Predicting Loan Payback 101

> Playground Series S5E11

Table of Contents

Competition Goal

Predict the **probability** that a borrower will pay back their loan based on:

- Financial metrics (income, debt, credit score)
- Loan characteristics (amount, interest rate, purpose)
- Personal information (gender, marital status, education, employment)

Evaluation: Area Under the ROC Curve (AUC-ROC)

1 Introduction & Setup

In [1]:
```
# Core libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
from scipy import stats
from scipy.stats import skew, kurtosis
from scipy.stats import rankdata

# Preprocessing
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import roc_auc_score, roc_curve, confusion_matrix

# Models
```

```
import lightgbm as lgb
import xgboost as xgb
from catboost import CatBoostClassifier

# Settings
warnings.filterwarnings('ignore')
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 100)
sns.set_style('whitegrid')
plt.rcParams['figure.figsize'] = (12, 6)

# Seed for reproducibility
SEED = 42
np.random.seed(SEED)
```

In [2]:
```
# Configuration
class Config:
    """Configuration class for hyperparameters and settings"""
    N_SPLITS = 5
    SEED = 42
    TARGET = 'loan_paid_back'
    VERBOSE = True

    # Model weights for ensemble (will be optimized later)
    WEIGHTS = {
        'lgb': 0.33,
        'xgb': 0.33,
        'cat': 0.34
    }

config = Config()
print("Configuration loaded successfully!")
print(f"   - Number of folds: {config.N_SPLITS}")
print(f"   - Random seed: {config.SEED}")
print(f"   - Target variable: {config.TARGET}")
```

```
Configuration loaded successfully!
   - Number of folds: 5
   - Random seed: 42
   - Target variable: loan_paid_back
```

2 Data Loading & Overview

In [3]:
```
# Load datasets
train = pd.read_csv('/kaggle/input/playground-series-s5e11/train.csv')
test = pd.read_csv('/kaggle/input/playground-series-s5e11/test.csv')
sample_submission = pd.read_csv('/kaggle/input/playground-series-s5e11/sample_su


print(f" Train shape: {train.shape}")
print(f" Test shape: {test.shape}")
print(f" Sample submission shape: {sample_submission.shape}")
print(f"\n Total rows: {train.shape[0]:,}")
print(f" Total features: {train.shape[1] - 2} (excluding id and target)")
print(f" Test samples to predict: {test.shape[0]:,}")
```

```
Train shape: (593994, 13)
Test shape: (254569, 12)
Sample submission shape: (254569, 2)

Total rows: 593,994
Total features: 11 (excluding id and target)
Test samples to predict: 254,569
```

In [4]:
```
# First look at the data
print("TRAIN DATA PREVIEW")
display(train.head(10))

print("TEST DATA PREVIEW")
display(test.head(10))
```

```
TRAIN DATA PREVIEW
```

	id	annual_income	debt_to_income_ratio	credit_score	loan_amount	interest_rate	gender
0	0	29367.99	0.084	736	2528.42	13.67	Female
1	1	22108.02	0.166	636	4593.10	12.92	Male
2	2	49566.20	0.097	694	17005.15	9.76	Male
3	3	46858.25	0.065	533	4682.48	16.10	Female
4	4	25496.70	0.053	665	12184.43	10.21	Male
5	5	44940.30	0.058	653	12159.92	12.24	Male
6	6	61574.16	0.042	696	16907.71	13.52	Other
7	7	45953.31	0.100	654	10111.62	12.82	Female
8	8	30592.29	0.132	713	7522.36	9.48	Male
9	9	17342.45	0.121	548	9653.48	16.04	Female

```
TEST DATA PREVIEW
```

	id	annual_income	debt_to_income_ratio	credit_score	loan_amount	interest_rate	ge
0	593994	28781.05	0.049	626	11461.42	14.73	Fe
1	593995	46626.39	0.093	732	15492.25	12.85	Fe
2	593996	54954.89	0.367	611	3796.41	13.29	
3	593997	25644.63	0.110	671	6574.30	9.57	Fe
4	593998	25169.64	0.081	688	17696.89	12.80	Fe
5	593999	45302.90	0.060	675	8106.78	13.74	Fe
6	594000	27676.47	0.061	714	8242.26	13.87	Fe
7	594001	38216.91	0.095	719	3765.50	15.10	
8	594002	25650.59	0.101	664	20310.64	11.74	
9	594003	62497.03	0.207	651	5177.58	13.90	Fe

In [5]:
```
# Data types and memory usage
print("\n" + "="*80)
print("DATA INFO - TRAIN")
print("="*80)
train.info()

print("\n" + "="*80)
print("DATA INFO - TEST")
print("="*80)
test.info()
```

```
================================================================================
DATA INFO - TRAIN
================================================================================
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 593994 entries, 0 to 593993
Data columns (total 13 columns):
 #   Column                Non-Null Count   Dtype  
---  ------                --------------   -----  
 0   id                    593994 non-null  int64  
 1   annual_income         593994 non-null  float64
 2   debt_to_income_ratio  593994 non-null  float64
 3   credit_score          593994 non-null  int64  
 4   loan_amount           593994 non-null  float64
 5   interest_rate         593994 non-null  float64
 6   gender                593994 non-null  object 
 7   marital_status        593994 non-null  object 
 8   education_level       593994 non-null  object 
 9   employment_status     593994 non-null  object 
 10  loan_purpose          593994 non-null  object 
 11  grade_subgrade        593994 non-null  object 
 12  loan_paid_back        593994 non-null  float64
dtypes: float64(5), int64(2), object(6)
memory usage: 58.9+ MB

================================================================================
DATA INFO - TEST
================================================================================
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 254569 entries, 0 to 254568
Data columns (total 12 columns):
 #   Column                Non-Null Count   Dtype  
---  ------                --------------   -----  
 0   id                    254569 non-null  int64  
 1   annual_income         254569 non-null  float64
 2   debt_to_income_ratio  254569 non-null  float64
 3   credit_score          254569 non-null  int64  
 4   loan_amount           254569 non-null  float64
 5   interest_rate         254569 non-null  float64
 6   gender                254569 non-null  object 
 7   marital_status        254569 non-null  object 
 8   education_level       254569 non-null  object 
 9   employment_status     254569 non-null  object 
 10  loan_purpose          254569 non-null  object 
 11  grade_subgrade        254569 non-null  object 
dtypes: float64(4), int64(2), object(6)
memory usage: 23.3+ MB
```

In [6]:

```python
# Check for missing values

missing_train = train.isnull().sum()
missing_test = test.isnull().sum()

missing_df = pd.DataFrame({
    'Feature': train.columns,
    'Train Missing': missing_train.values,
    'Train Missing %': (missing_train.values / len(train) * 100).round(2),
    'Test Missing': [missing_test.get(col, 0) for col in train.columns],
    'Test Missing %': [(missing_test.get(col, 0) / len(test) * 100) for col in t
})
```

```
missing_summary = missing_df[(missing_df['Train Missing'] > 0) | (missing_df['Te

if len(missing_summary) > 0:
    display(missing_summary.style.background_gradient(cmap='Reds'))
else:
    print(" No missing values found in train or test data!")
    print(" Data quality is excellent - ready for modeling!")
```

```
No missing values found in train or test data!
Data quality is excellent - ready for modeling!
```

In [7]:
```
# Check for duplicates
train_duplicates = train.duplicated().sum()
test_duplicates = test.duplicated().sum()

print(f"Train duplicates: {train_duplicates}")
print(f"Test duplicates: {test_duplicates}")

if train_duplicates == 0 and test_duplicates == 0:
    print(" No duplicates found!")
```

```
Train duplicates: 0
Test duplicates: 0
 No duplicates found!
```

In [8]:
```
# Target distribution
print("TARGET DISTRIBUTION ANALYSIS")
print("="*80)

target_counts = train[config.TARGET].value_counts()
target_pct = train[config.TARGET].value_counts(normalize=True) * 100

target_summary = pd.DataFrame({
    'Value': target_counts.index,
    'Count': target_counts.values,
    'Percentage': target_pct.values
})

display(target_summary.style.background_gradient(cmap='Blues'))

print(f"\nTarget Statistics:")
print(f"   - Mean: {train[config.TARGET].mean():.4f}")
print(f"   - Median: {train[config.TARGET].median():.4f}")
print(f"   - Std: {train[config.TARGET].std():.4f}")

# Check for class imbalance
imbalance_ratio = target_counts.min() / target_counts.max()
print(f"\n  Class Balance Ratio: {imbalance_ratio:.3f}")
if imbalance_ratio < 0.5:
    print("  Dataset is imbalanced - consider using stratified sampling")
else:
    print(" Dataset is relatively balanced")
```

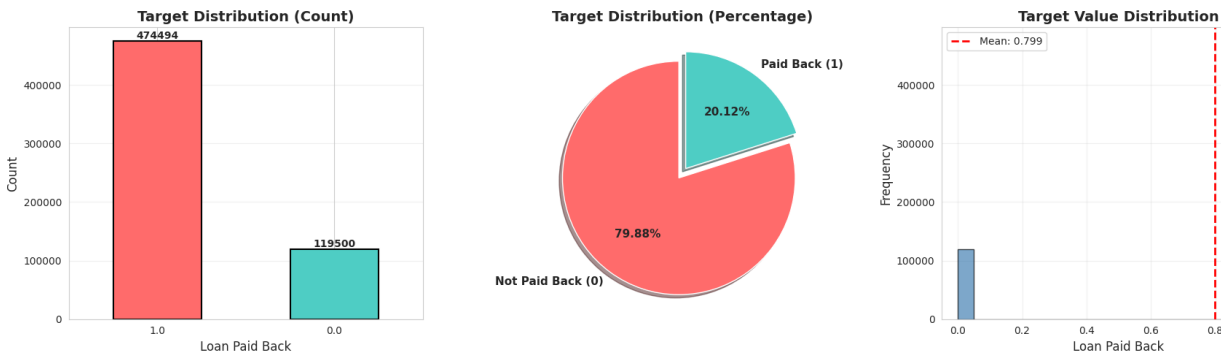
```
TARGET DISTRIBUTION ANALYSIS
================================================================================
```

	Value	Count	Percentage
0	1.000000	474494	79.881952
1	0.000000	119500	20.118048

```
Target Statistics:
   - Mean: 0.7988
   - Median: 1.0000
   - Std: 0.4009

 Class Balance Ratio: 0.252
 Dataset is imbalanced - consider using stratified sampling
```

In [9]:

```python
# Visualize target distribution
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

# Count plot
colors = ['#FF6B6B', '#4ECDC4']
train[config.TARGET].value_counts().plot(kind='bar', ax=axes[0], color=colors, e
axes[0].set_title('Target Distribution (Count)', fontsize=14, fontweight='bold')
axes[0].set_xlabel('Loan Paid Back', fontsize=12)
axes[0].set_ylabel('Count', fontsize=12)
axes[0].tick_params(rotation=0)
axes[0].grid(alpha=0.3, axis='y')

# Add value labels on bars
for container in axes[0].containers:
    axes[0].bar_label(container, fmt='%d', fontsize=10, fontweight='bold')

# Pie chart
explode = (0.05, 0.05)
axes[1].pie(train[config.TARGET].value_counts(),
            labels=['Not Paid Back (0)', 'Paid Back (1)'],
            autopct='%1.2f%%',
            colors=colors,
            explode=explode,
            shadow=True,
            startangle=90,
            textprops={'fontsize': 11, 'fontweight': 'bold'})
axes[1].set_title('Target Distribution (Percentage)', fontsize=14, fontweight='b

# Distribution plot
axes[2].hist(train[config.TARGET], bins=20, color='steelblue', edgecolor='black'
axes[2].axvline(train[config.TARGET].mean(), color='red', linestyle='--', linewi
axes[2].set_title('Target Value Distribution', fontsize=14, fontweight='bold')
axes[2].set_xlabel('Loan Paid Back', fontsize=12)
axes[2].set_ylabel('Frequency', fontsize=12)
axes[2].legend(fontsize=10)
axes[2].grid(alpha=0.3)

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



3 Exploratory Data Analysis (EDA)

3.1 Feature Type Identification

In [10]:
```
# Separate numerical and categorical columns
numerical_cols = train.select_dtypes(include=[np.number]).columns.tolist()
numerical_cols.remove('id')
if config.TARGET in numerical_cols:
    numerical_cols.remove(config.TARGET)

categorical_cols = train.select_dtypes(include=['object']).columns.tolist()

print("FEATURE TYPE SUMMARY")
print("="*80)
print(f"\n Numerical features ({len(numerical_cols)}):")
for i, col in enumerate(numerical_cols, 1):
    print(f"   {i}. {col}")

print(f"\n Categorical features ({len(categorical_cols)}):")
for i, col in enumerate(categorical_cols, 1):
    print(f"   {i}. {col}")

print(f"\n Total predictive features: {len(numerical_cols) + len(categorical_col
```

```
FEATURE TYPE SUMMARY
================================================================================

 Numerical features (5):
   1. annual_income
   2. debt_to_income_ratio
   3. credit_score
   4. loan_amount
   5. interest_rate

 Categorical features (6):
   1. gender
   2. marital_status
   3. education_level
   4. employment_status
   5. loan_purpose
   6. grade_subgrade

 Total predictive features: 11
```

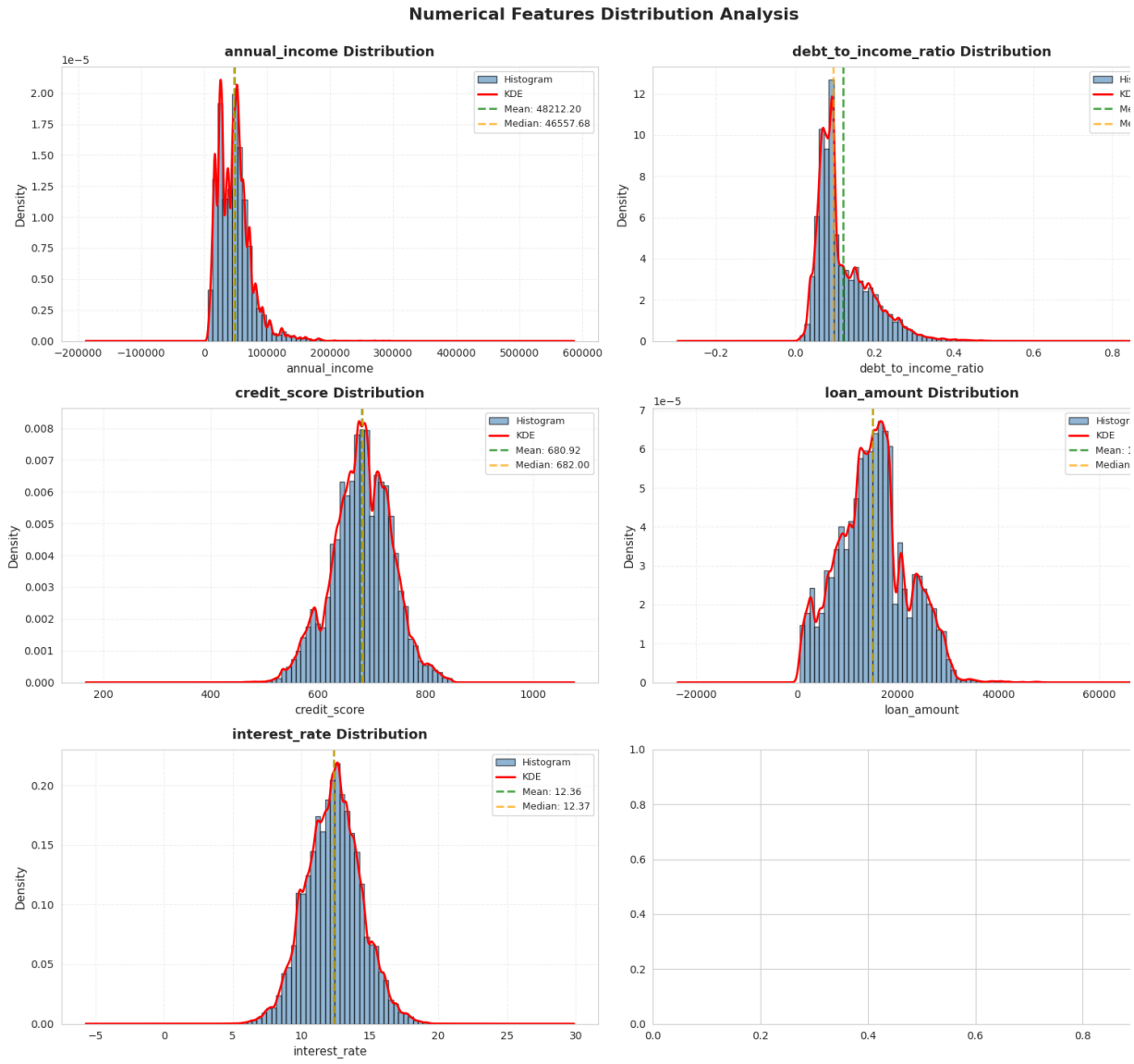
3.2 Numerical Features Analysis

In [11]:
```
# Statistical summary of numerical features
print("NUMERICAL FEATURES - STATISTICAL SUMMARY")
print("="*80)

numerical_stats = train[numerical_cols].describe().T
numerical_stats['missing'] = train[numerical_cols].isnull().sum().values
numerical_stats['skewness'] = train[numerical_cols].skew().values
numerical_stats['kurtosis'] = train[numerical_cols].kurtosis().values

display(numerical_stats.style.background_gradient(cmap='coolwarm', subset=['mean
```

```
NUMERICAL FEATURES - STATISTICAL SUMMARY
================================================================================
```

	count	mean	std	min	25%
annual_income	593994.000000	48212.202976	26711.942078	6002.430000	27934.400000
debt_to_income_ratio	593994.000000	0.120696	0.068573	0.011000	0.072000
credit_score	593994.000000	680.916009	55.424956	395.000000	646.000000
loan_amount	593994.000000	15020.297629	6926.530568	500.090000	10279.620000
interest_rate	593994.000000	12.356345	2.008959	3.200000	10.990000

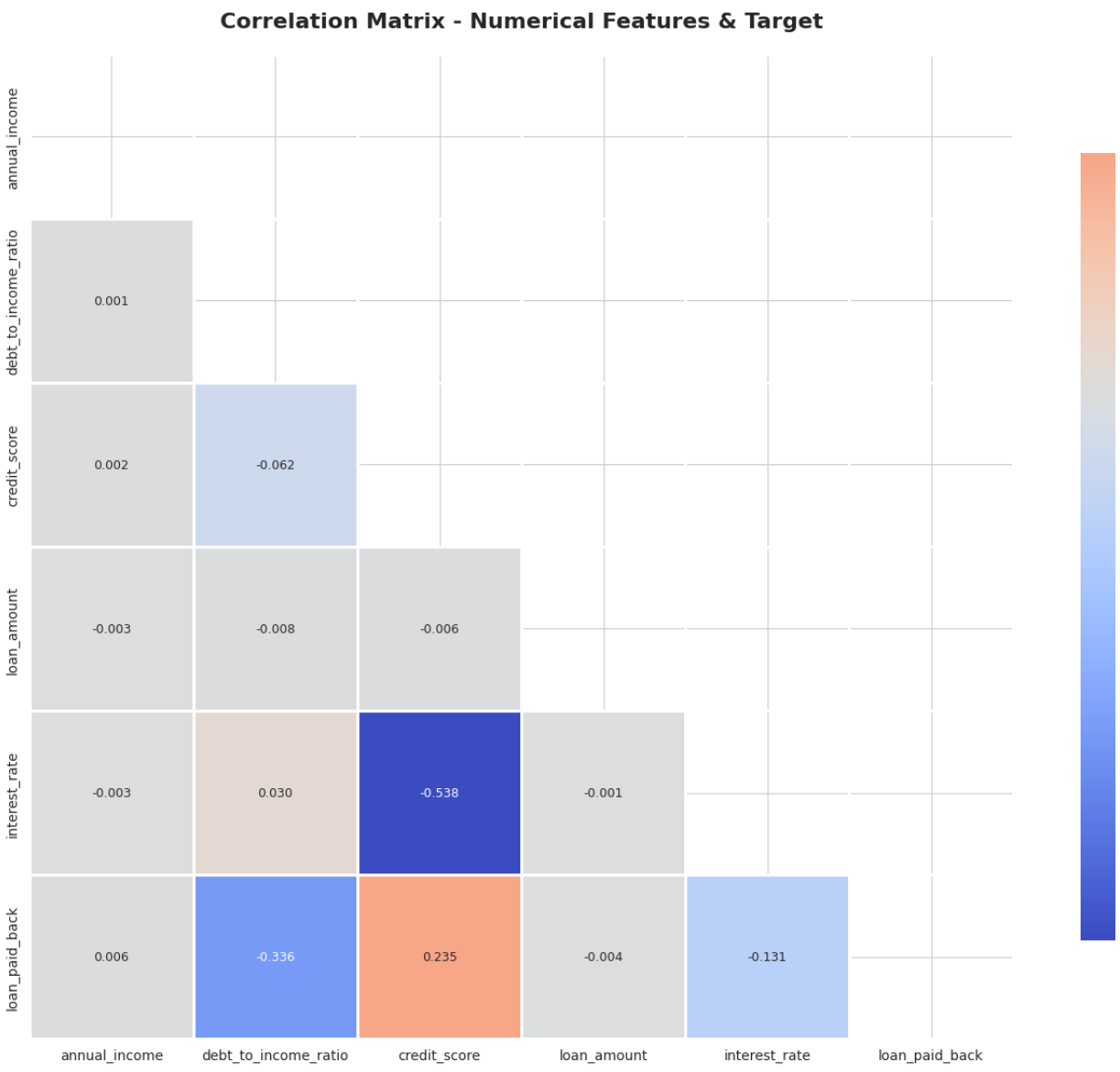
In [12]:
```
# Distribution of numerical features
fig, axes = plt.subplots(3, 2, figsize=(16, 14))
axes = axes.flatten()

for idx, col in enumerate(numerical_cols):
    # Plot distribution with KDE
    axes[idx].hist(train[col], bins=50, alpha=0.6, color='steelblue', edgecolor=

    # Add KDE
    train[col].plot(kind='kde', ax=axes[idx], color='red', linewidth=2, label='K

    axes[idx].set_title(f'{col} Distribution', fontsize=13, fontweight='bold', p
    axes[idx].set_xlabel(col, fontsize=11)
    axes[idx].set_ylabel('Density', fontsize=11)
    axes[idx].grid(alpha=0.3, linestyle='--')

    # Add statistics box
    mean_val = train[col].mean()
    median_val = train[col].median()
    std_val = train[col].std()

    axes[idx].axvline(mean_val, color='green', linestyle='--', linewidth=2, alph
    axes[idx].axvline(median_val, color='orange', linestyle='--', linewidth=2, a

    axes[idx].legend(fontsize=9, loc='upper right')

plt.suptitle('Numerical Features Distribution Analysis', fontsize=16, fontweight
plt.tight_layout()
plt.show()
```



In [13]:
```python
# Correlation analysis
print("CORRELATION ANALYSIS")
print("="*80)

plt.figure(figsize=(14, 11))
correlation_matrix = train[numerical_cols + [config.TARGET]].corr()

# Create mask for upper triangle
mask = np.triu(np.ones_like(correlation_matrix, dtype=bool))

# Create heatmap
sns.heatmap(correlation_matrix,
            mask=mask,
            annot=True,
            fmt='.3f',
            cmap='coolwarm',
            center=0,
            square=True,
            linewidths=1,
            cbar_kws={"shrink": 0.8, "label": "Correlation Coefficient"},
            annot_kws={"size": 9})

plt.title('Correlation Matrix - Numerical Features & Target', fontsize=16, fontw
plt.tight_layout()
plt.show()
```

```
# Feature importance based on correlation with target
target_corr = correlation_matrix[config.TARGET].drop(config.TARGET).sort_values(

print("\n Correlation with Target (sorted by absolute value):")
target_corr_abs = target_corr.abs().sort_values(ascending=False)
for feature, corr_val in target_corr_abs.items():
    actual_corr = target_corr[feature]
    print(f"   {feature:30s}: {actual_corr:7.4f} (abs: {corr_val:.4f})")
```

```
CORRELATION ANALYSIS
================================================================================
```

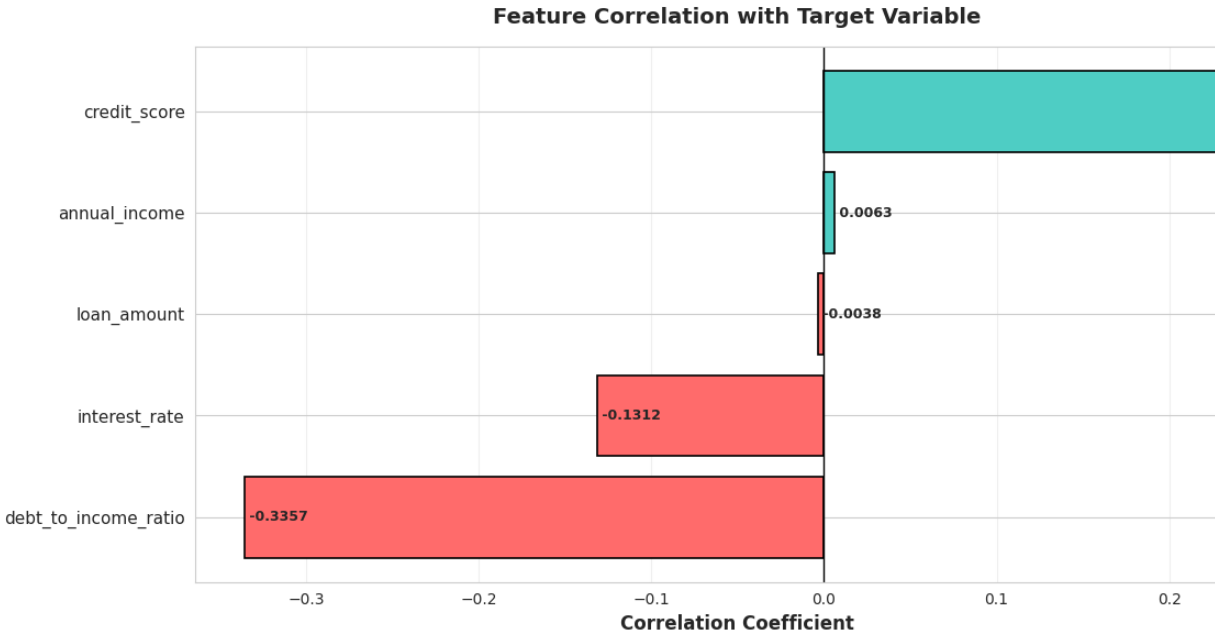



```
Correlation with Target (sorted by absolute value):
  debt_to_income_ratio          : -0.3357 (abs: 0.3357)
  credit_score                  :  0.2346 (abs: 0.2346)
  interest_rate                 : -0.1312 (abs: 0.1312)
  annual_income                 :  0.0063 (abs: 0.0063)
  loan_amount                   : -0.0038 (abs: 0.0038)
```

In [14]:

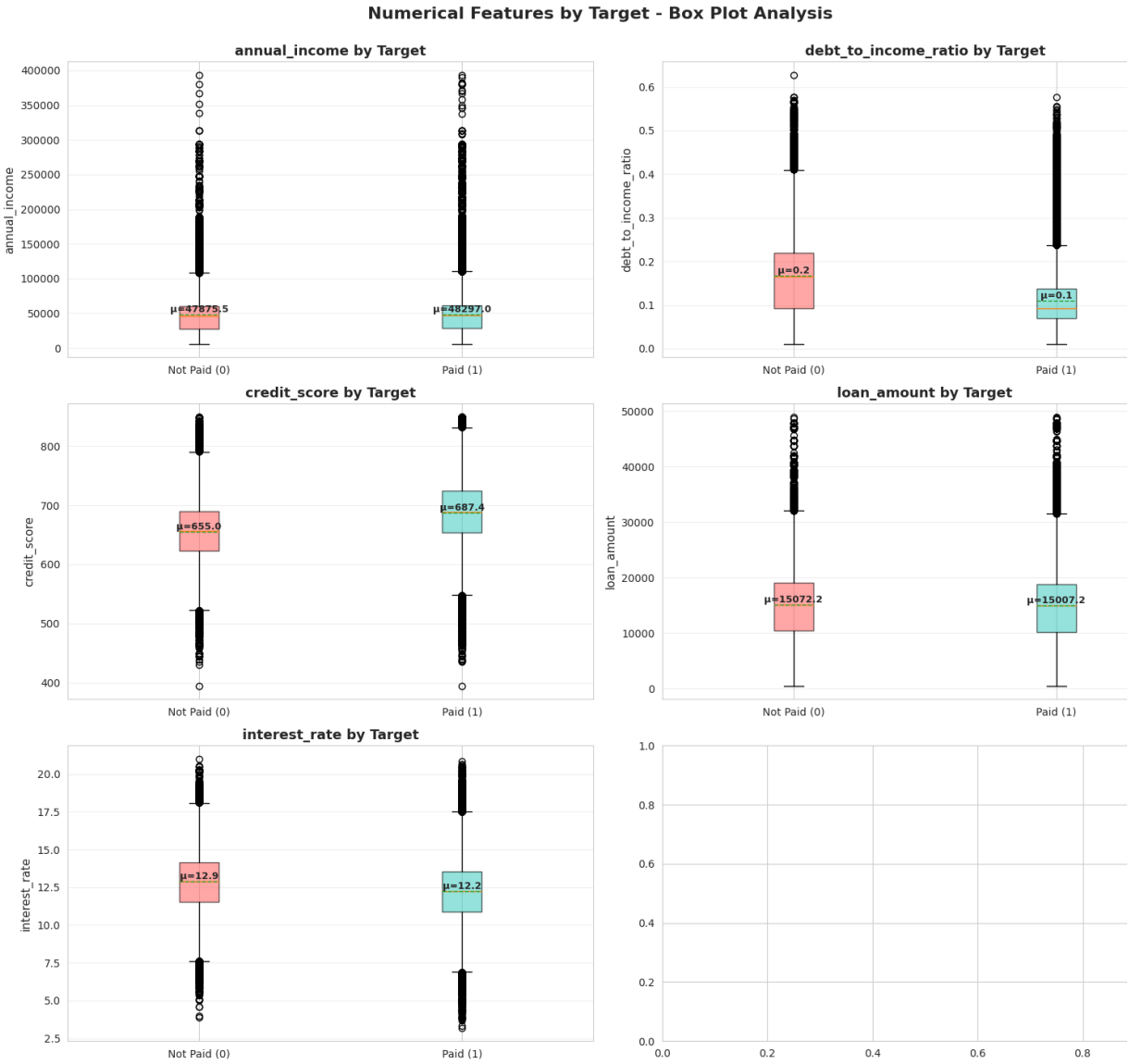
```
# Visualize correlation with target
plt.figure(figsize=(12, 6))
target_corr_df = target_corr.sort_values()
colors_list = ['#FF6B6B' if x < 0 else '#4ECDC4' for x in target_corr_df.values]

plt.barh(range(len(target_corr_df)), target_corr_df.values, color=colors_list, e
plt.yticks(range(len(target_corr_df)), target_corr_df.index, fontsize=11)
```

```
plt.xlabel('Correlation Coefficient', fontsize=12, fontweight='bold')
plt.title('Feature Correlation with Target Variable', fontsize=14, fontweight='b
plt.axvline(x=0, color='black', linestyle='-', linewidth=1)
plt.grid(alpha=0.3, axis='x')

# Add value labels
for i, v in enumerate(target_corr_df.values):
    plt.text(v, i, f' {v:.4f}', va='center', fontsize=9, fontweight='bold')

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



In [15]:
```
# Box plots by target - showing distribution differences
fig, axes = plt.subplots(3, 2, figsize=(16, 14))
axes = axes.flatten()

for idx, col in enumerate(numerical_cols):
    # Create box plot
    data_to_plot = [train[train[config.TARGET] == 0][col].dropna(),
                    train[train[config.TARGET] == 1][col].dropna()]

    bp = axes[idx].boxplot(data_to_plot,
                           labels=['Not Paid (0)', 'Paid (1)'],
                           patch_artist=True,
                           showmeans=True,
                           meanline=True)

    # Color the boxes
    colors = ['#FF6B6B', '#4ECDC4']
    for patch, color in zip(bp['boxes'], colors):
        patch.set_facecolor(color)
        patch.set_alpha(0.6)

    axes[idx].set_title(f'{col} by Target', fontsize=13, fontweight='bold')
    axes[idx].set_ylabel(col, fontsize=11)
    axes[idx].grid(alpha=0.3, axis='y')

    # Add mean values as text
    mean_0 = train[train[config.TARGET] == 0][col].mean()
    mean_1 = train[train[config.TARGET] == 1][col].mean()
```

```
    axes[idx].text(1, mean_0, f'μ={mean_0:.1f}', ha='center', va='bottom', fonts
    axes[idx].text(2, mean_1, f'μ={mean_1:.1f}', ha='center', va='bottom', fonts

plt.suptitle('Numerical Features by Target - Box Plot Analysis', fontsize=16, fc
plt.tight_layout()
plt.show()
```



3.3 Categorical Features Analysis

In [16]:

```
# Categorical features summary
print("CATEGORICAL FEATURES - DETAILED ANALYSIS")
print("="*80)

for col in categorical_cols:
    print(f"Feature: {col.upper()}")
    print(f"{'='*80}")
    print(f"Unique values: {train[col].nunique()}")
    print(f"Most common: {train[col].mode()[0]}")
    print(f"\nValue Counts:")

    value_counts_df = pd.DataFrame({
        'Value': train[col].value_counts().index,
        'Count': train[col].value_counts().values,
        'Percentage': (train[col].value_counts(normalize=True) * 100).values
```

```
    })
    display(value_counts_df.head(10).style.background_gradient(cmap='Blues', sub
```

```
CATEGORICAL FEATURES - DETAILED ANALYSIS
================================================================================
Feature: GENDER
================================================================================
Unique values: 3
Most common: Female

Value Counts:
```

	Value	Count	Percentage
0	Female	306175	51.545133
1	Male	284091	47.827251
2	Other	3728	0.627616

```
Feature: MARITAL_STATUS
================================================================================
Unique values: 4
Most common: Single

Value Counts:
```

	Value	Count	Percentage
0	Single	288843	48.627259
1	Married	277239	46.673704
2	Divorced	21312	3.587915
3	Widowed	6600	1.111122

```
Feature: EDUCATION_LEVEL
================================================================================
Unique values: 5
Most common: Bachelor's

Value Counts:
```

	Value	Count	Percentage
0	Bachelor's	279606	47.072193
1	High School	183592	30.908056
2	Master's	93097	15.673054
3	Other	26677	4.491123
4	PhD	11022	1.855574

```
Feature: EMPLOYMENT_STATUS
================================================================================
Unique values: 5
Most common: Employed

Value Counts:
```

	Value	Count	Percentage
0	Employed	450645	75.866928
1	Unemployed	62485	10.519467
2	Self-employed	52480	8.835106
3	Retired	16453	2.769893
4	Student	11931	2.008606

```
Feature: LOAN_PURPOSE
================================================================================
Unique values: 8
Most common: Debt consolidation

Value Counts:
```

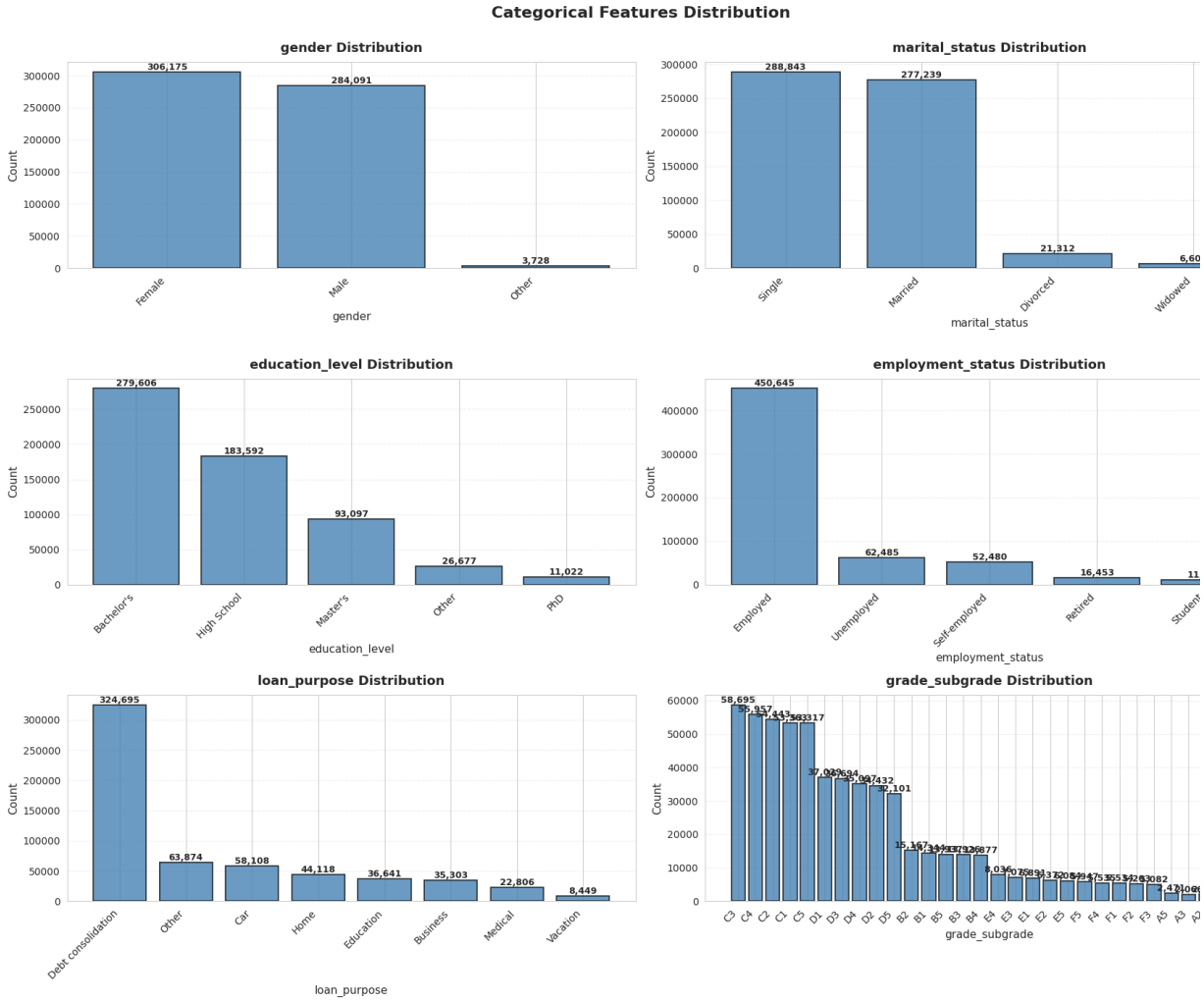
	Value	Count	Percentage
0	Debt consolidation	324695	54.663010
1	Other	63874	10.753307
2	Car	58108	9.782590
3	Home	44118	7.427348
4	Education	36641	6.168581
5	Business	35303	5.943326
6	Medical	22806	3.839433
7	Vacation	8449	1.422405

```
Feature: GRADE_SUBGRADE
================================================================================
Unique values: 30
Most common: C3

Value Counts:
```

	Value	Count	Percentage
0	C3	58695	9.881413
1	C4	55957	9.420466
2	C2	54443	9.165581
3	C1	53363	8.983761
4	C5	53317	8.976017
5	D1	37029	6.233901
6	D3	36694	6.177503
7	D4	35097	5.908646
8	D2	34432	5.796692
9	D5	32101	5.404263

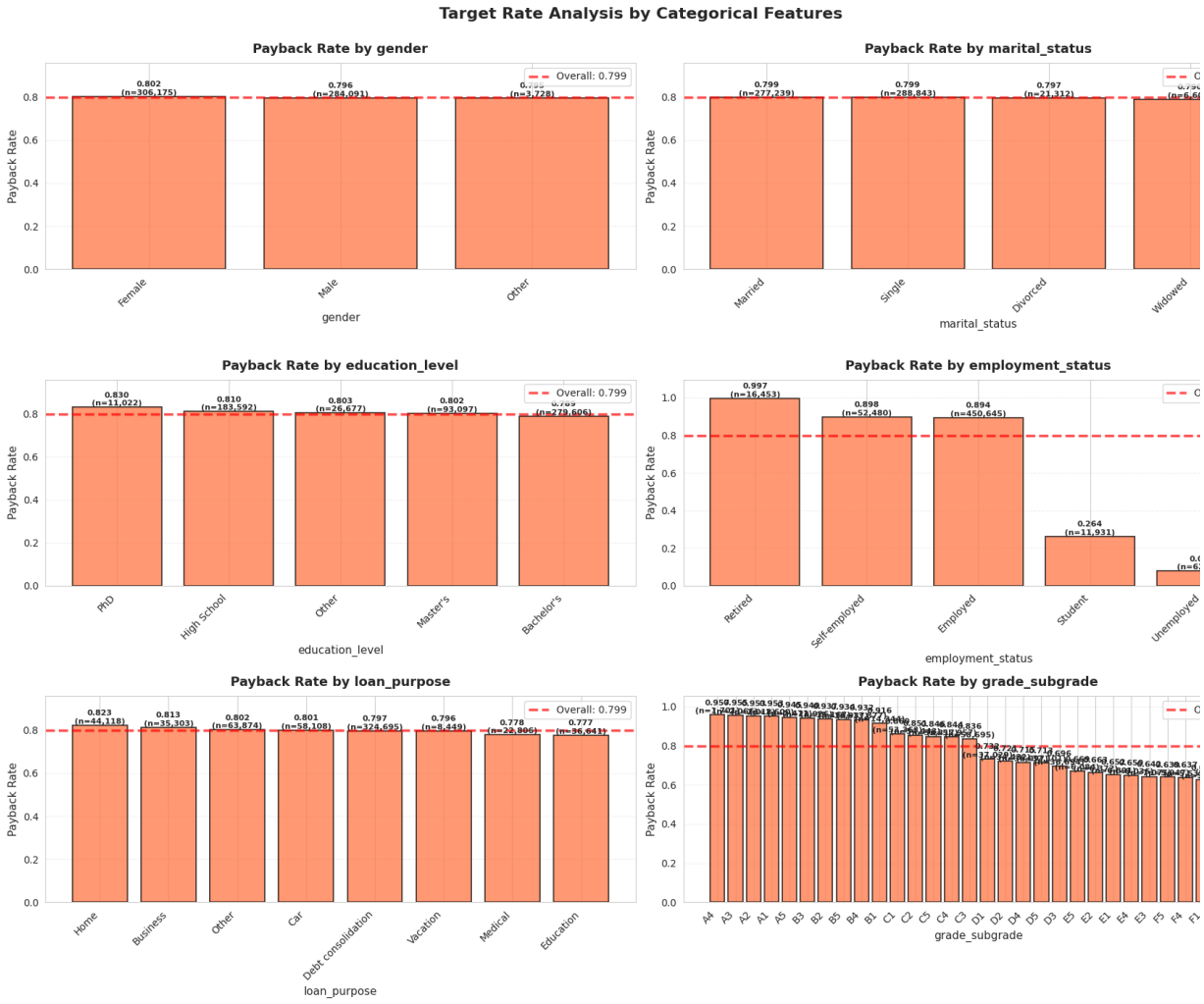
In [17]:

```
# Visualize categorical features distribution
fig, axes = plt.subplots(3, 2, figsize=(18, 14))
axes = axes.flatten()

for idx, col in enumerate(categorical_cols):
    value_counts = train[col].value_counts()

    # Create bar plot
    bars = axes[idx].bar(range(len(value_counts)), value_counts.values,
                         color='steelblue', edgecolor='black', linewidth=1.2, al

    axes[idx].set_title(f'{col} Distribution', fontsize=13, fontweight='bold', p
    axes[idx].set_xlabel(col, fontsize=11)
    axes[idx].set_ylabel('Count', fontsize=11)
    axes[idx].set_xticks(range(len(value_counts)))
    axes[idx].set_xticklabels(value_counts.index, rotation=45, ha='right')
    axes[idx].grid(alpha=0.3, axis='y', linestyle='--')

    # Add value labels on bars
    for bar in bars:
        height = bar.get_height()
        axes[idx].text(bar.get_x() + bar.get_width()/2., height,
                       f'{int(height):,}',
                       ha='center', va='bottom', fontsize=9, fontweight='bold')

plt.suptitle('Categorical Features Distribution', fontsize=16, fontweight='bold'
plt.tight_layout()
plt.show()
```



In [18]:

```
# Target rate by categorical features
fig, axes = plt.subplots(3, 2, figsize=(18, 14))
axes = axes.flatten()

overall_mean = train[config.TARGET].mean()

for idx, col in enumerate(categorical_cols):
    target_rate = train.groupby(col)[config.TARGET].agg(['mean', 'count']).sort_

    # Create bar plot
    bars = axes[idx].bar(range(len(target_rate)), target_rate['mean'].values,
                         color='coral', edgecolor='black', linewidth=1.2, alpha=

    axes[idx].set_title(f'Payback Rate by {col}', fontsize=13, fontweight='bold'
    axes[idx].set_xlabel(col, fontsize=11)
    axes[idx].set_ylabel('Payback Rate', fontsize=11)
    axes[idx].set_xticks(range(len(target_rate)))
    axes[idx].set_xticklabels(target_rate.index, rotation=45, ha='right')
    axes[idx].axhline(y=overall_mean, color='red', linestyle='--',
                      linewidth=2.5, alpha=0.7, label=f'Overall: {overall_mean:.
    axes[idx].legend(fontsize=10, loc='best')
    axes[idx].grid(alpha=0.3, axis='y', linestyle='--')
    axes[idx].set_ylim([0, max(target_rate['mean'].max() * 1.1, overall_mean * 1

    # Add value labels with sample counts
    for i, (bar, count) in enumerate(zip(bars, target_rate['count'].values)):
        height = bar.get_height()
        axes[idx].text(bar.get_x() + bar.get_width()/2., height,
                       f'{height:.3f}\n(n={count:,})',
                       ha='center', va='bottom', fontsize=8, fontweight='bold')

plt.suptitle('Target Rate Analysis by Categorical Features', fontsize=16, fontwe
plt.tight_layout()
plt.show()
```



3.4 Advanced Statistical Analysis

In [19]:
```
# Skewness and Kurtosis detailed analysis
print("SKEWNESS AND KURTOSIS ANALYSIS")
print("="*80)

skew_kurt_df = pd.DataFrame({
    'Feature': numerical_cols,
    'Skewness': [skew(train[col]) for col in numerical_cols],
    'Kurtosis': [kurtosis(train[col]) for col in numerical_cols],
    'Min': [train[col].min() for col in numerical_cols],
    'Max': [train[col].max() for col in numerical_cols],
    'Range': [train[col].max() - train[col].min() for col in numerical_cols]
})

# Add interpretation
skew_kurt_df['Skew_Interpretation'] = skew_kurt_df['Skewness'].apply(
    lambda x: 'Right-skewed' if x > 1 else ('Left-skewed' if x < -1 else 'Symmet
)

skew_kurt_df = skew_kurt_df.sort_values('Skewness', key=abs, ascending=False)

display(skew_kurt_df.style.background_gradient(cmap='coolwarm', subset=['Skewnes

print("\n Interpretation Guide:")
print("    Skewness:")
print("      - Close to 0: Symmetric distribution (Normal-like)")
print("      - > 1: Highly right-skewed (long tail on right)")
print("      - < -1: Highly left-skewed (long tail on left)")
```

```
print("\n   Kurtosis:")
print("      - Close to 0: Normal-like tails")
print("      - > 0: Heavy tails (more outliers)")
print("      - < 0: Light tails (fewer outliers)")
```

```
SKEWNESS AND KURTOSIS ANALYSIS
================================================================================
```

	Feature	Skewness	Kurtosis	Min	Max	Range	Ske
0	annual_income	1.719504	7.091343	6002.430000	393381.740000	387379.310000	
1	debt_to_income_ratio	1.406676	2.335200	0.011000	0.627000	0.616000	
3	loan_amount	0.207359	-0.150151	500.090000	48959.950000	48459.860000	
2	credit_score	-0.166992	0.095951	395.000000	849.000000	454.000000	
4	interest_rate	0.049945	0.059787	3.200000	20.990000	17.790000	

```
Interpretation Guide:
   Skewness:
     - Close to 0: Symmetric distribution (Normal-like)
     - > 1: Highly right-skewed (long tail on right)
     - < -1: Highly left-skewed (long tail on left)

  Kurtosis:
     - Close to 0: Normal-like tails
     - > 0: Heavy tails (more outliers)
     - < 0: Light tails (fewer outliers)
```

In [20]:
```
# Outlier detection using IQR method
print("OUTLIER DETECTION (IQR METHOD)")
print("="*80)

outlier_summary = []

for col in numerical_cols:
    Q1 = train[col].quantile(0.25)
    Q3 = train[col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    outliers = train[(train[col] < lower_bound) | (train[col] > upper_bound)][cc
    outlier_count = len(outliers)
    outlier_pct = (outlier_count / len(train)) * 100

    outlier_summary.append({
        'Feature': col,
        'Q1': Q1,
        'Q3': Q3,
        'IQR': IQR,
        'Lower_Bound': lower_bound,
        'Upper_Bound': upper_bound,
        'Outlier_Count': outlier_count,
        'Outlier_Percentage': outlier_pct
    })

outlier_df = pd.DataFrame(outlier_summary).sort_values('Outlier_Percentage', asc
display(outlier_df.style.background_gradient(cmap='Reds', subset=['Outlier_Count
```

```
print(f"\n Outlier Summary:")
print(f"   - Total features with outliers: {(outlier_df['Outlier_Count'] > 0).su
print(f"   - Average outlier percentage: {outlier_df['Outlier_Percentage'].mean(
```

```
OUTLIER DETECTION (IQR METHOD)
================================================================================
```

	Feature	Q1	Q3	IQR	Lower_Bound	Upper_Bou
1	debt_to_income_ratio	0.072000	0.156000	0.084000	-0.054000	0.2820
0	annual_income	27934.400000	60981.320000	33046.920000	-21635.980000	110551.7000
2	credit_score	646.000000	719.000000	73.000000	536.500000	828.5000
4	interest_rate	10.990000	13.680000	2.690000	6.955000	17.7150
3	loan_amount	10279.620000	18858.580000	8578.960000	-2588.820000	31727.0200

```
Outlier Summary:
  - Total features with outliers: 5
  - Average outlier percentage: 1.60%
```

4 Feature Engineering

4.1 Advanced Feature Engineering Function

In [21]:

```python
def advanced_feature_engineering(df, is_train=True):
    """
    Comprehensive feature engineering for loan prediction

    This function creates multiple types of features:
    - Financial ratios and metrics
    - Risk scores and composite metrics
    - Interaction features
    - Binned/categorical versions of numerical features
    - Statistical aggregations
    - Domain-specific features

    Parameters:
    -----------
    df : pd.DataFrame
        Input dataframe (train or test)
    is_train : bool
        Whether this is training data

    Returns:
    --------
    df : pd.DataFrame
        Dataframe with engineered features
    """

    df = df.copy()

    print("FEATURE ENGINEERING PIPELINE")
    print("="*80)
    print(f"Starting features: {df.shape[1]}")

    # ========================================
    # 1. FINANCIAL RATIO FEATURES
```

```python
# ========================================
print("\n[1/11]  Creating financial ratio features...")

# Core financial ratios
df['loan_to_income_ratio'] = df['loan_amount'] / (df['annual_income'] + 1)
df['monthly_income'] = df['annual_income'] / 12
df['monthly_payment_estimate'] = (df['loan_amount'] * df['interest_rate']) /
df['payment_to_income_ratio'] = df['monthly_payment_estimate'] / (df['monthl

# Debt calculations
df['current_debt_amount'] = df['debt_to_income_ratio'] * df['annual_income']
df['total_debt_with_loan'] = df['current_debt_amount'] + df['loan_amount']
df['new_debt_to_income'] = df['total_debt_with_loan'] / (df['annual_income']
df['debt_increase_ratio'] = df['new_debt_to_income'] / (df['debt_to_income_r
df['debt_increase_amount'] = df['loan_amount']

# Disposable income
df['disposable_income'] = df['annual_income'] - df['current_debt_amount']
df['disposable_income_ratio'] = df['disposable_income'] / (df['annual_income
df['loan_to_disposable_income'] = df['loan_amount'] / (df['disposable_income
df['monthly_disposable_income'] = df['disposable_income'] / 12

# Payment burden
df['payment_to_disposable_ratio'] = df['monthly_payment_estimate'] / (df['mo
df['annual_payment_burden'] = df['monthly_payment_estimate'] * 12
df['payment_burden_ratio'] = df['annual_payment_burden'] / (df['annual_incom

print(f"✓ Created 16 financial ratio features")

# ========================================
# 2. CREDIT SCORE FEATURES
# ========================================
print("\n[2/11]  Creating credit score features...")

# Normalize and transform credit score
df['credit_score_normalized'] = df['credit_score'] / 850
df['credit_risk_score'] = 1 - df['credit_score_normalized']
df['credit_score_squared'] = df['credit_score'] ** 2
df['credit_score_log'] = np.log1p(df['credit_score'])

# Credit categories
df['credit_category'] = pd.cut(df['credit_score'],
                               bins=[0, 580, 670, 740, 800, 850],
                               labels=['poor', 'fair', 'good', 'very_good'

# Credit score bins
df['credit_bin'] = pd.cut(df['credit_score'], bins=10, labels=False)

# Interactions with other features
df['credit_income_interaction'] = df['credit_score'] * df['annual_income']
df['credit_times_dti'] = df['credit_score'] * df['debt_to_income_ratio']
df['credit_loan_interaction'] = df['credit_score'] * df['loan_amount']

print(f"✓ Created 9 credit score features")

# ========================================
# 3. INTEREST RATE FEATURES
# ========================================
print("\n[3/11]  Creating interest rate features...")
```

```python
    # Interest rate flags and categories
    df['high_interest_flag'] = (df['interest_rate'] > df['interest_rate'].median
    df['very_high_interest'] = (df['interest_rate'] > df['interest_rate'].quanti
    df['low_interest_flag'] = (df['interest_rate'] < df['interest_rate'].quantil

    # Interest cost calculations
    df['total_interest_cost'] = df['loan_amount'] * df['interest_rate'] / 100
    df['interest_burden'] = df['total_interest_cost'] / (df['annual_income'] + 1
    df['monthly_interest_cost'] = df['total_interest_cost'] / 12

    # Interest rate vs credit score (should be inversely related)
    df['interest_credit_mismatch'] = df['interest_rate'] * (1 - df['credit_score
    df['interest_credit_ratio'] = df['interest_rate'] / (df['credit_score'] / 10

    # Interest rate transformations
    df['interest_rate_squared'] = df['interest_rate'] ** 2
    df['interest_rate_log'] = np.log1p(df['interest_rate'])

    print(f"✓ Created 10 interest rate features")

    # ========================================
    # 4. COMPOSITE RISK SCORES
    # ========================================
    print("\n[4/11]   Creating composite risk scores...")

    # Multi-factor risk scores (weighted combinations)
    df['risk_score_v1'] = (
        df['debt_to_income_ratio'] * 0.25 +
        df['loan_to_income_ratio'] * 0.25 +
        df['credit_risk_score'] * 0.30 +
        (df['interest_rate'] / 100) * 0.20
    )

    df['risk_score_v2'] = (
        df['payment_to_income_ratio'] * 0.40 +
        df['new_debt_to_income'] * 0.35 +
        df['interest_burden'] * 0.25
    )

    df['risk_score_v3'] = (
        df['debt_to_income_ratio'] * 0.30 +
        df['payment_burden_ratio'] * 0.30 +
        df['credit_risk_score'] * 0.40
    )

    # Affordability score (higher is better)
    df['affordability_score'] = (
        df['credit_score_normalized'] * 0.40 +
        (1 - df['debt_to_income_ratio']) * 0.30 +
        df['disposable_income_ratio'] * 0.30
    )

    # Financial health score
    df['financial_health_score'] = (
        df['affordability_score'] * 0.60 -
        df['risk_score_v1'] * 0.40
    )

    print(f"✓ Created 5 composite risk scores")
```

```
    # Continue in next cell...
    return df
```

5 Complete Feature Engineering

In [22]:
```
def complete_feature_engineering(df):
    """
    Comprehensive feature engineering pipeline for loan prediction
    """
    df = df.copy()

    print(" FEATURE ENGINEERING PIPELINE")
    print("="*80)
    print(f"Starting features: {df.shape[1]}")

    # 1. FINANCIAL RATIOS
    print("\n[1/11]  Financial ratio features...")
    df['loan_to_income_ratio'] = df['loan_amount'] / (df['annual_income'] + 1)
    df['monthly_income'] = df['annual_income'] / 12
    df['monthly_payment_estimate'] = (df['loan_amount'] * df['interest_rate']) /
    df['payment_to_income_ratio'] = df['monthly_payment_estimate'] / (df['monthl
    df['current_debt_amount'] = df['debt_to_income_ratio'] * df['annual_income']
    df['total_debt_with_loan'] = df['current_debt_amount'] + df['loan_amount']
    df['new_debt_to_income'] = df['total_debt_with_loan'] / (df['annual_income']
    df['debt_increase_ratio'] = df['new_debt_to_income'] / (df['debt_to_income_r
    df['disposable_income'] = df['annual_income'] - df['current_debt_amount']
    df['disposable_income_ratio'] = df['disposable_income'] / (df['annual_income
    df['loan_to_disposable_income'] = df['loan_amount'] / (df['disposable_income
    df['monthly_disposable_income'] = df['disposable_income'] / 12
    df['payment_to_disposable_ratio'] = df['monthly_payment_estimate'] / (df['mo
    df['annual_payment_burden'] = df['monthly_payment_estimate'] * 12
    df['payment_burden_ratio'] = df['annual_payment_burden'] / (df['annual_incom
    print(f"✓ Created 15 features")

    # 2. CREDIT SCORE FEATURES
    print("[2/11]  Credit score features...")
    df['credit_score_normalized'] = df['credit_score'] / 850
    df['credit_risk_score'] = 1 - df['credit_score_normalized']
    df['credit_score_squared'] = df['credit_score'] ** 2
    df['credit_score_log'] = np.log1p(df['credit_score'])
    df['credit_category'] = pd.cut(df['credit_score'], bins=[0, 580, 670, 740, 8
                                   labels=['poor', 'fair', 'good', 'very_good'
    df['credit_income_interaction'] = df['credit_score'] * df['annual_income']
    df['credit_times_dti'] = df['credit_score'] * df['debt_to_income_ratio']
    df['credit_loan_interaction'] = df['credit_score'] * df['loan_amount']
    print(f"✓ Created 8 features")

    # 3. INTEREST RATE FEATURES
    print("[3/11]  Interest rate features...")
    df['high_interest_flag'] = (df['interest_rate'] > df['interest_rate'].median
    df['very_high_interest'] = (df['interest_rate'] > df['interest_rate'].quanti
    df['low_interest_flag'] = (df['interest_rate'] < df['interest_rate'].quantil
    df['total_interest_cost'] = df['loan_amount'] * df['interest_rate'] / 100
    df['interest_burden'] = df['total_interest_cost'] / (df['annual_income'] + 1
    df['interest_credit_mismatch'] = df['interest_rate'] * (1 - df['credit_score
    df['interest_credit_ratio'] = df['interest_rate'] / (df['credit_score'] / 10
    df['interest_rate_squared'] = df['interest_rate'] ** 2
```

```
    print(f"✓ Created 8 features")

    # 4. RISK SCORES
    print("[4/11]   Composite risk scores...")
    df['risk_score_v1'] = (df['debt_to_income_ratio'] * 0.25 + df['loan_to_incom
                           df['credit_risk_score'] * 0.30 + (df['interest_rate']
    df['risk_score_v2'] = (df['payment_to_income_ratio'] * 0.40 + df['new_debt_t
                           df['interest_burden'] * 0.25)
    df['affordability_score'] = (df['credit_score_normalized'] * 0.40 +
                                 (1 - df['debt_to_income_ratio']) * 0.30 +
                                 df['disposable_income_ratio'] * 0.30)
    df['financial_health_score'] = df['affordability_score'] * 0.60 - df['risk_s
    print(f"   ✓ Created 4 features")

    # 5. LOAN AMOUNT FEATURES
    print("[5/11]  Loan amount features...")
    df['loan_size'] = pd.cut(df['loan_amount'], bins=[0, 10000, 20000, 30000, np
                             labels=['small', 'medium', 'large', 'very_large'])
    df['loan_amount_squared'] = df['loan_amount'] ** 2
    df['loan_amount_log'] = np.log1p(df['loan_amount'])
    df['annual_income_log'] = np.log1p(df['annual_income'])
    df['loan_amount_sqrt'] = np.sqrt(df['loan_amount'])
    print(f"✓ Created 5 features")

    # 6. BINNING FEATURES
    print("[6/11]  Binned features...")
    df['income_decile'] = pd.qcut(df['annual_income'], q=10, labels=False, dupli
    df['credit_decile'] = pd.qcut(df['credit_score'], q=10, labels=False, duplic
    df['loan_decile'] = pd.qcut(df['loan_amount'], q=10, labels=False, duplicate
    df['dti_decile'] = pd.qcut(df['debt_to_income_ratio'], q=10, labels=False, d
    df['interest_decile'] = pd.qcut(df['interest_rate'], q=10, labels=False, dup
    print(f"✓ Created 5 features")

    # 7. INTERACTION FEATURES
    print("[7/11]  Interaction features...")
    df['income_x_credit'] = df['annual_income'] * df['credit_score']
    df['dti_x_interest'] = df['debt_to_income_ratio'] * df['interest_rate']
    df['loan_x_interest'] = df['loan_amount'] * df['interest_rate']
    df['income_x_dti'] = df['annual_income'] * df['debt_to_income_ratio']
    df['income_credit_loan'] = df['annual_income'] * df['credit_score'] * df['lo
    df['dti_interest_credit'] = df['debt_to_income_ratio'] * df['interest_rate']
    print(f"✓ Created 6 features")

    # 8. GRADE FEATURES
    print("[8/11]  Grade/subgrade features...")
    df['grade'] = df['grade_subgrade'].str[0]
    df['subgrade_num'] = df['grade_subgrade'].str[1:].astype(int)
    grade_map = {'A': 1, 'B': 2, 'C': 3, 'D': 4, 'E': 5, 'F': 6, 'G': 7}
    df['grade_numeric'] = df['grade'].map(grade_map)
    df['full_grade_score'] = df['grade_numeric'] * 10 + df['subgrade_num']
    df['grade_credit_ratio'] = df['full_grade_score'] / (df['credit_score'] / 10
    print(f"✓ Created 5 features")

    # 9. STATISTICAL AGGREGATIONS
    print("[9/11]  Statistical aggregations...")
    df['mean_financial_metrics'] = df[['debt_to_income_ratio', 'loan_to_income_r
                                       'payment_to_income_ratio']].mean(axis=1)
    df['max_financial_burden'] = df[['debt_to_income_ratio', 'loan_to_income_rat
                                     'payment_to_income_ratio']].max(axis=1)
```

```
    df['min_financial_burden'] = df[['debt_to_income_ratio', 'loan_to_income_rat
                                     'payment_to_income_ratio']].min(axis=1)
    df['std_financial_metrics'] = df[['debt_to_income_ratio', 'loan_to_income_ra
                                      'payment_to_income_ratio']].std(axis=1)
    print(f"✓ Created 4 features")

    # 10. CATEGORICAL COMBINATIONS
    print("[10/11]  Categorical combinations...")
    df['gender_marital'] = df['gender'] + '_' + df['marital_status']
    df['education_employment'] = df['education_level'] + '_' + df['employment_st
    df['gender_education'] = df['gender'] + '_' + df['education_level']
    df['marital_employment'] = df['marital_status'] + '_' + df['employment_statu
    df['purpose_grade'] = df['loan_purpose'] + '_' + df['grade']
    df['employment_purpose'] = df['employment_status'] + '_' + df['loan_purpose'
    print(f"✓ Created 6 features")

    # 11. ANOMALY FLAGS
    print("[11/11]  Anomaly detection flags...")
    df['extreme_dti'] = (df['debt_to_income_ratio'] > df['debt_to_income_ratio']
    df['low_income'] = (df['annual_income'] < df['annual_income'].quantile(0.25)
    df['large_loan'] = (df['loan_amount'] > df['loan_amount'].quantile(0.75)).as
    df['risky_combo_1'] = ((df['debt_to_income_ratio'] > 0.4) & (df['credit_scor
    df['risky_combo_2'] = ((df['loan_to_income_ratio'] > 0.5) & (df['interest_ra
    df['safe_combo'] = ((df['credit_score'] > 750) & (df['debt_to_income_ratio']
    df['high_risk_all'] = (df['extreme_dti'] & df['risky_combo_1']).astype(int)
    print(f"✓ Created 7 features")

    print("\n" + "="*80)
    print(f" Feature Engineering Complete!")
    print(f"   Final features: {df.shape[1]}")
    print(f"   New features: {df.shape[1] - 13}")
    print("="*80)

    return df

# Apply feature engineering
train_fe = complete_feature_engineering(train)
test_fe = complete_feature_engineering(test)
```

```
 FEATURE ENGINEERING PIPELINE
================================================================================
Starting features: 13

[1/11]  Financial ratio features...
✓ Created 15 features
[2/11]  Credit score features...
✓ Created 8 features
[3/11]  Interest rate features...
✓ Created 8 features
[4/11]   Composite risk scores...
  ✓ Created 4 features
[5/11]  Loan amount features...
✓ Created 5 features
[6/11]  Binned features...
✓ Created 5 features
[7/11]  Interaction features...
✓ Created 6 features
[8/11]  Grade/subgrade features...
✓ Created 5 features
[9/11]  Statistical aggregations...
✓ Created 4 features
[10/11]  Categorical combinations...
✓ Created 6 features
[11/11]  Anomaly detection flags...
✓ Created 7 features

================================================================================
 Feature Engineering Complete!
   Final features: 86
   New features: 73
================================================================================
 FEATURE ENGINEERING PIPELINE
================================================================================
Starting features: 12

[1/11]  Financial ratio features...
✓ Created 15 features
[2/11]  Credit score features...
✓ Created 8 features
[3/11]  Interest rate features...
✓ Created 8 features
[4/11]   Composite risk scores...
  ✓ Created 4 features
[5/11]  Loan amount features...
✓ Created 5 features
[6/11]  Binned features...
✓ Created 5 features
[7/11]  Interaction features...
✓ Created 6 features
[8/11]  Grade/subgrade features...
✓ Created 5 features
[9/11]  Statistical aggregations...
✓ Created 4 features
[10/11]  Categorical combinations...
✓ Created 6 features
[11/11]  Anomaly detection flags...
```

```
✓ Created 7 features

================================================================================
 Feature Engineering Complete!
   Final features: 85
   New features: 72
================================================================================
```

In [23]:
```python
# Encode categorical features
print("ENCODING CATEGORICAL FEATURES")
print("="*80)

categorical_features = train_fe.select_dtypes(include=['object', 'category']).cc

le_dict = {}
for col in categorical_features:
    le = LabelEncoder()
    train_fe[col] = le.fit_transform(train_fe[col].astype(str))
    test_fe[col] = le.transform(test_fe[col].astype(str))
    le_dict[col] = le
    print(f"✓ {col}: {len(le.classes_)} classes")

print(f"\n Encoded {len(categorical_features)} features")
```

```
ENCODING CATEGORICAL FEATURES
================================================================================
✓ gender: 3 classes
✓ marital_status: 4 classes
✓ education_level: 5 classes
✓ employment_status: 5 classes
✓ loan_purpose: 8 classes
✓ grade_subgrade: 30 classes
✓ credit_category: 5 classes
✓ loan_size: 4 classes
✓ grade: 6 classes
✓ gender_marital: 12 classes
✓ education_employment: 25 classes
✓ gender_education: 15 classes
✓ marital_employment: 20 classes
✓ purpose_grade: 48 classes
✓ employment_purpose: 40 classes

 Encoded 15 features
```

In [24]:
```python
# Prepare datasets
feature_cols = [col for col in train_fe.columns if col not in ['id', config.TARG

X = train_fe[feature_cols]
y = train_fe[config.TARGET]
X_test = test_fe[feature_cols]
test_ids = test_fe['id']

print("FINAL DATA READY")
print("="*80)
print(f"X: {X.shape}")
print(f"y: {y.shape}")
print(f"X_test: {X_test.shape}")
print(f"Features: {len(feature_cols)}")
```

```
FINAL DATA READY
================================================================================
X: (593994, 84)
y: (593994,)
X_test: (254569, 84)
Features: 84
```

6 Model Training

6.1 LightGBM

In [25]:

```python
def train_lightgbm(X, y, X_test, n_splits=5):
    skf = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=SEED)
    oof_preds = np.zeros(len(X))
    test_preds = np.zeros(len(X_test))
    feature_importance = pd.DataFrame()

    params = {
        'objective': 'binary',
        'metric': 'auc',
        'boosting_type': 'gbdt',
        'num_leaves': 31,
        'learning_rate': 0.05,
        'feature_fraction': 0.8,
        'bagging_fraction': 0.8,
        'bagging_freq': 5,
        'max_depth': -1,
        'min_child_samples': 20,
        'reg_alpha': 0.1,
        'reg_lambda': 0.1,
        'random_state': SEED,
        'verbose': -1,
        'n_jobs': -1
    }

    fold_scores = []

    for fold, (train_idx, val_idx) in enumerate(skf.split(X, y)):
        print(f"\n{'='*80}")
        print(f"Fold {fold + 1}/{n_splits}")
        print(f"{'='*80}")

        X_train, X_val = X.iloc[train_idx], X.iloc[val_idx]
        y_train, y_val = y.iloc[train_idx], y.iloc[val_idx]

        train_data = lgb.Dataset(X_train, label=y_train)
        val_data = lgb.Dataset(X_val, label=y_val, reference=train_data)

        model = lgb.train(
            params, train_data, num_boost_round=2000,
            valid_sets=[train_data, val_data],
            callbacks=[lgb.early_stopping(100), lgb.log_evaluation(200)]
        )

        oof_preds[val_idx] = model.predict(X_val, num_iteration=model.best_itera
        test_preds += model.predict(X_test, num_iteration=model.best_iteration)

        score = roc_auc_score(y_val, oof_preds[val_idx])
```

```
        fold_scores.append(score)
        print(f"Fold {fold + 1} AUC: {score:.6f}")

        fold_importance = pd.DataFrame({
            'feature': X.columns,
            'importance': model.feature_importance(importance_type='gain'),
            'fold': fold + 1
        })
        feature_importance = pd.concat([feature_importance, fold_importance])

    overall_score = roc_auc_score(y, oof_preds)
    print(f"\n{'='*80}")
    print(f"LightGBM OOF AUC: {overall_score:.6f}")
    print(f"Mean: {np.mean(fold_scores):.6f} (+/- {np.std(fold_scores):.6f})")
    print(f"{'='*80}")

    return oof_preds, test_preds, feature_importance, overall_score

print("\n Training LightGBM...")
lgb_oof, lgb_test, lgb_importance, lgb_score = train_lightgbm(X, y, X_test, n_sp
```

```
 Training LightGBM...

================================================================================
Fold 1/5
================================================================================
Training until validation scores don't improve for 100 rounds
[200]   training's auc: 0.922458        valid_1's auc: 0.920609
[400]   training's auc: 0.927984        valid_1's auc: 0.921725
[600]   training's auc: 0.932593        valid_1's auc: 0.922334
[800]   training's auc: 0.936646        valid_1's auc: 0.922532
[1000]  training's auc: 0.940265        valid_1's auc: 0.922572
Early stopping, best iteration is:
[939]   training's auc: 0.939269        valid_1's auc: 0.922648
Fold 1 AUC: 0.922648

================================================================================
Fold 2/5
================================================================================
Training until validation scores don't improve for 100 rounds
[200]   training's auc: 0.922217        valid_1's auc: 0.919529
[400]   training's auc: 0.927921        valid_1's auc: 0.920848
[600]   training's auc: 0.932595        valid_1's auc: 0.92136
[800]   training's auc: 0.936633        valid_1's auc: 0.921527
[1000]  training's auc: 0.940178        valid_1's auc: 0.921595
Early stopping, best iteration is:
[1002]  training's auc: 0.940226        valid_1's auc: 0.921607
Fold 2 AUC: 0.921607

================================================================================
Fold 3/5
================================================================================
Training until validation scores don't improve for 100 rounds
[200]   training's auc: 0.923091        valid_1's auc: 0.918563
[400]   training's auc: 0.928569        valid_1's auc: 0.919715
[600]   training's auc: 0.933091        valid_1's auc: 0.920057
[800]   training's auc: 0.936995        valid_1's auc: 0.920196
Early stopping, best iteration is:
[781]   training's auc: 0.936648        valid_1's auc: 0.920235
Fold 3 AUC: 0.920235

================================================================================
Fold 4/5
================================================================================
Training until validation scores don't improve for 100 rounds
[200]   training's auc: 0.92277 valid_1's auc: 0.919465
[400]   training's auc: 0.928099        valid_1's auc: 0.920572
[600]   training's auc: 0.932731        valid_1's auc: 0.921026
[800]   training's auc: 0.93667 valid_1's auc: 0.921111
Early stopping, best iteration is:
[837]   training's auc: 0.937359        valid_1's auc: 0.921156
Fold 4 AUC: 0.921156

================================================================================
Fold 5/5
================================================================================
Training until validation scores don't improve for 100 rounds
[200]   training's auc: 0.922846        valid_1's auc: 0.919093
[400]   training's auc: 0.928396        valid_1's auc: 0.92016
[600]   training's auc: 0.932858        valid_1's auc: 0.920519
[800]   training's auc: 0.936968        valid_1's auc: 0.92043
```

```
Early stopping, best iteration is:
[700]   training's auc: 0.934905        valid_1's auc: 0.920559
Fold 5 AUC: 0.920559

================================================================================
LightGBM OOF AUC: 0.921238
Mean: 0.921241 (+/- 0.000848)
================================================================================
```

6.2 XGBoost

In [26]:

```python
def train_xgboost(X, y, X_test, n_splits=5):
    skf = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=SEED)
    oof_preds = np.zeros(len(X))
    test_preds = np.zeros(len(X_test))

    params = {
        'objective': 'binary:logistic',
        'eval_metric': 'auc',
        'max_depth': 6,
        'learning_rate': 0.05,
        'subsample': 0.8,
        'colsample_bytree': 0.8,
        'min_child_weight': 1,
        'reg_alpha': 0.1,
        'reg_lambda': 0.1,
        'random_state': SEED,
        'tree_method': 'hist',
        'n_jobs': -1
    }

    fold_scores = []

    for fold, (train_idx, val_idx) in enumerate(skf.split(X, y)):
        print(f"\nFold {fold + 1}/{n_splits}")

        X_train, X_val = X.iloc[train_idx], X.iloc[val_idx]
        y_train, y_val = y.iloc[train_idx], y.iloc[val_idx]

        model = xgb.XGBClassifier(**params, n_estimators=2000)
        model.fit(
            X_train, y_train,
            eval_set=[(X_val, y_val)],
            early_stopping_rounds=100,
            verbose=200
        )

        oof_preds[val_idx] = model.predict_proba(X_val)[:, 1]
        test_preds += model.predict_proba(X_test)[:, 1] / n_splits

        score = roc_auc_score(y_val, oof_preds[val_idx])
        fold_scores.append(score)
        print(f"Fold {fold + 1} AUC: {score:.6f}")

    overall_score = roc_auc_score(y, oof_preds)
    print(f"\nXGBoost OOF AUC: {overall_score:.6f}")
    print(f"Mean: {np.mean(fold_scores):.6f} (+/- {np.std(fold_scores):.6f})")

    return oof_preds, test_preds, overall_score
```

```
print("\n Training XGBoost...")
xgb_oof, xgb_test, xgb_score = train_xgboost(X, y, X_test, n_splits=config.N_SPL
```

```
 Training XGBoost...

Fold 1/5
[0]     validation_0-auc:0.90842
[200]   validation_0-auc:0.91920
[400]   validation_0-auc:0.92114
[600]   validation_0-auc:0.92173
[800]   validation_0-auc:0.92195
[1000]  validation_0-auc:0.92198
[1061]  validation_0-auc:0.92197
Fold 1 AUC: 0.922051

Fold 2/5
[0]     validation_0-auc:0.90715
[200]   validation_0-auc:0.91851
[400]   validation_0-auc:0.92049
[600]   validation_0-auc:0.92105
[800]   validation_0-auc:0.92129
[952]   validation_0-auc:0.92131
Fold 2 AUC: 0.921321

Fold 3/5
[0]     validation_0-auc:0.90624
[200]   validation_0-auc:0.91705
[400]   validation_0-auc:0.91899
[600]   validation_0-auc:0.91949
[800]   validation_0-auc:0.91967
[1000]  validation_0-auc:0.91970
[1005]  validation_0-auc:0.91970
Fold 3 AUC: 0.919786

Fold 4/5
[0]     validation_0-auc:0.90729
[200]   validation_0-auc:0.91802
[400]   validation_0-auc:0.91991
[600]   validation_0-auc:0.92041
[800]   validation_0-auc:0.92067
[911]   validation_0-auc:0.92064
Fold 4 AUC: 0.920682

Fold 5/5
[0]     validation_0-auc:0.90698
[200]   validation_0-auc:0.91750
[400]   validation_0-auc:0.91941
[600]   validation_0-auc:0.91988
[800]   validation_0-auc:0.91995
[814]   validation_0-auc:0.91996
Fold 5 AUC: 0.920034

XGBoost OOF AUC: 0.920772
Mean: 0.920775 (+/- 0.000832)
```

6.3 CatBoost

In [27]:
```
def train_catboost(X, y, X_test, n_splits=5):
    skf = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=SEED)
    oof_preds = np.zeros(len(X))
```

```
    test_preds = np.zeros(len(X_test))

    params = {
        'iterations': 2000,
        'learning_rate': 0.05,
        'depth': 6,
        'l2_leaf_reg': 3,
        'random_seed': SEED,
        'loss_function': 'Logloss',
        'eval_metric': 'AUC',
        'early_stopping_rounds': 100,
        'verbose': 200,
        'task_type': 'CPU'
    }

    fold_scores = []

    for fold, (train_idx, val_idx) in enumerate(skf.split(X, y)):
        print(f"\nFold {fold + 1}/{n_splits}")

        X_train, X_val = X.iloc[train_idx], X.iloc[val_idx]
        y_train, y_val = y.iloc[train_idx], y.iloc[val_idx]

        model = CatBoostClassifier(**params)
        model.fit(X_train, y_train, eval_set=(X_val, y_val), use_best_model=True

        oof_preds[val_idx] = model.predict_proba(X_val)[:, 1]
        test_preds += model.predict_proba(X_test)[:, 1] / n_splits

        score = roc_auc_score(y_val, oof_preds[val_idx])
        fold_scores.append(score)
        print(f"Fold {fold + 1} AUC: {score:.6f}")

    overall_score = roc_auc_score(y, oof_preds)
    print(f"\nCatBoost OOF AUC: {overall_score:.6f}")
    print(f"Mean: {np.mean(fold_scores):.6f} (+/- {np.std(fold_scores):.6f})")

    return oof_preds, test_preds, overall_score

print("\n Training CatBoost...")
cat_oof, cat_test, cat_score = train_catboost(X, y, X_test, n_splits=config.N_SP
```

```
 Training CatBoost...

Fold 1/5
0:      test: 0.8962237 best: 0.8962237 (0)     total: 159ms    remaining: 5m 17s
200:    test: 0.9166326 best: 0.9166326 (200)   total: 18.9s    remaining: 2m 49s
400:    test: 0.9190889 best: 0.9190889 (400)   total: 37.1s    remaining: 2m 27s
600:    test: 0.9202976 best: 0.9202976 (600)   total: 55.5s    remaining: 2m 9s
800:    test: 0.9210231 best: 0.9210241 (799)   total: 1m 14s   remaining: 1m 50s
1000:   test: 0.9215404 best: 0.9215404 (1000)  total: 1m 32s   remaining: 1m 32s
1200:   test: 0.9218876 best: 0.9218884 (1199)  total: 1m 51s   remaining: 1m 14s
1400:   test: 0.9221825 best: 0.9221834 (1393)  total: 2m 9s    remaining: 55.5s
1600:   test: 0.9223320 best: 0.9223399 (1594)  total: 2m 28s   remaining: 37s
1800:   test: 0.9224416 best: 0.9224496 (1742)  total: 2m 46s   remaining: 18.4s
Stopped by overfitting detector  (100 iterations wait)

bestTest = 0.9224699402
bestIteration = 1815

Shrink model to first 1816 iterations.
Fold 1 AUC: 0.922470

Fold 2/5
0:      test: 0.8962284 best: 0.8962284 (0)     total: 93.9ms   remaining: 3m 7s
200:    test: 0.9160382 best: 0.9160382 (200)   total: 18.9s    remaining: 2m 49s
400:    test: 0.9185711 best: 0.9185711 (400)   total: 37.4s    remaining: 2m 29s
600:    test: 0.9197117 best: 0.9197132 (598)   total: 55.8s    remaining: 2m 9s
800:    test: 0.9203443 best: 0.9203444 (799)   total: 1m 14s   remaining: 1m 50s
1000:   test: 0.9208306 best: 0.9208323 (998)   total: 1m 32s   remaining: 1m 32s
1200:   test: 0.9211800 best: 0.9211800 (1200)  total: 1m 51s   remaining: 1m 13s
1400:   test: 0.9215593 best: 0.9215593 (1400)  total: 2m 9s    remaining: 55.5s
1600:   test: 0.9218036 best: 0.9218043 (1598)  total: 2m 28s   remaining: 37s
1800:   test: 0.9219552 best: 0.9219555 (1799)  total: 2m 47s   remaining: 18.5s
1999:   test: 0.9221707 best: 0.9221802 (1964)  total: 3m 5s    remaining: 0us

bestTest = 0.9221802166
bestIteration = 1964

Shrink model to first 1965 iterations.
Fold 2 AUC: 0.922180

Fold 3/5
0:      test: 0.8955160 best: 0.8955160 (0)     total: 94.5ms   remaining: 3m 8s
200:    test: 0.9147760 best: 0.9147760 (200)   total: 18.7s    remaining: 2m 47s
400:    test: 0.9169415 best: 0.9169441 (399)   total: 37s      remaining: 2m 27s
600:    test: 0.9180095 best: 0.9180095 (600)   total: 55.5s    remaining: 2m 9s
800:    test: 0.9187641 best: 0.9187641 (800)   total: 1m 13s   remaining: 1m 50s
1000:   test: 0.9193447 best: 0.9193524 (996)   total: 1m 32s   remaining: 1m 31s
1200:   test: 0.9198186 best: 0.9198186 (1200)  total: 1m 50s   remaining: 1m 13s
1400:   test: 0.9202217 best: 0.9202255 (1379)  total: 2m 9s    remaining: 55.4s
1600:   test: 0.9203857 best: 0.9203857 (1600)  total: 2m 28s   remaining: 37s
1800:   test: 0.9205754 best: 0.9205767 (1797)  total: 2m 47s   remaining: 18.5s
1999:   test: 0.9206892 best: 0.9206897 (1998)  total: 3m 5s    remaining: 0us

bestTest = 0.9206896846
bestIteration = 1998

Shrink model to first 1999 iterations.
Fold 3 AUC: 0.920690

Fold 4/5
```

```
0:      test: 0.8962283 best: 0.8962283 (0)     total: 90.8ms   remaining: 3m 1s
200:    test: 0.9154144 best: 0.9154144 (200)   total: 18.8s    remaining: 2m 48s
400:    test: 0.9179507 best: 0.9179507 (400)   total: 37.2s    remaining: 2m 28s
600:    test: 0.9191922 best: 0.9191922 (600)   total: 55.6s    remaining: 2m 9s
800:    test: 0.9199550 best: 0.9199550 (800)   total: 1m 14s   remaining: 1m 51s
1000:   test: 0.9203871 best: 0.9203871 (1000)  total: 1m 32s   remaining: 1m 32s
1200:   test: 0.9208488 best: 0.9208499 (1199)  total: 1m 51s   remaining: 1m 14s
1400:   test: 0.9210482 best: 0.9210506 (1397)  total: 2m 9s    remaining: 55.5s
1600:   test: 0.9212370 best: 0.9212407 (1587)  total: 2m 28s   remaining: 36.9s
1800:   test: 0.9213944 best: 0.9213984 (1774)  total: 2m 46s   remaining: 18.4s
1999:   test: 0.9214630 best: 0.9214705 (1971)  total: 3m 5s    remaining: 0us

bestTest = 0.9214705041
bestIteration = 1971

Shrink model to first 1972 iterations.
Fold 4 AUC: 0.921471

Fold 5/5
0:      test: 0.8976247 best: 0.8976247 (0)     total: 93.9ms   remaining: 3m 7s
200:    test: 0.9154369 best: 0.9154369 (200)   total: 18.7s    remaining: 2m 47s
400:    test: 0.9176253 best: 0.9176257 (399)   total: 36.8s    remaining: 2m 26s
600:    test: 0.9184853 best: 0.9184853 (600)   total: 55.3s    remaining: 2m 8s
800:    test: 0.9191284 best: 0.9191284 (800)   total: 1m 13s   remaining: 1m 50s
1000:   test: 0.9194614 best: 0.9194742 (975)   total: 1m 32s   remaining: 1m 31s
1200:   test: 0.9198864 best: 0.9198864 (1200)  total: 1m 50s   remaining: 1m 13s
1400:   test: 0.9202549 best: 0.9202628 (1388)  total: 2m 9s    remaining: 55.2s
1600:   test: 0.9204375 best: 0.9204375 (1600)  total: 2m 27s   remaining: 36.8s
1800:   test: 0.9205863 best: 0.9205932 (1769)  total: 2m 46s   remaining: 18.3s
1999:   test: 0.9207115 best: 0.9207135 (1998)  total: 3m 4s    remaining: 0us

bestTest = 0.920713461
bestIteration = 1998

Shrink model to first 1999 iterations.
Fold 5 AUC: 0.920713

CatBoost OOF AUC: 0.921500
Mean: 0.921505 (+/- 0.000732)
```

7 Model Evaluation

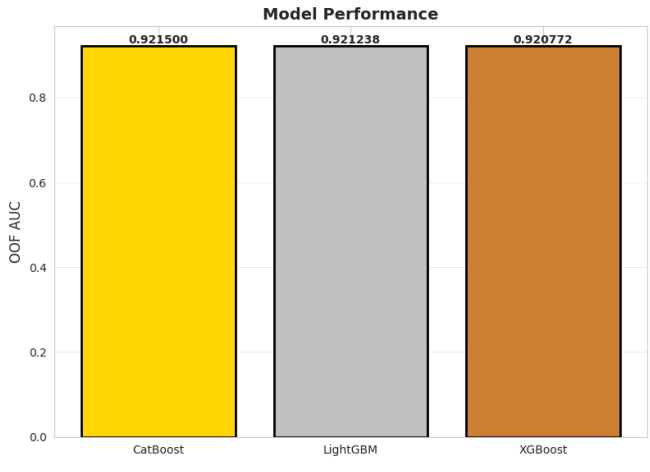
In [28]:

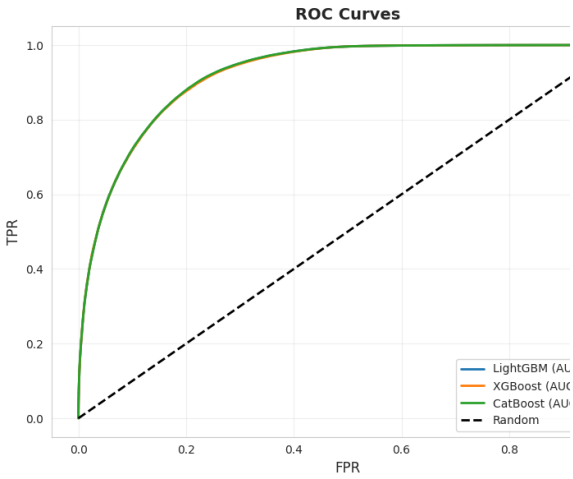
```python
# Compare models
comparison = pd.DataFrame({
    'Model': ['LightGBM', 'XGBoost', 'CatBoost'],
    'OOF AUC': [lgb_score, xgb_score, cat_score]
}).sort_values('OOF AUC', ascending=False)

print("\n" + "="*80)
print("MODEL COMPARISON")
print("="*80)
display(comparison.style.background_gradient(cmap='Greens'))

# Visualize
fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# Bar chart
axes[0].bar(comparison['Model'], comparison['OOF AUC'],
```

```
            color=['#FFD700', '#C0C0C0', '#CD7F32'], edgecolor='black', linewidt
axes[0].set_title('Model Performance', fontsize=14, fontweight='bold')
axes[0].set_ylabel('OOF AUC', fontsize=12)
axes[0].grid(alpha=0.3, axis='y')
for i, (model, score) in enumerate(zip(comparison['Model'], comparison['OOF AUC'
    axes[0].text(i, score, f'{score:.6f}', ha='center', va='bottom', fontweight=

# ROC curves
for name, oof in [('LightGBM', lgb_oof), ('XGBoost', xgb_oof), ('CatBoost', cat_
    fpr, tpr, _ = roc_curve(y, oof)
    axes[1].plot(fpr, tpr, linewidth=2, label=f'{name} (AUC={roc_auc_score(y, oc

axes[1].plot([0, 1], [0, 1], 'k--', linewidth=2, label='Random')
axes[1].set_xlabel('FPR', fontsize=12)
axes[1].set_ylabel('TPR', fontsize=12)
axes[1].set_title('ROC Curves', fontsize=14, fontweight='bold')
axes[1].legend(loc='lower right')
axes[1].grid(alpha=0.3)

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

```
================================================================================
MODEL COMPARISON
================================================================================
```

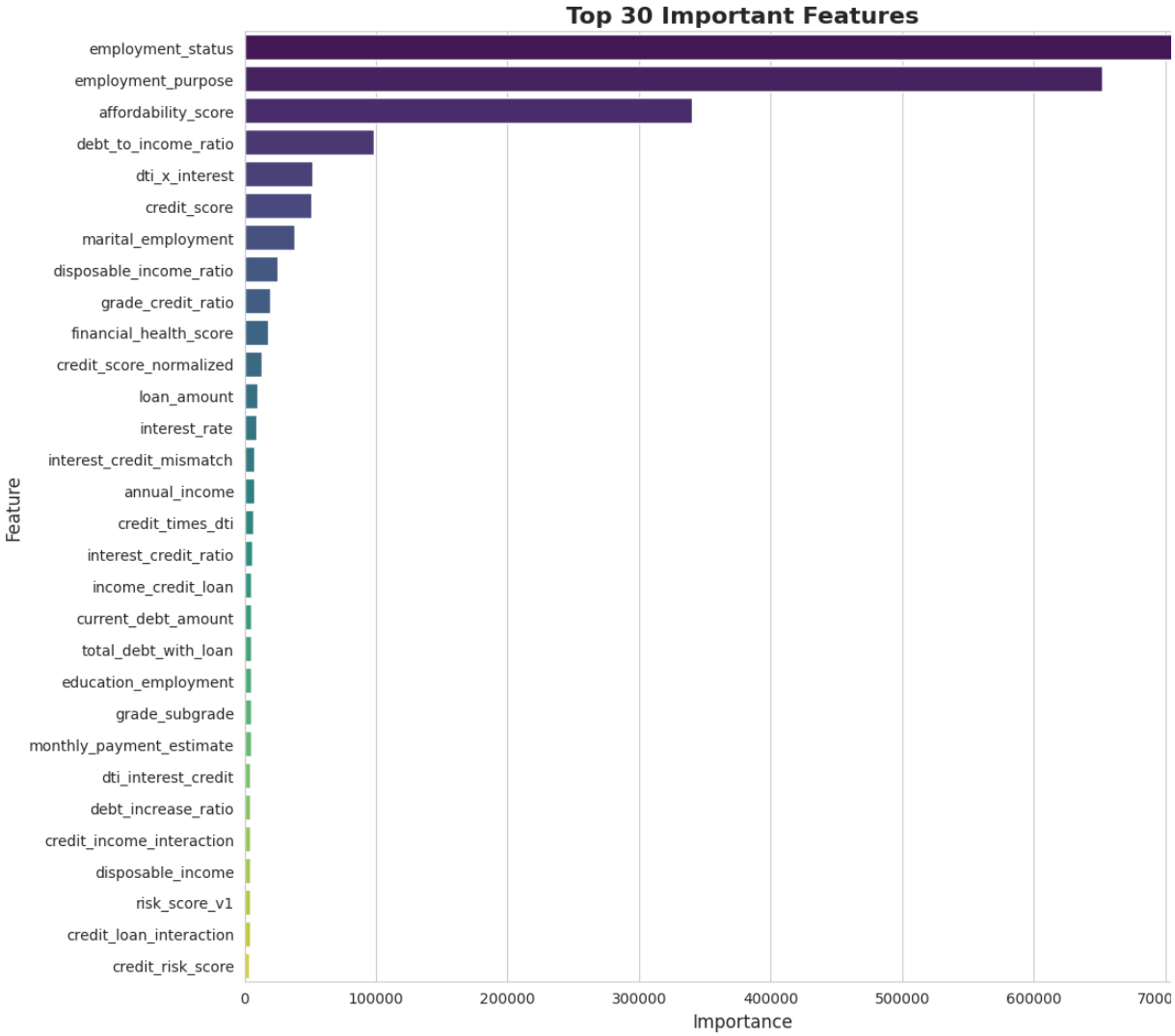
	Model	OOF AUC
2	CatBoost	0.921500
0	LightGBM	0.921238
1	XGBoost	0.920772





In [29]:
```
# Feature importance from LightGBM
importance_df = lgb_importance.groupby('feature')['importance'].mean().sort_valu

plt.figure(figsize=(12, 10))
top_n = 30
sns.barplot(data=importance_df.head(top_n), y='feature', x='importance', palette
plt.title(f'Top {top_n} Important Features', fontsize=16, fontweight='bold')
plt.xlabel('Importance', fontsize=12)
plt.ylabel('Feature', fontsize=12)
plt.tight_layout()
plt.show()
```

```
print("\nTop 15 Features:")
display(importance_df.head(15))
```




```
Top 15 Features:
```

	feature	importance
0	employment_status	721310.407070
1	employment_purpose	652337.187065
2	affordability_score	340654.714206
3	debt_to_income_ratio	98325.080585
4	dti_x_interest	51612.726006
5	credit_score	51282.968501
6	marital_employment	38501.806038
7	disposable_income_ratio	25081.298757
8	grade_credit_ratio	20066.436865
9	financial_health_score	17985.806864
10	credit_score_normalized	13155.560860
11	loan_amount	9923.455219
12	interest_rate	9164.197720
13	interest_credit_mismatch	7500.020375
14	annual_income	7322.225762

8 Ensemble Methods

In [30]:
```
print("CREATING ENSEMBLE")
print("="*80)

# Simple average
simple_oof = (lgb_oof + xgb_oof + cat_oof) / 3
simple_test = (lgb_test + xgb_test + cat_test) / 3
simple_score = roc_auc_score(y, simple_oof)

# Weighted average
total = lgb_score + xgb_score + cat_score
w_lgb = lgb_score / total
w_xgb = xgb_score / total
w_cat = cat_score / total

weighted_oof = lgb_oof * w_lgb + xgb_oof * w_xgb + cat_oof * w_cat
weighted_test = lgb_test * w_lgb + xgb_test * w_xgb + cat_test * w_cat
weighted_score = roc_auc_score(y, weighted_oof)

# Rank average
rank_oof = (rankdata(lgb_oof) + rankdata(xgb_oof) + rankdata(cat_oof)) / (3 * le
rank_test = (rankdata(lgb_test) + rankdata(xgb_test) + rankdata(cat_test)) / (3
rank_score = roc_auc_score(y, rank_oof)

ensemble_results = pd.DataFrame({
    'Ensemble': ['Simple Average', 'Weighted Average', 'Rank Average'],
```

```
    'OOF AUC': [simple_score, weighted_score, rank_score]
}).sort_values('OOF AUC', ascending=False)

print("\nEnsemble Results:")
display(ensemble_results.style.background_gradient(cmap='Greens'))

print(f"\nWeights: LGB={w_lgb:.3f}, XGB={w_xgb:.3f}, CAT={w_cat:.3f}")

# Choose best
best_idx = ensemble_results['OOF AUC'].idxmax()
best_name = ensemble_results.loc[best_idx, 'Ensemble']
best_score = ensemble_results.loc[best_idx, 'OOF AUC']

if best_name == 'Simple Average':
    final_preds = simple_test
elif best_name == 'Weighted Average':
    final_preds = weighted_test
else:
    final_preds = rank_test

print(f"\n Best: {best_name} (AUC: {best_score:.6f})")
```

```
CREATING ENSEMBLE
================================================================================

Ensemble Results:
```

	Ensemble	OOF AUC
2	Rank Average	0.921759
1	Weighted Average	0.921746
0	Simple Average	0.921746

```
Weights: LGB=0.333, XGB=0.333, CAT=0.333

 Best: Rank Average (AUC: 0.921759)
```

9 Submission Generation

In [31]:

```
# # Create submission
# submission = pd.DataFrame({
#     'id': test_ids,
#     config.TARGET: final_preds
# })

# submission.to_csv('submission.csv', index=False)

# print("SUBMISSION CREATED")
# print("="*80)
# print(f"File: submission.csv")
# print(f"Shape: {submission.shape}")
# print(f"\nPreview:")
# display(submission.head(10))

# print(f"\nStatistics:")
# print(submission[config.TARGET].describe())
```

```
# # Visualize
# fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# axes[0].hist(submission[config.TARGET], bins=50, color='steelblue', edgecolor=
# axes[0].axvline(submission[config.TARGET].mean(), color='red', linestyle='--',
#                 label=f'Mean: {submission[config.TARGET].mean():.4f}')
# axes[0].set_title('Prediction Distribution', fontsize=14, fontweight='bold')
# axes[0].set_xlabel('Probability', fontsize=12)
# axes[0].set_ylabel('Frequency', fontsize=12)
# axes[0].legend()
# axes[0].grid(alpha=0.3)

# axes[1].boxplot(submission[config.TARGET], vert=True, patch_artist=True,
#                 boxprops=dict(facecolor='lightblue', alpha=0.7))
# axes[1].set_title('Box Plot', fontsize=14, fontweight='bold')
# axes[1].set_ylabel('Probability', fontsize=12)
# axes[1].grid(alpha=0.3, axis='y')

# plt.tight_layout()
# plt.show()

# print("\n READY FOR SUBMISSION!")
```

10 Conclusion & Final Summary

In [32]:

```
# print(" FINAL SUMMARY")

# print("\n MODEL SCORES:")
# print(f"LightGBM:          {lgb_score:.6f}")
# print(f"XGBoost:           {xgb_score:.6f}")
# print(f"CatBoost:          {cat_score:.6f}")
# print(f"Simple Ensemble:   {simple_score:.6f}")
# print(f"Weighted Ensemble: {weighted_score:.6f}")
# print(f"Rank Ensemble:     {rank_score:.6f}")

# print(f"\n BEST MODEL: {best_name}")
# print(f" BEST AUC: {best_score:.6f}")

# print("\n KEY INSIGHTS:")
# print(f"\n1. Feature Engineering:")
# print(f"• Created {len(feature_cols)} features from 13 original")
# print(f"• Financial ratios were most important")
# print(f"• Risk scores captured complex patterns")

# print(f"\n2. Model Performance:")
# best_single = max(lgb_score, xgb_score, cat_score)
# improvement = ((best_score - best_single) / best_single) * 100
# print(f"• Best single: {best_single:.6f}")
# print(f"• Ensemble gain: +{improvement:.4f}%")
# print(f"• 5-fold CV for robustness")

# print(f"\n3. Top 5 Features:")
# for i, feat in enumerate(importance_df.head(5)['feature'], 1):
#     print(f"   {i}. {feat}")

# print("\n4. Improvements for Future:")
# print("• Hyperparameter optimization")
```

```
# print("• Add original dataset")
# print("• Stacking ensemble")
# print("• Neural network models")
# print("• Feature selection")

# # print("\n" + "="*80)
# # print(" COMPLETE - READY FOR KAGGLE SUBMISSION!")
# # print("="*80)
# # print(f"\n File: submission.csv")
# # print(f" Rows: {len(submission):,}")
# # print(f" Expected LB: ~{best_score:.4f}")
# # print(f"\n Good luck, {test.shape[0]:,} predictions ready!")
```

Connect with Me

Feel free to follow me on these platforms:

Reference: @yeoyunsianggeremie https://www.kaggle.com/code/yeoyunsianggeremie/ps5e11-ai-solution-single-xgb/notebook

In [33]:
```
import os
import sys
import time
import json
import logging
from pathlib import Path
import shutil

import numpy as np
import pandas as pd

from sklearn.model_selection import StratifiedKFold, train_test_split
from sklearn.metrics import roc_auc_score
from sklearn.preprocessing import QuantileTransformer, PowerTransformer, KBinsDi

import xgboost as xgb
import optuna  # tuning used only in FULL mode

# -------------------------
# Setup logging early
# -------------------------
BASE_DIR = Path("/kaggle/input/playground-series-s5e11")
OUTPUT_DIR = Path(".")
OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
LOG_FILE = OUTPUT_DIR / "code_8_1_v4.txt"
SUBMISSION_PATH = OUTPUT_DIR / "submission_4.csv"

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s | %(levelname)s | %(message)s",
    handlers=[
        logging.FileHandler(LOG_FILE, mode="w", encoding="utf-8"),
```

```
        logging.StreamHandler(sys.stdout),
    ],
)
print("Log file initialized at %s", LOG_FILE)

HF_TOKEN = os.environ.get("HF_TOKEN", "")
print("HF_TOKEN present: %s", "yes" if HF_TOKEN else "no")

# -------------------------
# Device selection for XGBoost (purpose: choose CUDA if available)
# -------------------------
def detect_cuda_available() -> bool:
    exe = shutil.which("nvidia-smi")
    if exe is None:
        return False
    out = os.popen(f"{exe} -L").read().strip()
    return len(out) > 0

CUDA_AVAILABLE = detect_cuda_available()
XGB_DEVICE = "cuda:0" if CUDA_AVAILABLE else "cpu"
if CUDA_AVAILABLE:
    print("CUDA detected. Using device='%s' with tree_method='hist'.", XGB_DEVIC
else:
    print("CUDA not detected. Using CPU (device='cpu') with tree_method='hist'."

# -------------------------
# Competition schema
# -------------------------
TRAIN_PATH = BASE_DIR / "train.csv"
TEST_PATH = BASE_DIR / "test.csv"
SAMPLE_SUB_PATH = BASE_DIR / "sample_submission.csv"

TARGET_COL = "loan_paid_back"   # binary classification; metric: ROC AUC
ID_COL = "id"
FOLD_COL = "fold"
META_COLS = {TARGET_COL, ID_COL, FOLD_COL}

# Optional numeric columns for special transforms (if present)
C_INCOME = "annual_income"
C_DTI = "debt_to_income_ratio"

# -------------------------
# Load data (purpose: read CSVs; inputs: train/test paths)
# -------------------------
train = pd.read_csv(TRAIN_PATH)
test = pd.read_csv(TEST_PATH)
sample_sub = pd.read_csv(SAMPLE_SUB_PATH)
print("Loaded data. Train shape: %s | Test shape: %s", train.shape, test.shape)
assert TARGET_COL in train.columns, f"Missing target column '{TARGET_COL}' in tr
assert ID_COL in train.columns and ID_COL in test.columns, "Missing id column in

# -------------------------
# Typing helpers and encoders (purpose: reusable feature builders; inputs: DataF
# -------------------------
def get_cat_num_cols(df: pd.DataFrame, target_col: str, id_col: str, exclude: se
    cols = [c for c in df.columns if c not in exclude]
    cat_cols = [c for c in cols if df[c].dtype == "object" or str(df[c].dtype).s
    num_cols = [c for c in cols if c not in cat_cols]
    return cat_cols, num_cols
```

```python
def pick_top_cats(cat_cols, df, k=6, exclude: set = None):
    exclude = exclude or set()
    cands = []
    for c in cat_cols:
        if c in exclude:
            continue
        n_unique = df[c].nunique(dropna=True)
        if 2 <= n_unique <= 200:
            cands.append((c, n_unique))
    cands.sort(key=lambda t: (-t[1], t[0]))
    sel = [c for c, _ in cands[:k]]
    if len(sel) < min(k, len(cat_cols)):
        rest = [c for c in cat_cols if c not in sel and c not in exclude]
        sel += rest[: (k - len(sel))]
    return sel[:k]

def pick_top_nums(num_cols, df, k=5, exclude: set = None):
    exclude = exclude or set()
    stats = []
    for c in num_cols:
        if c in exclude:
            continue
        series = df[c]
        if series.dtype.kind not in "biufc":
            continue
        nunq = series.nunique(dropna=True)
        if nunq <= 2:
            continue
        var = series.var(skipna=True)
        stats.append((c, 0.0 if pd.isna(var) else float(var)))
    stats.sort(key=lambda t: -t[1])
    return [c for c, _ in stats[:k]]

def add_missing_indicators(df: pd.DataFrame, exclude_cols):
    for c in df.columns:
        if c in exclude_cols:
            continue
        ind_name = f"{c}__isna"
        if ind_name not in df.columns:
            df[ind_name] = df[c].isna().astype(np.int8)
    return df

def frequency_encode(train_pool: pd.DataFrame, series: pd.Series):
    counts = train_pool[series.name].value_counts(dropna=False)
    return counts.to_dict()

def compute_te_map(x: pd.Series, y: pd.Series, m: float = 10.0):
    df = pd.DataFrame({"x": x, "y": y})
    gr = df.groupby("x")["y"].agg(["mean", "count"])
    global_mean = float(y.mean())
    smooth = (gr["mean"] * gr["count"] + global_mean * m) / (gr["count"] + m)
    return smooth.to_dict(), global_mean

def oof_target_encode(train_pool_df, y, col, folds, m=10.0):
    oof = pd.Series(index=train_pool_df.index, dtype="float32")
    for f, (tr_idx, va_idx) in folds.items():
        tr_df = train_pool_df.loc[tr_idx]
        tr_y = y.loc[tr_idx]
        mp, gmean = compute_te_map(tr_df[col], tr_y, m)
        oof.loc[va_idx] = train_pool_df.loc[va_idx, col].map(mp).fillna(gmean).a
```

```
    full_map, full_gmean = compute_te_map(train_pool_df[col], y, m)
    return oof, full_map, full_gmean

def compute_woe_map(x: pd.Series, y: pd.Series, eps: float = 0.5):
    df = pd.DataFrame({"x": x, "y": y})
    pos = df.groupby("x")["y"].sum(min_count=1)
    cnt = df.groupby("x")["y"].count()
    neg = cnt - pos
    total_pos = float(pos.sum())
    total_neg = float(neg.sum())
    dist_pos = (pos + eps) / (total_pos + eps * len(pos))
    dist_neg = (neg + eps) / (total_neg + eps * len(neg))
    woe = np.log((dist_pos) / (dist_neg))
    mapping = woe.to_dict()
    iv = ((dist_pos - dist_neg) * woe).sum()
    return mapping, float(iv)

def oof_woe_encode(train_pool_df, y, col, folds, eps=0.5):
    oof = pd.Series(index=train_pool_df.index, dtype="float32")
    for f, (tr_idx, va_idx) in folds.items():
        tr_df = train_pool_df.loc[tr_idx]
        tr_y = y.loc[tr_idx]
        mp, _iv = compute_woe_map(tr_df[col], tr_y, eps)
        # Clip WOE values to stabilize
        mp = {k: float(np.clip(v, -3.0, 3.0)) for k, v in mp.items()}
        oof.loc[va_idx] = train_pool_df.loc[va_idx, col].map(mp).fillna(0.0).ast
    full_map, iv_full = compute_woe_map(train_pool_df[col], y, eps)
    full_map = {k: float(np.clip(v, -3.0, 3.0)) for k, v in full_map.items()}
    return oof, full_map, iv_full

def fit_kbins(train_pool_series, n_bins=10):
    med = float(np.nanmedian(train_pool_series.values))
    tr_vals = train_pool_series.fillna(med).values.reshape(-1, 1)
    enc = KBinsDiscretizer(n_bins=n_bins, encode="ordinal", strategy="quantile")
    enc.fit(tr_vals)
    return enc, med

def transform_kbins(enc, med, series):
    vals = series.fillna(med).values.reshape(-1, 1)
    b = enc.transform(vals).astype("float32").reshape(-1)
    b = np.where(np.isfinite(b), b, -1.0)
    return pd.Series(b, index=series.index, dtype="float32")

def fit_rank_gaussian(train_pool_series, random_state=2025):
    med = float(np.nanmedian(train_pool_series.values))
    tr_vals = train_pool_series.fillna(med).values.reshape(-1, 1)
    qt = QuantileTransformer(n_quantiles=min(1000, len(tr_vals)), output_distrib
    qt.fit(tr_vals)
    return qt, med

def transform_rank_gaussian(qt, med, series):
    vals = series.fillna(med).values.reshape(-1, 1)
    out = qt.transform(vals).astype("float32").reshape(-1)
    return pd.Series(out, index=series.index, dtype="float32")

def fit_yeojohnson(train_pool_series):
    med = float(np.nanmedian(train_pool_series.values))
    tr_vals = train_pool_series.fillna(med).values.reshape(-1, 1)
    pt = PowerTransformer(method="yeo-johnson", standardize=True)
    pt.fit(tr_vals)
```

```
    return pt, med

def transform_yeojohnson(pt, med, series):
    vals = series.fillna(med).values.reshape(-1, 1)
    out = pt.transform(vals).astype("float32").reshape(-1)
    return pd.Series(out, index=series.index, dtype="float32")

def group_mean_deviation(train_pool_df, val_df, test_df, cat_cols, num_cols):
    # Fit group means on train_pool and map to val/test; guard meta columns.
    for c in cat_cols:
        if c in META_COLS or c not in train_pool_df.columns:
            continue
        for n in num_cols:
            if n in META_COLS or n not in train_pool_df.columns:
                continue
            gname = f"{n}__gm_{c}"
            devname = f"{n}__dev_{c}"
            grp = train_pool_df.groupby(c, observed=True)[n].mean()
            global_mean = float(train_pool_df[n].mean())
            train_pool_df[gname] = train_pool_df[c].map(grp).fillna(global_mean)
            val_df[gname] = val_df[c].map(grp).fillna(global_mean).astype("float
            test_df[gname] = test_df[c].map(grp).fillna(global_mean).astype("flo
            train_pool_df[devname] = (train_pool_df[n] - train_pool_df[gname]).a
            val_df[devname] = (val_df[n] - val_df[gname]).astype("float32")
            test_df[devname] = (test_df[n] - test_df[gname]).astype("float32")
    return train_pool_df, val_df, test_df

def group_percentile_feature(train_pool_df, val_df, test_df, group_col, value_co
    if group_col not in train_pool_df.columns or value_col not in train_pool_df.
        print("Percentile feature skipped (missing): %s within %s", value_col, g
        return train_pool_df, val_df, test_df
    edges_dict = {}
    for g, sub in train_pool_df[[group_col, value_col]].dropna().groupby(group_c
        vals = sub[value_col].values
        if len(vals) < 2:
            continue
        qs = np.linspace(0.0, 1.0, q + 1)
        try_edges = np.quantile(vals, qs)
        edges = try_edges.copy()
        for i in range(1, len(edges)):
            if edges[i] <= edges[i - 1]:
                edges[i] = np.nextafter(edges[i - 1], float("inf"))
        edges_dict[g] = edges

    def apply_edges(df_in: pd.DataFrame):
        out = pd.Series(index=df_in.index, dtype="float32")
        out.iloc[:] = np.nan
        for g, idx in df_in.groupby(group_col, observed=True).groups.items():
            e = edges_dict.get(g, None)
            if e is None:
                out.loc[idx] = 0.5
                continue
            v = df_in.loc[idx, value_col].fillna(e[0]).values
            bins = np.digitize(v, e[1:-1], right=True)
            denom = max(1, len(e) - 2)
            out.loc[idx] = bins.astype("float32") / float(denom)
        out.fillna(0.5, inplace=True)
        return out

    train_pool_df[feature_name] = apply_edges(train_pool_df[[group_col, value_co
```

```
    val_df[feature_name] = apply_edges(val_df[[group_col, value_col]].copy())
    test_df[feature_name] = apply_edges(test_df[[group_col, value_col]].copy())
    return train_pool_df, val_df, test_df

# -------------------------
# Global feature selections (purpose: choose candidate categorical and numeric c
# -------------------------
exclude_for_typing = {TARGET_COL, ID_COL, FOLD_COL}
all_cat, all_num = get_cat_num_cols(train, TARGET_COL, ID_COL, exclude=exclude_f
sel_cat = pick_top_cats(all_cat, train, k=6, exclude=META_COLS)
sel_num_for_deviation = pick_top_nums(all_num, train, k=5, exclude=META_COLS)
transform_targets = [c for c in [C_INCOME, C_DTI] if c in train.columns]
all_features_for_te = [c for c in (all_cat + all_num) if c not in META_COLS]
print("Selected categoricals (≤6): %s", sel_cat)
print("Selected numeric for group-mean deviations (≤5): %s", sel_num_for_deviati
print("Numeric transform targets: %s", transform_targets)
print("All features for target encoding (%d): %s", len(all_features_for_te), all

# -------------------------
# Preprocess for arbitrary held-out fold (purpose: per-fold encoders; inputs: tr
# -------------------------
def preprocess_for_outer_fold(train_df, test_df, held_out_fold, sel_cat, sel_num
    """Fit encoders/transforms on train_pool=all folds except held_out_fold; app
    if held_out_fold == -1:
        tr_pool = train_df.copy()
        va = train_df.iloc[0:0].copy()  # empty
    else:
        tr_pool = train_df.loc[train_df[FOLD_COL] != held_out_fold].copy()
        va = train_df.loc[train_df[FOLD_COL] == held_out_fold].copy()
    te = test_df.copy()
    y_pool = tr_pool[TARGET_COL].astype(int)

    # Inner folds based on existing assignment in tr_pool
    inner_fold_ids = sorted(int(f) for f in tr_pool[FOLD_COL].unique().tolist())
    inner_folds = {}
    for f in inner_fold_ids:
        inner_tr_idx = tr_pool.index[tr_pool[FOLD_COL] != f]
        inner_va_idx = tr_pool.index[tr_pool[FOLD_COL] == f]
        inner_folds[int(f)] = (inner_tr_idx, inner_va_idx)

    # Frequency encoding
    for c in sel_cat:
        if c not in tr_pool.columns:
            continue
        mapping = frequency_encode(tr_pool, tr_pool[c])
        tr_pool[f"{c}__freq"] = tr_pool[c].map(mapping).fillna(0).astype("float3
        if len(va) > 0:
            va[f"{c}__freq"] = va[c].map(mapping).fillna(0).astype("float32")
        te[f"{c}__freq"] = te[c].map(mapping).fillna(0).astype("float32")

    # OOF TE on ALL features (categorical + numerical)
    for c in all_features_for_te:
        if c not in tr_pool.columns:
            continue
        oof_te, te_map, te_g = oof_target_encode(tr_pool, y_pool, c, inner_folds
        tr_pool[f"{c}__te_m10"] = oof_te.astype("float32")
        if len(va) > 0:
            va[f"{c}__te_m10"] = va[c].map(te_map).fillna(te_g).astype("float32"
        te[f"{c}__te_m10"] = te[c].map(te_map).fillna(te_g).astype("float32")
```

```
    # OOF WOE (clip WOE) - only for categorical features
    for c in sel_cat:
        if c not in tr_pool.columns:
            continue
        oof_woe, woe_map, _iv = oof_woe_encode(tr_pool, y_pool, c, inner_folds,
        tr_pool[f"{c}__woe"] = oof_woe.astype("float32")
        if len(va) > 0:
            va[f"{c}__woe"] = va[c].map(woe_map).fillna(0.0).astype("float32")
        te[f"{c}__woe"] = te[c].map(woe_map).fillna(0.0).astype("float32")

    # Numeric transforms on income & DTI
    for col in transform_targets:
        if col not in tr_pool.columns:
            continue
        enc, med = fit_kbins(tr_pool[col], n_bins=10)
        tr_pool[f"{col}__qbin10"] = transform_kbins(enc, med, tr_pool[col])
        if len(va) > 0:
            va[f"{col}__qbin10"] = transform_kbins(enc, med, va[col])
        te[f"{col}__qbin10"] = transform_kbins(enc, med, te[col])

        qt, med_q = fit_rank_gaussian(tr_pool[col])
        tr_pool[f"{col}__rgauss"] = transform_rank_gaussian(qt, med_q, tr_pool[c
        if len(va) > 0:
            va[f"{col}__rgauss"] = transform_rank_gaussian(qt, med_q, va[col])
        te[f"{col}__rgauss"] = transform_rank_gaussian(qt, med_q, te[col])

        pt, med_p = fit_yeojohnson(tr_pool[col])
        tr_pool[f"{col}__yeoj"] = transform_yeojohnson(pt, med_p, tr_pool[col])
        if len(va) > 0:
            va[f"{col}__yeoj"] = transform_yeojohnson(pt, med_p, va[col])
        te[f"{col}__yeoj"] = transform_yeojohnson(pt, med_p, te[col])

    # Group mean deviations
    tr_pool, va, te = group_mean_deviation(tr_pool, va, te, sel_cat, sel_num_for

    # Percentile features
    if "credit_score" in tr_pool.columns and "grade_subgrade" in tr_pool.columns
        tr_pool, va, te = group_percentile_feature(tr_pool, va, te, "grade_subgr
    if "credit_score" in tr_pool.columns and "education_level" in tr_pool.column
        tr_pool, va, te = group_percentile_feature(tr_pool, va, te, "education_l

    # Missingness indicators
    tr_pool = add_missing_indicators(tr_pool, exclude_cols=META_COLS)
    if len(va) > 0:
        va = add_missing_indicators(va, exclude_cols=META_COLS)
    te = add_missing_indicators(te, exclude_cols={ID_COL})

    # Feature list: original numeric (excluding raw categoricals/meta) + enginee
    excl = {TARGET_COL, ID_COL, FOLD_COL}
    raw_cat, raw_num = get_cat_num_cols(train_df, TARGET_COL, ID_COL, exclude=ex
    raw_num_cols = [c for c in raw_num if c not in META_COLS]

    eng_cols = [c for c in tr_pool.columns if (
        c not in train_df.columns or
        c.endswith("__freq") or c.endswith("__te_m10") or c.endswith("__woe") or
        "__gm_" in c or "__dev_" in c or
        c.endswith("__qbin10") or c.endswith("__rgauss") or c.endswith("__yeoj")
        c.endswith("__isna") or
        c.endswith("__pctl_in_grade") or c.endswith("__pctl_in_edu")
    )]
```

```
    feature_cols = sorted(set(raw_num_cols + eng_cols))
    feature_cols = [c for c in feature_cols if (c not in META_COLS and not c.end

    X_tr = tr_pool[feature_cols].copy()
    y_tr = tr_pool[TARGET_COL].astype(int).copy()
    if len(va) > 0:
        X_va = va[feature_cols].copy()
        y_va = va[TARGET_COL].astype(int).copy()
    else:
        X_va = va  # empty
        y_va = va  # empty
    X_te = te[feature_cols].copy()
    return X_tr, y_tr, X_va, y_va, X_te, feature_cols

# -------------------------
# XGBoost params and trainers
# -------------------------
def build_xgb_params(base_lr=0.05, n_estimators=1500, early_stopping_rounds=100)
    params = dict(
        booster="gbtree",
        objective="binary:logistic",
        eval_metric="auc",
        tree_method="hist",
        device=XGB_DEVICE,   # 'cuda:0' or 'cpu'
        learning_rate=base_lr,
        max_depth=6,
        min_child_weight=8,
        subsample=0.8,
        colsample_bytree=0.8,
        colsample_bylevel=0.8,
        gamma=0.0,
        reg_lambda=1.0,
        reg_alpha=0.0,
        max_bin=256,
        grow_policy="depthwise",
        random_state=2025,
        n_estimators=n_estimators,
        n_jobs=0,
        early_stopping_rounds=early_stopping_rounds,
        verbosity=1,
    )
    return params

def train_xgb_single(X_tr, y_tr, X_va, y_va, params, label="baseline"):
    t0 = time.time()
    clf = xgb.XGBClassifier(**params)
    clf.fit(X_tr, y_tr, eval_set=[(X_va, y_va)], verbose=False)
    best_it = getattr(clf, "best_iteration", None)
    y_pred_va = clf.predict_proba(X_va, iteration_range=(0, best_it + 1) if best
    val_auc = roc_auc_score(y_va, y_pred_va)
    elapsed = time.time() - t0
    print("XGB %s: val AUC=%.6f | best_iteration=%s | time=%.1fs", label, val_au
    return clf, val_auc, best_it, elapsed

def optuna_tune_xgb(X_tr, y_tr, X_va, y_va, base_params, time_budget_sec=300):
    print("Optuna tuning start (budget=%ds).", time_budget_sec)
    study = optuna.create_study(direction="maximize", study_name="xgb_ps_s5e11_v

    def objective(trial: optuna.trial.Trial):
        p = base_params.copy()
```

```
        p.update({
            "learning_rate": trial.suggest_float("learning_rate", 0.02, 0.12),
            "max_depth": trial.suggest_int("max_depth", 4, 9),
            "min_child_weight": trial.suggest_float("min_child_weight", 2.0, 12.
            "subsample": trial.suggest_float("subsample", 0.6, 1.0),
            "colsample_bytree": trial.suggest_float("colsample_bytree", 0.6, 1.0
            "colsample_bylevel": trial.suggest_float("colsample_bylevel", 0.6, 1
            "reg_lambda": trial.suggest_float("reg_lambda", 0.5, 5.0, log=True),
            "reg_alpha": trial.suggest_float("reg_alpha", 0.0, 2.0),
            "gamma": trial.suggest_float("gamma", 0.0, 5.0),
            "max_bin": trial.suggest_categorical("max_bin", [128, 256, 512]),
            "n_estimators": trial.suggest_int("n_estimators", 600, 1500),
        })
        # Keep device and tree_method fixed
        p["tree_method"] = base_params["tree_method"]
        p["device"] = base_params["device"]
        p["random_state"] = 2025
        p["early_stopping_rounds"] = base_params["early_stopping_rounds"]

        model = xgb.XGBClassifier(**p)
        model.fit(X_tr, y_tr, eval_set=[(X_va, y_va)], verbose=False)
        best_it = getattr(model, "best_iteration", None)
        y_pred = model.predict_proba(X_va, iteration_range=(0, best_it + 1) if b
        auc = roc_auc_score(y_va, y_pred)
        return auc

    study.optimize(objective, n_trials=200, timeout=time_budget_sec, gc_after_tr
    best_params = study.best_params
    best_value = study.best_value
    print("Optuna best AUC=%.6f with params=%s", best_value, json.dumps(best_par

    tuned_params = base_params.copy()
    tuned_params.update(best_params)
    # Retrain on the same fold-0 split to verify
    model = xgb.XGBClassifier(**tuned_params)
    t0 = time.time()
    model.fit(X_tr, y_tr, eval_set=[(X_va, y_va)], verbose=False)
    best_it = getattr(model, "best_iteration", None)
    y_pred = model.predict_proba(X_va, iteration_range=(0, best_it + 1) if best_
    auc = roc_auc_score(y_va, y_pred)
    elapsed = time.time() - t0
    print("Tuned retrain: val AUC=%.6f | best_iteration=%s | retrain_time=%.1fs"
    return model, auc, best_it, tuned_params

# -------------------------
# CV trainer and final refit
# -------------------------
def assign_outer_folds(df: pd.DataFrame, n_splits=5, seed=2025):
    skf = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=seed)
    folds = np.full(len(df), -1, dtype=int)
    for i, (_, va_idx) in enumerate(skf.split(df.drop(columns=[TARGET_COL]), df[
        folds[va_idx] = i
    out = df.copy()
    out[FOLD_COL] = folds
    return out

def train_xgb_cv_and_predict(train_df, test_df, params, n_splits=5, debug=False)
    print("Starting %d-fold CV training with per-fold encoders.", n_splits)
    train_df = assign_outer_folds(train_df, n_splits=n_splits, seed=2025)
    oof = np.zeros(len(train_df), dtype=np.float32)
```

```
    test_preds = []
    fold_aucs = []
    best_its = []

    for f in range(n_splits):
        print("Fold %d/%d: preprocessing (fit on train folds only).", f+1, n_spl
        X_tr, y_tr, X_va, y_va, X_te, feats = preprocess_for_outer_fold(
            train_df, test_df, held_out_fold=f,
            sel_cat=sel_cat, sel_num_for_deviation=sel_num_for_deviation, transf
            all_features_for_te=all_features_for_te
        )
        params_use = params.copy()
        if debug:
            # Downsample training to 1000 rows in DEBUG to save time
            if len(X_tr) > 1000:
                X_tr, _, y_tr, _ = train_test_split(X_tr, y_tr, test_size=(1.0 -
            params_use["n_estimators"] = min(200, params_use.get("n_estimators",
            params_use["early_stopping_rounds"] = min(20, params_use.get("early_

        print("Fold %d: training XGBoost.", f+1)
        clf = xgb.XGBClassifier(**params_use)
        clf.fit(X_tr, y_tr, eval_set=[(X_va, y_va)], verbose=False)
        best_it = getattr(clf, "best_iteration", None)
        y_va_pred = clf.predict_proba(X_va, iteration_range=(0, best_it + 1) if
        fold_auc = roc_auc_score(y_va, y_va_pred)
        fold_aucs.append(fold_auc)
        oof[train_df.index[train_df[FOLD_COL] == f]] = y_va_pred
        y_te_pred = clf.predict_proba(X_te, iteration_range=(0, best_it + 1) if
        test_preds.append(y_te_pred)
        best_its.append(best_it if best_it is not None else params_use.get("n_es
        print("Fold %d: AUC=%.6f | best_iteration=%s", f+1, fold_auc, str(best_i

    oof_auc = roc_auc_score(train_df[TARGET_COL].values, oof)
    y_test_cv = np.mean(np.vstack(test_preds), axis=0)
    print("CV complete. OOF AUC=%.6f | per-fold AUCs=%s | median best_it=%d",
                oof_auc, [round(a, 6) for a in fold_aucs], int(np.median(best_i
    return y_test_cv, oof_auc, int(np.median(best_its)), feats

def refit_full_and_predict(train_df, test_df, params, debug=False):
    print("Refit on all training data with inner OOF encoders; no held-out valid
    # Assign inner folds (for encoders) deterministically
    train_df_full = assign_outer_folds(train_df, n_splits=5, seed=2025)
    X_tr_full, y_tr_full, X_va_dummy, y_va_dummy, X_te_full, feats_full = preprc
        train_df_full, test_df, held_out_fold=-1,
        sel_cat=sel_cat, sel_num_for_deviation=sel_num_for_deviation, transform_
        all_features_for_te=all_features_for_te
    )
    params_use = params.copy()
    if debug:
        if len(X_tr_full) > 1000:
            X_tr_full, _, y_tr_full, _ = train_test_split(X_tr_full, y_tr_full,
        params_use["n_estimators"] = min(200, params_use.get("n_estimators", 150
        params_use["early_stopping_rounds"] = min(20, params_use.get("early_stop

    # For full refit, use training data as eval_set just to track rounds; accept
    clf_full = xgb.XGBClassifier(**params_use)
    clf_full.fit(X_tr_full, y_tr_full, eval_set=[(X_tr_full, y_tr_full)], verbos
    best_it_full = getattr(clf_full, "best_iteration", None)
    y_test_full = clf_full.predict_proba(X_te_full, iteration_range=(0, best_it_
    print("Full refit complete. best_iteration=%s | n_features=%d", str(best_it_
```

```
    return y_test_full, best_it_full, feats_full

# -------------------------
# Main pipeline runs twice: DEBUG then FULL
# -------------------------
def run_pipeline(DEBUG: bool):
    mode = "DEBUG" if DEBUG else "FULL"
    print("===== Running in %s mode =====", mode)

    # Create a single 5-fold assignment for baseline/tuning on fold 0
    train_folds = assign_outer_folds(train.copy(), n_splits=5, seed=2025)
    # Preprocess for fold 0 for baseline/tuning
    X_tr0, y_tr0, X_va0, y_va0, X_te0, feats0 = preprocess_for_outer_fold(
        train_folds, test.copy(), held_out_fold=0,
        sel_cat=sel_cat, sel_num_for_deviation=sel_num_for_deviation, transform_
        all_features_for_te=all_features_for_te
    )

    # Baseline params (reduced trees in DEBUG)
    if DEBUG:
        base_params = build_xgb_params(base_lr=0.05, n_estimators=150, early_stc
        # Downsample training to 1000 rows for the initial fold-0 baseline
        if len(X_tr0) > 1000:
            X_tr0, _, y_tr0, _ = train_test_split(X_tr0, y_tr0, test_size=(1.0 -
    else:
        base_params = build_xgb_params(base_lr=0.05, n_estimators=1500, early_st

    print("Baseline training on fold 0 (purpose: establish reference AUC).")
    model_base, auc_base, best_it_base, t_base = train_xgb_single(X_tr0, y_tr0,

    # Tuning only in FULL mode
    if not DEBUG:
        model_tuned, auc_tuned, best_it_tuned, tuned_params = optuna_tune_xgb(
            X_tr0, y_tr0, X_va0, y_va0, base_params, time_budget_sec=300
        )
        if auc_tuned >= auc_base:
            final_params = tuned_params
            print("Selected tuned params (AUC=%.6f >= baseline %.6f).", auc_tune
        else:
            final_params = base_params
            print("Selected baseline params (AUC=%.6f > tuned %.6f).", auc_base,
    else:
        final_params = base_params
        print("DEBUG mode: tuning skipped; using baseline params.")

    # CV training + predictions
    y_test_cv, oof_auc, median_best_it, feats_cv = train_xgb_cv_and_predict(
        train.copy(), test.copy(), final_params, n_splits=5, debug=DEBUG
    )

    if DEBUG:
        print("DEBUG mode: submission generation skipped per requirements.")
        return

    # Full refit + predictions
    y_test_full, best_it_full, feats_full = refit_full_and_predict(
        train.copy(), test.copy(), final_params, debug=False
    )

    # Blend CV ensemble with full-refit model (simple mean)
```

```
    y_test_final = 0.5 * y_test_cv + 0.5 * y_test_full
    y_test_final = np.clip(y_test_final, 1e-9, 1 - 1e-9)

    # Write submission
    submission = pd.DataFrame({ID_COL: test[ID_COL].values, TARGET_COL: y_test_f
    submission.to_csv(SUBMISSION_PATH, index=False)
    print("Submission written to %s", SUBMISSION_PATH)

    # Log prediction distribution
    desc = pd.Series(y_test_final).describe(percentiles=[0.01, 0.05, 0.1, 0.5, 0
    print("Prediction distribution summary:\n%s", desc.to_string())
    print("Run complete: OOF AUC=%.6f | median_best_it(CV)=%d | device=%s", oof_

# -------------------------
# Execute: DEBUG then FULL
# -------------------------
run_pipeline(DEBUG=True)   # no submission
run_pipeline(DEBUG=False)  # produce submission_4.csv
```

```
Log file initialized at %s code_8_1_v4.txt
HF_TOKEN present: %s no
CUDA detected. Using device='%s' with tree_method='hist'. cuda:0
Loaded data. Train shape: %s | Test shape: %s (593994, 13) (254569, 12)
Selected categoricals (≤6): %s ['grade_subgrade', 'loan_purpose', 'education_level',
'employment_status', 'marital_status', 'gender']
Selected numeric for group-mean deviations (≤5): %s ['annual_income', 'loan_amount',
'credit_score', 'interest_rate', 'debt_to_income_ratio']
Numeric transform targets: %s ['annual_income', 'debt_to_income_ratio']
All features for target encoding (%d): %s 11 ['gender', 'marital_status',
'education_level', 'employment_status', 'loan_purpose', 'grade_subgrade', 'annual_inc
'debt_to_income_ratio', 'credit_score', 'loan_amount', 'interest_rate']
===== Running in %s mode ===== DEBUG
Baseline training on fold 0 (purpose: establish reference AUC).
XGB %s: val AUC=%.6f | best_iteration=%s | time=%.1fs baseline-fold0 0.90845363233785
1.6991572380065918
DEBUG mode: tuning skipped; using baseline params.
Starting %d-fold CV training with per-fold encoders. 5
Fold %d/%d: preprocessing (fit on train folds only). 1 5
Fold %d: training XGBoost. 1
Fold %d: AUC=%.6f | best_iteration=%s 1 0.9084536323378551 42
Fold %d/%d: preprocessing (fit on train folds only). 2 5
Fold %d: training XGBoost. 2
Fold %d: AUC=%.6f | best_iteration=%s 2 0.9062593710617952 55
Fold %d/%d: preprocessing (fit on train folds only). 3 5
Fold %d: training XGBoost. 3
Fold %d: AUC=%.6f | best_iteration=%s 3 0.9093426942654426 63
Fold %d/%d: preprocessing (fit on train folds only). 4 5
Fold %d: training XGBoost. 4
Fold %d: AUC=%.6f | best_iteration=%s 4 0.9135301554469205 52
Fold %d/%d: preprocessing (fit on train folds only). 5 5
Fold %d: training XGBoost. 5
Fold %d: AUC=%.6f | best_iteration=%s 5 0.9124716123305614 35
CV complete. OOF AUC=%.6f | per-fold AUCs=%s | median best_it=%d 0.9076668387004748
[0.908454, 0.906259, 0.909343, 0.91353, 0.912472] 52
DEBUG mode: submission generation skipped per requirements.
===== Running in %s mode ===== FULL
Baseline training on fold 0 (purpose: establish reference AUC).
[I 2025-11-05 02:56:04,396] A new study created in memory with name: xgb_ps_s5e11_v4
```

```
XGB %s: val AUC=%.6f | best_iteration=%s | time=%.1fs baseline-fold0 0.9258297237922
17.159825563430786
Optuna tuning start (budget=%ds). 300
```

```
[I 2025-11-05 02:56:15,683] Trial 0 finished with value: 0.9251235376381876 and paran
{'learning_rate': 0.07597080804370578, 'max_depth': 9, 'min_child_weight':
9.09693121238876, 'subsample': 0.946986861279399, 'colsample_bytree': 0.866771736960
'colsample_bylevel': 0.9408076727446717, 'reg_lambda': 2.152470308732518, 'reg_alpha
1.4389492105751618, 'gamma': 2.448104718769653, 'max_bin': 256, 'n_estimators': 672}.
is trial 0 with value: 0.9251235376381876.
[I 2025-11-05 02:56:25,933] Trial 1 finished with value: 0.9254989735618944 and paran
{'learning_rate': 0.09722093141505489, 'max_depth': 5, 'min_child_weight':
10.322352100549104, 'subsample': 0.988289977802986, 'colsample_bytree': 0.81520372723
'colsample_bylevel': 0.8930477855847261, 'reg_lambda': 0.5201241418231672, 'reg_alpha
1.101847637729537, 'gamma': 4.990516091104718, 'max_bin': 256, 'n_estimators': 619}.
is trial 1 with value: 0.9254989735618944.
[I 2025-11-05 02:56:36,270] Trial 2 finished with value: 0.9254222033281716 and paran
{'learning_rate': 0.10351141495205216, 'max_depth': 7, 'min_child_weight':
7.388492767933842, 'subsample': 0.613171600794527, 'colsample_bytree': 0.736408514112
'colsample_bylevel': 0.7056757866054771, 'reg_lambda': 0.5455539648211561, 'reg_alpha
1.7624903548749233, 'gamma': 3.09713599089052, 'max_bin': 128, 'n_estimators': 942}.
is trial 1 with value: 0.9254989735618944.
[I 2025-11-05 02:56:51,527] Trial 3 finished with value: 0.9257853374702133 and paran
{'learning_rate': 0.053936004107600716, 'max_depth': 7, 'min_child_weight':
8.910703598269102, 'subsample': 0.7865528246783687, 'colsample_bytree': 0.676938351
'colsample_bylevel': 0.6098703283806709, 'reg_lambda': 3.651128864481156, 'reg_alpha
1.4557214674101113, 'gamma': 2.9117530799497415, 'max_bin': 256, 'n_estimators': 863
is trial 3 with value: 0.9257853374702133.
[I 2025-11-05 02:57:07,898] Trial 4 finished with value: 0.925816959946979 and parame
{'learning_rate': 0.07195733802041049, 'max_depth': 4, 'min_child_weight':
4.1380972752185965, 'subsample': 0.829155607117936, 'colsample_bytree': 0.617554484
'colsample_bylevel': 0.6522333987823193, 'reg_lambda': 0.5602369899273998, 'reg_alpha
1.9937148726951104, 'gamma': 4.695348776052236, 'max_bin': 128, 'n_estimators': 901}.
is trial 4 with value: 0.925816959946979.
[I 2025-11-05 02:57:27,702] Trial 5 finished with value: 0.9261042554777794 and paran
{'learning_rate': 0.04606413764727155, 'max_depth': 4, 'min_child_weight':
7.445770070386859, 'subsample': 0.8920864624055421, 'colsample_bytree': 0.678342531
'colsample_bylevel': 0.6064298244062202, 'reg_lambda': 0.7477790062732108, 'reg_alpha
0.13247295506571, 'gamma': 0.7130546517085395, 'max_bin': 256, 'n_estimators': 1316}.
is trial 5 with value: 0.9261042554777794.
[I 2025-11-05 02:57:39,981] Trial 6 finished with value: 0.9254298447929292 and paran
{'learning_rate': 0.07880097364192855, 'max_depth': 8, 'min_child_weight':
4.58403629294317, 'subsample': 0.7116460151781706, 'colsample_bytree': 0.826844429330
'colsample_bylevel': 0.8610741445979126, 'reg_lambda': 1.2889392684125038, 'reg_alpha
0.8669542278981235, 'gamma': 4.279026309992301, 'max_bin': 512, 'n_estimators': 1031
is trial 5 with value: 0.9261042554777794.
[I 2025-11-05 02:57:59,032] Trial 7 finished with value: 0.9258813845294498 and paran
{'learning_rate': 0.039230075347521, 'max_depth': 6, 'min_child_weight':
3.0050478053965026, 'subsample': 0.8822642355533182, 'colsample_bytree':
0.9501802659411878, 'colsample_bylevel': 0.9508111161531669, 'reg_lambda':
4.120217264736194, 'reg_alpha': 0.40961119989151684, 'gamma': 0.751404026946062, 'max
256, 'n_estimators': 1301}. Best is trial 5 with value: 0.9261042554777794.
[I 2025-11-05 02:58:18,774] Trial 8 finished with value: 0.925990473642072 and parame
{'learning_rate': 0.04529073262686532, 'max_depth': 5, 'min_child_weight':
3.202792671856019, 'subsample': 0.7163700906550228, 'colsample_bytree': 0.956206729560
'colsample_bylevel': 0.942695978199033, 'reg_lambda': 0.6368297108803681, 'reg_alpha
1.8826964557395702, 'gamma': 3.1930221197361712, 'max_bin': 256, 'n_estimators': 1282
Best is trial 5 with value: 0.9261042554777794.
[I 2025-11-05 02:58:30,585] Trial 9 finished with value: 0.9250942263611598 and paran
{'learning_rate': 0.09170855944868678, 'max_depth': 8, 'min_child_weight':
8.713843677373426, 'subsample': 0.6675472830844859, 'colsample_bytree': 0.998128664277
'colsample_bylevel': 0.8066906753798886, 'reg_lambda': 2.6079872411779843, 'reg_alpha
1.6475180989928107, 'gamma': 2.636736549959433, 'max_bin': 128, 'n_estimators': 1217
is trial 5 with value: 0.9261042554777794.
```

```
[I 2025-11-05 02:59:00,914] Trial 10 finished with value: 0.9260797874913126 and
parameters: {'learning_rate': 0.024535515750080884, 'max_depth': 4, 'min_child_weight
11.719040453280044, 'subsample': 0.8971450529782905, 'colsample_bytree':
0.7231173465855045, 'colsample_bylevel': 0.7317908974380087, 'reg_lambda':
1.0579125925106108, 'reg_alpha': 0.0031254453636913038, 'gamma': 0.05840509621143575,
'max_bin': 512, 'n_estimators': 1469}. Best is trial 5 with value: 0.9261042554777794.
[I 2025-11-05 02:59:32,424] Trial 11 finished with value: 0.9260230358979759 and
parameters: {'learning_rate': 0.022351359171438646, 'max_depth': 4, 'min_child_weight
11.285838569093588, 'subsample': 0.8958703062428688, 'colsample_bytree':
0.7261193642385914, 'colsample_bylevel': 0.726809342464943, 'reg_lambda':
1.0696899644777185, 'reg_alpha': 0.026843596078676698, 'gamma': 0.11262761888265667,
'max_bin': 512, 'n_estimators': 1500}. Best is trial 5 with value: 0.9261042554777794.
[I 2025-11-05 03:00:01,575] Trial 12 finished with value: 0.926058179184644 and param
{'learning_rate': 0.029066740530560138, 'max_depth': 5, 'min_child_weight':
5.542144800666126, 'subsample': 0.8539207598258259, 'colsample_bytree': 0.6068685500
'colsample_bylevel': 0.7405962245084058, 'reg_lambda': 0.8925006099532176, 'reg_alpha
0.06678198017968395, 'gamma': 1.2688921816178715, 'max_bin': 512, 'n_estimators': 149
Best is trial 5 with value: 0.9261042554777794.
[I 2025-11-05 03:00:22,753] Trial 13 finished with value: 0.9260797579509877 and
parameters: {'learning_rate': 0.058095867659045355, 'max_depth': 4, 'min_child_weight
6.604460382097686, 'subsample': 0.92632878669274, 'colsample_bytree': 0.719635275149
'colsample_bylevel': 0.6049937527761484, 'reg_lambda': 0.8508117218326079, 'reg_alpha
0.4015298246123288, 'gamma': 1.535573633681844, 'max_bin': 512, 'n_estimators': 1390}
is trial 5 with value: 0.9261042554777794.
[I 2025-11-05 03:00:44,096] Trial 14 finished with value: 0.9259782655517353 and
parameters: {'learning_rate': 0.03529133281996172, 'max_depth': 6, 'min_child_weight
11.290466804885138, 'subsample': 0.7758865519760907, 'colsample_bytree': 0.667225460
'colsample_bylevel': 0.6817221955862797, 'reg_lambda': 1.445579583541291, 'reg_alpha
0.41252948774987436, 'gamma': 0.06356223314731368, 'max_bin': 512, 'n_estimators': 11
Best is trial 5 with value: 0.9261042554777794.
[I 2025-11-05 03:01:13,075] Trial 15 finished with value: 0.9258591481602041 and
parameters: {'learning_rate': 0.02122824428088507, 'max_depth': 4, 'min_child_weight
7.369093703539113, 'subsample': 0.9991101368149037, 'colsample_bytree': 0.761897459
'colsample_bylevel': 0.7726977978702707, 'reg_lambda': 0.824799210046169, 'reg_alpha
0.7196159627172842, 'gamma': 0.9633976522099137, 'max_bin': 256, 'n_estimators': 1378
Best is trial 5 with value: 0.9261042554777794.
```

```
Optuna best AUC=%.6f with params=%s 0.9261042554777794 {"learning_rate":
0.04606413764727155, "max_depth": 4, "min_child_weight": 7.445770070386859, "subsampl
0.8920864624055421, "colsample_bytree": 0.6783425313085594, "colsample_bylevel":
0.6064298244062202, "reg_lambda": 0.7477790062732108, "reg_alpha": 0.13247295506571,
"gamma": 0.7130546517085395, "max_bin": 256, "n_estimators": 1316}
Tuned retrain: val AUC=%.6f | best_iteration=%s | retrain_time=%.1fs 0.92610425547777
20.80276608467102
Selected tuned params (AUC=%.6f >= baseline %.6f). 0.9261042554777794 0.925829723792
Starting %d-fold CV training with per-fold encoders. 5
Fold %d/%d: preprocessing (fit on train folds only). 1 5
Fold %d: training XGBoost. 1
Fold %d: AUC=%.6f | best_iteration=%s 1 0.9261042554777794 821
Fold %d/%d: preprocessing (fit on train folds only). 2 5
Fold %d: training XGBoost. 2
Fold %d: AUC=%.6f | best_iteration=%s 2 0.9259141024231841 1148
Fold %d/%d: preprocessing (fit on train folds only). 3 5
Fold %d: training XGBoost. 3
Fold %d: AUC=%.6f | best_iteration=%s 3 0.9263278960177042 1050
Fold %d/%d: preprocessing (fit on train folds only). 4 5
Fold %d: training XGBoost. 4
Fold %d: AUC=%.6f | best_iteration=%s 4 0.9271705209956536 914
Fold %d/%d: preprocessing (fit on train folds only). 5 5
Fold %d: training XGBoost. 5
Fold %d: AUC=%.6f | best_iteration=%s 5 0.9277248276083434 1030
CV complete. OOF AUC=%.6f | per-fold AUCs=%s | median best_it=%d 0.9266406384229645
[0.926104, 0.925914, 0.926328, 0.927171, 0.927725] 1030
Refit on all training data with inner OOF encoders; no held-out validation.
Full refit complete. best_iteration=%s | n_features=%d 1315 198
Submission written to %s submission_4.csv
Prediction distribution summary:
%s count    254569.000000
mean          0.799724
std           0.301906
min           0.000212
1%            0.001860
5%            0.011903
10%           0.133671
50%           0.938899
90%           0.993974
95%           0.997125
99%           0.999267
max           0.999986
Run complete: OOF AUC=%.6f | median_best_it(CV)=%d | device=%s 0.9266406384229645 103
cuda:0
```