Importing libraries

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

In [149…
```
df = pd.read_csv('loan_data.csv')
```

EDA

In [150…
```
df
```

Out[150…

	person_age	person_gender	person_education	person_income	person_emp_exp	pe
0	22.0	female	Master	71948.0	0	
1	21.0	female	High School	12282.0	0	
2	25.0	female	High School	12438.0	3	
3	23.0	female	Bachelor	79753.0	0	
4	24.0	male	Master	66135.0	1	
...	...	...	...	...	...	
44995	27.0	male	Associate	47971.0	6	
44996	37.0	female	Associate	65800.0	17	
44997	33.0	male	Associate	56942.0	7	
44998	29.0	male	Bachelor	33164.0	4	
44999	24.0	male	High School	51609.0	1	

45000 rows × 14 columns

In [151…
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45000 entries, 0 to 44999
Data columns (total 14 columns):
 #   Column                          Non-Null Count  Dtype  
---  ------                          --------------  -----  
 0   person_age                      45000 non-null  float64
 1   person_gender                   45000 non-null  object 
 2   person_education                45000 non-null  object 
 3   person_income                   45000 non-null  float64
 4   person_emp_exp                  45000 non-null  int64  
 5   person_home_ownership           45000 non-null  object 
 6   loan_amnt                       45000 non-null  float64
 7   loan_intent                     45000 non-null  object 
 8   loan_int_rate                   45000 non-null  float64
 9   loan_percent_income             45000 non-null  float64
 10  cb_person_cred_hist_length      45000 non-null  float64
 11  credit_score                    45000 non-null  int64  
 12  previous_loan_defaults_on_file  45000 non-null  object 
 13  loan_status                     45000 non-null  int64  
dtypes: float64(6), int64(3), object(5)
memory usage: 4.8+ MB
```

In [152…

```python
df.describe()
```

Out[152…

	person_age	person_income	person_emp_exp	loan_amnt	loan_int_rate	loan_pe
count	45000.000000	4.500000e+04	45000.000000	45000.000000	45000.000000	
mean	27.764178	8.031905e+04	5.410333	9583.157556	11.006606	
std	6.045108	8.042250e+04	6.063532	6314.886691	2.978808	
min	20.000000	8.000000e+03	0.000000	500.000000	5.420000	
25%	24.000000	4.720400e+04	1.000000	5000.000000	8.590000	
50%	26.000000	6.704800e+04	4.000000	8000.000000	11.010000	
75%	30.000000	9.578925e+04	8.000000	12237.250000	12.990000	
max	144.000000	7.200766e+06	125.000000	35000.000000	20.000000	

Check the people whose age greater than 100

In [153…

```python
df.loc[df["person_age"] > 100]
```

Out[153…

	person_age	person_gender	person_education	person_income	person_emp_exp	pe
81	144.0	male	Bachelor	300616.0	125	
183	144.0	male	Associate	241424.0	121	
575	123.0	female	High School	97140.0	101	
747	123.0	male	Bachelor	94723.0	100	
32297	144.0	female	Associate	7200766.0	124	
37930	116.0	male	Bachelor	5545545.0	93	
38113	109.0	male	High School	5556399.0	85	

The maximum age value of 144 is indeed outlier, as it isn't reasonable. To handle this, we removed Outliers, Remove ages above a certain threshold, like who are greater than 100 years, to make it more reasonable.

In [154…

```
df = df.loc[~(df["person_age"] > 100)]
```

Check the null values & dublicates

In [155…

```
df.isna().sum()
```

Out[155…

```
person_age                        0
person_gender                     0
person_education                  0
person_income                     0
person_emp_exp                    0
person_home_ownership             0
loan_amnt                         0
loan_intent                       0
loan_int_rate                     0
loan_percent_income               0
cb_person_cred_hist_length        0
credit_score                      0
previous_loan_defaults_on_file    0
loan_status                       0
dtype: int64
```

In [156…

```
df.duplicated().sum()
```

Out[156…

```
0
```

Count unique values of categorical features

In [157…

```
for category in df.columns:
    if df[category].dtype == 'object':
```

```
        print(f"Value counts for column: {category}")
        print(df[category].value_counts())
        print("\n")
```

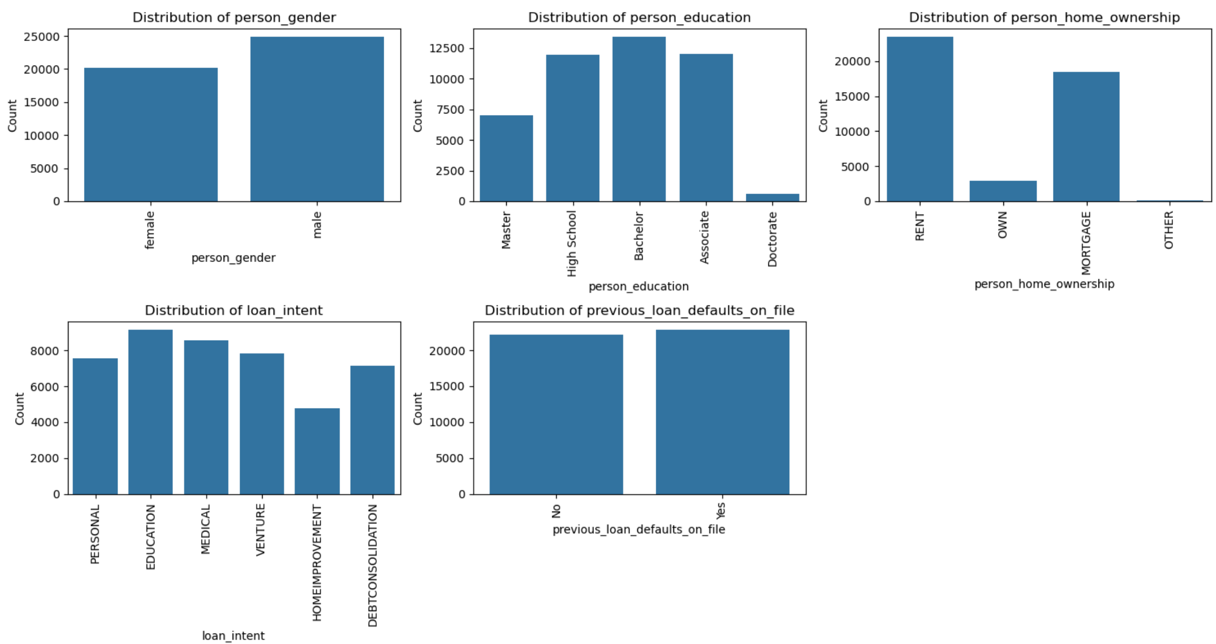
```
Value counts for column: person_gender
person_gender
male      24836
female    20157
Name: count, dtype: int64


Value counts for column: person_education
person_education
Bachelor       13396
Associate      12026
High School    11970
Master          6980
Doctorate        621
Name: count, dtype: int64


Value counts for column: person_home_ownership
person_home_ownership
RENT        23440
MORTGAGE    18485
OWN          2951
OTHER         117
Name: count, dtype: int64


Value counts for column: loan_intent
loan_intent
EDUCATION            9151
MEDICAL              8548
VENTURE              7815
PERSONAL             7551
DEBTCONSOLIDATION    7145
HOMEIMPROVEMENT      4783
Name: count, dtype: int64


Value counts for column: previous_loan_defaults_on_file
previous_loan_defaults_on_file
Yes    22856
No     22137
Name: count, dtype: int64
```

Visualization

Visualise categorical columns distributions

In [158…

```
fig, axes = plt.subplots(2, 3, figsize=(15, 8))
axes = axes.flatten()
i = 0
for category in df.columns:
    if df[category].dtype == 'object':
        sns.countplot(x=df[category], ax=axes[i])
        axes[i].set_title(f"Distribution of {category}", fontsize=12)
        axes[i].set_xlabel(category)
        axes[i].set_ylabel("Count")
        axes[i].tick_params(axis='x', rotation=90)
        i += 1

# Hide any remaining unused axes
for j in range(i, len(axes)):
    fig.delaxes(axes[j])

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

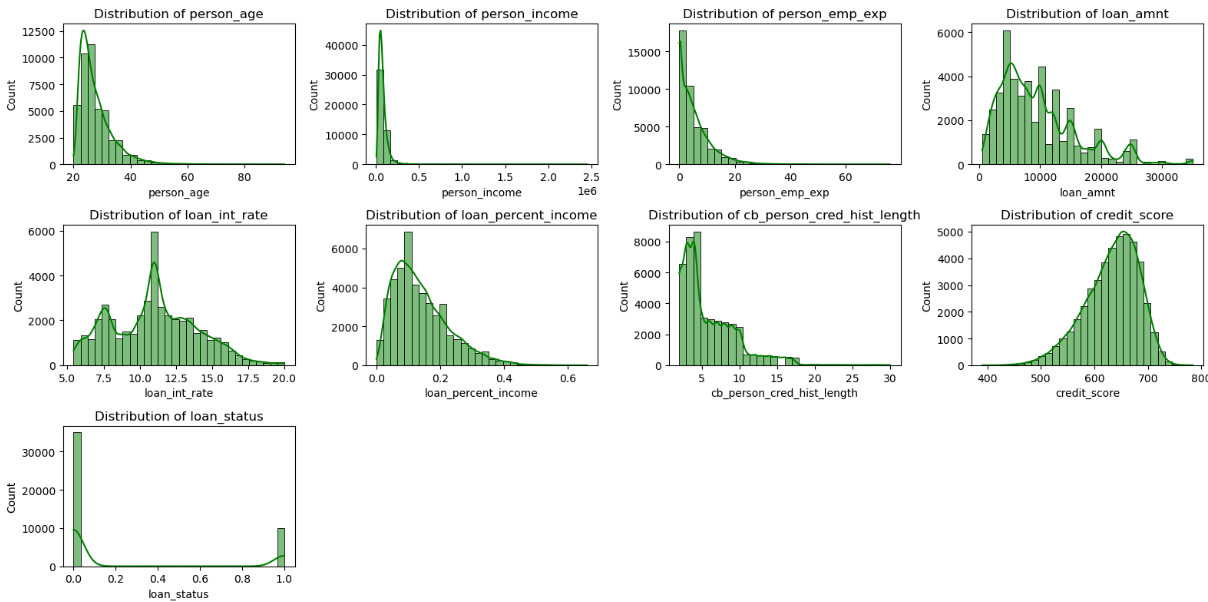


Do the same for numerical columns

In [159…

```
fig, axes = plt.subplots(3, 4, figsize=(16, 8))
axes = axes.flatten()
i = 0
for category in df.columns:
    if np.issubdtype(df[category].dtype, np.number):
        sns.histplot(df[category], kde=True, bins=30, ax=axes[i], color="green")
        axes[i].set_title(f"Distribution of {category}", fontsize=12)
        axes[i].set_xlabel(category)
        axes[i].set_ylabel("Count")
        i += 1

# Hide any remaining unused axes
for j in range(i, len(axes)):
    fig.delaxes(axes[j])
```

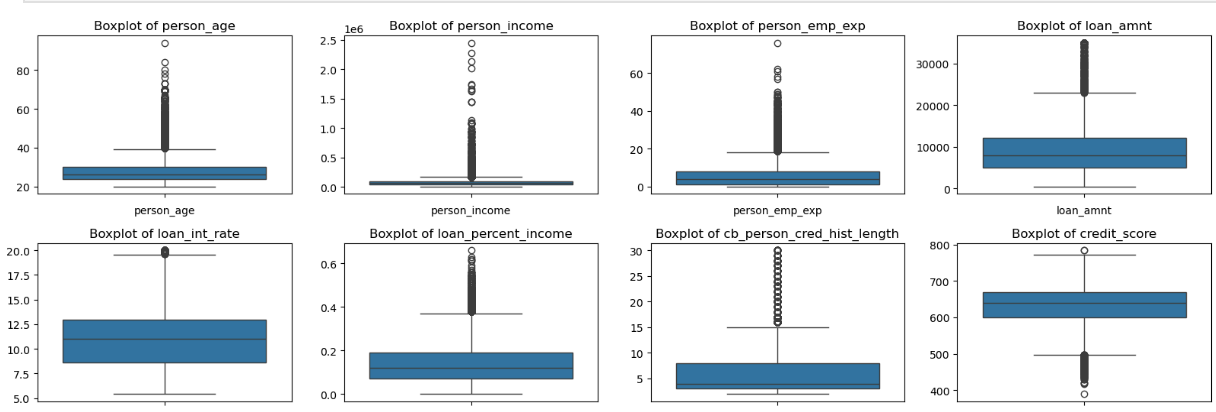
```python
plt.tight_layout()
plt.show()
```



Draw boxplots of numerical columns

In [160…

```python
fig, axes = plt.subplots(3, 4, figsize=(16, 8))
axes = axes.flatten()
i = 0
for category in df.columns:
    if np.issubdtype(df[category].dtype, np.number):
        sns.boxplot(df[category], ax = axes[i])
        axes[i].set_title(f"Boxplot of {category}", fontsize = 12)
        axes[i].set_xlabel(category)
        axes[i].set_ylabel("")
        i += 1

# Hide any remaining unused axes
for j in range(i - 1, len(axes)):
    fig.delaxes(axes[j])

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



The view of the boxplots indicate that the dataset contain a lot of outliers but they just seem like that and they shouldn't be removed as they are real observed values and indictae real life situations

Pairplot of every numeric feature with themselves and with the target, the main diagonal is the kde distribtution of each one of the two opposited features and the color of the points indicate a third dimension which is the target

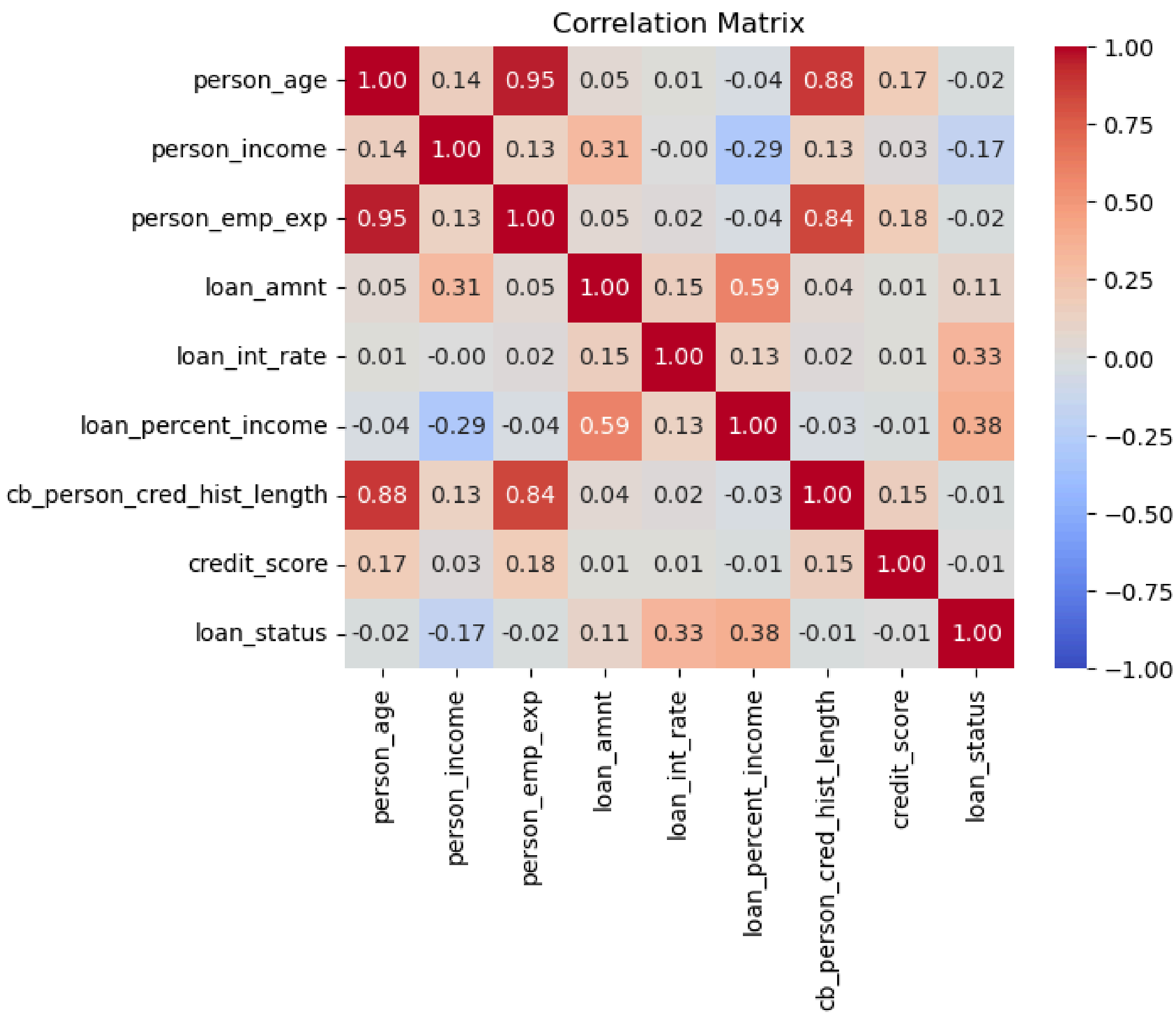
In [129…

```
sns.pairplot(df, hue="loan_status", corner = False, palette="coolwarm")
plt.legend(title = "Loan status", fontsize = 20)
```

Down here the correlation matrix of numeric features with each other which indicates the strength of the relationship between the two opposited features the value of correlation takes a value between -1 and 1 which by as the value gets closer to -1 this indicates strong negative relationship and as it gets closer to 1 it then indicates strong positive relationship and as the value gets closer to 0 then this means that the relationship is week in either way negative or positive

In [161…

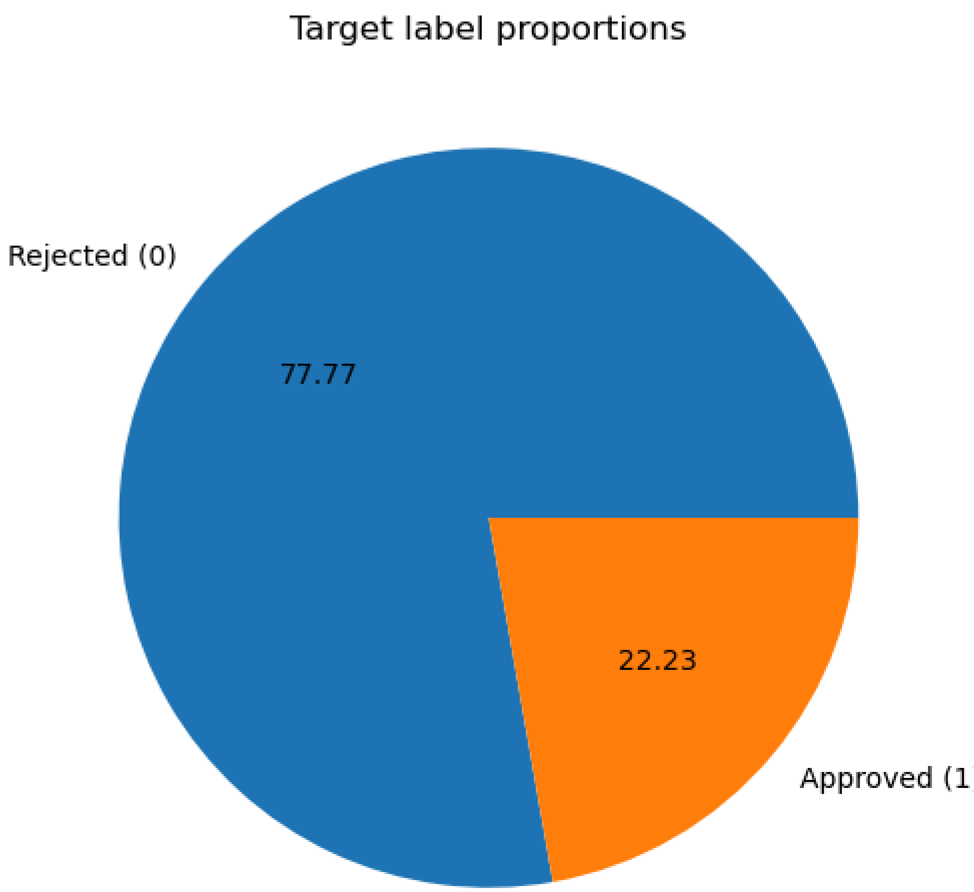
```
corr = df.corr(numeric_only = True)
sns.heatmap(corr, annot=True, cmap="coolwarm", fmt=".2f", vmax = 1, vmin = -1)
plt.title("Correlation Matrix")
plt.show()
```



The following pie chart shows the overall distribution of our target label

In [162…

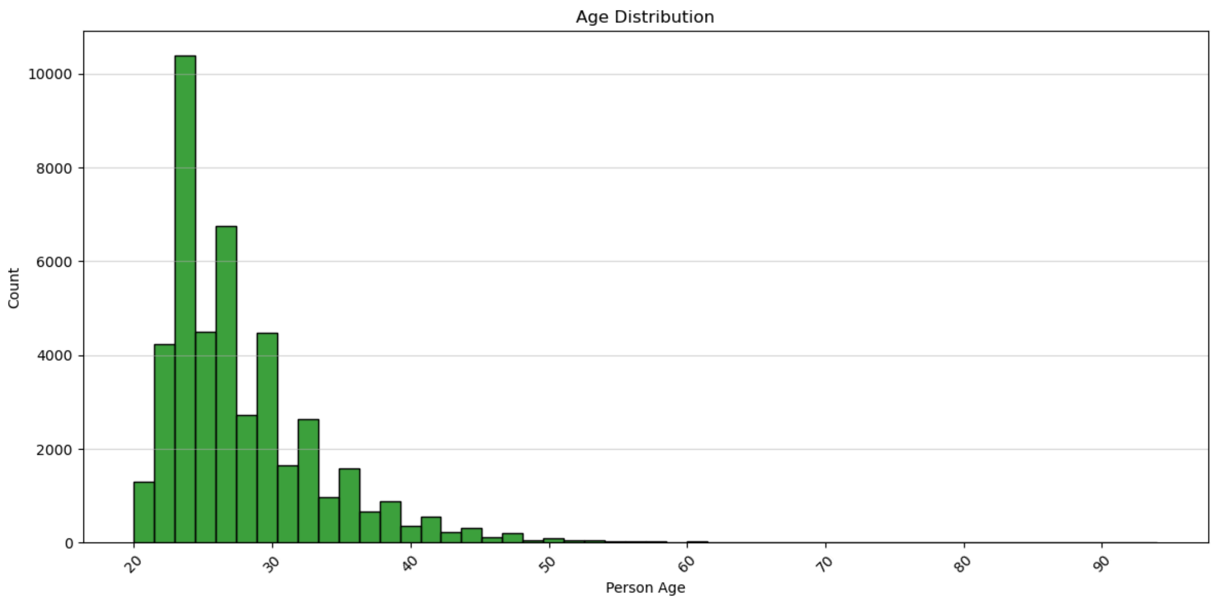
```
plt.figure(figsize=(12, 6))
label_prop = df['loan_status'].value_counts()

plt.pie(label_prop.values, labels=['Rejected (0)', 'Approved (1)'], autopct='%.2f')
plt.title('Target label proportions')
plt.show()
```



The age distribution down below indicates that the dominant age group in our dataset is ages between 20 and 30 years old

In [163…

```
plt.figure(figsize=(12, 6))
sns.histplot(df["person_age"], bins=50, kde=False, color='green')
plt.title("Age Distribution")
plt.xlabel("Person Age")
plt.ylabel("Count")
plt.xticks(rotation=45)
plt.grid(axis='y', alpha=0.5)
plt.tight_layout()
plt.show()
```

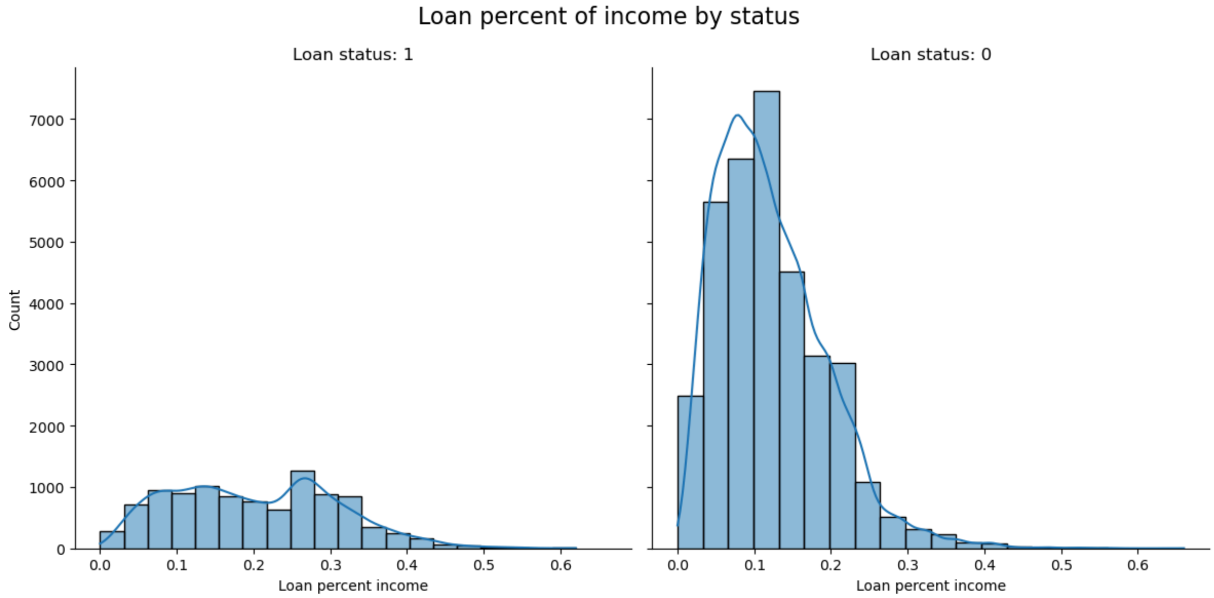


The following two plots shows the loan amount as a percentage of annual income and by this the lender can conclude whether the customer is able to make up for the loan or will not be able to by shorter terms "affordability"

In [164…

```
fig, ax = plt.subplots(1, 2, sharey=True, sharex=True, figsize=(12, 6))

for i, status in enumerate([1, 0]):
    sns.histplot(
        x='loan_percent_income',
        kde=True,
        bins=20,
        palette="muted",
        data=df.query(f"loan_status == {status}"),
        ax=ax[i]
    )
    ax[i].set_title(f"Loan status: {status}")
    ax[i].set_xlabel("Loan percent income")

fig.suptitle("Loan percent of income by status", fontsize=16)

plt.tight_layout()
sns.despine()
plt.show()
```

```
C:\Users\AhMeD\AppData\Local\Temp\ipykernel_10720\1144256084.py:4: UserWarning: Igno
ring `palette` because no `hue` variable has been assigned.
  sns.histplot(
C:\Users\AhMeD\AppData\Local\Temp\ipykernel_10720\1144256084.py:4: UserWarning: Igno
ring `palette` because no `hue` variable has been assigned.
  sns.histplot(
```

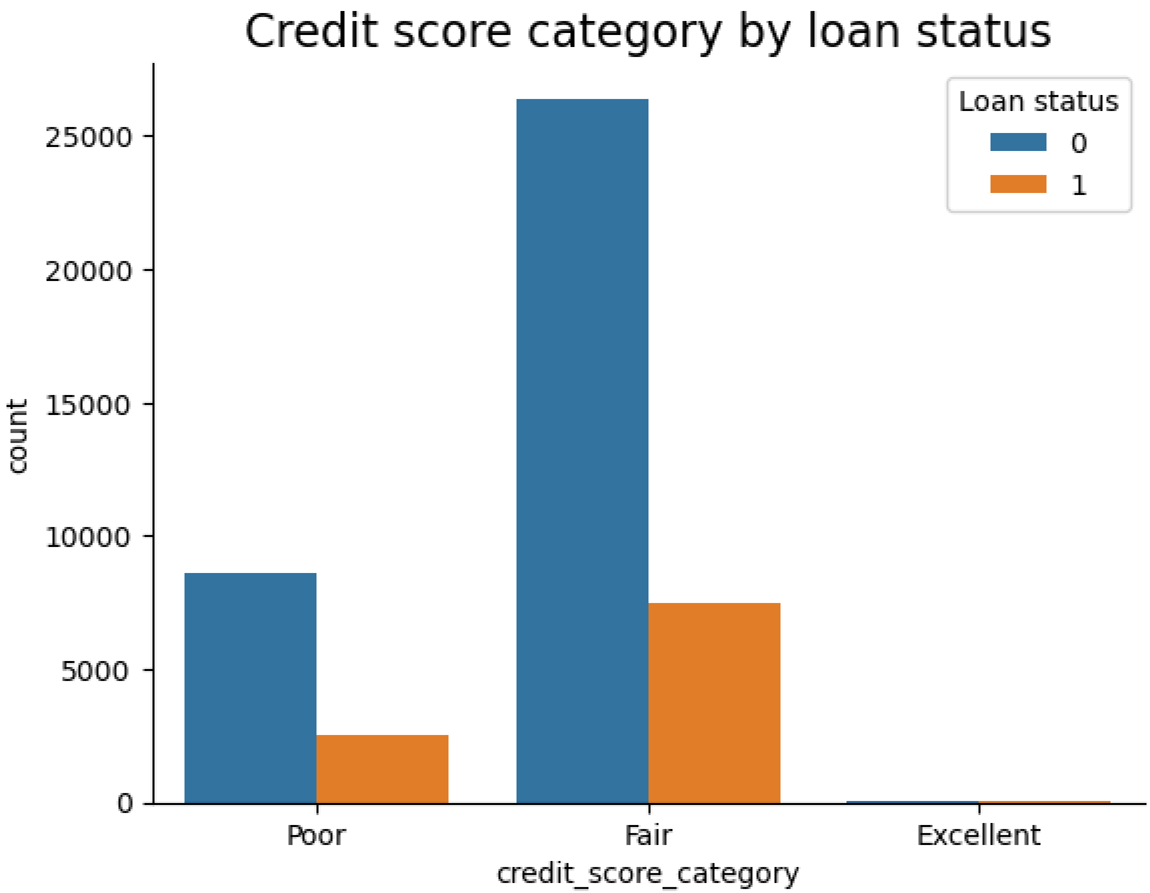


The credit score distribution which is categorised into three classes (Poor, Fair, and Excellent) based on the score, scores from 0 - 600 are considered poor and scores that lie between 600 - 750 are considered Fair and scores exceeding 750 are excellent, the credit score indicates the worthieness of the customer of the loan based on his credit history length and so as this value gets higher the loan approval is recommended

In [165…

```
bins = [0, 600, 750, 850]
labels = ["Poor", "Fair", "Excellent"]

new_df = df.copy()
new_df['credit_score_category'] = pd.cut(df['credit_score'], bins=bins, labels=labe

sns.countplot(
    x='credit_score_category',
    hue='loan_status',
    data=new_df
)

plt.title("Credit score category by loan status", fontsize=16)
plt.legend(title="Loan status")
sns.despine()
plt.show()
```

```
C:\Users\AhMeD\AppData\Local\Temp\ipykernel_10720\3816762142.py:4: SettingWithCopyWa
rning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy
  df['credit_score_category'] = pd.cut(df['credit_score'], bins=bins, labels=labels)
```



Feature Engineering

Apply Label Encoding to Convert each unique category into an integer

In [134…

```
# dataframe before encoding
df
```

Out[134…

	person_age	person_gender	person_education	person_income	person_emp_exp	pe
0	22.0	female	Master	71948.0	0	
1	21.0	female	High School	12282.0	0	
2	25.0	female	High School	12438.0	3	
3	23.0	female	Bachelor	79753.0	0	
4	24.0	male	Master	66135.0	1	
...	...	...	...	...	...	
44995	27.0	male	Associate	47971.0	6	
44996	37.0	female	Associate	65800.0	17	
44997	33.0	male	Associate	56942.0	7	
44998	29.0	male	Bachelor	33164.0	4	
44999	24.0	male	High School	51609.0	1	

44993 rows × 14 columns

In [135…

```
# Information about the df
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 44993 entries, 0 to 44999
Data columns (total 14 columns):
 #   Column                          Non-Null Count  Dtype  
---  ------                          --------------  -----  
 0   person_age                      44993 non-null  float64
 1   person_gender                   44993 non-null  object 
 2   person_education                44993 non-null  object 
 3   person_income                   44993 non-null  float64
 4   person_emp_exp                  44993 non-null  int64  
 5   person_home_ownership           44993 non-null  object 
 6   loan_amnt                       44993 non-null  float64
 7   loan_intent                     44993 non-null  object 
 8   loan_int_rate                   44993 non-null  float64
 9   loan_percent_income             44993 non-null  float64
 10  cb_person_cred_hist_length      44993 non-null  float64
 11  credit_score                    44993 non-null  int64  
 12  previous_loan_defaults_on_file  44993 non-null  object 
 13  loan_status                     44993 non-null  int64  
dtypes: float64(6), int64(3), object(5)
memory usage: 5.1+ MB
```

In [136…

```
from sklearn.preprocessing import LabelEncoder

# Initialize LabelEncoder

label_encoder = LabelEncoder()
```

```
encoded_df = df.copy()

# Identify categorical columns
categorical_columns = encoded_df.select_dtypes(include=['object']).columns

# Apply LabelEncoder to a categorical column
for col in categorical_columns:
    encoded_df[col] = label_encoder.fit_transform(encoded_df[col])

# print the encoded_df
encoded_df
```

Out[136…

	person_age	person_gender	person_education	person_income	person_emp_exp	pe
0	22.0	0	4	71948.0	0	
1	21.0	0	3	12282.0	0	
2	25.0	0	3	12438.0	3	
3	23.0	0	1	79753.0	0	
4	24.0	1	4	66135.0	1	
...	...	...	...	...	...	
44995	27.0	1	0	47971.0	6	
44996	37.0	0	0	65800.0	17	
44997	33.0	1	0	56942.0	7	
44998	29.0	1	1	33164.0	4	
44999	24.0	1	3	51609.0	1	

44993 rows × 14 columns

Machine Learning Algorithms

1) Naive Bayes

In [137…

```
from sklearn.model_selection import cross_validate, train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

In [138…

```
X = encoded_df.drop(columns=['loan_status'])
y = encoded_df['loan_status']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_s
```

In [139…
```
scoring = ['accuracy', 'precision', 'recall', 'f1_weighted']
```

In [140…
```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_train)
```

In [141…
```
naive_bayes = GaussianNB()
nb_scores = cross_validate(naive_bayes, X_scaled, y_train, scoring = scoring, retur

print("\nNaive bayes scores:")
for metric in scoring:
    print(f"{metric.capitalize()}: {nb_scores['test_' + metric].mean():.4f}")
```

```
Naive bayes scores:
Accuracy: 0.7303
Precision: 0.4518
Recall: 0.9969
F1_weighted: 0.7529
```

2) KNN

import the used libraries

In [ ]:
```
from sklearn.preprocessing import MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_validate, GridSearchCV
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
import math
```

In [ ]:
```
mod_df = encoded_df.copy()
```

define x as independent variables and y as output/label/target

In [ ]:
```
X = mod_df.iloc[:, :-1].values
y = mod_df.iloc[:, -1].values
```

In [ ]:
```
X
```

```
array([[ 22.,   0.,   4., ...,   3., 561.,   0.],
       [ 21.,   0.,   3., ...,   2., 504.,   1.],
       [ 25.,   0.,   3., ...,   3., 635.,   0.],
       ...,
       [ 33.,   1.,   0., ...,  10., 668.,   0.],
       [ 29.,   1.,   1., ...,   6., 604.,   0.],
       [ 24.,   1.,   3., ...,   3., 628.,   0.]])
```

define the range of K from 1 to the sqrt of len of dataset, then make scorers directory which store accuracy, precision, recall and f1_score of each K, then scalling the data using min_max scaler

In [ ]:
```
k_values = [i for i in range (1,int(math.sqrt(len(df)))+1)]
scorers = {
    'accuracy': make_scorer(accuracy_score),
    'precision': make_scorer(precision_score, average='weighted'),
    'recall': make_scorer(recall_score, average='weighted'),
    'f1_score': make_scorer(f1_score, average='weighted')
}

scaler = MinMaxScaler()
X = scaler.fit_transform(X)
```

split data into 80% for train and 20% for test

In [ ]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20, random_
```

for each k i store its accuracy, precision, recall and f1_score and use cross validation to get best k

In [ ]:
```
results = []
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_validate(knn, X_train, y_train, cv=5,  scoring=scorers)
    results.append({
        'k': k,
        'accuracy': np.mean(scores['test_accuracy']),
        'precision': np.mean(scores['test_precision']),
        'recall': np.mean(scores['test_recall']),
        'f1_score': np.mean(scores['test_f1_score'])
    })
```

make the results as dataframe with metrics corresponding to each K

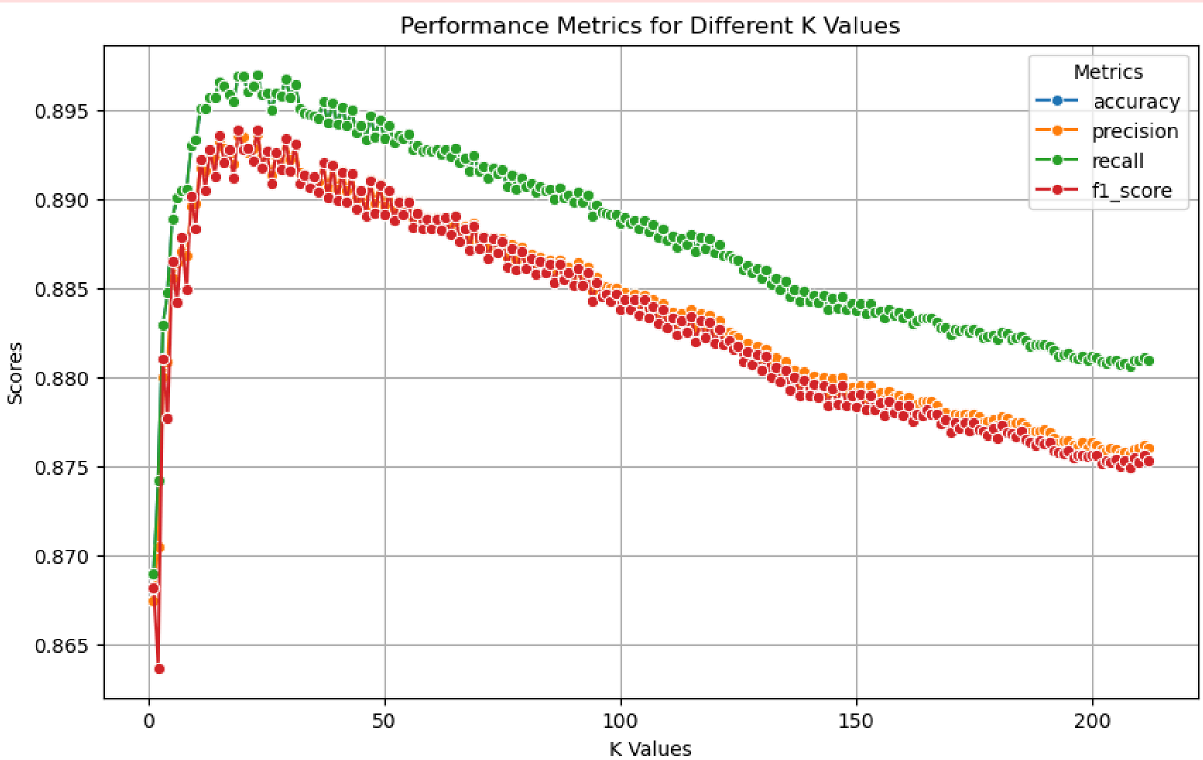
In [ ]:
```
results_df = pd.DataFrame(results)

print(results_df)
```

```
       k  accuracy  precision    recall  f1_score
0      1  0.868984   0.867486  0.868984  0.868162
1      2  0.874180   0.870467  0.874180  0.863698
2      3  0.882933   0.880023  0.882933  0.881048
3      4  0.884767   0.880882  0.884767  0.877681
4      5  0.888852   0.885567  0.888852  0.886478
..   ...       ...        ...       ...       ...
207  208  0.880627   0.875677  0.880627  0.874921
208  209  0.880905   0.875984  0.880905  0.875460
209  210  0.880932   0.876010  0.880932  0.875277
210  211  0.881071   0.876158  0.881071  0.875647
211  212  0.880932   0.876006  0.880932  0.875328

[212 rows x 5 columns]
```

visualizes how the performance metrcis change as K varies, to help identify the best K for the model

In [ ]:
```python
metrics = ['accuracy', 'precision', 'recall', 'f1_score']

plt.figure(figsize=(10, 6))

for metric in metrics:
    sns.lineplot(x=results_df['k'], y=results_df[metric], marker='o', label=metric)

# Customize the plot
plt.xlabel("K Values")
plt.ylabel("Scores")
plt.title("Performance Metrics for Different K Values")
plt.legend(title="Metrics")
plt.grid(True)
plt.show()
```

```
c:\Users\Mega Store\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarn
ing: use_inf_as_na option is deprecated and will be removed in a future version. Con
vert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
c:\Users\Mega Store\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarn
ing: use_inf_as_na option is deprecated and will be removed in a future version. Con
vert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
c:\Users\Mega Store\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarn
ing: use_inf_as_na option is deprecated and will be removed in a future version. Con
vert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
c:\Users\Mega Store\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarn
ing: use_inf_as_na option is deprecated and will be removed in a future version. Con
vert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
c:\Users\Mega Store\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarn
ing: use_inf_as_na option is deprecated and will be removed in a future version. Con
vert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
c:\Users\Mega Store\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarn
ing: use_inf_as_na option is deprecated and will be removed in a future version. Con
vert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
c:\Users\Mega Store\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarn
ing: use_inf_as_na option is deprecated and will be removed in a future version. Con
vert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
c:\Users\Mega Store\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarn
ing: use_inf_as_na option is deprecated and will be removed in a future version. Con
vert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```



Check Data Balance, which used to know if we will choose the k with highest f1_score or the highest accuracy, and as we see the data tend to be imbalanced as the two class are 77.77 and 22.22 percentage, so we tend to take the k with highest f1_score

In [ ]:
```
unique_classes, class_counts = np.unique(y, return_counts=True)

class_distribution = pd.DataFrame({
    'Class': unique_classes,
    'Count': class_counts,
    'Percentage': (class_counts / len(y)) * 100
})

print(class_distribution)

threshold = 40
is_imbalanced = (class_distribution['Percentage'] < threshold).any()

if is_imbalanced:
    print("The dataset is imbalanced.")
else:
    print("The dataset is balanced.")
```

```
   Class  Count  Percentage
0      0  34993    77.77432
1      1  10000    22.22568
The dataset is imbalanced.
```

By normalizing the metrics to the range zero and one, all metrics are brought to the same scale, allowing for a fair combination, The combined score is calculated as the average of the normalized metrics, representing an aggregate measure of performance.Finally, the code identifies the value of k that maximizes this combined score

In [ ]:
```
results_df['normalized_accuracy'] = results_df['accuracy'] / results_df['accuracy']
results_df['normalized_precision'] = results_df['precision'] / results_df['precisio
results_df['normalized_recall'] = results_df['recall'] / results_df['recall'].max()
results_df['normalized_f1_score'] = results_df['f1_score'] / results_df['f1_score']

results_df['combined_score'] = (
    results_df['normalized_accuracy'] +
    results_df['normalized_precision'] +
    results_df['normalized_recall'] +
    results_df['normalized_f1_score']
) / 4

highest_k = results_df.loc[results_df['combined_score'].idxmax(), 'k']

print(f"highest k value based on combined score: {highest_k}")
```

```
highest k value based on combined score: 23
```

or as our classlabel is imbalanced, so we seek to get the k with highest F1_score

In [ ]:
```
best_k = results_df.loc[results_df['f1_score'].idxmax(), 'k']

print(f"Best k value based on highest F1-score: {best_k}")
```

```
Best k value based on highest F1-score: 19
```

In [ ]:
```
print(X_train)
```

```
[[0.13513514 1.         0.25       ... 0.17857143 0.73604061 0.        ]
 [0.25675676 1.         0.25       ... 0.53571429 0.70304569 1.        ]
 [0.22972973 1.         1.         ... 0.39285714 0.86548223 1.        ]
 ...
 [0.09459459 0.         1.         ... 0.10714286 0.67766497 0.        ]
 [0.09459459 1.         0.25       ... 0.17857143 0.76903553 0.        ]
 [0.02702703 0.         0.75       ... 0.         0.55076142 0.        ]]
```

use the best choosen k and try with metric minkowski with p=2 (euclidean distance) and train the model with the x_train and y_train

In [ ]:
```
classifier = KNeighborsClassifier(n_neighbors = 19, metric = 'minkowski', p = 2)
classifier.fit(X_train, y_train)
```

```
▾ KNeighborsClassifier
KNeighborsClassifier(n_neighbors=19)
```

use the model to predict using the testing data, then print it and its coresponding actual y_test

In [ ]:
```
y_pred = classifier.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1))
```

```
[[0 1]
 [0 0]
 [0 0]
 ...
 [0 0]
 [1 1]
 [1 1]]
```

In [ ]:
```
comparison = pd.DataFrame({
    'y_pred': y_pred,
    'y_test': y_test
})

print(comparison)
```

```
      y_pred  y_test
0          0       1
1          0       0
2          0       0
3          1       1
4          0       0
...      ...     ...
8994       0       0
8995       0       0
8996       0       0
8997       1       1
8998       1       1

[8999 rows x 2 columns]
```

print the confusion matrix and other metric like accuracy, recall, precision and f1_score for evalution of KNN performance

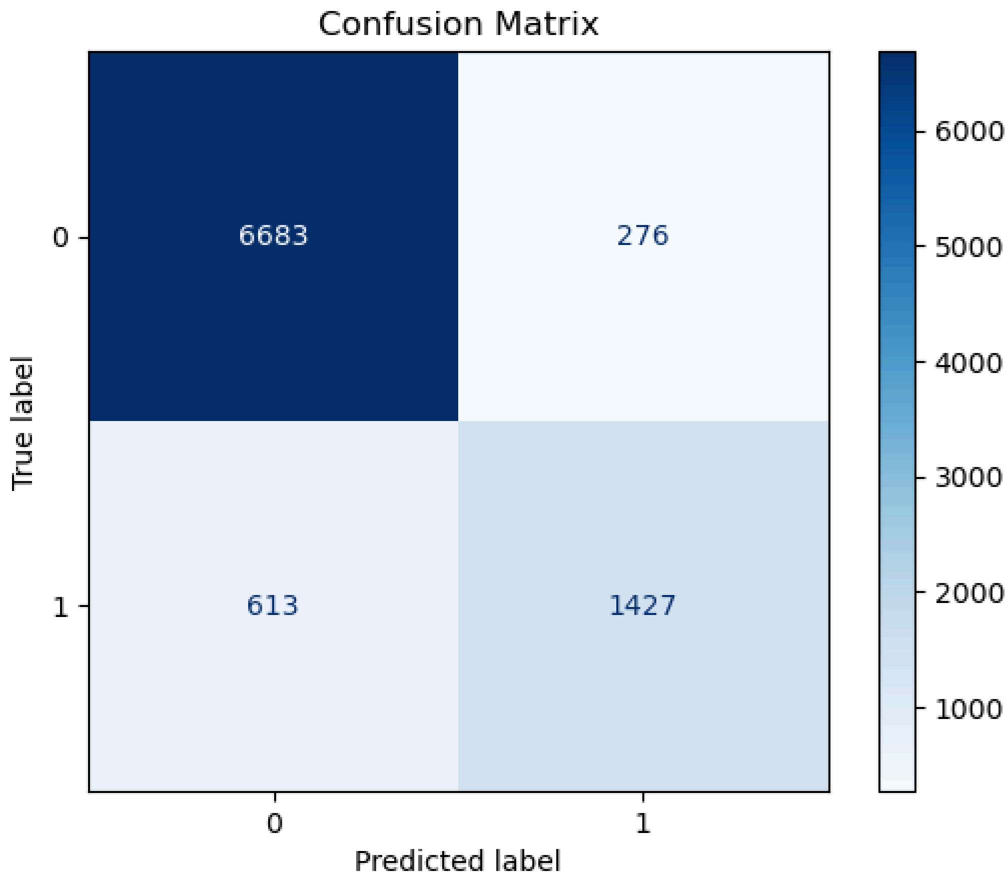
In [ ]:
```
cm = confusion_matrix(y_test, y_pred)
print(cm)
# [[TN  FP]
#  [FN  TP]]
```

```
[[6683  276]
 [ 613 1427]]
```

In [ ]:
```
tn, fp, fn, tp = cm.ravel()
print(f"True Negatives: {tn}")
print(f"False Positives: {fp}")
print(f"False Negatives: {fn}")
print(f"True Positives: {tp}")
```

```
True Negatives: 6683
False Positives: 276
False Negatives: 613
True Positives: 1427
```

In [ ]:
```
class_names = sorted(set(y_test))
ConfusionMatrixDisplay.from_predictions(y_test, y_pred, display_labels=class_names,
plt.title('Confusion Matrix')
plt.show()
```



make comparison between each matric distance and study which will result into more good metrics like f1_score and others, and after observing the results, manhatten and eclidean are slightly better than others distance as : Euclidean and Manhattan distances perform better when features are independent and not redundant. In high-dimensional data , other metrics like hamming can struggle because: it isn't suitable for high dimensional data and limited as the most of the dataset aren't binary or categorical. Manhattan distance is more robust in oure case because it sums individual feature differences

In [ ]:
```
metrics = ['euclidean', 'manhattan', 'minkowski', 'chebyshev', 'hamming', 'cosine']
metric_results = []

for metric in metrics:
    knn = KNeighborsClassifier(n_neighbors=19, metric=metric)
    knn.fit(X_train, y_train)

    y_test_pred = knn.predict(X_test)

    accuracy = accuracy_score(y_test, y_test_pred)
    precision = precision_score(y_test, y_test_pred, average='weighted')
    recall = recall_score(y_test, y_test_pred, average='weighted')
    f1 = f1_score(y_test, y_test_pred, average='weighted')
```

```
    metric_results.append({'metric': metric, 'accuracy': accuracy, 'precision': pre
```

In [ ]:
```
metric_comparison=pd.DataFrame(metric_results)
print(metric_comparison)
```

```
      metric  accuracy  precision    recall  f1_score
0  euclidean  0.901211   0.898288  0.901211  0.897932
1  manhattan  0.902767   0.899987  0.902767  0.899387
2  minkowski  0.901211   0.898288  0.901211  0.897932
3  chebyshev  0.895544   0.892187  0.895544  0.891901
4    hamming  0.840427   0.830310  0.840427  0.830945
5     cosine  0.899100   0.896153  0.899100  0.896527
```

so i used the best metric distance and best k then i will use the grid search to optimize the model's performance

In [ ]:
```
knn = KNeighborsClassifier(n_neighbors=19, metric='manhattan')
knn.fit(X_train, y_train)
```

```
▾ KNeighborsClassifier
KNeighborsClassifier(metric='manhattan', n_neighbors=19)
```

the Hyperparameters here are settings for KNN model that are not learned during training .Hyperparameter tuning searches for the combination of hyperparameters that gives the best performance on the training data, using cross-validation

In [ ]:
```
param_grid = {
    'weights': ['uniform', 'distance'],
    'leaf_size': [20, 30, 40, 50],
}
grid_search = GridSearchCV(estimator=knn, param_grid=param_grid, cv=5)
grid_search.fit(X_train, y_train)
```

```
▸ GridSearchCV
▸ estimator: KNeighborsClassifier
    ▸ KNeighborsClassifier
```

the weighted value Handles Imbalanced Data, Ensures classes with more samples have a fair impact, then check the improved F1_score and accuracy and other metrics.

In [ ]:
```
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_
print("Best Parameters:", best_params)
```

```python
# Evaluate the model on the test set
y_pred = best_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_test_pred, average='weighted')
recall = recall_score(y_test, y_test_pred, average='weighted')
F_1 = f1_score(y_test, y_pred, average='weighted')
print("Test Set F1_score:", F_1)
print("Test Set Accuracy:", accuracy)
print("Test Set Recall:", recall)
print("Test Set precision:", precision)
```

```
Best Parameters: {'leaf_size': 20, 'weights': 'distance'}
Test Set F1_score: 0.9008526619441675
Test Set Accuracy: 0.9041004556061785
Test Set Recall: 0.8990998999888876
Test Set precision: 0.8961528441140358
```

In [ ]:
```python
comparison_after = pd.DataFrame({
    'y_pred': y_pred,
    'y_test': y_test
})
```

In [ ]:
```python
print(comparison_after)
```

```
      y_pred  y_test
0          0       1
1          0       0
2          0       0
3          1       1
4          0       0
...      ...     ...
8994       0       0
8995       0       0
8996       0       0
8997       1       1
8998       1       1

[8999 rows x 2 columns]
```

3) Decision Tree

x & y split

In [ ]:
```python
# select all columns (features) except the target 'loan_state'
x = encoded_df.loc[:, encoded_df.columns != 'loan_status']
x.head()
```

	person_age	person_gender	person_education	person_income	person_emp_exp	person_h
0	22.0	0	4	71948.0	0	
1	21.0	0	3	12282.0	0	
2	25.0	0	3	12438.0	3	
3	23.0	0	1	79753.0	0	
4	24.0	1	4	66135.0	1	

In [ ]:
```
# dim of the features
x.shape
```

```
(44993, 13)
```

- No. of observations = 44993
- No. of features = 13

In [ ]:
```
# select the target column 'loan_state'
y = encoded_df['loan_status']
y.head()
```

```
0    1
1    0
2    1
3    1
4    1
Name: loan_status, dtype: int64
```

- 1 --> approved loan
- 0 --> rejected loan

train & test split

In [ ]:
```
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets (80% train, 20% test)
x_train, x_test, y_train, y_test = train_test_split(x, y,test_size=0.2,random_state

# training and testing dataset size
print(f"Training set size: {x_train.shape[0]}")
print(f"Testing set size: {x_test.shape[0]}")
```

```
Training set size: 35994
Testing set size: 8999
```

Training Classification Tree

In [ ]:
```
from sklearn.tree import DecisionTreeClassifier

# Instantiate the Decision Tree model
model = DecisionTreeClassifier(random_state=0)

# Train the model with training data
model.fit(x_train, y_train)
```

```
▾ DecisionTreeClassifier
DecisionTreeClassifier(random_state=0)
```

Evaluate the model performance using evaluation metrics

In [ ]:
```
# Check the distribution of the target variable 'loan_status'
target_distribution = (y.value_counts(normalize=True) * 100).round(2)
print("Target distribution:\n", target_distribution.astype(str) + '%')
```

```
Target distribution:
 loan_status
0    77.77%
1    22.23%
Name: proportion, dtype: object
```

- As we can see, the target distribution is imbalanced with the majority class being "rejected" (77.77%) and the minority class being "approved" (22.23%)
  - For imbalanced data, accuracy may not be the best metric because it could be skewed towards the majority class
  - So, we will use F1-score instead as it combines the benefits of precision and recall into one metric

---

- In addition to F1-score, we are also interested in Precision and Recall:
  - Precision is important when we want to avoid false positives, i.e., incorrectly classifying loans as approved when they should be rejected.
  - Recall is important when we want to minimize false negatives, i.e., making sure that loans that should be approved are not missed.

In [ ]:
```
from sklearn.metrics import precision_score, recall_score, f1_score

# Make predictions on the test set
y_pred = model.predict(x_test)


# Calculate F1-score
f1 = f1_score(y_test, y_pred)
print(f"F1-score: {f1:.3f}")

# Calculate Precision
```

```
precision = precision_score(y_test, y_pred)
print(f"Precision: {precision:.3f}")

# Calculate Recall
recall = recall_score(y_test, y_pred)
print(f"Recall: {recall:.3f}")
```

```
F1-score: 0.774
Precision: 0.770
Recall: 0.778
```

Evaluation Metrics Results:

- **F1-score: 0.774**
  - The F1-score is a harmonic mean of precision and recall, balancing both metrics.
  - A value of 0.774 indicates a good balance between identifying approved loans correctly (recall) and avoiding false approvals (precision).
- **Precision: 0.770**
  - Precision measures how many of the loans predicted as "approved" are actually "approved."
  - A precision of 0.77 means that 77% of the loans we classified as "approved" were indeed correct.
  - This indicates a moderate ability to avoid false positives (incorrectly approving rejected loans).
- **Recall: 0.778**
  - Recall measures how many of the actual "approved" loans were correctly identified by the model.
  - A recall of 0.778 means the model captured 77.8% of all loans that should have been approved.
  - This indicates a reasonable ability to avoid false negatives (missing loans that should be approved).

However, these metrics are not yet optimal as they are based on the default hyperparameters of the Decision Tree model. In the next steps, we will optimize the model by experimenting with and tuning hyperparameters to achieve better performance

Calculate the Confusion Matrix

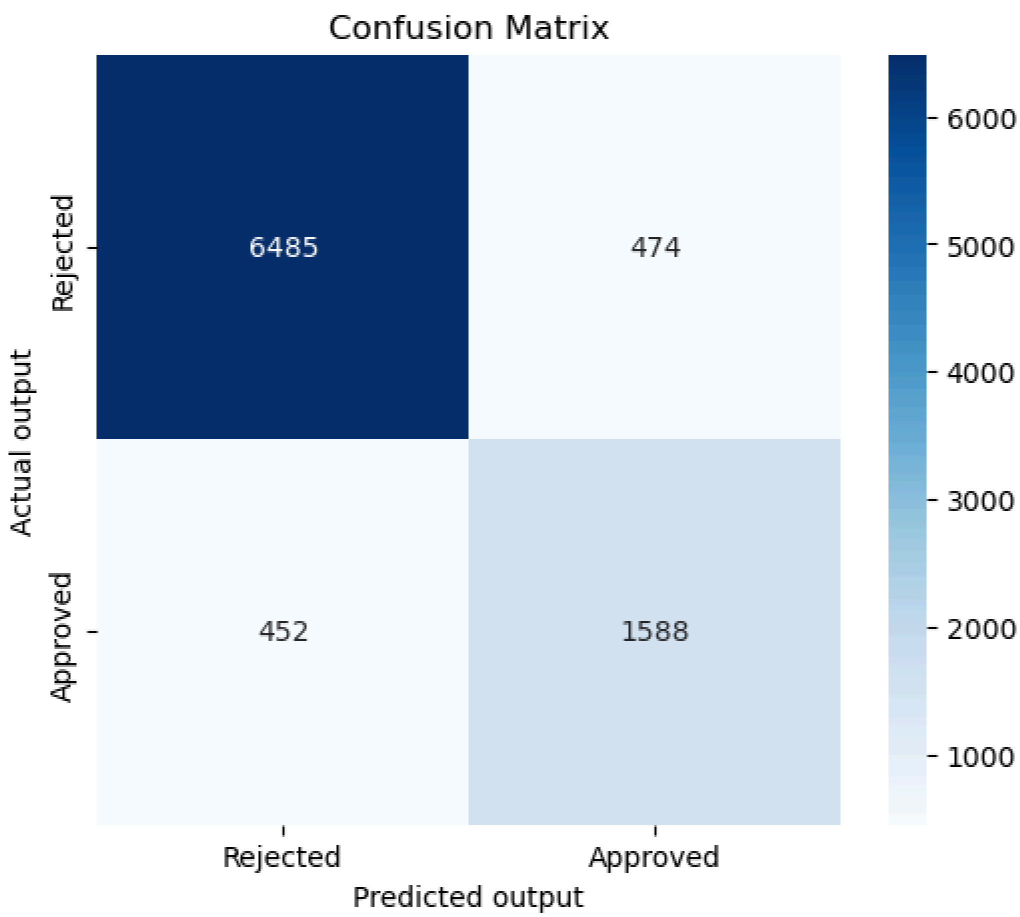
In [ ]:
```
from sklearn.metrics import confusion_matrix

# Calculate the confusion matrix
cm = confusion_matrix(y_test, y_pred)
```

```
# Display the confusion matrix
print("Confusion Matrix:\n", cm)
```

```
Confusion Matrix:
 [[6485  474]
 [ 452 1588]]
```

Visualize the Confusion Matrix

In [ ]:
```
# Plot the confusion matrix
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["Rejected", "Approv
plt.xlabel("Predicted output")
plt.ylabel("Actual output")
plt.title("Confusion Matrix")
plt.show()
```



Manual Hyperparameter Tuning of Decision Tree Classifier

- In this step, we will manually adjust the values of key hyperparameters to analyze their impact on the model's performance.

In [ ]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import f1_score, precision_score, recall_score

# Define hyperparameter combinations to try manually
```

```python
hyperparameters = [
    {'max_depth': 3, 'min_samples_split': 2, 'criterion': 'gini'},
    {'max_depth': 5, 'min_samples_split': 10, 'criterion': 'gini'},
    {'max_depth': 10, 'min_samples_split': 2, 'criterion': 'entropy'},
    {'max_depth': None, 'min_samples_split': 2, 'criterion': 'gini'},
    {'max_depth': 7, 'min_samples_split': 5, 'criterion': 'entropy'}
]

# Loop through different hyperparameter settings
for hyper_para in hyperparameters:
    print(f"\nTesting hyperparameters: {hyper_para}")

    # Create model with current hyperparameters
    model = DecisionTreeClassifier(max_depth=hyper_para["max_depth"],
                                   min_samples_split=hyper_para["min_samples_split"
                                   criterion=hyper_para["criterion"],
                                   random_state=0)

    # Train the model
    model.fit(x_train, y_train)

    # Predict on test data
    y_pred = model.predict(x_test)

    # Evaluate model performance in each iteration
    # Calculate F1-score
    f1 = f1_score(y_test, y_pred)
    print(f"F1-score: {f1:.3f}")

    # Calculate Precision
    precision = precision_score(y_test, y_pred)
    print(f"Precision: {precision:.3f}")

    # Calculate Recall
    recall = recall_score(y_test, y_pred)
    print(f"Recall: {recall:.3f}")

    print("-" * 90)
```

```
Testing hyperparameters: {'max_depth': 3, 'min_samples_split': 2, 'criterion': 'gin
i'}
F1-score: 0.742
Precision: 0.869
Recall: 0.647
--------------------------------------------------------------------------------
------

Testing hyperparameters: {'max_depth': 5, 'min_samples_split': 10, 'criterion': 'gin
i'}
F1-score: 0.798
Precision: 0.895
Recall: 0.721
--------------------------------------------------------------------------------
------

Testing hyperparameters: {'max_depth': 10, 'min_samples_split': 2, 'criterion': 'ent
ropy'}
F1-score: 0.799
Precision: 0.906
Recall: 0.715
--------------------------------------------------------------------------------
------

Testing hyperparameters: {'max_depth': None, 'min_samples_split': 2, 'criterion': 'g
ini'}
F1-score: 0.774
Precision: 0.770
Recall: 0.778
--------------------------------------------------------------------------------
------

Testing hyperparameters: {'max_depth': 7, 'min_samples_split': 5, 'criterion': 'entr
opy'}
F1-score: 0.797
Precision: 0.923
Recall: 0.702
--------------------------------------------------------------------------------
------
```

Insights and Observations from Manual Hyperparameter Tuning

In this step, we experimented with different hyperparameter combinations to understand their impact on the performance of the Decision Tree model. Below are the results and key insights:

Hyperparameters: {'max_depth': 3, 'min_samples_split': 2, 'criterion': 'gini'}

- **F1-score: 0.742**
- **Precision: 0.869**

- **Recall: 0.647**
- This combination results in a relatively low F1-score, indicating a moderate balance between precision and recall.
- **Precision is high** (86.9%), suggesting that most of the positive predictions made by the model are correct.
- However, the **Recall is low** (64.7%), meaning a significant portion of the positives is being missed by the model, leading to a higher number of false negatives.

Hyperparameters: {'max_depth': 5, 'min_samples_split': 10, 'criterion': 'gini'}

- **F1-score: 0.798**
- **Precision: 0.895**
- **Recall: 0.721**
- With these settings, the F1-score improves to 0.798, indicating a better balance between precision and recall.
- **Precision remains high** (89.5%), and **Recall increases** to 72.1%, suggesting that this combination is more effective at correctly identifying positives without significantly increasing false positives.

Hyperparameters: {'max_depth': 10, 'min_samples_split': 2, 'criterion': 'entropy'}

- **F1-score: 0.799**
- **Precision: 0.906**
- **Recall: 0.715**
- This combination gives the best **Precision (90.6%)**, indicating that the model is more accurate in its positive predictions.
- The **Recall is moderate** (71.5%), meaning the model misses some positives but is quite effective at avoiding false positives.
- Overall, this setup results in the best trade-off between precision and recall so far.

Hyperparameters: {'max_depth': None, 'min_samples_split': 2, 'criterion': 'gini'}

- **F1-score: 0.774**
- **Precision: 0.770**

- **Recall: 0.778**
- This combination, with an unlimited **max_depth**, results in an F1-score of 0.774, which is relatively balanced.
- **Precision (77.0%)** and **Recall (77.8%)** are close to each other, indicating that the model is performing reasonably well in both avoiding false positives and capturing most positives.
- This is a good default combination that strikes a reasonable balance between precision and recall.

Hyperparameters: {'max_depth': 7, 'min_samples_split': 5, 'criterion': 'entropy'}

- **F1-score: 0.797**
- **Precision: 0.923**
- **Recall: 0.702**
- The F1-score is slightly improved at 0.797, and **Precision remains high** (92.3%).
- However, **Recall** is somewhat lower at 70.2%, meaning the model still misses a portion of the positives.
- This trade-off between high precision and moderate recall could be further optimized.

Key Insights:

1. **Max Depth**: Increasing the max depth generally improves the F1-score and recall, but it can lead to overfitting if taken too far. The optimal depth depends on the data complexity.
2. **Min Samples Split**: Lower values of `min_samples_split` allow the model to make finer splits, which helps capture more details but can lead to overfitting if the tree is too complex.
3. **Criterion**: The change from `gini` to `entropy` has a noticeable impact on precision and recall. The `entropy` criterion often leads to better precision but can sometimes slightly lower recall.
4. **F1-score**: The highest F1-score was achieved with a combination of `max_depth=10`, `min_samples_split=2`, and `criterion='entropy'`. This combination produced the best balance of precision and recall, though recall could still be improved further.
5. **Model Performance**: While the model performance improved with each new combination, there is still potential for further optimization, particularly in improving recall without sacrificing precision.

In conclusion, while the current hyperparameter tuning has provided useful insights, further improvements can be made. To achieve the best performance, we will now focus on hyperparameter optimization using `Grid Search` to systematically find the optimal combination of hyperparameters.

Gird Search

In [ ]:
```
from sklearn.metrics import accuracy_score, confusion_matrix, f1_score
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier


param_grid = {
    'max_depth': [3, 5, 10, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4,],
    'criterion': ['gini', 'entropy'],
    'random_state':[0]
}

# Initialize the decision tree classifier
clftree = DecisionTreeClassifier()

# Perform grid search with f1_weighted as the scoring metric
grid_search = GridSearchCV(estimator=clftree, param_grid=param_grid, cv=5, scoring=
grid_search.fit(x_train, y_train)

# Best hyperparameters from grid search
print("Best Parameters:", grid_search.best_params_)

# Evaluate the best model on the test set
best_model = grid_search.best_estimator_
y_test_pred = best_model.predict(x_test)  # Make predictions on the test set

# Calculate F1 Score for the test set
print("F1 Score:", f1_score(y_test, y_test_pred, average='weighted'))  # Ensure you
```

```
Best Parameters: {'criterion': 'gini', 'max_depth': 10, 'min_samples_leaf': 1, 'min_
samples_split': 10, 'random_state': 0}
F1 Score: 0.9162292660544267
```

Grid Search

This code demonstrates how to optimize hyperparameters for a Decision Tree Classifier using **GridSearchCV**, with the `f1_weighted` scoring metric, which is particularly useful for imbalanced datasets.

Key Steps:

1. **Importing Libraries**:
   The necessary libraries are imported to build, optimize, and evaluate the Decision Tree Classifier. Metrics like accuracy, confusion matrix, and F1 score are used to measure model performance.

2. **Defining the Parameter Grid**:
   The `param_grid` defines the hyperparameters to test and their respective ranges:

   - `max_depth`: Limits the depth of the tree to control overfitting and underfitting. Includes specific values (3, 5, 10) and `None` for unlimited depth.
   - `min_samples_split`: Ensures splits occur only when the number of samples in a node meets or exceeds this threshold. Common values like 2, 5, and 10 are included.
   - `min_samples_leaf`: Specifies the minimum number of samples required in a leaf node to prevent overfitting.
   - `criterion`: Tests two split criteria: "gini" (Gini impurity) and "entropy" (information gain).
   - `random_state`: Ensures reproducibility by fixing the random seed.
3. **Initializing the Decision Tree Classifier**:
   A Decision Tree Classifier is created without predefined hyperparameters. This model will be optimized during the grid search.

4. **Performing Grid Search**:
   The `GridSearchCV` object is used to exhaustively search through the hyperparameter combinations:

   - Cross-validation (`cv=5`) divides the training data into 5 folds to evaluate each combination.
   - The `f1_weighted` scoring metric is used, making it ideal for imbalanced datasets as it accounts for both precision and recall, weighted by class support.
   - Parallel processing (`n_jobs=-1`) speeds up the computation.
5. **Retrieving the Best Hyperparameters**:
   After fitting the model, the combination of hyperparameters yielding the best `f1_weighted` score is displayed.

6. **Evaluating the Best Model**:
   The best model is tested on unseen data (test set) to assess its performance. Predictions are made using the optimized classifier, and the F1 score is calculated for the test set.

Highlights:

- **Hyperparameter Optimization**: Grid search systematically tests multiple combinations of hyperparameters to find the best-performing configuration.
- **Weighted F1 Score**: The chosen scoring metric balances precision and recall for each class, addressing the challenges of imbalanced datasets.

- **Cross-Validation**: Ensures that the hyperparameter tuning process generalizes well to unseen data.

This approach provides a structured way to tune and evaluate a Decision Tree Classifier, making it robust and reliable for real-world datasets. Adjusting the parameter ranges in the grid can further tailor the process to your specific problem.

```
=================================================================
```

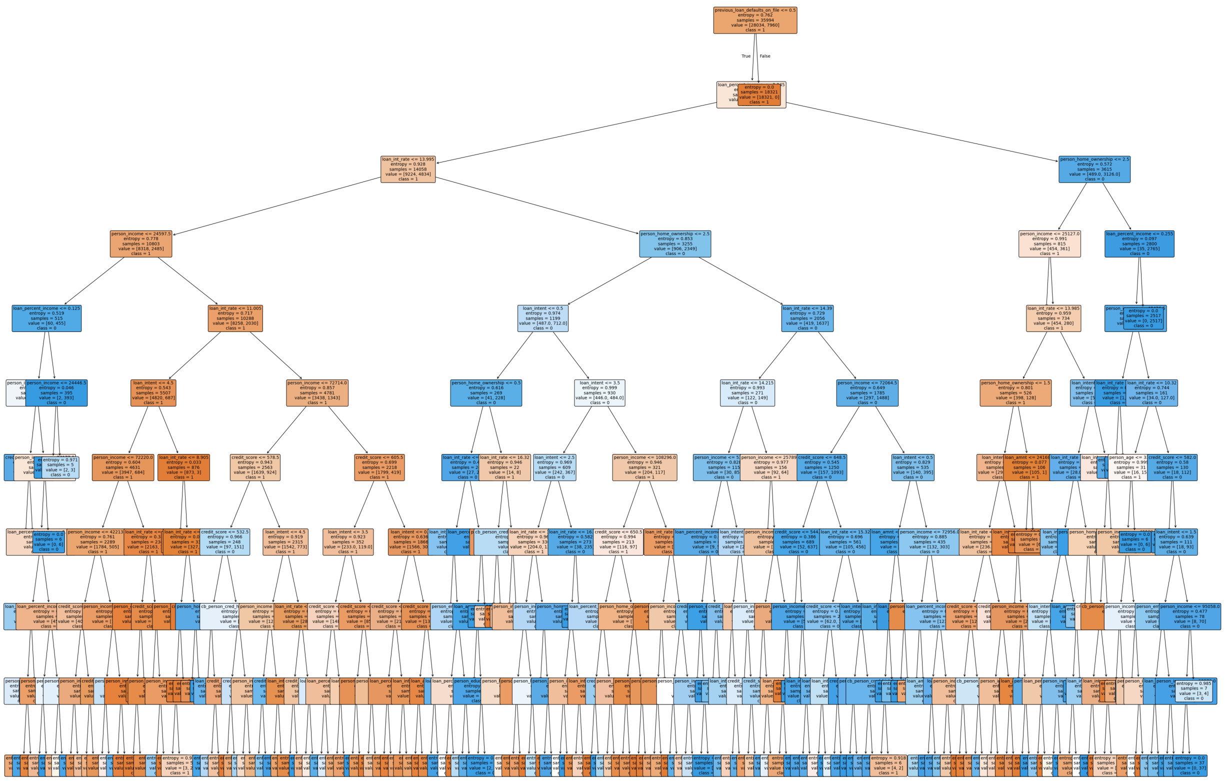
Now we will Apply the best-performing Decision Tree model to the testing set for predictions depending on the output of Grid Search.

In [ ]:
```
model = DecisionTreeClassifier(
    max_depth=10,
    min_samples_split=10,
    min_samples_leaf=4,
    criterion='entropy',
    random_state=0
)


# Train the model
model.fit(x_train, y_train)

# Predict on test data
y_pred = model.predict(x_test)

# Evaluate model performance in each iteration
# Calculate F1-score
f1 = f1_score(y_test, y_pred, average='weighted')
print(f"F1-score: {f1:.3f}")

# Calculate Precision
precision = precision_score(y_test, y_pred)
print(f"Precision: {precision:.3f}")

# Calculate Recall
recall = recall_score(y_test, y_pred)
print(f"Recall: {recall:.3f}")
```

```
F1-score: 0.915
Precision: 0.906
Recall: 0.714
```

In [ ]:
```
# Feature names
feature_names = x.columns.tolist()

# Class names
class_names = y.unique().astype(str).tolist()
```

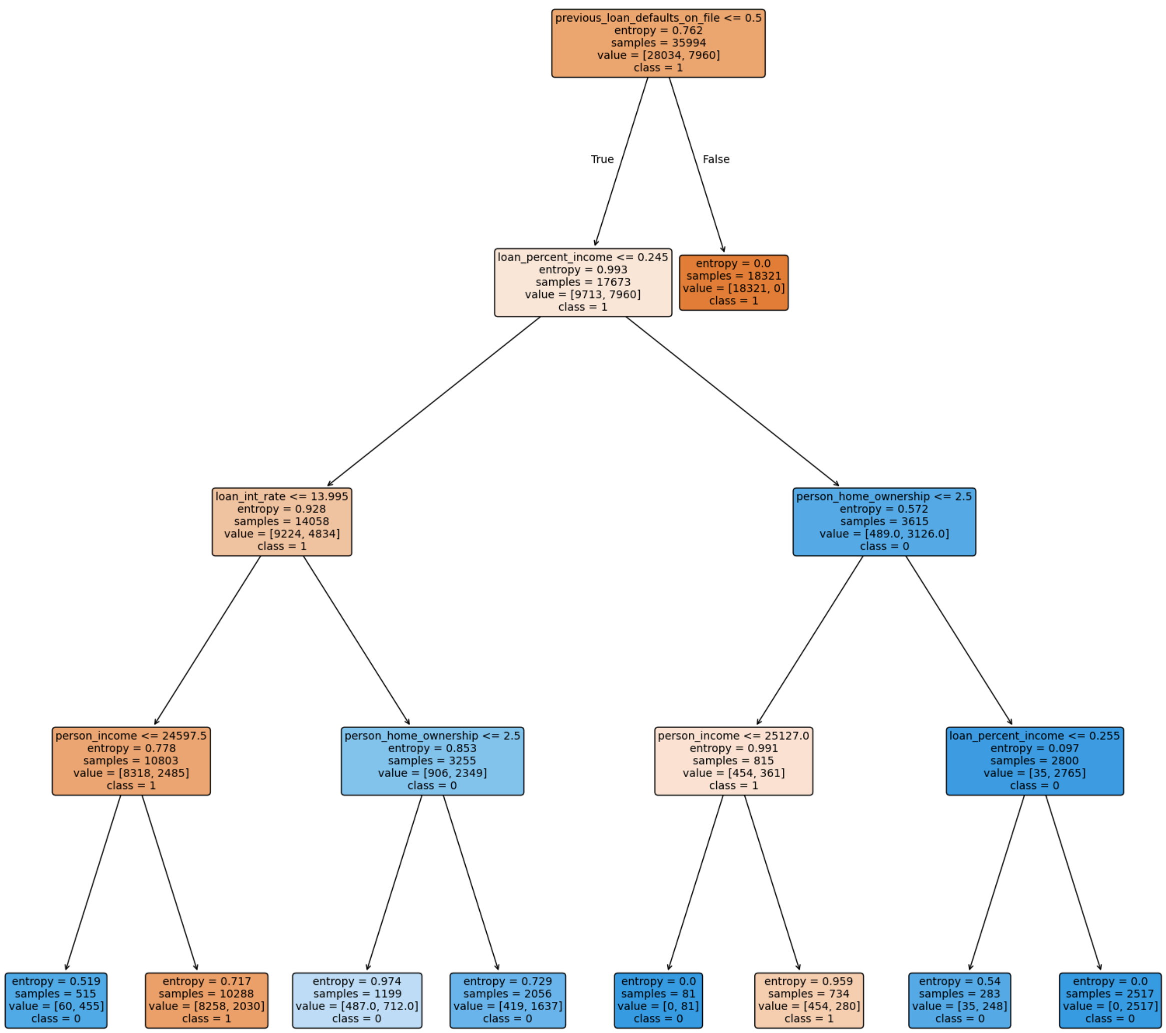
```python
# Visualize the tree
import matplotlib.pyplot as plt
from sklearn import tree

plt.figure(figsize=(50, 35))
tree.plot_tree(model,
        feature_names=feature_names,
        class_names=class_names,
        filled=True,
        rounded=True,
        fontsize=10)
plt.show()
```



for more readable tree we will edit max_depth to became 4

In [ ]:
```python
clftree = DecisionTreeClassifier(
    max_depth=4,
    criterion='entropy',
    random_state=0
)



# Feature names
feature_names = x.columns.tolist()

# Class names
class_names = y.unique().astype(str).tolist()

# Fit Decision Tree
clftree.fit(x_train, y_train)
```

```python
# Visualize the tree
import matplotlib.pyplot as plt
from sklearn import tree

plt.figure(figsize=(20, 20))
tree.plot_tree(clftree,
        feature_names=feature_names,
        class_names=class_names,
        filled=True,
        rounded=True,
        fontsize=10)
plt.show()
```



Insights

1. **Root Node Analysis (Top Node)**

- The **root node** (`previous_loan_defaults_on_file <= 0.5`) is the most important feature, as it is used for the first split.
- **Interpretation**:
  - If `previous_loan_defaults_on_file` is less than or equal to 0.5, it leads to the left child node (class distribution is mixed).
  - If greater, it leads to the right child node where all samples belong to **class = 1**.
- **Insight**: A **history of defaults** is a strong indicator of loan status. If someone has previous defaults, they are highly likely to default again.

2. **Left Branch Analysis (Loan Percent Income)**

   - For individuals **without previous defaults**, the next split occurs at `loan_percent_income <= 0.245`.
   - **Insights**:
     - Lower loan-to-income ratios (**<= 0.245**) are generally associated with better outcomes (non-default).
     - Higher loan-to-income ratios lead to further splits where factors such as **loan interest rate** and **income** determine the outcomes.

3. **Loan Interest Rate and Income Splits**

   - For individuals with high `loan_percent_income` (> 0.245), the decision tree further splits on **loan interest rate** (`loan_int_rate <= 13.985`).
     - Lower interest rates tend to correlate with better loan outcomes (class = 0).
     - Higher interest rates combined with lower incomes increase the likelihood of default.
   - **Key Insight**: High loan-to-income ratios, high interest rates, and low income levels create a **high-risk scenario** for default.

4. **Right Branch Analysis (Previous Defaults Exists)**

   - When `previous_loan_defaults_on_file > 0.5`, the right branch directly predicts **class = 1** (default).
   - **Insight**: A history of prior defaults is a highly **dominant factor** in predicting future loan defaults. No further splits are necessary because the outcome is clear.

5. **Blue Nodes - Class = 0 (Non-default Cases)**

   - Nodes that are blue represent regions where individuals are more likely to **not default** (class = 0).
   - Key features contributing to non-default cases include:
     - Lower `loan_percent_income`
     - Higher `person_income`
     - Lower loan interest rates
     - Stable **home ownership status**
   - **Insight**: Financial stability factors, such as higher income and lower loan burdens, play a critical role in ensuring loan repayment success.

6. **Entropy and Sample Sizes**

- **Entropy**: Measures uncertainty in a node (lower entropy = purer splits). Nodes with `entropy = 0` indicate perfect classification.
  - For instance, at the right branch where `previous_loan_defaults_on_file > 0.5`, the entropy is **0.0** because all samples belong to **class = 1**.
- **Sample Sizes**: Nodes with larger sample sizes provide stronger predictions and generalize better.

Key Insights from the Tree:

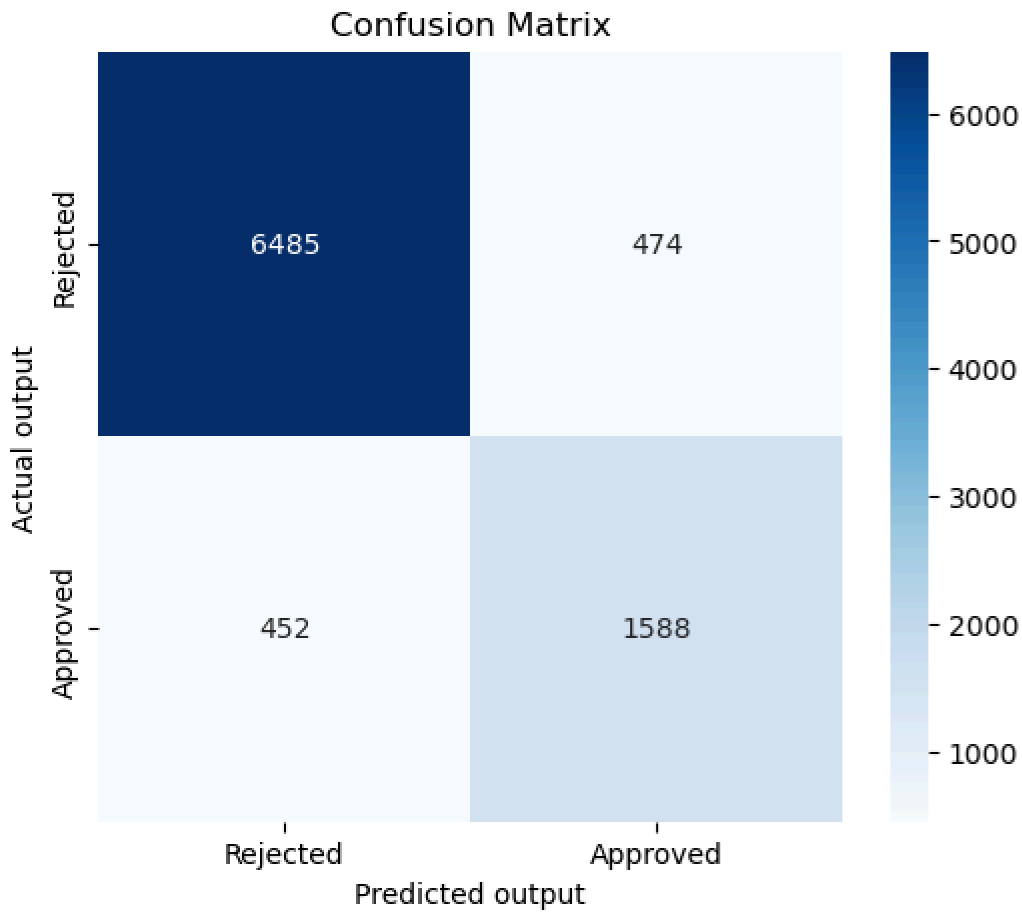
1. **Previous Loan Defaults**: A history of defaults is the most influential factor in predicting loan outcomes.
2. **Loan Percent Income**: Individuals allocating a large percentage of their income to loans are at higher risk of default.
3. **Interest Rates and Income**: High interest rates combined with low income exacerbate the risk of loan default.
4. **Entropy and Class Purity**: Nodes with low entropy provide strong predictions with clear class distributions.

By analyzing the splits and feature thresholds, we uncover patterns in the data that explain why certain individuals default on loans while others do not. These insights are valuable for making informed decisions regarding **loan approvals**, **risk management**, and **customer profiling**.

the final Decision Tree to implement

In [ ]:
```
# Evaluate model performance in each iteration
# Calculate F1-score
f1 = f1_score(y_test, y_pred, average='weighted')
print(f"F1-score: {f1:.3f}")

# Calculate Precision
precision = precision_score(y_test, y_pred)
print(f"Precision: {precision:.3f}")

# Calculate Recall
recall = recall_score(y_test, y_pred)
print(f"Recall: {recall:.3f}")

# Plot the confusion matrix
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["Rejected", "Approv
plt.xlabel("Predicted output")
plt.ylabel("Actual output")
plt.title("Confusion Matrix")
plt.show()
```

```
F1-score: 0.915
Precision: 0.906
Recall: 0.714
```



4) SVM

In [ ]:
```
from sklearn import svm
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.model_selection import cross_val_score,cross_validate
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
```

In [ ]:
```
df = pd.read_csv("loan_data.csv")
df.head()
```

	person_age	person_gender	person_education	person_income	person_emp_exp	person_h
0	22.0	female	Master	71948.0	0	
1	21.0	female	High School	12282.0	0	
2	25.0	female	High School	12438.0	3	
3	23.0	female	Bachelor	79753.0	0	
4	24.0	male	Master	66135.0	1	

In [ ]:
```
df['person_gender'] =(df['person_gender']=='male').astype(int)
df['previous_loan_defaults_on_file'] =(df['previous_loan_defaults_on_file']=='Yes')
df['person_education'] = LabelEncoder().fit_transform(df['person_education'])
df['person_home_ownership'] = LabelEncoder().fit_transform(df['person_home_ownershi
df['loan_intent'] = LabelEncoder().fit_transform(df['loan_intent'])
```

In [ ]:
```
df.head()
```

	person_age	person_gender	person_education	person_income	person_emp_exp	person_h
0	22.0	0	4	71948.0	0	
1	21.0	0	3	12282.0	0	
2	25.0	0	3	12438.0	3	
3	23.0	0	1	79753.0	0	
4	24.0	1	4	66135.0	1	

In [ ]:
```
len(df)
```

45000

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

In [ ]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.35, random_
```

In [ ]:
```
from sklearn.model_selection import GridSearchCV
```

In [ ]:
```
parameters = {'kernel':[ 'rbf'], 'C':[1,2, 5] ,'degree':[3,4,5],'cache_size':[200,2
svc = svm.SVC()
clf = GridSearchCV(svc, parameters,cv=2 ,refit='accuracy',verbose=10,scoring = ['ac
clf.fit(X_train, y_train)
```

```
Fitting 2 folds for each of 27 candidates, totalling 54 fits
[CV 1/2; 1/27] START C=1, cache_size=200, degree=3, kernel=rbf.................
[CV 1/2; 1/27] END C=1, cache_size=200, degree=3, kernel=rbf; accuracy: (train=0.77
8, test=0.778) f1: (train=0.015, test=0.016) total time=  15.3s
[CV 2/2; 1/27] START C=1, cache_size=200, degree=3, kernel=rbf.................
[CV 2/2; 1/27] END C=1, cache_size=200, degree=3, kernel=rbf; accuracy: (train=0.80
1, test=0.802) f1: (train=0.239, test=0.251) total time=  15.3s
[CV 1/2; 2/27] START C=1, cache_size=200, degree=4, kernel=rbf.................
[CV 1/2; 2/27] END C=1, cache_size=200, degree=4, kernel=rbf; accuracy: (train=0.77
8, test=0.778) f1: (train=0.015, test=0.016) total time=  15.1s
[CV 2/2; 2/27] START C=1, cache_size=200, degree=4, kernel=rbf.................
[CV 2/2; 2/27] END C=1, cache_size=200, degree=4, kernel=rbf; accuracy: (train=0.80
1, test=0.802) f1: (train=0.239, test=0.251) total time=  15.3s
[CV 1/2; 3/27] START C=1, cache_size=200, degree=5, kernel=rbf.................
[CV 1/2; 3/27] END C=1, cache_size=200, degree=5, kernel=rbf; accuracy: (train=0.77
8, test=0.778) f1: (train=0.015, test=0.016) total time=  15.2s
[CV 2/2; 3/27] START C=1, cache_size=200, degree=5, kernel=rbf.................
[CV 2/2; 3/27] END C=1, cache_size=200, degree=5, kernel=rbf; accuracy: (train=0.80
1, test=0.802) f1: (train=0.239, test=0.251) total time=  15.3s
[CV 1/2; 4/27] START C=1, cache_size=2000, degree=3, kernel=rbf................
[CV 1/2; 4/27] END C=1, cache_size=2000, degree=3, kernel=rbf; accuracy: (train=0.77
8, test=0.778) f1: (train=0.015, test=0.016) total time=  15.2s
[CV 2/2; 4/27] START C=1, cache_size=2000, degree=3, kernel=rbf................
[CV 2/2; 4/27] END C=1, cache_size=2000, degree=3, kernel=rbf; accuracy: (train=0.80
1, test=0.802) f1: (train=0.239, test=0.251) total time=  15.6s
[CV 1/2; 5/27] START C=1, cache_size=2000, degree=4, kernel=rbf................
[CV 1/2; 5/27] END C=1, cache_size=2000, degree=4, kernel=rbf; accuracy: (train=0.77
8, test=0.778) f1: (train=0.015, test=0.016) total time=  15.3s
[CV 2/2; 5/27] START C=1, cache_size=2000, degree=4, kernel=rbf................
[CV 2/2; 5/27] END C=1, cache_size=2000, degree=4, kernel=rbf; accuracy: (train=0.80
1, test=0.802) f1: (train=0.239, test=0.251) total time=  15.5s
[CV 1/2; 6/27] START C=1, cache_size=2000, degree=5, kernel=rbf................
[CV 1/2; 6/27] END C=1, cache_size=2000, degree=5, kernel=rbf; accuracy: (train=0.77
8, test=0.778) f1: (train=0.015, test=0.016) total time=  15.2s
[CV 2/2; 6/27] START C=1, cache_size=2000, degree=5, kernel=rbf................
[CV 2/2; 6/27] END C=1, cache_size=2000, degree=5, kernel=rbf; accuracy: (train=0.80
1, test=0.802) f1: (train=0.239, test=0.251) total time=  15.7s
[CV 1/2; 7/27] START C=1, cache_size=6000, degree=3, kernel=rbf................
[CV 1/2; 7/27] END C=1, cache_size=6000, degree=3, kernel=rbf; accuracy: (train=0.77
8, test=0.778) f1: (train=0.015, test=0.016) total time=  15.3s
[CV 2/2; 7/27] START C=1, cache_size=6000, degree=3, kernel=rbf................
[CV 2/2; 7/27] END C=1, cache_size=6000, degree=3, kernel=rbf; accuracy: (train=0.80
1, test=0.802) f1: (train=0.239, test=0.251) total time=  15.6s
[CV 1/2; 8/27] START C=1, cache_size=6000, degree=4, kernel=rbf................
[CV 1/2; 8/27] END C=1, cache_size=6000, degree=4, kernel=rbf; accuracy: (train=0.77
8, test=0.778) f1: (train=0.015, test=0.016) total time=  15.3s
[CV 2/2; 8/27] START C=1, cache_size=6000, degree=4, kernel=rbf................
[CV 2/2; 8/27] END C=1, cache_size=6000, degree=4, kernel=rbf; accuracy: (train=0.80
1, test=0.802) f1: (train=0.239, test=0.251) total time=  15.6s
[CV 1/2; 9/27] START C=1, cache_size=6000, degree=5, kernel=rbf................
[CV 1/2; 9/27] END C=1, cache_size=6000, degree=5, kernel=rbf; accuracy: (train=0.77
8, test=0.778) f1: (train=0.015, test=0.016) total time=  15.4s
[CV 2/2; 9/27] START C=1, cache_size=6000, degree=5, kernel=rbf................
[CV 2/2; 9/27] END C=1, cache_size=6000, degree=5, kernel=rbf; accuracy: (train=0.80
1, test=0.802) f1: (train=0.239, test=0.251) total time=  15.6s
[CV 1/2; 10/27] START C=2, cache_size=200, degree=3, kernel=rbf................
```

```
[CV 1/2; 10/27] END C=2, cache_size=200, degree=3, kernel=rbf; accuracy: (train=0.80
2, test=0.800) f1: (train=0.243, test=0.223) total time=  15.5s
[CV 2/2; 10/27] START C=2, cache_size=200, degree=3, kernel=rbf.................
[CV 2/2; 10/27] END C=2, cache_size=200, degree=3, kernel=rbf; accuracy: (train=0.80
8, test=0.811) f1: (train=0.308, test=0.324) total time=  16.5s
[CV 1/2; 11/27] START C=2, cache_size=200, degree=4, kernel=rbf.................
[CV 1/2; 11/27] END C=2, cache_size=200, degree=4, kernel=rbf; accuracy: (train=0.80
2, test=0.800) f1: (train=0.243, test=0.223) total time=  15.5s
[CV 2/2; 11/27] START C=2, cache_size=200, degree=4, kernel=rbf.................
[CV 2/2; 11/27] END C=2, cache_size=200, degree=4, kernel=rbf; accuracy: (train=0.80
8, test=0.811) f1: (train=0.308, test=0.324) total time=  15.0s
[CV 1/2; 12/27] START C=2, cache_size=200, degree=5, kernel=rbf.................
[CV 1/2; 12/27] END C=2, cache_size=200, degree=5, kernel=rbf; accuracy: (train=0.80
2, test=0.800) f1: (train=0.243, test=0.223) total time=  15.4s
[CV 2/2; 12/27] START C=2, cache_size=200, degree=5, kernel=rbf.................
[CV 2/2; 12/27] END C=2, cache_size=200, degree=5, kernel=rbf; accuracy: (train=0.80
8, test=0.811) f1: (train=0.308, test=0.324) total time=  15.1s
[CV 1/2; 13/27] START C=2, cache_size=2000, degree=3, kernel=rbf................
[CV 1/2; 13/27] END C=2, cache_size=2000, degree=3, kernel=rbf; accuracy: (train=0.8
02, test=0.800) f1: (train=0.243, test=0.223) total time=  15.6s
[CV 2/2; 13/27] START C=2, cache_size=2000, degree=3, kernel=rbf................
[CV 2/2; 13/27] END C=2, cache_size=2000, degree=3, kernel=rbf; accuracy: (train=0.8
08, test=0.811) f1: (train=0.308, test=0.324) total time=  15.2s
[CV 1/2; 14/27] START C=2, cache_size=2000, degree=4, kernel=rbf................
[CV 1/2; 14/27] END C=2, cache_size=2000, degree=4, kernel=rbf; accuracy: (train=0.8
02, test=0.800) f1: (train=0.243, test=0.223) total time=  15.5s
[CV 2/2; 14/27] START C=2, cache_size=2000, degree=4, kernel=rbf................
[CV 2/2; 14/27] END C=2, cache_size=2000, degree=4, kernel=rbf; accuracy: (train=0.8
08, test=0.811) f1: (train=0.308, test=0.324) total time=  15.6s
[CV 1/2; 15/27] START C=2, cache_size=2000, degree=5, kernel=rbf................
[CV 1/2; 15/27] END C=2, cache_size=2000, degree=5, kernel=rbf; accuracy: (train=0.8
02, test=0.800) f1: (train=0.243, test=0.223) total time=  15.6s
[CV 2/2; 15/27] START C=2, cache_size=2000, degree=5, kernel=rbf................
[CV 2/2; 15/27] END C=2, cache_size=2000, degree=5, kernel=rbf; accuracy: (train=0.8
08, test=0.811) f1: (train=0.308, test=0.324) total time=  15.3s
[CV 1/2; 16/27] START C=2, cache_size=6000, degree=3, kernel=rbf................
[CV 1/2; 16/27] END C=2, cache_size=6000, degree=3, kernel=rbf; accuracy: (train=0.8
02, test=0.800) f1: (train=0.243, test=0.223) total time=  15.7s
[CV 2/2; 16/27] START C=2, cache_size=6000, degree=3, kernel=rbf................
[CV 2/2; 16/27] END C=2, cache_size=6000, degree=3, kernel=rbf; accuracy: (train=0.8
08, test=0.811) f1: (train=0.308, test=0.324) total time=  15.3s
[CV 1/2; 17/27] START C=2, cache_size=6000, degree=4, kernel=rbf................
[CV 1/2; 17/27] END C=2, cache_size=6000, degree=4, kernel=rbf; accuracy: (train=0.8
02, test=0.800) f1: (train=0.243, test=0.223) total time=  15.8s
[CV 2/2; 17/27] START C=2, cache_size=6000, degree=4, kernel=rbf................
[CV 2/2; 17/27] END C=2, cache_size=6000, degree=4, kernel=rbf; accuracy: (train=0.8
08, test=0.811) f1: (train=0.308, test=0.324) total time=  15.3s
[CV 1/2; 18/27] START C=2, cache_size=6000, degree=5, kernel=rbf................
[CV 1/2; 18/27] END C=2, cache_size=6000, degree=5, kernel=rbf; accuracy: (train=0.8
02, test=0.800) f1: (train=0.243, test=0.223) total time=  15.7s
[CV 2/2; 18/27] START C=2, cache_size=6000, degree=5, kernel=rbf................
[CV 2/2; 18/27] END C=2, cache_size=6000, degree=5, kernel=rbf; accuracy: (train=0.8
08, test=0.811) f1: (train=0.308, test=0.324) total time=  15.4s
[CV 1/2; 19/27] START C=5, cache_size=200, degree=3, kernel=rbf.................
```

```
[CV 1/2; 19/27] END C=5, cache_size=200, degree=3, kernel=rbf; accuracy: (train=0.81
0, test=0.807) f1: (train=0.318, test=0.301) total time=  15.3s
[CV 2/2; 19/27] START C=5, cache_size=200, degree=3, kernel=rbf.................
[CV 2/2; 19/27] END C=5, cache_size=200, degree=3, kernel=rbf; accuracy: (train=0.81
2, test=0.814) f1: (train=0.357, test=0.365) total time=  15.2s
[CV 1/2; 20/27] START C=5, cache_size=200, degree=4, kernel=rbf.................
[CV 1/2; 20/27] END C=5, cache_size=200, degree=4, kernel=rbf; accuracy: (train=0.81
0, test=0.807) f1: (train=0.318, test=0.301) total time=  15.2s
[CV 2/2; 20/27] START C=5, cache_size=200, degree=4, kernel=rbf.................
[CV 2/2; 20/27] END C=5, cache_size=200, degree=4, kernel=rbf; accuracy: (train=0.81
2, test=0.814) f1: (train=0.357, test=0.365) total time=  15.2s
[CV 1/2; 21/27] START C=5, cache_size=200, degree=5, kernel=rbf.................
[CV 1/2; 21/27] END C=5, cache_size=200, degree=5, kernel=rbf; accuracy: (train=0.81
0, test=0.807) f1: (train=0.318, test=0.301) total time=  15.3s
[CV 2/2; 21/27] START C=5, cache_size=200, degree=5, kernel=rbf.................
[CV 2/2; 21/27] END C=5, cache_size=200, degree=5, kernel=rbf; accuracy: (train=0.81
2, test=0.814) f1: (train=0.357, test=0.365) total time=  15.1s
[CV 1/2; 22/27] START C=5, cache_size=2000, degree=3, kernel=rbf................
[CV 1/2; 22/27] END C=5, cache_size=2000, degree=3, kernel=rbf; accuracy: (train=0.8
10, test=0.807) f1: (train=0.318, test=0.301) total time=  15.6s
[CV 2/2; 22/27] START C=5, cache_size=2000, degree=3, kernel=rbf................
[CV 2/2; 22/27] END C=5, cache_size=2000, degree=3, kernel=rbf; accuracy: (train=0.8
12, test=0.814) f1: (train=0.357, test=0.365) total time=  15.9s
[CV 1/2; 23/27] START C=5, cache_size=2000, degree=4, kernel=rbf................
[CV 1/2; 23/27] END C=5, cache_size=2000, degree=4, kernel=rbf; accuracy: (train=0.8
10, test=0.807) f1: (train=0.318, test=0.301) total time=  16.0s
[CV 2/2; 23/27] START C=5, cache_size=2000, degree=4, kernel=rbf................
[CV 2/2; 23/27] END C=5, cache_size=2000, degree=4, kernel=rbf; accuracy: (train=0.8
12, test=0.814) f1: (train=0.357, test=0.365) total time=  15.7s
[CV 1/2; 24/27] START C=5, cache_size=2000, degree=5, kernel=rbf................
[CV 1/2; 24/27] END C=5, cache_size=2000, degree=5, kernel=rbf; accuracy: (train=0.8
10, test=0.807) f1: (train=0.318, test=0.301) total time=  15.9s
[CV 2/2; 24/27] START C=5, cache_size=2000, degree=5, kernel=rbf................
[CV 2/2; 24/27] END C=5, cache_size=2000, degree=5, kernel=rbf; accuracy: (train=0.8
12, test=0.814) f1: (train=0.357, test=0.365) total time=  15.8s
[CV 1/2; 25/27] START C=5, cache_size=6000, degree=3, kernel=rbf................
[CV 1/2; 25/27] END C=5, cache_size=6000, degree=3, kernel=rbf; accuracy: (train=0.8
10, test=0.807) f1: (train=0.318, test=0.301) total time=  16.5s
[CV 2/2; 25/27] START C=5, cache_size=6000, degree=3, kernel=rbf................
[CV 2/2; 25/27] END C=5, cache_size=6000, degree=3, kernel=rbf; accuracy: (train=0.8
12, test=0.814) f1: (train=0.357, test=0.365) total time=  16.3s
[CV 1/2; 26/27] START C=5, cache_size=6000, degree=4, kernel=rbf................
[CV 1/2; 26/27] END C=5, cache_size=6000, degree=4, kernel=rbf; accuracy: (train=0.8
10, test=0.807) f1: (train=0.318, test=0.301) total time=  16.6s
[CV 2/2; 26/27] START C=5, cache_size=6000, degree=4, kernel=rbf................
[CV 2/2; 26/27] END C=5, cache_size=6000, degree=4, kernel=rbf; accuracy: (train=0.8
12, test=0.814) f1: (train=0.357, test=0.365) total time=  17.3s
[CV 1/2; 27/27] START C=5, cache_size=6000, degree=5, kernel=rbf................
[CV 1/2; 27/27] END C=5, cache_size=6000, degree=5, kernel=rbf; accuracy: (train=0.8
10, test=0.807) f1: (train=0.318, test=0.301) total time=  17.2s
[CV 2/2; 27/27] START C=5, cache_size=6000, degree=5, kernel=rbf................
[CV 2/2; 27/27] END C=5, cache_size=6000, degree=5, kernel=rbf; accuracy: (train=0.8
12, test=0.814) f1: (train=0.357, test=0.365) total time=  17.3s
```

```
GridSearchCV(cv=2, estimator=SVC(),
             param_grid={'C': [1, 2, 5], 'cache_size': [200, 2000, 6000],
                         'degree': [3, 4, 5], 'kernel': ['rbf']},
             refit='accuracy', return_train_score=True,
             scoring=['accuracy', 'f1'], verbose=10)
```

In [ ]:
```
clf.best_score_
```

```
0.8102222222222222
```

In [ ]:
```
clf.best_params_
```

```
{'C': 5, 'cache_size': 200, 'degree': 3, 'kernel': 'rbf'}
```

In [ ]:
```
svc2=svm.SVC(C=5,cache_size=100000,degree=3,kernel='rbf')
```

In [ ]:
```
svc2.fit(X_train,y_train)
```

```
SVC(C=5, cache_size=100000)
```

In [ ]:
```
y_pred=svc2.predict(X_test)
```

In [ ]:
```
cm = confusion_matrix(y_test, y_pred)
print(cm)
accuracy_score(y_test, y_pred)
```

```
[[12010   288]
 [ 2655   797]]
0.8131428571428572
```

- from what we see it is very hard to train SVM mode on the data cause how computionally entinsive to run it on kernals like ('poly', 'linear') so we trid our best to get the best f1 score and accuracy out of the 'rbf' model using grid search alogrthim

we can conclude that svm isn't suitable for our task

Comparative Analysis

ML Algorithm comparison Table

In [4]:
```
metrics = pd.DataFrame({'Algorithm': ['KNN', 'Decision Tree', 'SVM', 'Naive Bayes'],
                        'Accuracy': ['0.904', '0.918', '0.813','0.730'],
                        'Precision': ['0.896', '0.906', '0.735', '0.452'],
                        'Recall': ['0.899', '0.714', '0.231', '0.997'],
                        'F1-Score': ['0.901', '0.915', '0.351','0.753']})

metrics
```

Out[4]:

	Algorithm	Accuracy	Precision	Recall	F1-Score
0	KNN	0.904	0.896	0.899	0.901
1	Decision Tree	0.918	0.906	0.714	0.915
2	SVM	0.813	0.735	0.231	0.351
3	Naive Bayes	0.730	0.452	0.997	0.753

Model Performance Insights:

- For imbalanced data, accuracy may not be the best metric because it could be skewed towards the majority class
  - So, we will use F1-score instead as it combines the benefits of precision and recall into one metric
  - So Accuracy values is misleading we will not consider it
- In addition to F1-score, we are also interested in Precision and Recall:
  - Precision is important when we want to avoid FP, i.e., incorrectly classifying loans as approved when they should be rejected.
  - Recall is important when we want to minimize false negatives, i.e., making sure that loans that should be approved are not missed.
  - `But precision in the loan context is important than recall`

1. **K-Nearest Neighbors (KNN)**

- **F1 Score:** 0.9008
- **Accuracy:** 0.9041
- **Precision:** 0.8961
- **Recall:** 0.8991

Key Insights:

- Balanced performance across all metrics.
- Handles imbalance effectively but could improve recall for critical cases.
- If the dataset has balanced classes, KNN appears to be a reliable choice. However, if the dataset has class imbalance, this could explain why precision is marginally lower than recall
- KNN is a simple and adaptable algorithm that performs well on imbalanced datasets by achieving a strong recall and balanced F1-score, making it effective in identifying minority class instances.

---

2. **Decision Tree**

- **F1 Score:** 0.915
- **Accuracy:** 0.918
- **Precision:** 0.906
- **Recall:** 0.714

Key Insights:

- **High Precision**: Indicates strong performance in avoiding FP, favoring correct predictions over identifying all positives However, high Precision sometimes comes at the cost of lower Recall.
- **F1-Score**: A high F1-Score demonstrates a good trade-off between Precision and Recall. Focusing in both FP & FN
- The significant gap between Precision (0.906) and Recall (0.714) suggests the model may be prioritizing avoiding FP over identifying all positives.

---

3. **Support Vector Machine (SVM)**

- **F1 Score:** 0.3513
- **Accuracy:** 0.8131
- **Precision:** 0.7346
- **Recall:** 0.2309

Key Insights:

- Low recall highlights difficulty in identifying positive cases.
- Sensitive to class imbalance
- Could benefit from kernel adjustments or dimensionality reduction.
- Computationally expensive to run other karnels adjustments
- the deficiency of recall score lead to a bad overall f1-score

---

4. **Naive Bayes**

- **F1 Score:** 0.7529
- **Accuracy:** 0.7303
- **Precision:** 0.4518
- **Recall:** 0.9969

Key Insights:

- Excels in recall, effectively identifying nearly all positive cases.
- Very low precision, leading to many FP.
- Suitable for use cases where missing positives is highly undesirable.

Analysis

- **KNN** and the **Decision Tree** both offer high Precision, with Decision Tree slightly outperforming KNN in F1-Score.

- For applications where **identifying all positives (Recall)** is critical:

  - **Naive Bayes** has the highest Recall (0.9969), but it suffers from very low Precision (0.4518), leading to a high number of false positives.
  - This makes it less suitable unless the cost of missing positives significantly outweighs the cost of false positives.
- **Algorithm Adaptability**:

  - **Decision Tree** is less computationally expensive than **SVM** and more interpretable, making it a practical and adaptable choice for diverse datasets.

Final Recommendation

- The **Decision Tree** is the best choice overall due to its strong F1-Score, adaptability, and balanced approach to managing false positives and false negatives.

- If you are Focusing on Reducing FN and you don't care about FP, consider **Naive Bayes** for its exceptionally high Recall.