job	Credit_amount	Duration	Sex_female	Sex_male	Housing_free	Housing_own	Housing_rent	Saving_accounts_Other	Saving_accounts_little	Saving_accounts_moderate	Saving_accounts_
0	2.766456	0.146949	-0.745131	-1.236478	False	True	False	True	False	True	False	False	
1	-1.191404	0.146949	0.949817	2.248194	True	False	False	True	False	False	True	False	
2	1.183312	-1.383771	-0.416562	-0.738668	False	True	False	True	False	False	True	False	
3	0.831502	0.146949	1.634247	1.750384	False	True	True	False	False	False	True	False	
4	1.535122	0.146949	0.566664	0.256953	False	True	True	False	False	False	True	False	
...	...	...	...	...	...	...	...	...	...	...	...	...	
995	-0.399832	-1.383771	-0.544162	-0.738668	True	False	False	True	False	False	True	False	
996	0.391740	1.677670	0.207612	0.754763	False	True	False	True	False	False	True	False	
997	0.215835	0.146949	-0.874503	-0.738668	False	True	False	True	False	False	True	False	
998	-1.103451	0.146949	-0.505528	1.999289	False	True	True	False	False	False	True	False	
999	-0.751642	0.146949	0.462457	1.999289	False	True	False	True	False	False	False	True	

1000 rows × 26 columns

```
0      0
1      1
2      0
3      0
4      1
      ..
995    0
996    0
997    0
998    1
999    0
Name: Risk, Length: 1000, dtype: int64
```

Splitting the dataset into training and testing :

In [39]:
```
X_train, X_test, Y_train, Y_test = train_test_split(X_2, Y, test_size=0.2, stratify = Y, random_state=78)

print('Train Test split ratio is : [80, 20]','\n')
print("Training set:")
print("X_train shape:", X_train.shape)
print("y_train shape:", Y_train.shape)


print("\nTesting set:")
print("X_test shape:", X_test.shape)
print("y_test shape:", Y_test.shape)
```

```
Train Test split ratio is : [80, 20]

Training set:
X_train shape: (800, 26)
y_train shape: (800,)

Testing set:
X_test shape: (200, 26)
y_test shape: (200,)
```

In [40]:
```
X_train
```

Out[40]:

	Age	Job	Credit_amount	Duration	Sex_female	Sex_male	Housing_free	Housing_own	Housing_rent	Saving_accounts_Other	Saving_accounts_little	Saving_accounts_moderate	Saving_accounts_
523	-1.103451	0.146949	0.076823	0.256953	True	False	False	True	False	False	False	True	
434	-0.927547	0.146949	-0.402385	-0.987573	False	True	False	True	False	False	True	False	
522	1.535122	0.146949	1.363807	2.248194	False	True	True	False	False	False	True	False	
721	-1.015499	0.146949	-1.006002	-1.236478	True	False	False	False	True	False	False	False	
830	0.743550	0.146949	-0.317673	0.256953	False	True	False	True	False	False	False	False	
...	...	...	...	...	...	...	...	...	...	...	...	...	
761	-1.015499	0.146949	-0.406638	-0.240857	True	False	False	False	True	False	True	False	
925	0.919455	0.146949	-0.912429	-0.738668	False	True	False	True	False	False	True	False	
620	-0.751642	0.146949	0.134951	0.008048	False	True	False	True	False	False	True	False	
327	-0.135974	0.146949	-0.618950	0.256953	True	False	False	True	False	False	False	False	
284	0.127883	0.146949	0.215056	0.256953	False	True	False	True	False	False	False	True	

800 rows × 26 columns

Implimenting the KFOLD Cross-Validation

Fit the train dataset to all models and output the score for the test dataset. Choose the model with the highest score for further processing.

In [41]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier, AdaBoostClassifier, ExtraTreesClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis, LinearDiscriminantAnalysis
from sklearn.gaussian_process import GaussianProcessClassifier


classifier_models = {
    "Logistic Regression": LogisticRegression(),
    "K-Nearest Neighbors": KNeighborsClassifier(),
    "Support Vector Machine": SVC(),
    "Decision Tree": DecisionTreeClassifier(),
    "Random Forest": RandomForestClassifier(),
    "Gradient Boosting": GradientBoostingClassifier(),
    "AdaBoost": AdaBoostClassifier(),
    "Extra Trees": ExtraTreesClassifier(),
    "Gaussian Naive Bayes": GaussianNB(),
    "Multi-layer Perceptron": MLPClassifier(),
    "Quadratic Discriminant Analysis": QuadraticDiscriminantAnalysis(),
    "Linear Discriminant Analysis": LinearDiscriminantAnalysis(),
    "Gaussian Process Classifier": GaussianProcessClassifier()
}
```

In [42]:
```
classifier_accuracy = []
for model_name, model in classifier_models.items():
    accuracy = model.fit(X_train, Y_train).score(X_test, Y_test) * 100
    print(f"{model_name}: {accuracy}")
    classifier_accuracy.append((model_name, accuracy))
```

```
Logistic Regression: 72.0
K-Nearest Neighbors: 66.5
Support Vector Machine: 73.5
Decision Tree: 69.5
Random Forest: 72.5
Gradient Boosting: 75.0
AdaBoost: 75.0
Extra Trees: 71.0
Gaussian Naive Bayes: 70.0
Multi-layer Perceptron: 74.0
Quadratic Discriminant Analysis: 55.50000000000001
Linear Discriminant Analysis: 72.0
Gaussian Process Classifier: 71.5
```

In [43]:
```
#classifier_accuracy
```

In [44]:
```
sorted_accuracy = sorted(classifier_accuracy, key=lambda x: x[1], reverse=True)

for model_name, accuracy in sorted_accuracy:
    print(f"{model_name}: {accuracy}")
```

```
Gradient Boosting: 75.0
AdaBoost: 75.0
Multi-layer Perceptron: 74.0
Support Vector Machine: 73.5
Random Forest: 72.5
Logistic Regression: 72.0
Linear Discriminant Analysis: 72.0
Gaussian Process Classifier: 71.5
Extra Trees: 71.0
Gaussian Naive Bayes: 70.0
Decision Tree: 69.5
K-Nearest Neighbors: 66.5
Quadratic Discriminant Analysis: 55.50000000000001
```

Here after analysing the scores of all classifier model best score is obtained from (Random Forest: 75.5). So moving forward with the Random Forest.

Defined KFold and model:

In [45]:
```
kf = KFold(n_splits=5, shuffle=True, random_state=42)
model = RandomForestClassifier()
```

Def function for evaluating the model using Kfold :

In [150…
```
def evaluate_model_with_kfold(model, X, Y, kf):
    print("Model Evaluation using KFold", '\n', "." * 110)
    print("Algorithm".ljust(30), "Mean Cross validation Score".ljust(35), "Standard Deviation Cross validation Score", '\n')

    cv_scores = []
    for train_index, test_index in kf.split(X, Y):
        x_train, x_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = Y.iloc[train_index], Y.iloc[test_index]
        model.fit(x_train, y_train)
        val_score = model.score(x_test, y_test)
        cv_scores.append(val_score)

    mean_cv_score = np.mean(cv_scores) * 100
    std_cv_score = np.std(cv_scores)

    print(f"{model.__class__.__name__}".ljust(30), str(mean_cv_score).ljust(35), str(std_cv_score))

    return model, cv_scores
```

In [151…
```
trained_model, cv_scores, = evaluate_model_with_kfold(model, X_2, Y, kf)
print('\n', "Cross validation score of the different folds", cv_scores)
```

```
Model Evaluation using KFold
 ..............................................................................................................
Algorithm                      Mean Cross validation Score         Standard Deviation Cross validation Score

RandomForestClassifier         74.00000000000001                   0.033166247903553984

 Cross validation score of the different folds [0.76, 0.755, 0.675, 0.765, 0.745]
```

The output represent the Mean cross validation and STD score for the "Random Forest Classifier", using kfold CV.

```
- The Mean CV score is 74.00 percent, with STD of 0.04
- Cross-validation scores indicate how the model performed on different data segments.
- The results range from 0.67 to 0.76, with an average cross-validation score of about 74.00 . This fluctuation demonstrates that the model's performance varies between subsets of the data.
- The less Mean CV score might be because of class imbalance.
```

So to balance out the dataset we can apply the smote technique after spilitting the dataset into training and testing sets :

```
- Bcecause It's important to apply SMOTE only to the training data to avoid data leakage.
```

Evaluating the model on testing dataset

In [156…
```
y_pred = trained_model.predict(X_test)

print("Accuracy Score:", accuracy_score(Y_test, y_pred))
print("\nClassification Report:")
print(classification_report(Y_test, y_pred))
print("Confusion Matrix:")
print(confusion_matrix(Y_test, y_pred))
```

```
Accuracy Score: 0.93

Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.97      0.95       140
           1       0.93      0.83      0.88        60

    accuracy                           0.93       200
   macro avg       0.93      0.90      0.91       200
weighted avg       0.93      0.93      0.93       200

Confusion Matrix:
[[136   4]
 [ 10  50]]
```

Accuracy Score and Classification Report:

```
- The accuracy score of the classification model is 0.93, which indicates a high level of overall correctness.
- The Classification Report provides detailed metrics for two classes (labeled as “0” and “1”):
- Here the lables ("0" is Good and "1" is bad)
```

1. For class 0:
   - Precision: 0.93
   - Recall: 0.97
   - F1-score: 0.95
   - Support: 140 instances
2. For class 1:
   - Precision: 0.93
   - Recall: 0.83
   - F1-score: 0.88
   - Support: 60 instances

Confusion Matrix:

```
- The confusion matrix shows the the model had identified for,
```

1. class 0 (good) :
   - The model has identified 136 values correctly out of 140.
2. class 1 (bad) :
   - Similarly, for class one 50 was correctly identified and only 10 was wrong guessed.

Interpretation:

```
- True Positives (TP) for class 0: 136
- False Positives (FP) for class 0: 4
- True Positives (TP) for class 1: 50
- False Positives (FP) for class 1: 10
```

The model appears to do better in correctly recognizing instances of class 0 (greater recall and f1-score) than class 1. To summarize, our classification model performs well, particularly for class 0, but there is potential for improvement in class 1 predictions.

Area under the Receiver Operating Characteristic Curve :

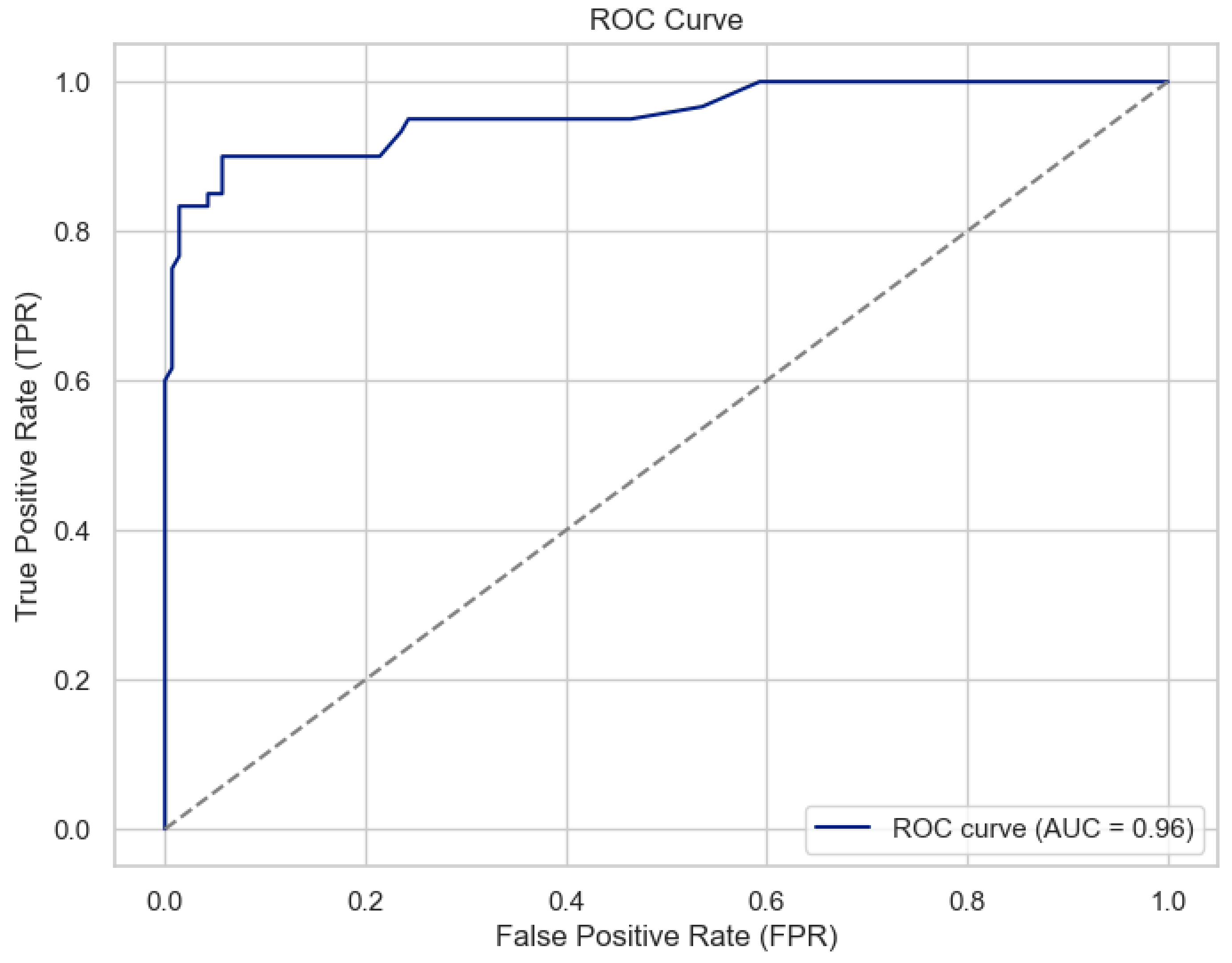
In [157…
```
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve, auc
```

```
y_probabilities = model.predict_proba(X_test)[:, 1]

fpr, tpr, thresholds = roc_curve(Y_test, y_probabilities)

auc_roc = auc(fpr, tpr)
print("AUC-ROC Score:", auc_roc)
```

```
AUC-ROC Score: 0.9556547619047618
```

In [158…

```
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='b', label='ROC curve (AUC = {:.2f})'.format(auc_roc))
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('ROC Curve')
plt.legend(loc='lower right')
plt.show()
```



AUC-ROC is a performance metric used in machine learning to evaluate binary classification models.

- This metric helps in evaluating the ability of a model to distinguish between positive and negative.
- The AUC (Area Under the Curve) of the ROC (Receiver Operating Characteristic) curve display in above graph is approximately 0.955.
- This high AUC value indicates that the model's performance is excellent, as it is close to 1.
- The ROC curve visually represents the trade-off between the true positive rate (TPR) and the false positive rate (FPR) for different classification thresholds.
- The model demonstrates strong discriminatory power in distinguishing between positive and negative instances.