# Kaggle Playground - Loan Payback Prediction ( - Optuna)

## 1. Importing Libraries

In [ ]:
```
# Core Data Science Libraries
import numpy as np
import pandas as pd
import warnings
import gc

# Visualization Libraries
import matplotlib.pyplot as plt
import seaborn as sns

# Scikit-learn for Preprocessing and Modeling
from sklearn.model_selection import StratifiedKFold, train_test_split
from sklearn.preprocessing import StandardScaler, OrdinalEncoder
from sklearn.metrics import roc_auc_score
from scipy.stats import rankdata

# Machine Learning Models
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
import lightgbm as lgb
from catboost import CatBoostClassifier, Pool

# Hyperparameter Tuning
import optuna

# Notebook settings
warnings.filterwarnings('ignore')
pd.set_option('display.max_columns', None)
optuna.logging.set_verbosity(optuna.logging.WARNING)
```

## 2. Loading Dataset

In [ ]:
```
# Define file paths
TRAIN_PATH = "/kaggle/input/playground-series-s5e11/train.csv"
TEST_PATH = "/kaggle/input/playground-series-s5e11/test.csv"
SUBMISSION_PATH = "/kaggle/input/playground-series-s5e11/sample_submission.csv"

# Load the datasets
train = pd.read_csv(TRAIN_PATH)
test = pd.read_csv(TEST_PATH)
sample_submission = pd.read_csv(SUBMISSION_PATH)
```

In [ ]:
```
print("Train shape:", train.shape)
print("Test shape:", test.shape)
```

## 3. Feature Engineering

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

```python
# 1. FINANCIAL RATIOS
df['loan_to_income_ratio'] = df['loan_amount'] / (df['annual_income'] + 1)
df['monthly_income'] = df['annual_income'] / 12
# Simplified approximation from source
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['mc
df['annual_payment_burden'] = df['monthly_payment_estimate'] * 12
df['payment_burden_ratio'] = df['annual_payment_burden'] / (df['annual_incom

# 2. 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', 'e
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']

# 3. 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

# 4. 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

# 5. 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'])

# 6. BINNING 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

    # 7. 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[
    df['dti_interest_credit'] = (df['debt_to_income_ratio'] * df['interest_rate'

    # 8. GRADE FEATURES
    df['grade'] = df['grade_subgrade'].str[0]
    df['subgrade_num'] = pd.to_numeric(df['grade_subgrade'].str[1:], errors='coe
    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

    # 9. STATISTICAL AGGREGATIONS
    financial_metrics = ['debt_to_income_ratio', 'loan_to_income_ratio', 'paymen
    df['mean_financial_metrics'] = df[financial_metrics].mean(axis=1)
    df['max_financial_burden'] = df[financial_metrics].max(axis=1)
    df['min_financial_burden'] = df[financial_metrics].min(axis=1)
    df['std_financial_metrics'] = df[financial_metrics].std(axis=1)

    # 10. 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'

    # 11. ANOMALY 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)

    return df
```

In [ ]:
```
print("Starting Feature Engineering...")
train_fe = complete_feature_engineering(train)
test_fe = complete_feature_engineering(test)
print("Feature Engineering Complete.")
gc.collect()
```

## 4. Preprocessing

In [ ]:
```
TARGET = 'loan_paid_back'
y = train[TARGET]

# Handle NaNs created during FE (e.g., subgrade_num)
train_fe['subgrade_num'] = train_fe['subgrade_num'].fillna(0)
```

```
test_fe['subgrade_num'] = test_fe['subgrade_num'].fillna(0)

# Identify categorical and numerical features
cat_features = train_fe.select_dtypes(include=['object', 'category']).columns.to
num_features = train_fe.select_dtypes(include=np.number).columns.tolist()

# Remove target and ID from feature lists
num_features = [col for col in num_features if col not in [TARGET, 'id']]
original_cat_features = train.select_dtypes(include=['object']).columns.tolist()
cat_features = list(set(cat_features + original_cat_features) - {'grade_subgrade

print(f"Categorical Features: {cat_features}")
print(f"Numerical Features: {len(num_features)}")

# Ordinal Encoding for all categorical features
encoder = OrdinalEncoder(handle_unknown='use_encoded_value', unknown_value=-1)
train_fe[cat_features] = encoder.fit_transform(train_fe[cat_features])
test_fe[cat_features] = encoder.transform(test_fe[cat_features])

# Standard Scaling for numerical features
scaler = StandardScaler()
train_fe[num_features] = scaler.fit_transform(train_fe[num_features])
test_fe[num_features] = scaler.transform(test_fe[num_features])

# Combine features for modeling
features = num_features + cat_features
X = train_fe[features]
X_test = test_fe[features]

print(f"Final feature count: {len(features)}")
gc.collect()
```

## 5. Hyperparameter Tuning (Optuna)

In [ ]:
```
# Create a single validation split for quick Optuna tuning
# We will use a full CV later with the best params
X_train_tune, X_val_tune, y_train_tune, y_val_tune = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

RANDOM_STATE = 42
N_OPTUNA_TRIALS = 20 # Increase this for better results (e.g., 50-100)

# Get feature indices for CatBoost
cat_features_indices = [X.columns.get_loc(c) for c in cat_features if c in X]
```

In [ ]:
```
def objective_lgbm(trial):
    param = {
        'device': 'gpu', 'gpu_platform_id': 0, 'gpu_device_id': 0,
        'objective': 'binary', 'metric': 'auc',
        'boosting_type': 'gbdt',
        'n_estimators': trial.suggest_int('n_estimators', 500, 2000, step=100),
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.1, log=Tru
        'num_leaves': trial.suggest_int('num_leaves', 20, 100),
        'max_depth': trial.suggest_int('max_depth', 5, 12),
        'lambda_l1': trial.suggest_float('lambda_l1', 1e-3, 10.0, log=True),
        'lambda_l2': trial.suggest_float('lambda_l2', 1e-3, 10.0, log=True),
        'feature_fraction': trial.suggest_float('feature_fraction', 0.5, 1.0),
        'bagging_fraction': trial.suggest_float('bagging_fraction', 0.5, 1.0),
        'bagging_freq': trial.suggest_int('bagging_freq', 1, 7),
```

```
        'min_child_samples': trial.suggest_int('min_child_samples', 20, 100),
        'random_state': RANDOM_STATE, 'verbosity': -1, 'n_jobs': -1
    }

    model = LGBMClassifier(**param)
    model.fit(X_train_tune, y_train_tune,
              eval_set=[(X_val_tune, y_val_tune)],
              eval_metric='auc',
              callbacks=[lgb.early_stopping(100, verbose=False)])

    y_pred_proba = model.predict_proba(X_val_tune)[:, 1]
    auc = roc_auc_score(y_val_tune, y_pred_proba)
    return auc

print("Tuning LGBMClassifier...")
study_lgbm = optuna.create_study(direction='maximize')
study_lgbm.optimize(objective_lgbm, n_trials=N_OPTUNA_TRIALS)
best_params_lgb = study_lgbm.best_params
print(f"Best LGBM AUC: {study_lgbm.best_value}")
```

In [ ]:

```
def objective_xgb(trial):
    param = {
        'tree_method': 'gpu_hist', 'predictor': 'gpu_predictor', 'gpu_id': 0,
        'objective': 'binary:logistic', 'eval_metric': 'auc',
        'lambda': trial.suggest_loguniform('lambda', 1e-3, 10.0),
        'alpha': trial.suggest_loguniform('alpha', 1e-3, 10.0),
        'colsample_bytree': trial.suggest_categorical('colsample_bytree', [0.5,
        'subsample': trial.suggest_categorical('subsample', [0.5, 0.7, 0.9, 1.0]
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.1, log=Tru
        'n_estimators': trial.suggest_int('n_estimators', 500, 2000, step=100),
        'max_depth': trial.suggest_int('max_depth', 4, 10),
        'min_child_weight': trial.suggest_int('min_child_weight', 1, 100),
        'random_state': RANDOM_STATE, 'n_jobs': -1
    }

    model = XGBClassifier(**param)
    model.fit(X_train_tune, y_train_tune,
              eval_set=[(X_val_tune, y_val_tune)],
              early_stopping_rounds=100,
              verbose=False)

    y_pred_proba = model.predict_proba(X_val_tune)[:, 1]
    auc = roc_auc_score(y_val_tune, y_pred_proba)
    return auc

print("Tuning XGBClassifier...")
study_xgb = optuna.create_study(direction='maximize')
study_xgb.optimize(objective_xgb, n_trials=N_OPTUNA_TRIALS)
best_params_xgb = study_xgb.best_params
print(f"Best XGB AUC: {study_xgb.best_value}")
```

In [ ]:

```
def objective_cat(trial):
    param = {
        'task_type': 'GPU', 'devices': '0',
        'loss_function': 'Logloss', 'eval_metric': 'AUC',
        'iterations': trial.suggest_int('iterations', 500, 2000),
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.1, log=Tru
        'depth': trial.suggest_int('depth', 4, 10),
        'l2_leaf_reg': trial.suggest_float('l2_leaf_reg', 1.0, 10.0, log=True),
```

```
        'random_strength': trial.suggest_float('random_strength', 1e-3, 10.0, lo
        'bagging_temperature': trial.suggest_float('bagging_temperature', 0.0, 1
        'border_count': trial.suggest_int('border_count', 128, 254),
        'random_seed': RANDOM_STATE, 'logging_level': 'Silent',
        'early_stopping_rounds': 100
    }

    train_pool = Pool(X_train_tune, y_train_tune, cat_features=cat_features_indi
    val_pool = Pool(X_val_tune, y_val_tune, cat_features=cat_features_indices)

    model = CatBoostClassifier(**param)
    model.fit(train_pool, eval_set=val_pool, verbose=0)

    y_pred_proba = model.predict_proba(val_pool)[:, 1]
    auc = roc_auc_score(y_val_tune, y_pred_proba)
    return auc

print("Tuning CatBoostClassifier...")
study_cat = optuna.create_study(direction='maximize')
study_cat.optimize(objective_cat, n_trials=N_OPTUNA_TRIALS)
best_params_cat = study_cat.best_params
print(f"Best CAT AUC: {study_cat.best_value}")
```

## 6. Model Training (Cross-Validation with Tuned Params)

In [ ]:
```
N_SPLITS = 10

skf = StratifiedKFold(n_splits=N_SPLITS, shuffle=True, random_state=RANDOM_STATE

# OOF and Test Predictions Arrays
lgb_oof = np.zeros(len(X))
xgb_oof = np.zeros(len(X))
cat_oof = np.zeros(len(X))

lgb_test = np.zeros(len(X_test))
xgb_test = np.zeros(len(X_test))
cat_test = np.zeros(len(X_test))

# Score Lists
lgb_scores = []
xgb_scores = []
cat_scores = []

# --- Add final GPU/Fixed params to the tuned params ---
best_params_lgb.update({'device': 'gpu', 'gpu_platform_id': 0, 'gpu_device_id':
best_params_xgb.update({'tree_method': 'gpu_hist', 'predictor': 'gpu_predictor',
best_params_cat.update({'task_type': 'GPU', 'devices': '0', 'loss_function': 'Lo
```

In [ ]:
```
for fold, (train_idx, val_idx) in enumerate(skf.split(X, y)):
    print(f"--- Fold {fold+1}/{N_SPLITS} ---")

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

    # --- 1. LightGBM ---
    print("Training LGBM...")
    lgb = LGBMClassifier(**best_params_lgb)
    lgb.fit(X_train, y_train,
            eval_set=[(X_val, y_val)],
            eval_metric='auc',
```

```
            callbacks=[lgb.early_stopping(100, verbose=False)])

    lgb_oof[val_idx] = lgb.predict_proba(X_val)[:, 1]
    lgb_test += lgb.predict_proba(X_test)[:, 1] / N_SPLITS
    lgb_scores.append(roc_auc_score(y_val, lgb_oof[val_idx]))

    # --- 2. XGBoost ---
    print("Training XGBoost...")
    xgb = XGBClassifier(**best_params_xgb)
    xgb.fit(X_train, y_train,
            eval_set=[(X_val, y_val)],
            early_stopping_rounds=100,
            verbose=False)

    xgb_oof[val_idx] = xgb.predict_proba(X_val)[:, 1]
    xgb_test += xgb.predict_proba(X_test)[:, 1] / N_SPLITS
    xgb_scores.append(roc_auc_score(y_val, xgb_oof[val_idx]))

    # --- 3. CatBoost ---
    print("Training CatBoost...")
    train_pool = Pool(X_train, y_train, cat_features=cat_features_indices)
    val_pool = Pool(X_val, y_val, cat_features=cat_features_indices)

    cat = CatBoostClassifier(**best_params_cat)
    cat.fit(train_pool, eval_set=val_pool, verbose=0)

    cat_oof[val_idx] = cat.predict_proba(val_pool)[:, 1]
    cat_test += cat.predict_proba(X_test)[:, 1] / N_SPLITS
    cat_scores.append(roc_auc_score(y_val, cat_oof[val_idx]))

    print(f"Fold {fold+1} Scores: LGB={lgb_scores[-1]:.6f}, XGB={xgb_scores[-1]:
    gc.collect()

# --- Final OOF Scores ---
lgb_score = roc_auc_score(y, lgb_oof)
xgb_score = roc_auc_score(y, xgb_oof)
cat_score = roc_auc_score(y, cat_oof)

print(f"\n--- Overall OOF Scores ---")
print(f"LightGBM OOF AUC: {lgb_score:.6f}")
print(f"XGBoost OOF AUC:  {xgb_score:.6f}")
print(f"CatBoost OOF AUC: {cat_score:.6f}")
```

## 7. Ensembling

In [ ]:
```
# Model Comparison
print("\nMODEL COMPARISON")
comparison = pd.DataFrame({
    'Model': ['LightGBM', 'XGBoost', 'CatBoost'],
    'OOF AUC': [lgb_score, xgb_score, cat_score]
}).sort_values('OOF AUC', ascending=False)
print(comparison)
```

In [ ]:
```
# Create Ensemble
print("\nCREATING ENSEMBLE")

# 1. 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)
```

```
# 2. Weighted Average
total_auc = lgb_score + xgb_score + cat_score
w_lgb = lgb_score / total_auc
w_xgb = xgb_score / total_auc
w_cat = cat_score / total_auc
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)

# 3. 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)
```

In [ ]:
```
# Ensemble Results
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:")
print(ensemble_results)
print(f"\nWeights: LGB={w_lgb:.3f}, XGB={w_xgb:.3f}, CAT={w_cat:.3f}")
```

In [ ]:
```
# 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"\nBest Ensemble: {best_name} (AUC: {best_score:.6f})")
```

## 8. Submission

In [ ]:
```
print("\n--- 8. Submission ---")
submission = pd.DataFrame({'id': test['id'], TARGET: final_preds})
submission.to_csv('submission.csv', index=False)
print("Submission file created successfully!")
display(submission.head())
```

In [ ]:
```
plt.figure(figsize=(8, 5))
sns.histplot(submission[TARGET], bins=50, kde=True)
plt.title('Distribution of Final Predictions')
plt.xlabel('Predicted Probability')
plt.ylabel('Frequency')
plt.show()
```