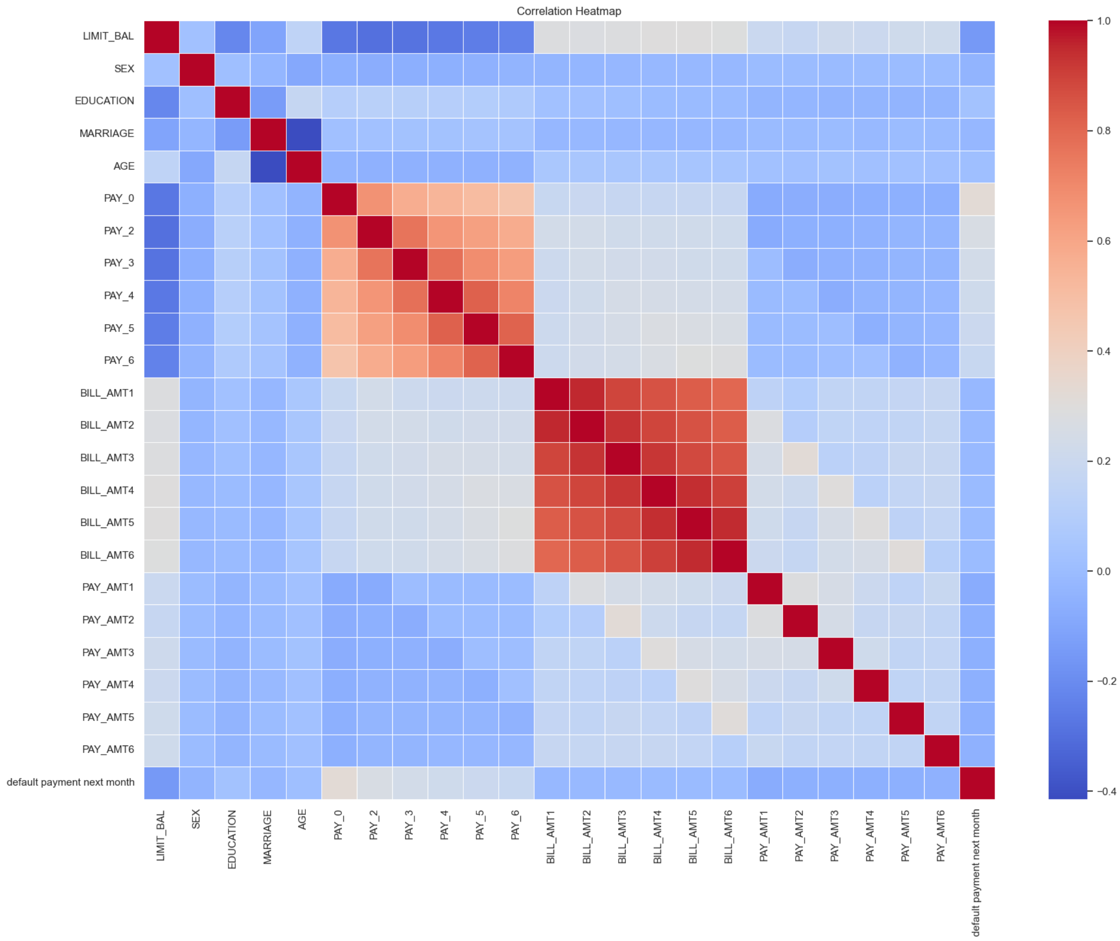
FinalTeamProject

June 21, 2025

[216]:
```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

[217]:
```python
df = pd.read_csv('Datasets/Credit.csv')
```

[218]:
```python
# Load the dataset -- Skip first metadata row
df = pd.read_csv('Datasets/Credit.csv')

# Drop the ID column as it's not useful for analysis
df_cleaned = df.drop(columns=["ID"])

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

# Generating Correlation Heatmap
plt.figure(figsize=(18, 14))
correlation_matrix = df_cleaned.corr()
sns.heatmap(correlation_matrix, annot=False, cmap="coolwarm", fmt=".2f",␣
 ↪linewidths=0.5)
plt.title("Correlation Heatmap")
plt.tight_layout()
plt.show()

# Printing my explanation
print("explanation")
print()
print("The correlation heatmap of the dataset reveals relationships between␣
 ↪features such as: High correlation among BILL_AMT variables (e.g.,␣
 ↪BILL_AMT1, BILL_AMT2, etc.)")
print("The correlation heatmap also indicates a positive correlation between␣
 ↪LIMIT_BAL and PAY_AMT values, a Mmoderate positive correlation between past␣
 ↪payment statuses (PAY_0 to PAY_6) and default likelihood")
```

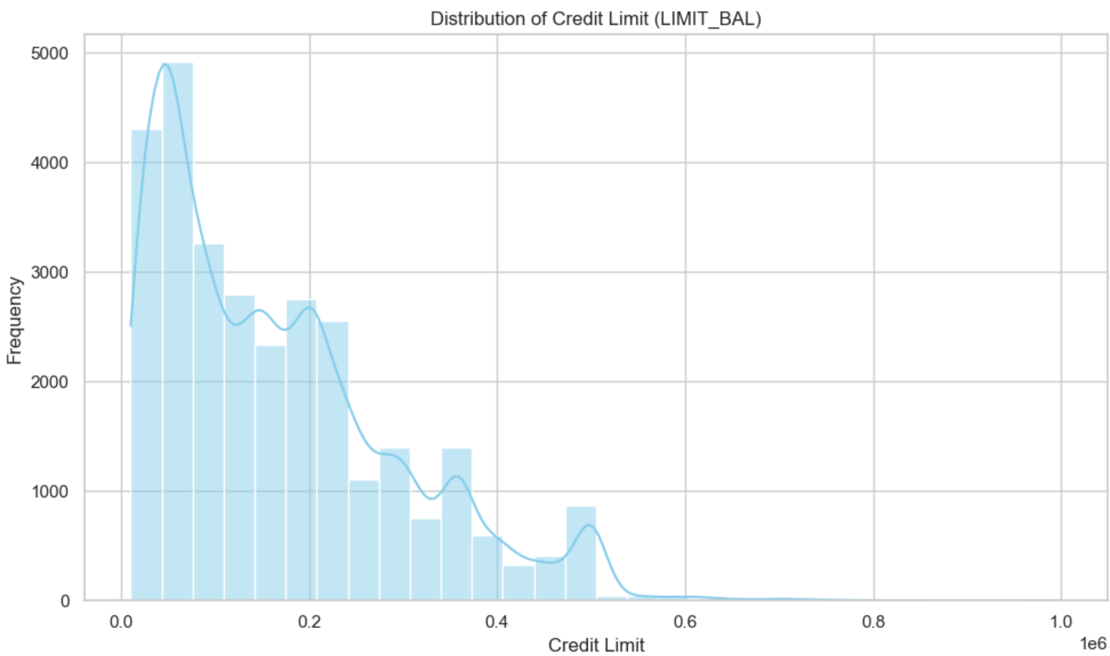


explanation

The correlation heatmap of the dataset reveals relationships between features such as: High correlation among BILL_AMT variables (e.g., BILL_AMT1, BILL_AMT2, etc.)
The correlation heatmap also indicates a positive correlation between LIMIT_BAL and PAY_AMT values, a Mmoderate positive correlation between past payment statuses (PAY_0 to PAY_6) and default likelihood

[219]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset (assuming the file is in the same directory) --Skip first metadata row
df = pd.read_csv('Datasets/Credit.csv')

# Drop the ID column as it's not useful for analysis
```

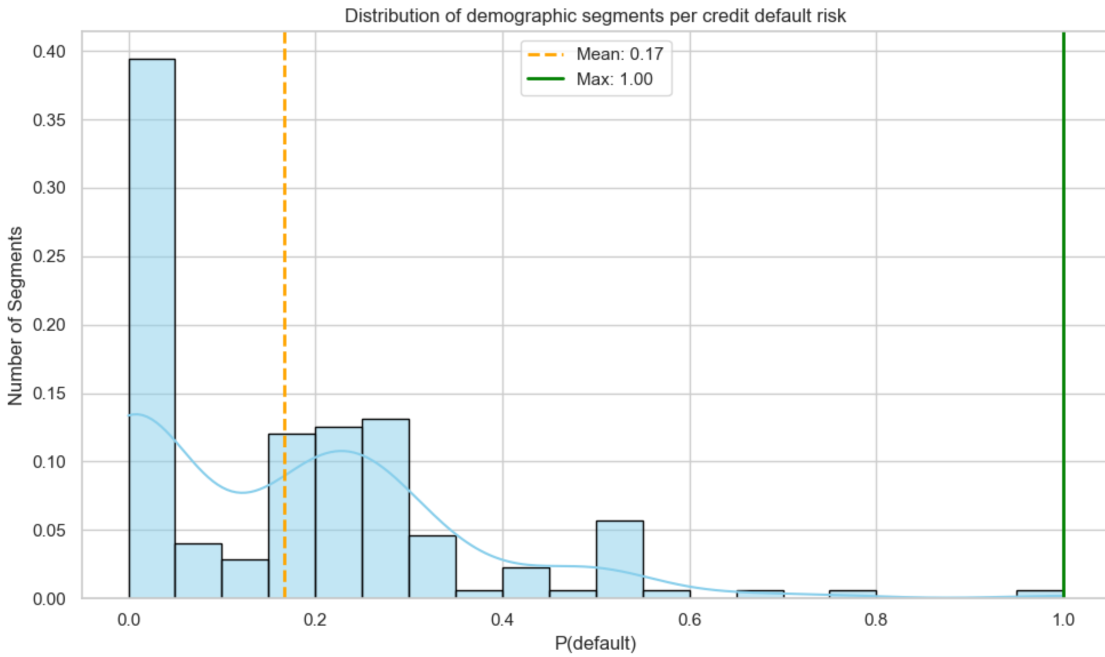
```python
df_cleaned = df.drop(columns=["ID"])

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

# --- Histogram of Credit Limit ---
plt.figure(figsize=(10, 6))
sns.histplot(df_cleaned["LIMIT_BAL"], bins=30, kde=True, color="skyblue")
plt.title("Distribution of Credit Limit (LIMIT_BAL)")
plt.xlabel("Credit Limit")
plt.ylabel("Frequency")
plt.tight_layout()
plt.show()

# Printing my explanation
print("Explanation -- The histogram of the credit limit (LIMIT_BAL)")
print()
print("Most credit limits are concentrated below 200,000 units.")
print("The distribution is right-skewed, indicating a smaller number of clients␣
↪with very high credit limits.")
```




```
Explanation -- The histogram of the credit limit (LIMIT_BAL)

Most credit limits are concentrated below 200,000 units.
The distribution is right-skewed, indicating a smaller number of clients with
very high credit limits.
```

[220]:
```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df2 = pd.read_csv('Datasets/Credit.csv')

# Create AGE groups
age_bins = [20, 30, 40, 50, 60, 70, 80]
age_labels = ['20s', '30s', '40s', '50s', '60s', '70s']
df2['AGE_GROUP'] = pd.cut(df2['AGE'], bins=age_bins, labels=age_labels,␣
↪right=False)

# Define population segments
segment_columns = ['SEX', 'EDUCATION', 'MARRIAGE', 'AGE_GROUP']
segment_group = df2.groupby(segment_columns)

# Count total and on-time payments per segment
segment_stats = segment_group['default payment next month'].agg(
    total='count',
    default=lambda x: (x == 1).sum()
).reset_index()

# Calculate Probability
segment_stats['probability'] = segment_stats['default'] / segment_stats['total']

# Plot histogram with KDE
plt.figure(figsize=(10, 6))
sns.histplot(segment_stats['probability'], bins=20, kde=True, color='skyblue',␣
↪edgecolor='black', stat='probability')

# Add vertical lines
mean_prob = segment_stats['probability'].mean()
max_prob = segment_stats['probability'].max()

plt.axvline(mean_prob, color='orange', linestyle='--', linewidth=2,␣
↪label=f'Mean: {mean_prob:.2f}')
plt.axvline(max_prob, color='green', linestyle='-', linewidth=2, label=f'Max:␣
↪{max_prob:.2f}')

# Labels and legend
plt.title("Distribution of demographic segments per credit default risk")
plt.xlabel("P(default)")
plt.ylabel("Number of Segments")
plt.legend()
plt.tight_layout()
plt.show()
```

```
/var/folders/ck/sr6gtz6n0jx9dmp9nlplxl_w0000gn/T/ipykernel_65554/2297665363.py:1
4: FutureWarning: The default of observed=False is deprecated and will be
changed to True in a future version of pandas. Pass observed=False to retain
current behavior or observed=True to adopt the future default and silence this
warning.
  segment_group = df2.groupby(segment_columns)
```



[221]:

```python
df.rename(columns={df.columns[-1]: 'default_status'}, inplace=True)

# Define the columns
bill_columns = ['BILL_AMT1', 'BILL_AMT2', 'BILL_AMT3', 'BILL_AMT4',␣
 ↪'BILL_AMT5', 'BILL_AMT6']
pay_columns = ['PAY_AMT1', 'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5',␣
 ↪'PAY_AMT6']

# Method 1: Linear decay weights (most recent gets highest weight)
# Weights: [6, 5, 4, 3, 2, 1] for [AMT1, AMT2, AMT3, AMT4, AMT5, AMT6]
linear_weights = np.array([6, 5, 4, 3, 2, 1])
linear_weights = linear_weights / linear_weights.sum()  # Normalize to sum to 1

print("Linear weights:", linear_weights)

# Calculate weighted averages
df['WEIGHTED_BILL_AMT'] = np.average(df[bill_columns], weights=linear_weights,␣
 ↪axis=1)
```

```python
df['WEIGHTED_PAY_AMT'] = np.average(df[pay_columns], weights=linear_weights, axis=1)

def create_numeric_percentile_bins(df, column_name, num_bins=4):
    """
    Create percentile bins with ascending numeric codes (1, 2, 3, 4)
    """
    # Create percentile bins and assign numeric labels
    binned_column = pd.qcut(df[column_name], q=num_bins, labels=range(1, num_bins + 1), duplicates='drop')

    # Get the actual bin edges for reference
    _, bin_edges = pd.qcut(df[column_name], q=num_bins, retbins=True, duplicates='drop')

    return binned_column.astype(int), bin_edges

# Apply numeric percentile binning
variables_to_bin = ['AGE', 'LIMIT_BAL', 'WEIGHTED_BILL_AMT', 'WEIGHTED_PAY_AMT']

print("Creating numeric percentile-based bins (1=lowest quartile, 4=highest quartile)...")
print("=" * 80)

for var in variables_to_bin:
    # Create numeric bins
    binned_col, edges = create_numeric_percentile_bins(df, var, num_bins=4)

    # Add the binned column to dataframe
    df[f'{var}_Q'] = binned_col

    # Print bin information
    print(f"\n{var}_Q:")
    print(f"  Overall range: {df[var].min():.2f} to {df[var].max():.2f}")
    print(f"  Quartile boundaries and coding:")

    for i in range(len(edges) - 1):
        quartile_num = i + 1
        start_val = edges[i]
        end_val = edges[i + 1]
        count = (df[f'{var}_Q'] == quartile_num).sum()
        percentage = count / len(df) * 100

        print(f"    {quartile_num}: {start_val:8.2f} to {end_val:8.2f} | {count:,} obs ({percentage:.1f}%)")

    # Show the numeric distribution
```

```
    print(f"  Value counts: {dict(df[f'{var}_Q'].value_counts().sort_index())}")



df.head()
```

```
Linear weights: [0.28571429 0.23809524 0.19047619 0.14285714 0.0952381
0.04761905]
Creating numeric percentile-based bins (1=lowest quartile, 4=highest
quartile)…
================================================================================

AGE_Q:
  Overall range: 21.00 to 79.00
  Quartile boundaries and coding:
    1:    21.00 to    28.00 | 8,013 obs (26.7%)
    2:    28.00 to    34.00 | 7,683 obs (25.6%)
    3:    34.00 to    41.00 | 6,854 obs (22.8%)
    4:    41.00 to    79.00 | 7,450 obs (24.8%)
  Value counts: {1: np.int64(8013), 2: np.int64(7683), 3: np.int64(6854), 4:
np.int64(7450)}

LIMIT_BAL_Q:
  Overall range: 10000.00 to 1000000.00
  Quartile boundaries and coding:
    1: 10000.00 to 50000.00 | 7,676 obs (25.6%)
    2: 50000.00 to 140000.00 | 7,614 obs (25.4%)
    3: 140000.00 to 240000.00 | 7,643 obs (25.5%)
    4: 240000.00 to 1000000.00 | 7,067 obs (23.6%)
  Value counts: {1: np.int64(7676), 2: np.int64(7614), 3: np.int64(7643), 4:
np.int64(7067)}

WEIGHTED_BILL_AMT_Q:
  Overall range: -29464.95 to 873217.38
  Quartile boundaries and coding:
    1: -29464.95 to  4888.90 | 7,500 obs (25.0%)
    2:  4888.90 to 21980.29 | 7,500 obs (25.0%)
    3: 21980.29 to 60405.44 | 7,500 obs (25.0%)
    4: 60405.44 to 873217.38 | 7,500 obs (25.0%)
  Value counts: {1: np.int64(7500), 2: np.int64(7500), 3: np.int64(7500), 4:
np.int64(7500)}

WEIGHTED_PAY_AMT_Q:
  Overall range: 0.00 to 805849.48
  Quartile boundaries and coding:
    1:     0.00 to  1228.08 | 7,500 obs (25.0%)
    2:  1228.08 to  2488.14 | 7,500 obs (25.0%)
    3:  2488.14 to  5696.19 | 7,500 obs (25.0%)
```

```
    4:  5696.19 to 805849.48 | 7,500 obs (25.0%)
  Value counts: {1: np.int64(7500), 2: np.int64(7500), 3: np.int64(7500), 4:
np.int64(7500)}
```

[221]:

```
   ID  LIMIT_BAL  SEX  EDUCATION  MARRIAGE  AGE  PAY_0  PAY_2  PAY_3  PAY_4  \
0   1      20000    2          2         1   24      2      2     -1     -1
1   2     120000    2          2         2   26     -1      2      0      0
2   3      90000    2          2         2   34      0      0      0      0
3   4      50000    2          2         1   37      0      0      0      0
4   5      50000    1          2         1   57     -1      0     -1      0

   …  PAY_AMT4  PAY_AMT5  PAY_AMT6  default_status  WEIGHTED_BILL_AMT  \
0  …         0         0         0               1        1987.809524
1  …      1000         0      2000               1        2639.619048
2  …      1000      1000      5000               0       18487.761905
3  …      1100      1069      1000               0       42508.380952
4  …      9000       689       679               0       16363.571429

   WEIGHTED_PAY_AMT  AGE_Q  LIMIT_BAL_Q  WEIGHTED_BILL_AMT_Q  \
0        164.047619      1            1                    1
1        666.666667      1            2                    1
2       1457.523810      2            2                    2
3       1587.285714      3            1                    3
4      12593.428571      4            1                    2

   WEIGHTED_PAY_AMT_Q
0                   1
1                   1
2                   2
3                   2
4                   4

[5 rows x 31 columns]
```

[222]:

```python
# replace -1 with 0
df['PAY_0'] = df['PAY_0'].replace(-1, 0)

# separate between train and test

train_df = df.sample(frac=0.7, random_state=42)
test_df = df.drop(train_df.index)

train_df.shape
```

[222]: (21000, 31)

[223]:

```python
# train logistic regression model

import statsmodels.formula.api as smf
import statsmodels.api as sm



model = smf.glm('default_status ~ LIMIT_BAL_Q + SEX + EDUCATION + MARRIAGE + AGE_Q + PAY_0 + WEIGHTED_BILL_AMT_Q + WEIGHTED_PAY_AMT_Q', data=train_df, family=sm.families.Binomial())

results = model.fit()

results.summary()
```

[223]:

Dep. Variable:	default_status	**No. Observations:**	21000
Model:	GLM	**Df Residuals:**	20991
Model Family:	Binomial	**Df Model:**	8
Link Function:	Logit	**Scale:**	1.0000
Method:	IRLS	**Log-Likelihood:**	-9560.6
Date:	Sat, 21 Jun 2025	**Deviance:**	19121.
Time:	13:57:33	**Pearson chi2:**	2.57e+04
No. Iterations:	5	**Pseudo R-squ. (CS):**	0.1311
Covariance Type:	nonrobust		

	coef	**std err**	**z**	**P>|z|**	**[0.025**	**0.975]**
Intercept	-0.1974	0.134	-1.474	0.140	-0.460	0.065
LIMIT_BAL_Q	-0.1390	0.019	-7.296	0.000	-0.176	-0.102
SEX	-0.1118	0.037	-3.014	0.003	-0.185	-0.039
EDUCATION	-0.0605	0.025	-2.415	0.016	-0.110	-0.011
MARRIAGE	-0.1505	0.039	-3.890	0.000	-0.226	-0.075
AGE_Q	0.0377	0.018	2.105	0.035	0.003	0.073
PAY_0	0.8464	0.021	39.870	0.000	0.805	0.888
WEIGHTED_BILL_AMT_Q	-0.0003	0.021	-0.013	0.990	-0.041	0.040
WEIGHTED_PAY_AMT_Q	-0.2598	0.022	-11.572	0.000	-0.304	-0.216

[224]:

```python
# analyze results

summary_df = pd.concat([results.params, results.pvalues], axis=1, keys=['coef', 'pvalue'])

# absolute value of the coefficients for sorting
summary_df = summary_df.assign(abs_coef=summary_df['coef'].abs())

# get labels of variables with p > 0.05
removed_labels = summary_df.index[summary_df['pvalue'] > 0.05].tolist()
```

```python
# keep only variables with p <= 0.05
summary_df = summary_df[summary_df['pvalue'] <= 0.05]

# sort by effect size
summary_df = summary_df.sort_values(by='abs_coef', ascending=False)

# rounding
summary_df['pvalue'] = summary_df['pvalue'].map('{:.5f}'.format)

# print labels of variables with p > 0.05
print("p > 0.05: \n\n{}".format(removed_labels))

print("\n--------------------------------\n")

print("Sorted by effect size: \n{}".format(summary_df))
print("\n--------------------------------\n")

# sort by pvalue
summary_df = summary_df.sort_values(by='pvalue', ascending=True)

print("\n--------------------------------\n")

print("Sorted by p-value: \n{}".format(summary_df))
print("\n--------------------------------\n")
```

```
p > 0.05:

['Intercept', 'WEIGHTED_BILL_AMT_Q']

--------------------------------


Sorted by effect size:
                       coef   pvalue  abs_coef
PAY_0              0.846440  0.00000  0.846440
WEIGHTED_PAY_AMT_Q -0.259782  0.00000  0.259782
MARRIAGE          -0.150458  0.00010  0.150458
LIMIT_BAL_Q       -0.138968  0.00000  0.138968
SEX               -0.111832  0.00258  0.111832
EDUCATION         -0.060463  0.01573  0.060463
AGE_Q              0.037691  0.03530  0.037691

--------------------------------



--------------------------------


Sorted by p-value:
                       coef   pvalue  abs_coef
```

```
PAY_0                 0.846440  0.00000  0.846440
WEIGHTED_PAY_AMT_Q -0.259782  0.00000  0.259782
LIMIT_BAL_Q        -0.138968  0.00000  0.138968
MARRIAGE           -0.150458  0.00010  0.150458
SEX                -0.111832  0.00258  0.111832
EDUCATION          -0.060463  0.01573  0.060463
AGE_Q               0.037691  0.03530  0.037691

--------------------------------
```

[225]:
```
odds_ratios = pd.Series(
    data=round(np.exp(summary_df['coef']), 2),
    index=summary_df.index,
    name='odds_ratio'
)

print(odds_ratios)
```

```
PAY_0                 2.33
WEIGHTED_PAY_AMT_Q    0.77
LIMIT_BAL_Q           0.87
MARRIAGE              0.86
SEX                   0.89
EDUCATION             0.94
AGE_Q                 1.04
Name: odds_ratio, dtype: float64
```

[226]:
```
# Make examples

class Person:

    def __init__(self, age, sex, education, marriage, limit_balance,␣
 ↪bill_amount, payment_amount, payment_history):
        self.age = age
        self.sex = sex
        self.education = education
        self.marriage = marriage
        self.limit_balance = limit_balance
        self.bill_amount = bill_amount
        self.payment_amount = payment_amount
        self.payment_history = payment_history

    def calculate_probability(self):
        intercept = results.params['Intercept']
        age_coef = results.params['AGE_Q']
        sex_coef = results.params['SEX']
        education_coef = results.params['EDUCATION']
```

```
        marriage_coef = results.params['MARRIAGE']
        limit_balance_coef = results.params['LIMIT_BAL_Q']
        bill_amount_coef = results.params['WEIGHTED_BILL_AMT_Q']
        payment_amount_coef = results.params['WEIGHTED_PAY_AMT_Q']
        payment_history_coef = results.params['PAY_0']

        probability = 1 / (1 + np.exp(-(intercept + age_coef * self.age +␣
 ↪sex_coef * self.sex + education_coef * self.education + marriage_coef * self.
 ↪marriage + limit_balance_coef * self.limit_balance + bill_amount_coef * self.
 ↪bill_amount + payment_amount_coef * self.payment_amount +␣
 ↪payment_history_coef * self.payment_history)))

        return probability



jake = Person(age=1, sex=1, education=0, marriage=0, limit_balance=1,␣
 ↪bill_amount=2, payment_amount=0, payment_history=0)
print("jake:", round(jake.calculate_probability(), 4))

john = Person(age=1, sex=1, education=4, marriage=3, limit_balance=1,␣
 ↪bill_amount=4, payment_amount=0, payment_history=8)
print("john:", round(john.calculate_probability(), 4))

penelope = Person(age=4, sex=2, education=1, marriage=1, limit_balance=4,␣
 ↪bill_amount=1, payment_amount=3, payment_history=0)
print("penelope:", round(penelope.calculate_probability(), 4))

ricardo = Person(age=1, sex=1, education=1, marriage=0, limit_balance=4,␣
 ↪bill_amount=4, payment_amount=1, payment_history=6)
print("ricardo:", round(ricardo.calculate_probability(), 4))

stella = Person(age=2, sex=2, education=3, marriage=2, limit_balance=1,␣
 ↪bill_amount=1, payment_amount=1, payment_history=0)
print("stella:", round(stella.calculate_probability(), 4))
```

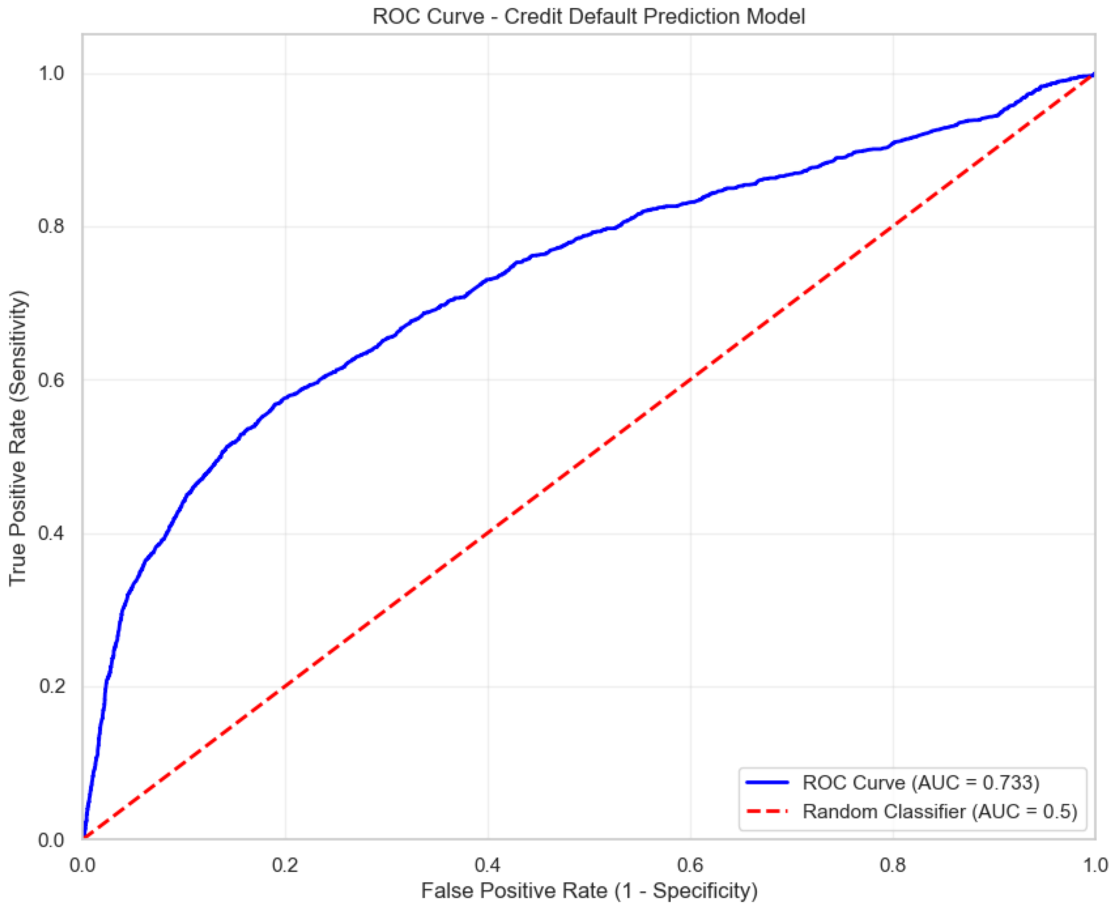
```
jake: 0.3987
john: 0.9966
penelope: 0.1398
ricardo: 0.9807
stella: 0.2267
```

[227]:
```
# calculate metrics

from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score, confusion_matrix, roc_auc_score, roc_curve
```

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


# Generate predictions on test set
# Get predicted probabilities
test_probabilities = results.predict(test_df)

# Convert probabilities to binary predictions using 0.5 threshold
test_predictions = (test_probabilities > 0.5).astype(int)

# Get actual values
test_actual = test_df['default_status'].values

print(f"Test set size: {len(test_df)}")
print(f"Number of actual defaults in test set: {sum(test_actual)}")
print(f"Number of predicted defaults: {sum(test_predictions)}")


# Calculate confusion matrix
cm = confusion_matrix(test_actual, test_predictions)
print("Confusion Matrix:")
print(cm)

# Extract components
tn, fp, fn, tp = cm.ravel()
print(f"\nBreakdown:")
print(f"True Negatives (TN): {tn}")
print(f"False Positives (FP): {fp}")
print(f"False Negatives (FN): {fn}")
print(f"True Positives (TP): {tp}")

# Calculate all performance metrics
accuracy = accuracy_score(test_actual, test_predictions)
precision = precision_score(test_actual, test_predictions)
sensitivity_recall = recall_score(test_actual, test_predictions)  # Same as␣
↪sensitivity
f1 = f1_score(test_actual, test_predictions)

# Calculate specificity manually (no direct sklearn function)
specificity = tn / (tn + fp)

print("=== MODEL PERFORMANCE METRICS ===")
print(f"Accuracy: {accuracy:.4f} ({accuracy*100:.2f}%)")
print(f"Precision: {precision:.4f} ({precision*100:.2f}%)")
print(f"Sensitivity (Recall): {sensitivity_recall:.4f} ({sensitivity_recall*100:
↪.2f}%)")
```

```python
print(f"Specificity: {specificity:.4f} ({specificity*100:.2f}%)")
print(f"F1-Score: {f1:.4f}")

print("\n=== METRIC INTERPRETATIONS ===")
print(f"• Accuracy: {accuracy*100:.1f}% of all predictions were correct")
print(f"• Precision: {precision*100:.1f}% of predicted defaults were actually defaults")
print(f"• Sensitivity: {sensitivity_recall*100:.1f}% of actual defaults were correctly identified")
print(f"• Specificity: {specificity*100:.1f}% of actual non-defaults were correctly identified")
print(f"• F1-Score: Harmonic mean of precision and recall = {f1:.3f}")

# Calculate AUC
auc = roc_auc_score(test_actual, test_probabilities)
print(f"AUC-ROC Score: {auc:.4f}")

# Generate ROC curve data
fpr, tpr, thresholds = roc_curve(test_actual, test_probabilities)

# Plot ROC curve
plt.figure(figsize=(10, 8))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC Curve (AUC = {auc:.3f})')
plt.plot([0, 1], [0, 1], color='red', lw=2, linestyle='--', label='Random Classifier (AUC = 0.5)')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate (1 - Specificity)')
plt.ylabel('True Positive Rate (Sensitivity)')
plt.title('ROC Curve - Credit Default Prediction Model')
plt.legend(loc="lower right")
plt.grid(True, alpha=0.3)
plt.show()

print(f"\n=== AUC INTERPRETATION ===")
if auc >= 0.9:
    interpretation = "Excellent"
elif auc >= 0.8:
    interpretation = "Good"
elif auc >= 0.7:
    interpretation = "Fair"
elif auc >= 0.6:
    interpretation = "Poor"
else:
    interpretation = "Very Poor"

print(f"AUC = {auc:.3f} indicates {interpretation} discriminatory ability")
```

```
Test set size: 9000
Number of actual defaults in test set: 2039
Number of predicted defaults: 738
Confusion Matrix:
[[6737  224]
 [1525  514]]

Breakdown:
True Negatives (TN): 6737
False Positives (FP): 224
False Negatives (FN): 1525
True Positives (TP): 514
=== MODEL PERFORMANCE METRICS ===
Accuracy: 0.8057 (80.57%)
Precision: 0.6965 (69.65%)
Sensitivity (Recall): 0.2521 (25.21%)
Specificity: 0.9678 (96.78%)
F1-Score: 0.3702

=== METRIC INTERPRETATIONS ===
• Accuracy: 80.6% of all predictions were correct
• Precision: 69.6% of predicted defaults were actually defaults
• Sensitivity: 25.2% of actual defaults were correctly identified
• Specificity: 96.8% of actual non-defaults were correctly identified
• F1-Score: Harmonic mean of precision and recall = 0.370
AUC-ROC Score: 0.7326
```




```
=== AUC INTERPRETATION ===
AUC = 0.733 indicates Fair discriminatory ability
```