In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.impute import SimpleImputer
import seaborn as sns
import numpy as np
import ydata_profiling
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
```

[Upgrade to ydata-sdk]

Improve your data and profiling with ydata-sdk, featuring data quality scoring, redundancy detection, outlier identification, text validation, and synthetic data generation.

In [2]:
```
df = pd.read_csv("Delinquency_prediction_dataset.csv")
df.head(10)
```

Out[2]:

	Customer_ID	Age	Income	Credit_Score	Credit_Utilization	Missed_Payments	Delinqu
0	CUST0001	56	165580.0	398.0	0.390502	3	
1	CUST0002	69	100999.0	493.0	0.312444	6	
2	CUST0003	46	188416.0	500.0	0.359930	0	
3	CUST0004	32	101672.0	413.0	0.371400	3	
4	CUST0005	60	38524.0	487.0	0.234716	2	
5	CUST0006	25	84042.0	700.0	0.650540	6	
6	CUST0007	38	35056.0	354.0	0.390581	3	
7	CUST0008	56	123215.0	415.0	0.532715	5	
8	CUST0009	36	66991.0	405.0	0.413035	5	
9	CUST0010	40	34870.0	679.0	0.361824	4	

In [3]:
```
from ydata_profiling import ProfileReport

# create the report
profile = ProfileReport(df, title="EDA Report")
profile.to_file("eda_report.html")
```

```
Summarize dataset:   0%|          | 0/5 [00:00<?, ?it/s]
```

```
  0%|
| 0/19 [00:00<?, ?it/s]
 26%|█████████████████████▍
| 5/19 [00:00<00:00, 36.86it/s]
 53%|███████████████████████████████████████████▎
| 10/19 [00:00<00:00, 40.94it/s]
100%|███████████████████████████████████████████████████████████████████████████████
███| 19/19 [00:00<00:00, 32.85it/s]
Generate report structure:   0%|          | 0/1 [00:00<?, ?it/s]
Render HTML:   0%|          | 0/1 [00:00<?, ?it/s]
Export report to file:   0%|          | 0/1 [00:00<?, ?it/s]
```

In [4]:
```
profile.to_notebook_iframe()
```

EDA Report

Overview

Brought to you by YData

Overview | Alerts 5 | Reproduction

Dataset statistics

Number of variables	19
Number of observations	500
Missing cells	70
Missing cells (%)	0.7%
Duplicate rows	0
Duplicate rows (%)	0.0%
Total size in memory	74.3 KiB
Average record size in memory	152.3 B

Variable types

Text	1
Numeric	8
Categorical	10

Variables

Income has 39 (7.8%) missing values Missing Loan_Balance has 29 (5.8%) missing values Missing Customer_ID has unique values Unique Missed_Payments has 77 (15.4%) zeros Zeros Account_Tenure has 28 (5.6%) zeros Zeros

In [5]:
```
#changing EMP, employed  labes to Employed
```

```
df['Employment_Status'].replace({
    'employed': 'Employed',
    'EMP': 'Employed'
},inplace=True)
```

In [6]:
```
df['Employment_Status'].value_counts()
```

Out[6]:
```
Employment_Status
Employed         240
Unemployed        93
retired           87
Self-employed     80
Name: count, dtype: int64
```

In [7]:
```
# Number of records (rows, columns)
df.shape
```

Out[7]:
```
(500, 19)
```

In [8]:
```
# Column names and data types
df.dtypes
```

Out[8]:
```
Customer_ID               object
Age                        int64
Income                   float64
Credit_Score             float64
Credit_Utilization       float64
Missed_Payments            int64
Delinquent_Account         int64
Loan_Balance             float64
Debt_to_Income_Ratio     float64
Employment_Status         object
Account_Tenure             int64
Credit_Card_Type          object
Location                  object
Month_1                   object
Month_2                   object
Month_3                   object
Month_4                   object
Month_5                   object
Month_6                   object
dtype: object
```

In [9]:
```
# Check for missing values
df.isnull().sum()
```

Out[9]:
```
Customer_ID              0
Age                      0
Income                  39
Credit_Score             2
Credit_Utilization       0
Missed_Payments          0
Delinquent_Account       0
Loan_Balance            29
Debt_to_Income_Ratio     0
Employment_Status        0
Account_Tenure           0
Credit_Card_Type         0
Location                 0
Month_1                  0
Month_2                  0
Month_3                  0
Month_4                  0
Month_5                  0
Month_6                  0
dtype: int64
```

In [10]:
```
# Percentage of missing values per column
df.isnull().mean() * 100
```

Out[10]:
```
Customer_ID             0.0
Age                     0.0
Income                  7.8
Credit_Score            0.4
Credit_Utilization      0.0
Missed_Payments         0.0
Delinquent_Account      0.0
Loan_Balance            5.8
Debt_to_Income_Ratio    0.0
Employment_Status       0.0
Account_Tenure          0.0
Credit_Card_Type        0.0
Location                0.0
Month_1                 0.0
Month_2                 0.0
Month_3                 0.0
Month_4                 0.0
Month_5                 0.0
Month_6                 0.0
dtype: float64
```

In [11]:
```
# Check for duplicate rows
df.duplicated().sum()
```

Out[11]:
```
0
```

Data imputation income and loan balance missing values replaced with median credit score missing values replaced with mean

In [12]:
```
# Impute 'Income' and 'Loan_Balance' with median
median_imputer = SimpleImputer(strategy="median")
```

```
df[['Income', 'Loan_Balance']] = median_imputer.fit_transform(df[['Income', 'Loan_B

# Impute 'Credit_Score' with mean
mean_imputer = SimpleImputer(strategy="mean")
df[['Credit_Score']] = mean_imputer.fit_transform(df[['Credit_Score']])

# Confirm no missing values remain
print(df.isnull().sum())
```

```
Customer_ID            0
Age                    0
Income                 0
Credit_Score           0
Credit_Utilization     0
Missed_Payments        0
Delinquent_Account     0
Loan_Balance           0
Debt_to_Income_Ratio   0
Employment_Status      0
Account_Tenure         0
Credit_Card_Type       0
Location               0
Month_1                0
Month_2                0
Month_3                0
Month_4                0
Month_5                0
Month_6                0
dtype: int64
```

In [13]:

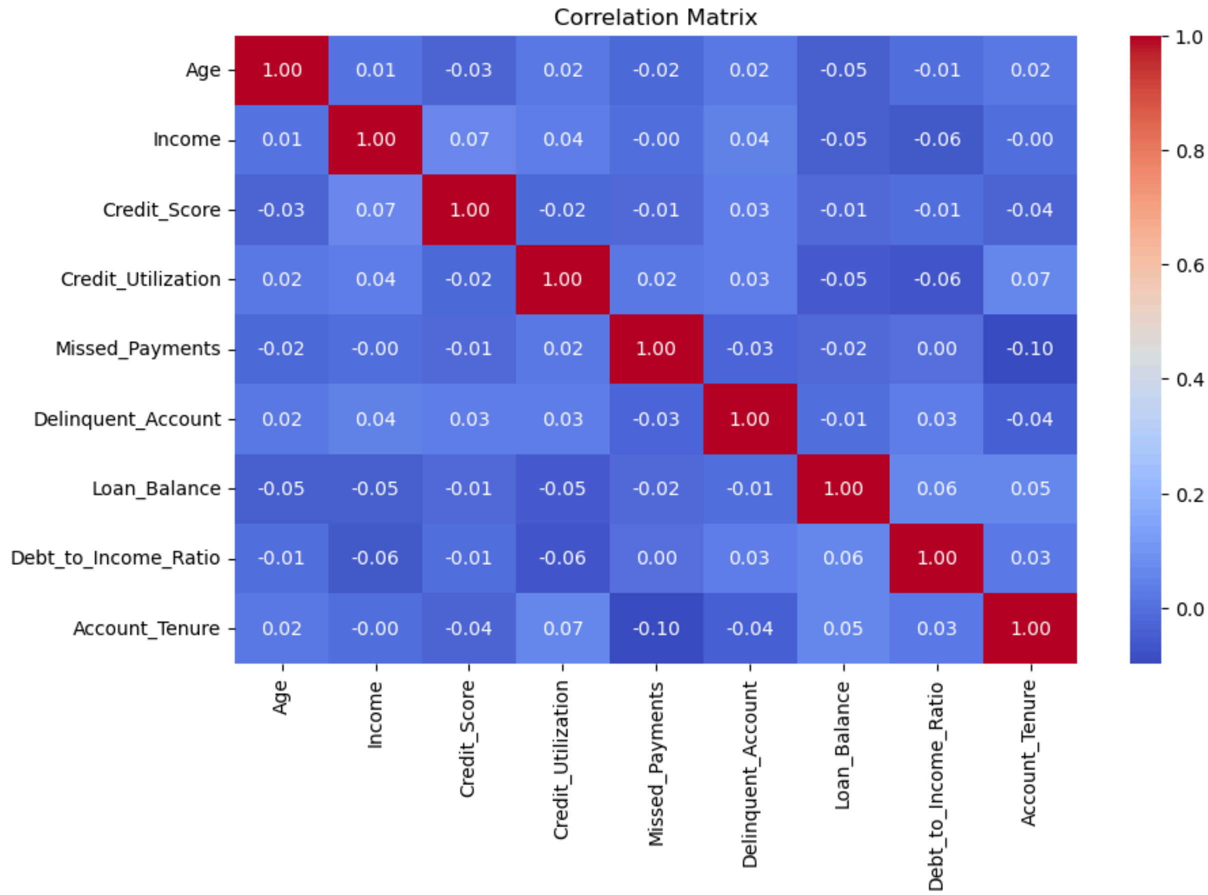
```
#  Keep only numeric columns
numeric_df = df.select_dtypes(include=['int64', 'float64'])

# Correlation matrix
corr = numeric_df.corr()
```

In [14]:

```
%matplotlib inline
```

In [15]:

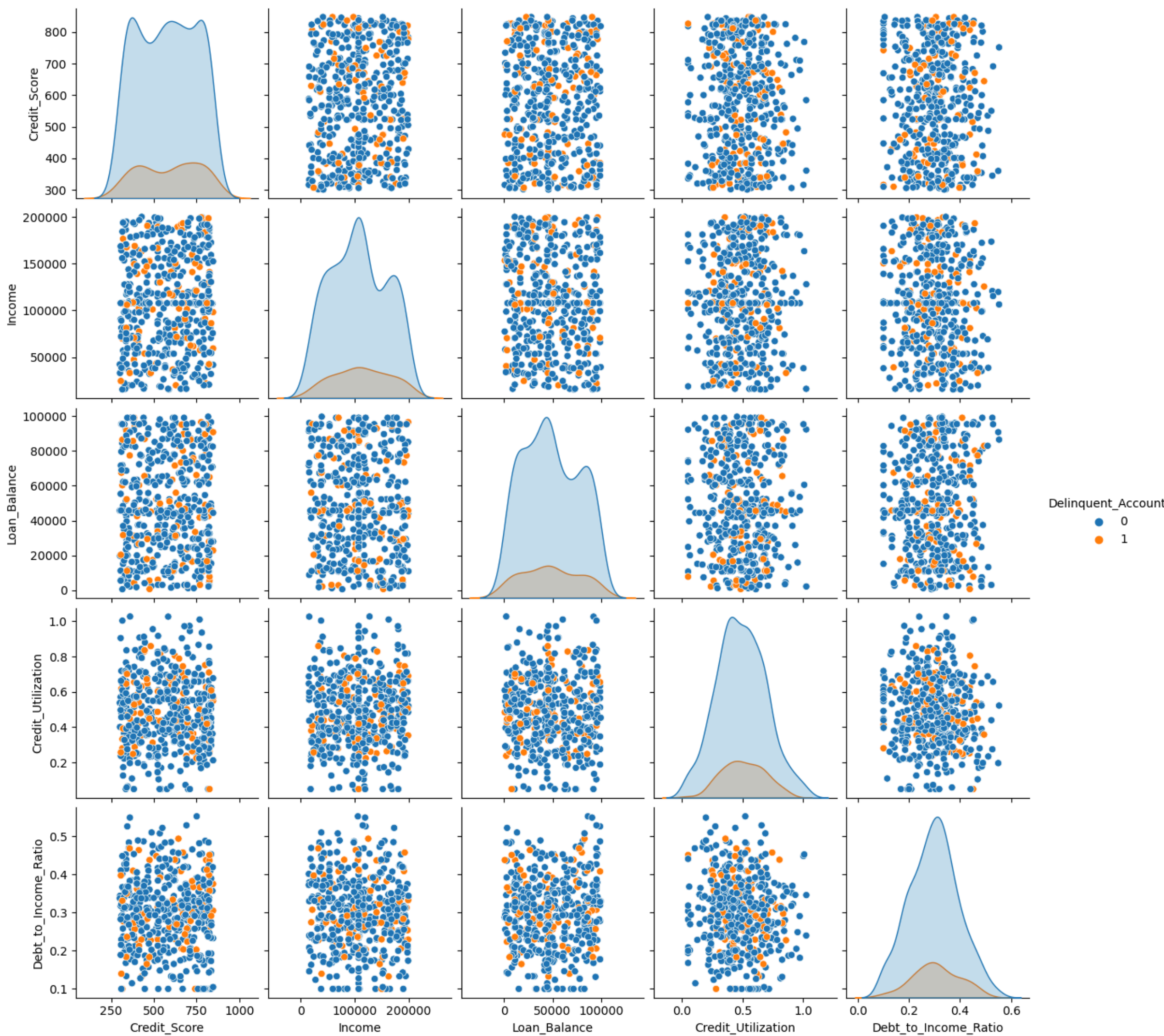
```
# Correlation matrix (numerical features only)
corr = numeric_df.corr()

# Heatmap of correlations
plt.figure(figsize=(10,6))
sns.heatmap(corr, annot=True, fmt=".2f", cmap="coolwarm")
plt.title("Correlation Matrix")
plt.show()
```



In [16]:
```
#correlation of each variable with delinquency
corr_with_target = corr['Delinquent_Account'].sort_values(ascending=False)
print(corr_with_target)
```

```
Delinquent_Account      1.000000
Income                  0.043991
Credit_Score            0.034820
Debt_to_Income_Ratio    0.034386
Credit_Utilization      0.034224
Age                     0.022508
Loan_Balance           -0.005438
Missed_Payments        -0.026478
Account_Tenure         -0.039829
Name: Delinquent_Account, dtype: float64
```

In [17]:
```
# Pairplot for key risk factors
sns.pairplot(df[['Credit_Score', 'Income', 'Loan_Balance',
                 'Credit_Utilization', 'Debt_to_Income_Ratio',
                 'Delinquent_Account']], hue="Delinquent_Account")
plt.show()
```



In [18]:
```
#calculate and display delinquency rates (%) across different categories

categorical_cols = ['Employment_Status', 'Debt_to_Income_Ratio', 'Credit_Score' , '

for col in categorical_cols:
    delinquency_rate = df.groupby(col)['Delinquent_Account'].mean().sort_values(asc
    print(f"\nDelinquency Rate by {col}:\n", delinquency_rate)
```

```
Delinquency Rate by Employment_Status:
 Employment_Status
Unemployed       19.354839
Employed         16.250000
Self-employed    16.250000
retired          11.494253
Name: Delinquent_Account, dtype: float64

Delinquency Rate by Debt_to_Income_Ratio:
 Debt_to_Income_Ratio
0.313142    100.0
0.438178    100.0
0.243359    100.0
0.367570    100.0
0.366179    100.0
            ...
0.269624      0.0
0.269559      0.0
0.269109      0.0
0.268534      0.0
0.552956      0.0
Name: Delinquent_Account, Length: 487, dtype: float64

Delinquency Rate by Credit_Score:
 Credit_Score
378.0    100.0
412.0    100.0
445.0    100.0
805.0    100.0
731.0    100.0
         ...
526.0      0.0
528.0      0.0
534.0      0.0
535.0      0.0
567.0      0.0
Name: Delinquent_Account, Length: 235, dtype: float64

Delinquency Rate by Credit_Utilization:
 Credit_Utilization
0.448492    100.0
0.257505    100.0
0.427107    100.0
0.337007    100.0
0.575592    100.0
            ...
0.401246      0.0
0.400141      0.0
0.398533      0.0
0.397710      0.0
1.025843      0.0
Name: Delinquent_Account, Length: 492, dtype: float64
```

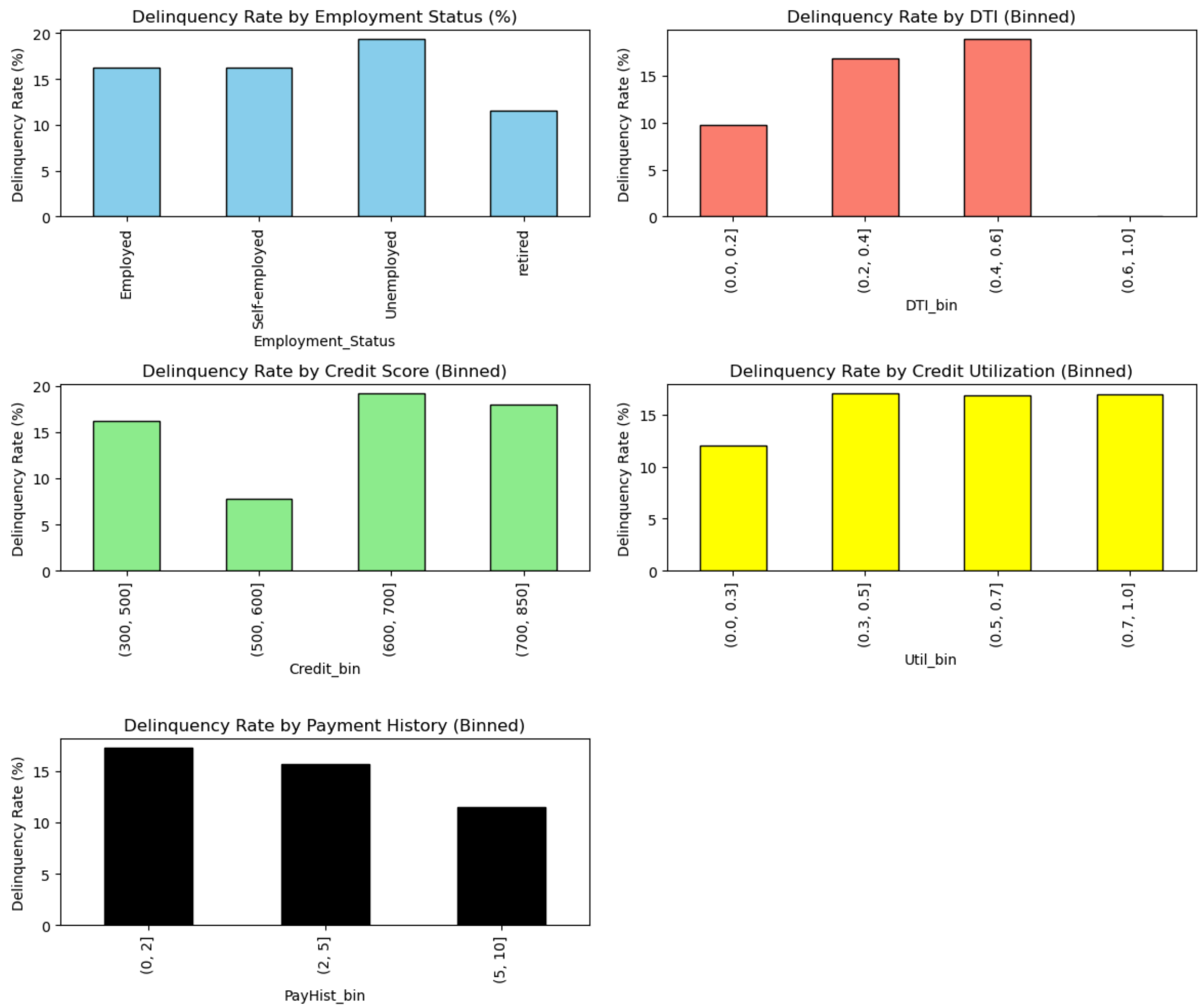
In [19]:

```python
# Employment Status
delinq_by_emp = df.groupby("Employment_Status")["Delinquent_Account"].mean() * 100
```

```
# Debt-to-Income Ratio (binning)
df["DTI_bin"] = pd.cut(df["Debt_to_Income_Ratio"], bins=[0,0.2,0.4,0.6,1.0])
delinq_by_dti = df.groupby("DTI_bin")["Delinquent_Account"].mean() * 100

# Credit Score (binning)
df["Credit_bin"] = pd.cut(df["Credit_Score"], bins=[300,500,600,700,850])
delinq_by_credit = df.groupby("Credit_bin")["Delinquent_Account"].mean() * 100

# Credit Utilization (binning)
df["Util_bin"] = pd.cut(df["Credit_Utilization"], bins=[0,0.3,0.5,0.7,1.0])
delinq_by_util = df.groupby("Util_bin")["Delinquent_Account"].mean() * 100

# Payment History (Missed Payments binning)
df["PayHist_bin"] = pd.cut(df["Missed_Payments"], bins=[0,2,5,10])
delinq_by_payhist = df.groupby("PayHist_bin")["Delinquent_Account"].mean() * 100
```

In [20]:

```
fig, axes = plt.subplots(3, 2, figsize=(12,10))

# Employment Status
delinq_by_emp.plot(kind="bar", ax=axes[0,0], color="skyblue", edgecolor="black")
axes[0,0].set_title("Delinquency Rate by Employment Status (%)")
axes[0,0].set_ylabel("Delinquency Rate (%)")

# Debt-to-Income Ratio
delinq_by_dti.plot(kind="bar", ax=axes[0,1], color="salmon", edgecolor="black")
axes[0,1].set_title("Delinquency Rate by DTI (Binned)")
axes[0,1].set_ylabel("Delinquency Rate (%)")

# Credit Score
delinq_by_credit.plot(kind="bar", ax=axes[1,0], color="lightgreen", edgecolor="blac
axes[1,0].set_title("Delinquency Rate by Credit Score (Binned)")
axes[1,0].set_ylabel("Delinquency Rate (%)")

# Credit Utilization
delinq_by_util.plot(kind="bar", ax=axes[1,1], color="yellow", edgecolor="black")
axes[1,1].set_title("Delinquency Rate by Credit Utilization (Binned)")
axes[1,1].set_ylabel("Delinquency Rate (%)")

# Payment History
delinq_by_payhist.plot(kind="bar", ax=axes[2,0], color="black", edgecolor="black")
axes[2,0].set_title("Delinquency Rate by Payment History (Binned)")
axes[2,0].set_ylabel("Delinquency Rate (%)")

# Hide empty subplot (bottom-right)
axes[2,1].axis("off")

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



In [21]:

```
df.head(10)
```

Out[21]:

	Customer_ID	Age	Income	Credit_Score	Credit_Utilization	Missed_Payments	Delinqu
0	CUST0001	56	165580.0	398.0	0.390502	3	
1	CUST0002	69	100999.0	493.0	0.312444	6	
2	CUST0003	46	188416.0	500.0	0.359930	0	
3	CUST0004	32	101672.0	413.0	0.371400	3	
4	CUST0005	60	38524.0	487.0	0.234716	2	
5	CUST0006	25	84042.0	700.0	0.650540	6	
6	CUST0007	38	35056.0	354.0	0.390581	3	
7	CUST0008	56	123215.0	415.0	0.532715	5	
8	CUST0009	36	66991.0	405.0	0.413035	5	
9	CUST0010	40	34870.0	679.0	0.361824	4	

10 rows × 23 columns

In [ ]:

```
!pip install pycaret
```

Modeling using Gen AI (pyncret)

In [23]:

```
# Import classification module
from pycaret.classification import *
```

In [24]:

```
from pycaret.classification import setup, compare_models, evaluate_model, predict_m

# Setup classification experiment
exp = setup(
    data = df,
    target = "Delinquent_Account",   # target variable
    session_id = 123,                # reproducibility
    train_size = 0.8,                # 80/20 split
    normalize = True,                # scale numeric features
    categorical_imputation = 'mode', # fill missing categorical values
    numeric_imputation = 'mean',     # fill missing numeric values
    verbose = False                  # suppress setup logs if you want a clean outp
)
```

In [25]:
```
# Compare models (AutoML)
best_model = compare_models(sort = 'AUC')  # sorts models by ROC AUC
```

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (Sec)
svm	SVM - Linear Kernel	0.8400	0.5323	0.0000	0.0000	0.0000	0.0000	0.0000	0.3240
knn	K Neighbors Classifier	0.8350	0.5303	0.0000	0.0000	0.0000	-0.0092	-0.0147	0.2510
et	Extra Trees Classifier	0.8400	0.5159	0.0000	0.0000	0.0000	0.0000	0.0000	0.4630
nb	Naive Bayes	0.8400	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000	0.2580
dt	Decision Tree Classifier	0.8400	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000	0.2740
ada	Ada Boost Classifier	0.8400	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000	0.2840
gbc	Gradient Boosting Classifier	0.8400	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000	0.7340
lda	Linear Discriminant Analysis	0.8400	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000	0.2960
dummy	Dummy Classifier	0.8400	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000	0.2460
qda	Quadratic Discriminant Analysis	0.8400	0.4942	0.0000	0.0000	0.0000	0.0000	0.0000	0.2750
ridge	Ridge Classifier	0.8400	0.4796	0.0000	0.0000	0.0000	0.0000	0.0000	0.2720
lr	Logistic Regression	0.8400	0.4790	0.0000	0.0000	0.0000	0.0000	0.0000	1.8750
rf	Random Forest Classifier	0.8400	0.4710	0.0000	0.0000	0.0000	0.0000	0.0000	0.3880

In [26]:
```
#  Evaluate model using  ROC, PR curve, confusion matrix
evaluate_model(best_model)
```

```
interactive(children=(ToggleButtons(description='Plot Type:', icons=('',), options=
(('Pipeline Plot', 'pipelin…
```

Interpreting model with SHAP. SHAP (SHapley Additive exPlanations) is a method to explain the predictions of a machine learning model by showing how much each feature contributes

In [27]:
```
# Only include tree-based models
best_model = compare_models(include=["lightgbm", "rf", "et", "dt"], sort="AUC")

# Now interpret works
interpret_model(best_model, plot="summary")
```

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (Sec)
et	Extra Trees Classifier	0.8400	0.5159	0.0000	0.0000	0.0000	0.0000	0.0000	0.4720
dt	Decision Tree Classifier	0.8400	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000	0.3160
rf	Random Forest Classifier	0.8400	0.4710	0.0000	0.0000	0.0000	0.0000	0.0000	0.4820

SHAP summary plot shows the overall impact of features on a machine learning model's output (which features are most influential and how they affect the predictions with the most impactful feature at the top)

High Impact Features (at the top) → Customer_ID, Month_6_Late. Wide spreads mean they can shift predictions strongly in either direction.

Customer_ID → unusually important; this suggests data leakage since IDs shouldn't affect outcomes. Needs investigation.

Month_6_Late → late in month 6 (red) raises default risk; not late (blue) lowers it.

Month_1_On-time → being on time lowers default risk; not on time raises it.

Location_Los Angeles → being in LA reduces risk; not in LA increases it.

Extra Trees, Decision Tree, Random Forest are showing high accuracy (0.84) but zero recall, precision, and F1 which signifies Severe class imbalance

Most customers in the dataset are not delinquent and models are just predicting "Not Delinquent" for everyone, which gives high accuracy but fails to detect delinquents (hence recall = 0).

AUC is near random (~0.5)Confirms the model isn't separating delinquent(Customers who are paying their credit obligations on time) vs. non-delinquent customers (Customers who have missed payments).

ACTION: Use Smote to solve the imbalance

In [28]:
```
#enable  balancing

from imblearn.over_sampling import SMOTE

exp = setup(
    data=df,
    target="Delinquent_Account",
    session_id=123,
    train_size=0.8,
    normalize=True,
    categorical_imputation="mode",
    numeric_imputation="mean",
    fix_imbalance=True,
    fix_imbalance_method=SMOTE()
)
```

	Description	Value
0	Session id	123
1	Target	Delinquent_Account
2	Target type	Binary
3	Original data shape	(500, 23)
4	Transformed data shape	(772, 56)
5	Transformed train set shape	(672, 56)
6	Transformed test set shape	(100, 56)
7	Numeric features	8
8	Categorical features	14
9	Rows with missing values	16.2%
10	Preprocess	True
11	Imputation type	simple
12	Numeric imputation	mean
13	Categorical imputation	mode
14	Maximum one-hot encoding	25
15	Encoding method	None
16	Fix imbalance	True
17	Fix imbalance method	SMOTE(k_neighbors=5, random_state=None, sampling_strategy='auto')
18	Normalize	True
19	Normalize method	zscore
20	Fold Generator	StratifiedKFold
21	Fold Number	10
22	CPU Jobs	-1
23	Use GPU	False
24	Log Experiment	False
25	Experiment Name	clf-default-name
26	USI	bd4a

Summary

Original data shape → (500, 23) Transformed data shape → (772, 56), SMOTE created synthetic samples of the minority class (delinquent customers/training data) to balance the dataset.

Train set (672, 56) / Test set (100, 56).Data was split into 80% train / 20% test after balancing.

In [29]:
```
#  Compare models after smote
best_model = compare_models(sort="Recall")
```

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (Sec)
knn	K Neighbors Classifier	0.3100	0.4568	0.6690	0.1435	0.2362	-0.0364	-0.0725	0.2660
lda	Linear Discriminant Analysis	0.5525	0.4398	0.3738	0.1422	0.2014	-0.0299	-0.0309	0.2640
svm	SVM - Linear Kernel	0.8375	0.5213	0.0167	0.0333	0.0222	0.0136	0.0146	0.3130
lr	Logistic Regression	0.8400	0.4506	0.0000	0.0000	0.0000	0.0000	0.0000	0.2620
nb	Naive Bayes	0.8400	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000	0.2970
dt	Decision Tree Classifier	0.8400	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000	0.2850
ridge	Ridge Classifier	0.8400	0.4622	0.0000	0.0000	0.0000	0.0000	0.0000	0.2970
rf	Random Forest Classifier	0.8400	0.4370	0.0000	0.0000	0.0000	0.0000	0.0000	0.4660
qda	Quadratic Discriminant Analysis	0.8400	0.4917	0.0000	0.0000	0.0000	0.0000	0.0000	0.2960
ada	Ada Boost Classifier	0.8400	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000	0.3170
gbc	Gradient Boosting Classifier	0.8400	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000	0.4090
et	Extra Trees Classifier	0.8400	0.5674	0.0000	0.0000	0.0000	0.0000	0.0000	0.4670
dummy	Dummy Classifier	0.8400	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000	0.2870

In [30]:
```
# Evaluate the best model
evaluate_model(best_model)
```

```
interactive(children=(ToggleButtons(description='Plot Type:', icons=('',), options=
(('Pipeline Plot', 'pipelin…
```

this plot provides a visual summary of all the data preprocessing and modeling steps that PyCaret performed behind the scenes to create the final model

Raw data: This is your initial, unprocessed dataset.

SimpleImputer(one for numerical, one for categorical): ensures data is clean before it's used for training.

OneHotEncoder: converts categorical data into a numerical format createing new binary columns that the machine learning model can understand.

TargetEncoder: replaces each category with a numerical value based on the mean of the target variable for that category.

FixImbalance: addresses the class imbalance in the target variable (Delinquent_Account)

StandardScaler: scales numerical features so they have a mean of 0 and a standard deviation of 1.

KNeighborsClassifier: where the model is trained on the preprocessed data.

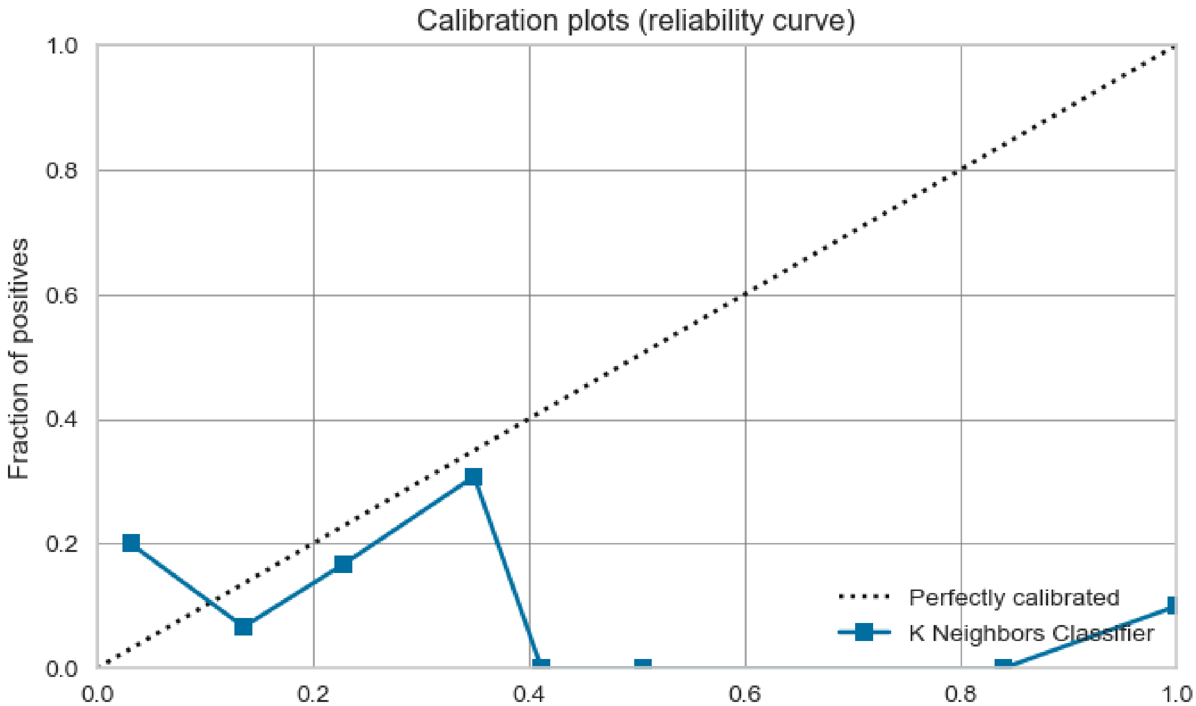
Hyperparameter Tuning

calibration (PyCaret, calibrate_model()) : improve the probability estimates of a trained classifier by applying probability calibration.Helps improve KNN model predictive performance , especially the k value.

In [31]:
```
# Create KNN model
knn = create_model('knn')

# Calibrate the KNN model
calibrated_knn = calibrate_model(knn, method='isotonic')

# Plot calibration curve
plot_model(calibrated_knn, plot='calibration')
```

Fold	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.3500	0.4069	0.5000	0.1154	0.1875	-0.0744	-0.1321
1	0.3500	0.4755	0.6667	0.1429	0.2353	-0.0156	-0.0306
2	0.2500	0.1912	0.5000	0.1000	0.1667	-0.1111	-0.2425
3	0.3500	0.6544	0.6667	0.1429	0.2353	-0.0156	-0.0306
4	0.4000	0.5539	0.6667	0.1538	0.2500	0.0083	0.0147
5	0.3750	0.4608	0.6667	0.1481	0.2424	-0.0040	-0.0075
6	0.2750	0.4740	0.5714	0.1333	0.2162	-0.0943	-0.1899
7	0.2500	0.3290	0.4286	0.1034	0.1667	-0.1605	-0.3058
8	0.3000	0.3377	0.5714	0.1379	0.2222	-0.0832	-0.1584
9	0.4500	0.5736	0.8571	0.2222	0.3529	0.1039	0.1791
Mean	0.3350	0.4457	0.6095	0.1400	0.2275	-0.0447	-0.0904
Std	0.0624	0.1285	0.1158	0.0326	0.0507	0.0713	0.1355

Fold	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.7250	0.4265	0.0000	0.0000	0.0000	-0.1579	-0.1588
1	0.7000	0.4853	0.0000	0.0000	0.0000	-0.1765	-0.1765
2	0.6000	0.1985	0.0000	0.0000	0.0000	-0.2308	-0.2425
3	0.8250	0.6275	0.3333	0.4000	0.3636	0.2632	0.2646
4	0.7750	0.7353	0.3333	0.2857	0.3077	0.1743	0.1751
5	0.6750	0.5123	0.1667	0.1111	0.1333	-0.0569	-0.0587
6	0.7250	0.5779	0.1429	0.1667	0.1538	-0.0092	-0.0092
7	0.6000	0.3571	0.1429	0.0909	0.1111	-0.1307	-0.1363
8	0.6250	0.3658	0.0000	0.0000	0.0000	-0.2295	-0.2303
9	0.8500	0.4784	0.2857	0.6667	0.4000	0.3296	0.3685
Mean	0.7100	0.4765	0.1405	0.1721	0.1470	-0.0224	-0.0204
Std	0.0838	0.1441	0.1323	0.2093	0.1499	0.1964	0.2056



In [32]:
```
#  Finalize model (train on full dataset)
final_model = finalize_model(best_model)
```

In [33]:
```
#  Save model for deployment
save_model(final_model, "customer_delinquency_model")
```

```
Transformation Pipeline and Model Successfully Saved
```

Out[33]:
```
(Pipeline(memory=Memory(location=None),
          steps=[('numerical_imputer',
                  TransformerWrapper(exclude=None,
                                     include=['Age', 'Income', 'Credit_Score',
                                              'Credit_Utilization',
                                              'Missed_Payments', 'Loan_Balance',
                                              'Debt_to_Income_Ratio',
                                              'Account_Tenure'],
                                     transformer=SimpleImputer(add_indicator=Fals
e,
                                                               copy=True,
                                                               fill_value=None,
                                                               keep_empty_features
=False,
                                                               missing_values=nan,
                                                               s...
                                     transformer=FixImbalancer(estimator=SMOTE(k_n
eighbors=5,
                                                                               ran
dom_state=None,
                                                                               sam
pling_strategy='auto')))),
                 ('normalize',
                  TransformerWrapper(exclude=None, include=None,
                                     transformer=StandardScaler(copy=True,
                                                                with_mean=True,
                                                                with_std=True))),
                 ('actual_estimator',
                  KNeighborsClassifier(algorithm='auto', leaf_size=30,
                                       metric='minkowski', metric_params=None,
                                       n_jobs=-1, n_neighbors=5, p=2,
                                       weights='uniform'))],
          verbose=False),
 'customer_delinquency_model.pkl')
```

In [34]:
```
#  Predict on new/unseen data
predictions = predict_model(final_model, data=df)
```

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	K Neighbors Classifier	0.6140	0.9943	1.0000	0.2930	0.4533	0.2735	0.3980

In [35]:
```
# Show sample predictions
predictions.head()
```

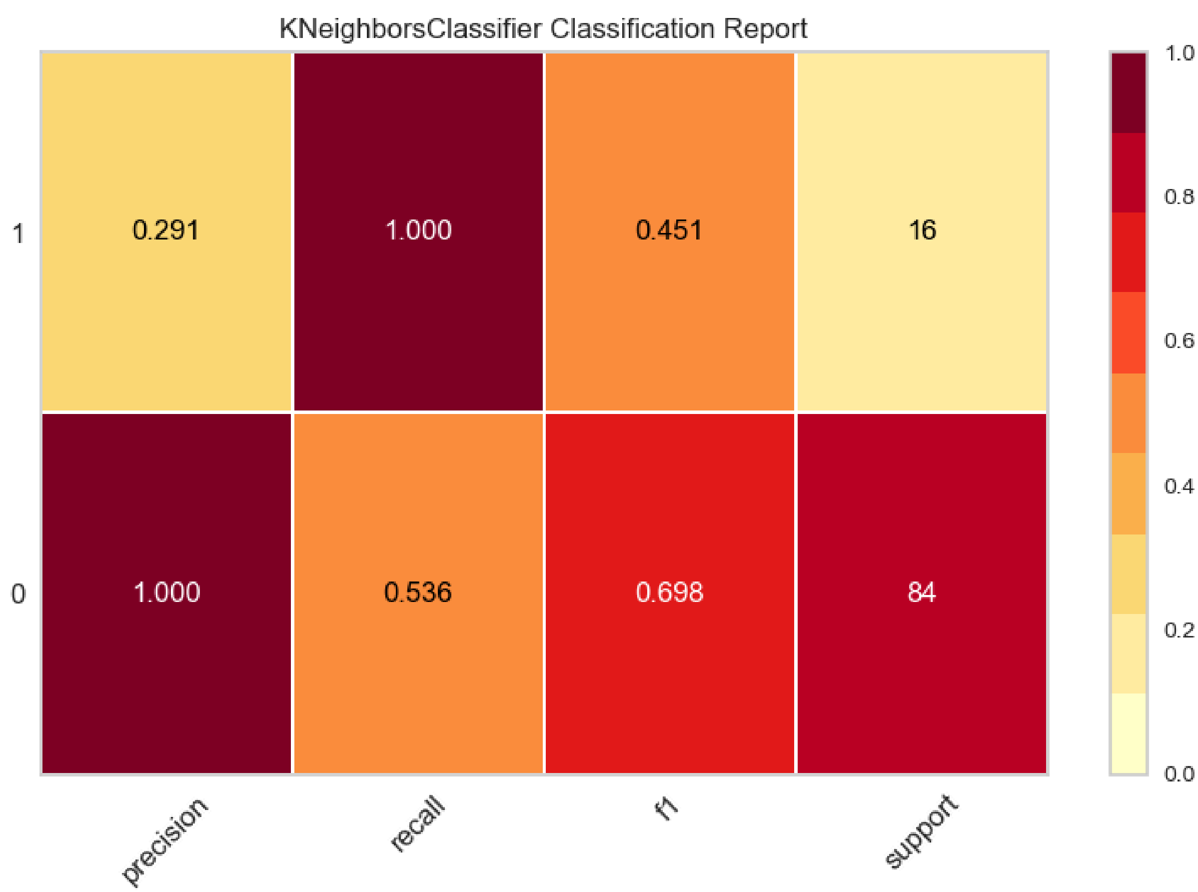
Out[35]:

	Customer_ID	Age	Income	Credit_Score	Credit_Utilization	Missed_Payments	Loan_Ba
0	CUST0001	56	165580.0	398.0	0.390502	3	16
1	CUST0002	69	100999.0	493.0	0.312444	6	17
2	CUST0003	46	188416.0	500.0	0.359930	0	13
3	CUST0004	32	101672.0	413.0	0.371400	3	88
4	CUST0005	60	38524.0	487.0	0.234716	2	13

5 rows × 25 columns

In [36]:

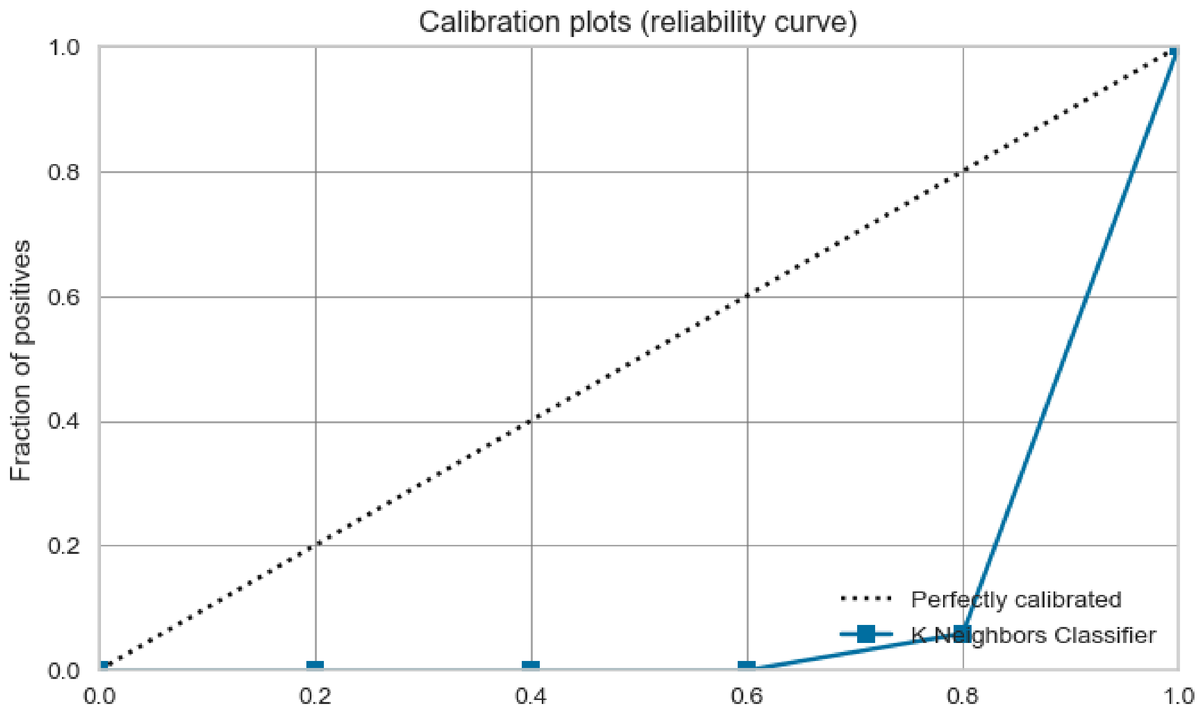
```
# Classification Report
plot_model(final_model, plot="class_report")
```



This KNN model prioritizes catching all delinquent customers (Recall = 100%), which is great for risk minimization.

But it sacrifices precision, meaning many safe customers get falsely flagged as delinquent.

In [37]:
```
#Probability Distribution
plot_model(final_model, plot="calibration")
```



In [38]:
```
# Sort customers by delinquency risk
risk_ranking = predictions.sort_values("prediction_score", ascending=False)
risk_ranking[["Customer_ID", "prediction_label", "prediction_score"]].head(10)
```

Out[38]:

	Customer_ID	prediction_label	prediction_score
93	CUST0094	0	1.0
251	CUST0252	1	1.0
455	CUST0456	0	1.0
257	CUST0258	1	1.0
105	CUST0106	0	1.0
253	CUST0254	1	1.0
452	CUST0453	1	1.0
109	CUST0110	0	1.0
1	CUST0002	1	1.0
457	CUST0458	0	1.0

In [ ]:

In [ ]: