

# Kaggle Playground

## Problem Statement / Real World Implementations

```
In [1]: # --- Add this cell at the very top ---  
!pip install optuna-integration[lightgbm]
```

Collecting optuna-integration[lightgbm]

Downloading optuna\_integration-4.5.0-py3-none-any.whl.metadata (12 kB)

Requirement already satisfied: optuna in /usr/local/lib/python3.11/dist-packages (from optuna-integration[lightgbm]) (4.5.0)

Requirement already satisfied: lightgbm in /usr/local/lib/python3.11/dist-packages (from optuna-integration[lightgbm]) (4.6.0)

Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/dist-packages (from optuna-integration[lightgbm]) (1.2.2)

Requirement already satisfied: numpy>=1.17.0 in /usr/local/lib/python3.11/dist-packages (from lightgbm->optuna-integration[lightgbm]) (1.26.4)

Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (from lightgbm->optuna-integration[lightgbm]) (1.15.3)

Requirement already satisfied: alembic>=1.5.0 in /usr/local/lib/python3.11/dist-packages (from optuna->optuna-integration[lightgbm]) (1.16.5)

Requirement already satisfied: colorlog in /usr/local/lib/python3.11/dist-packages (from optuna->optuna-integration[lightgbm]) (6.9.0)

Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packages (from optuna->optuna-integration[lightgbm]) (25.0)

Requirement already satisfied: sqlalchemy>=1.4.2 in /usr/local/lib/python3.11/dist-packages (from optuna->optuna-integration[lightgbm]) (2.0.41)

Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from optuna->optuna-integration[lightgbm]) (4.67.1)

Requirement already satisfied: PyYAML in /usr/local/lib/python3.11/dist-packages (from optuna->optuna-integration[lightgbm]) (6.0.3)

Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.11/dist-packages (from scikit-learn->optuna-integration[lightgbm]) (1.5.2)

Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn->optuna-integration[lightgbm]) (3.6.0)

Requirement already satisfied: Mako in /usr/local/lib/python3.11/dist-packages (from alembic>=1.5.0->optuna->optuna-integration[lightgbm]) (1.3.10)

Requirement already satisfied: typing-extensions>=4.12 in /usr/local/lib/python3.11/dist-packages (from alembic>=1.5.0->optuna->optuna-integration[lightgbm]) (4.15.0)

Requirement already satisfied: mkl\_fft in /usr/local/lib/python3.11/dist-packages (from numpy>=1.17.0->lightgbm->optuna-integration[lightgbm]) (1.3.8)

Requirement already satisfied: mkl\_random in /usr/local/lib/python3.11/dist-packages (from numpy>=1.17.0->lightgbm->optuna-integration[lightgbm]) (1.2.4)

Requirement already satisfied: mkl\_umath in /usr/local/lib/python3.11/dist-packages (from numpy>=1.17.0->lightgbm->optuna-integration[lightgbm]) (0.1.1)

Requirement already satisfied: mkl in /usr/local/lib/python3.11/dist-packages (from numpy>=1.17.0->lightgbm->optuna-integration[lightgbm]) (2025.2.0)

Requirement already satisfied: tbb4py in /usr/local/lib/python3.11/dist-packages (from numpy>=1.17.0->lightgbm->optuna-integration[lightgbm]) (2022.2.0)

Requirement already satisfied: mkl-service in /usr/local/lib/python3.11/dist-package: numpy>=1.17.0->lightgbm->optuna-integration[lightgbm]) (2.4.1)

Requirement already satisfied: greenlet>=1 in /usr/local/lib/python3.11/dist-package: sqlalchemy>=1.4.2->optuna->optuna-integration[lightgbm]) (3.2.3)

Requirement already satisfied: MarkupSafe>=0.9.2 in /usr/local/lib/python3.11/dist-p: (from Mako->alembic>=1.5.0->optuna->optuna-integration[lightgbm]) (3.0.2)

Requirement already satisfied: intel-openmp<2026,>=2024 in /usr/local/lib/python3.11, packages (from mkl->numpy>=1.17.0->lightgbm->optuna-integration[lightgbm]) (2024.2.0)

Requirement already satisfied: tbb==2022.\* in /usr/local/lib/python3.11/dist-package: mkl->numpy>=1.17.0->lightgbm->optuna-integration[lightgbm]) (2022.2.0)

Requirement already satisfied: tcmlib==1.\* in /usr/local/lib/python3.11/dist-package: tbb==2022.\*->mkl->numpy>=1.17.0->lightgbm->optuna-integration[lightgbm]) (1.4.0)

Requirement already satisfied: intel-cmplr-lib-rt in /usr/local/lib/python3.11/dist- packages (from mkl\_umath->numpy>=1.17.0->lightgbm->optuna-integration[lightgbm]) (20:

Requirement already satisfied: intel-cmplr-lib-ur==2024.2.0 in /usr/local/lib/python: dist-packages (from intel-openmp<2026,>=2024->mkl->numpy>=1.17.0->lightgbm->optuna- integration[lightgbm]) (2024.2.0)

Downloading optuna\_integration-4.5.0-py3-none-any.whl (99 kB)

99.1/99.1 kB 3.3 MB/s eta 0:00:00

Installing collected packages: optuna-integration

Successfully installed optuna-integration-4.5.0

```
In [2]: # --- 0. Setup ---
import numpy as np
import pandas as pd
import warnings
import optuna
import matplotlib.pyplot as plt
import seaborn as sns
from IPython.display import display

from sklearn.model_selection import KFold, train_test_split, StratifiedKFold
from sklearn.preprocessing import OrdinalEncoder, StandardScaler, RobustScaler,
from sklearn.metrics import roc_auc_score, mean_squared_error, mean_absolute_err

from xgboost import XGBRegressor
from lightgbm import LGBMRegressor
import lightgbm as lgbm # <-- THIS IS THE FIX FOR THE NameError
from catboost import CatBoostRegressor

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

```

N_TRIALS_LGBM = 50
N_TRIALS_XGB = 50
N_TRIALS_CATBOOST = 30 # CatBoost is slower
N_SPLITS_OOF_CHECK = 5 # For the OOF sanity check
N_SPLITS_TUNE = 5 # For cross-validation *inside* Optuna
GLOBAL_RANDOM_STATE = 42

print("Version 9 (Upgraded): AUC-Driven 3-Model CV-Tuned Blend")
print(f"--- 1. Loading Data ---")

# Define file paths
TRAIN_PATH = "/kaggle/input/playground-series-s5e11/train.csv"
TEST_PATH = "/kaggle/input/playground-series-s5e11/test.csv"

# Load the datasets
train_df = pd.read_csv(TRAIN_PATH)
test_df = pd.read_csv(TEST_PATH)

print(f"Train shape: {train_df.shape}, Test shape: {test_df.shape}")

# --- 2. Preprocessing & Feature Engineering (V4 DNA + V10 Experimental) ---
print("--- 2. Defining Preprocessing & Feature Engineering Pipelines ---")

def create_financial_features_v4(df):
    """ V4 Champion DNA Features """
    df_feat = df.copy()
    # Add 1e-6 to prevent division by zero
    df_feat['loan_to_income_ratio'] = df_feat['loan_amount'] / (df_feat['annual_
df_feat['interest_x_loan'] = df_feat['interest_rate'] * df_feat['loan_amount
df_feat['available_income'] = df_feat['annual_income'] * (1 - df_feat['debt_
df_feat['loan_to_available_income'] = df_feat['loan_amount'] / (df_feat['ava
df_feat.replace([np.inf, -np.inf], np.nan, inplace=True)
    return df_feat

def add_experimental_features(df):
    """ V10 Experimental Features (Point 1) """
    df_feat = df.copy()
    df_feat['interest_to_income_ratio'] = df_feat['interest_rate'] / (df_feat['a
df_feat['credit_score_x_interest'] = df_feat['credit_score'] * df_feat['inte
df_feat['credit_score_to_loan'] = df_feat['credit_score'] / (df_feat['loan_a
df_feat.replace([np.inf, -np.inf], np.nan, inplace=True)
    return df_feat

# --- FIX: Create map from TRAIN data ONLY to prevent Leakage ---
all_grades = train_df['grade_subgrade'].unique() # <-- Only train_df
grades_sorted = sorted([g for g in all_grades if pd.notna(g)])
ALL_GRADES_MAP = {grade: i for i, grade in enumerate(grades_sorted)}

def apply_logical_encoding(df, all_grades_map):
    """ Applies logical ordinal mapping as per V4 DNA """
    df_encoded = df.copy()

    education_map = {'Other': 0, 'High School': 1, "Bachelor's": 2, "Master's":
df_encoded['education_level'] = df_encoded['education_level'].map(education_

# Use the pre-computed all_grades_map
df_encoded['grade_subgrade'] = df_encoded['grade_subgrade'].map(all_grades_m

    return df_encoded

```

```

def preprocess(df, use_experimental_features=False, train_cols=None, encoder=None):
    """ Full preprocessing pipeline """
    df_proc = df.drop('id', axis=1, errors='ignore')

    # 1. V4 Features
    df_proc = create_financial_features_v4(df_proc)

    # 2. (Point 1) Experimental Features
    if use_experimental_features:
        df_proc = add_experimental_features(df_proc)

    # 3. Logical Encoding
    df_proc = apply_logical_encoding(df_proc, ALL_GRADES_MAP)

    # 4. Other Categorical Features
    categorical_cols = list(df_proc.select_dtypes(include='object').columns)

    if encoder is None:
        # Handle potential NaNs in categorical columns before fitting encoder
        df_proc[categorical_cols] = df_proc[categorical_cols].fillna('Missing')
        encoder = OrdinalEncoder(handle_unknown='use_encoded_value', unknown_value=-1)
        df_proc[categorical_cols] = encoder.fit_transform(df_proc[categorical_cols])
    else:
        df_proc[categorical_cols] = df_proc[categorical_cols].fillna('Missing')
        df_proc[categorical_cols] = encoder.transform(df_proc[categorical_cols])

    # 5. Handle NaNs with sentinel value -1 (V4 DNA)
    df_proc = df_proc.fillna(-1)

    # 6. Align columns
    if train_cols is not None:
        missing_cols = set(train_cols) - set(df_proc.columns)
        for c in missing_cols:
            df_proc[c] = -1 # Fill new columns with sentinel
        df_proc = df_proc[train_cols] # Ensure order and drop extra columns

    return df_proc, encoder

# --- 3. Feature & Scaler Selection (Point 1 & 2) ---
print("--- 3. Testing Feature Sets and Scalers (Points 1 & 2) ---")

# Prepare data for testing
y_full = train_df['loan_paid_back']

# Base V4 features
train_proc_v4, cat_encoder_v4 = preprocess(train_df.drop('loan_paid_back', axis=1),
                                           encoder=cat_encoder_v4)
X_train_v4, X_val_v4, y_train_v4, y_val_v4 = train_test_split(train_proc_v4, y_full,
                                                              test_size=0.2,
                                                              random_state=42)

# Experimental features
train_proc_exp, cat_encoder_exp = preprocess(train_df.drop('loan_paid_back', axis=1),
                                              encoder=cat_encoder_exp)
X_train_exp, X_val_exp, y_train_exp, y_val_exp = train_test_split(train_proc_exp, y_full,
                                                                    test_size=0.2,
                                                                    random_state=42)

# Scalers to test (Point 2)
scalers = {
    "Standard": StandardScaler(),
    "Robust": RobustScaler(),
    "Quantile": QuantileTransformer(output_distribution='normal', random_state=42)
}

results = []

```

```

# --- NOTE: We no longer define 'model_lgbm_default' out here ---

# Test V4 Features
print("Testing V4 Features with different scalers...")
for name, scaler_obj in scalers.items():
    # --- FIX: Re-initialize the model inside the loop ---
    model = LGBMRegressor(random_state=GLOBAL_RANDOM_STATE, device='gpu', verbose=0)

    X_tr_s = scaler_obj.fit_transform(X_train_v4)
    X_val_s = scaler_obj.transform(X_val_v4)
    model.fit(X_tr_s, y_train_v4)
    preds = model.predict(X_val_s)
    auc = roc_auc_score(y_val_v4, preds)
    results.append({"features": "V4_Baseline", "scaler": name, "auc": auc})

# Test Experimental Features (Point 1)
print("Testing V10 Experimental Features with different scalers...")
for name, scaler_obj in scalers.items():
    # --- FIX: Re-initialize the model inside the loop ---
    model = LGBMRegressor(random_state=GLOBAL_RANDOM_STATE, device='gpu', verbose=0)

    X_tr_s = scaler_obj.fit_transform(X_train_exp)
    X_val_s = scaler_obj.transform(X_val_exp)
    model.fit(X_tr_s, y_train_exp) # This will now work
    preds = model.predict(X_val_s)
    auc = roc_auc_score(y_val_exp, preds)
    results.append({"features": "V10_Experimental", "scaler": name, "auc": auc})

results_df = pd.DataFrame(results).sort_values(by="auc", ascending=False)
print("\n--- Scaler and Feature Validation Results ---")
print(results_df)

# Select best config
best_config = results_df.iloc[0]
best_feature_set_name = best_config['features']
BEST_SCALER_CLASS = scalers[best_config['scaler']].__class__ # Get the class, no need to import
USE_EXPERIMENTAL_FEATURES = (best_feature_set_name == "V10_Experimental")

# (Point 1 Check)
baseline_auc = results_df[results_df['features'] == 'V4_Baseline']['auc'].max()
exp_auc = results_df[results_df['features'] == 'V10_Experimental']['auc'].max()
auc_gain = exp_auc - baseline_auc

if (USE_EXPERIMENTAL_FEATURES and auc_gain > 0.002):
    print(f"\nDecision (Point 1): V10 Experimental features kept (AUC Gain: {auc_gain:.6f})")
else:
    if USE_EXPERIMENTAL_FEATURES:
        print(f"\nDecision (Point 1): V10 features gain ({auc_gain:.6f}) < 0.002, so we stick with V4 features")
    else:
        print(f"\nDecision (Point 1): V4 features performed best. Sticking with V4 features")
    USE_EXPERIMENTAL_FEATURES = False
    # Re-select the best scaler *just* for V4 features
    best_config_v4 = results_df[results_df['features'] == 'V4_Baseline'].iloc[0]
    BEST_SCALER_CLASS = scalers[best_config_v4['scaler']].__class__

print(f"Selected Features: {'V10_Experimental' if USE_EXPERIMENTAL_FEATURES else 'V4_Baseline'}")
print(f"Selected Scaler (Point 2): {BEST_SCALER_CLASS.__name__}")

# --- 4. OOF Sanity Check (Point 4) ---

```

```

print(f"\n--- 4. OOF Sanity Check ({N_SPLITS_OOF_CHECK}-Folds) ---")
# Prepare the chosen full dataset
train_proc_final, cat_encoder_final = preprocess(train_df.drop('loan_paid_back',
                                                             use_experimental_features=USE_E
y_full = train_df['loan_paid_back']
FINAL_TRAIN_COLS = train_proc_final.columns # Save final column set

kf = KFold(n_splits=N_SPLITS_OOF_CHECK, shuffle=True, random_state=GLOBAL_RANDOM
oof_preds = np.zeros(len(train_proc_final))
model_oof = LGBMRegressor(random_state=GLOBAL_RANDOM_STATE, device='gpu', verbos

for fold, (train_idx, val_idx) in enumerate(kf.split(train_proc_final, y_full)):
    print(f"Running OOF Fold {fold+1}/{N_SPLITS_OOF_CHECK}...")
    X_tr, X_v = train_proc_final.iloc[train_idx], train_proc_final.iloc[val_idx]
    y_tr, y_v = y_full.iloc[train_idx], y_full.iloc[val_idx]

    scaler_oof = BEST_SCALER_CLASS()
    X_tr_s = scaler_oof.fit_transform(X_tr)
    X_v_s = scaler_oof.transform(X_v)

    model_oof.fit(X_tr_s, y_tr)
    oof_preds[val_idx] = model_oof.predict(X_v_s)

oof_auc = roc_auc_score(y_full, oof_preds)
print(f"\nSanity Check (Point 4) OOF AUC: {oof_auc:.6f}")
print("(This confirms the 80/20 split is not unusually lucky)")

# --- 5. Unified AUC Tuning Function (UPGRADED to use CV) ---
print(f"\n--- 5. AUC-Driven Optuna Tuning (with {N_SPLITS_TUNE}-Fold CV) ---")

# We will tune on the full dataset, and the objective function
# will handle its own cross-validation.
X_train_final = train_proc_final
y_train_final = y_full

def train_auc_model(model_type="lgbm", n_trials=30):
    """
    Unified Optuna-AUC pipeline function
    UPGRADE: Uses K-Fold CV for robust tuning.
    """

    def objective(trial):
        # --- Create parameters ---
        if model_type == "lgbm":
            param = {
                'device': 'gpu', 'objective': 'regression_l1', 'metric': 'auc',
                'lambda_l1': trial.suggest_float('lambda_l1', 1e-5, 1.0, log=True),
                'lambda_l2': trial.suggest_float('lambda_l2', 1e-5, 1.0, log=True),
                'num_leaves': trial.suggest_int('num_leaves', 32, 256),
                'feature_fraction': trial.suggest_float('feature_fraction', 0.5,
                'bagging_fraction': trial.suggest_float('bagging_fraction', 0.5,
                'bagging_freq': trial.suggest_int('bagging_freq', 1, 7),
                'min_child_samples': trial.suggest_int('min_child_samples', 20,
                'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.05),
                'n_estimators': trial.suggest_int('n_estimators', 400, 1500),
                'max_depth': trial.suggest_int('max_depth', 4, 10),
                'random_state': GLOBAL_RANDOM_STATE, 'verbosity': -1, 'n_jobs':
            }

```

```

elif model_type == "xgb":
    param = {
        'tree_method': 'gpu_hist', 'predictor': 'gpu_predictor', 'gpu_id': 0,
        '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, 0.7, 1.0]),
        'subsample': trial.suggest_categorical('subsample', [0.6, 0.7, 0.8]),
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.05),
        'n_estimators': trial.suggest_int('n_estimators', 400, 1500),
        'max_depth': trial.suggest_int('max_depth', 4, 10),
        'min_child_weight': trial.suggest_int('min_child_weight', 1, 300),
        'random_state': GLOBAL_RANDOM_STATE, 'verbosity': 0
    }

elif model_type == "catboost":
    param = {
        'task_type': 'GPU', 'iterations': trial.suggest_int('iterations', 100, 1000),
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.05),
        'depth': trial.suggest_int('depth', 4, 10),
        'l2_leaf_reg': trial.suggest_loguniform('l2_leaf_reg', 1.0, 10.0),
        'random_strength': trial.suggest_loguniform('random_strength', 1.0, 10.0),
        'bagging_temperature': trial.suggest_loguniform('bagging_temperature', 1.0, 10.0),
        'eval_metric': 'AUC', 'od_type': 'Iter', 'od_wait': 50,
        'random_seed': GLOBAL_RANDOM_STATE, 'verbose': 0
    }

# --- Run 5-Fold Cross-Validation ---
kf = KFold(n_splits=N_SPLITS_TUNE, shuffle=True, random_state=GLOBAL_RANDOM_STATE)
fold_auc_scores = []

for fold, (train_idx, val_idx) in enumerate(kf.split(X_train_final, y_train_final)):
    X_tr, X_v = X_train_final.iloc[train_idx], X_train_final.iloc[val_idx]
    y_tr, y_v = y_train_final.iloc[train_idx], y_train_final.iloc[val_idx]

    # Scaler is fit *inside the fold* to prevent any leakage
    scaler_cv = BEST_SCALER_CLASS()
    X_tr_s = scaler_cv.fit_transform(X_tr)
    X_v_s = scaler_cv.transform(X_v)

    if model_type == "lgbm":
        model = LGBMRegressor(**param)
        model.fit(X_tr_s, y_tr,
                  eval_set=[(X_v_s, y_v)],
                  eval_metric='auc',
                  callbacks=[optuna.integration.LightGBMPruningCallback(trial, 'auc'),
                             lgbm.early_stopping(100, verbose=False)])

    elif model_type == "xgb":
        model = XGBRegressor(**param)
        model.fit(X_tr_s, y_tr,
                  eval_set=[(X_v_s, y_v)],
                  eval_metric='auc',
                  callbacks=[optuna.integration.XGBoostPruningCallback(trial, 'auc'),
                             xgb.early_stopping_rounds(100, verbose=0)])

    elif model_type == "catboost":
        model = CatBoostRegressor(**param)
        model.fit(X_tr_s, y_tr,
                  eval_set=[(X_v_s, y_v)],

```



```

        early_stopping_rounds=100,
        verbose=0)

    preds = model.predict(X_v_s)
    auc = roc_auc_score(y_v, preds)
    fold_auc_scores.append(auc)

    # Optuna Pruning
    trial.report(auc, fold)
    if trial.should_prune():
        raise optuna.exceptions.TrialPruned()

    # Return the average AUC across all 5 folds
    return np.mean(fold_auc_scores)

# --- Run the study ---
pruner = optuna.pruners.MedianPruner(n_warmup_steps=2) # Prune after 2 folds
study = optuna.create_study(direction='maximize', pruner=pruner)
study.optimize(objective, n_trials=n_trials, timeout=7200)

print(f"Best {model_type.upper()} CV AUC: {study.best_value:.6f}")
return study.best_params, study.best_value

# --- Run tuners (This part remains the same) ---
print(f"Tuning LGBM for CV AUC ({N_TRIALS_LGBM} trials)...")
params_lgbm, val_auc_lgbm = train_auc_model("lgbm", n_trials=N_TRIALS_LGBM)

print(f"Tuning XGB for CV AUC ({N_TRIALS_XGB} trials)...")
params_xgb, val_auc_xgb = train_auc_model("xgb", n_trials=N_TRIALS_XGB)

print(f"Tuning CatBoost for CV AUC ({N_TRIALS_CATBOOST} trials)...")
params_cat, val_auc_catboost = train_auc_model("catboost", n_trials=N_TRIALS_CATBOOST)

# --- 6. AUC-Weighted Ensemble & Submission (Point 3 & 5) ---
print("\n--- 6. Final Training, Blending, and Submission ---")

# (Point 3) Calculate 3-model AUC-based weights
print("Calculating AUC-based blend weights...")
total_auc = val_auc_lgbm + val_auc_xgb + val_auc_catboost
weight_lgbm = val_auc_lgbm / total_auc
weight_xgb = val_auc_xgb / total_auc
weight_catboost = val_auc_catboost / total_auc

print(f"LGBM Weight: {weight_lgbm:.4f} (CV AUC: {val_auc_lgbm:.6f})")
print(f"XGB Weight: {weight_xgb:.4f} (CV AUC: {val_auc_xgb:.6f})")
print(f"CAT Weight: {weight_catboost:.4f} (CV AUC: {val_auc_catboost:.6f})")

# Retrain models on 100% of the *best* feature set
print("Retraining models on 100% data...")
scaler_final = BEST_SCALER_CLASS()
X_full_final_s = scaler_final.fit_transform(train_proc_final)
y_full = train_df['loan_paid_back'] # Get the full target

# Initialize and train final models
lgbm_final = LGBMRegressor(**params_lgbm, device='gpu', verbosity=-1, n_jobs=-1)
lgbm_final.fit(X_full_final_s, y_full)

xgb_final = XGBRegressor(**params_xgb, tree_method='gpu_hist', predictor='gpu_pr
xgb_final.fit(X_full_final_s, y_full)

```

```

# CatBoost needs 'iterations' from its tuned params
if 'iterations' not in params_cat:
    params_cat['iterations'] = 1000 # Default if something went wrong
catboost_final = CatBoostRegressor(**params_cat, task_type='GPU', eval_metric='AUC')
catboost_final.fit(X_full_final_s, y_full)

# Prepare test data
print("Preprocessing test data...")
test_proc, _ = preprocess(test_df,
                           use_experimental_features=USE_EXPERIMENTAL_FEATURES,
                           train_cols=FINAL_TRAIN_COLS,
                           encoder=cat_encoder_final)
X_test_scaled = scaler_final.transform(test_proc)

# Predict
print("Generating predictions from all 3 models...")
preds_lgbm = lgbm_final.predict(X_test_scaled)
preds_xgb = xgb_final.predict(X_test_scaled)
preds_catboost = catboost_final.predict(X_test_scaled)

# Blend
blended_preds = (weight_lgbm * preds_lgbm) + \
                (weight_xgb * preds_xgb) + \
                (weight_catboost * preds_catboost)

# (Point 5) Clip predictions to prevent AUC penalties
clipped_preds = np.clip(blended_preds, 0.001, 0.999)

# Create submission file
print("Creating submission.csv...")
submission = pd.DataFrame({'id': test_df['id'], 'loan_paid_back': clipped_preds})
submission.to_csv('submission.csv', index=False)

print("✔ Version 9 (Upgraded) Complete. Submission file generated.")
display(submission.head())

# Display prediction distribution
print("\nFinal Prediction Distribution:")
sns.histplot(clipped_preds, bins=50, kde=True)
plt.title("Version 9 Final Blended Prediction Distribution")
plt.xlabel("Predicted loan_paid_back")
plt.show()

```

Version 9 (Upgraded): AUC-Driven 3-Model CV-Tuned Blend

--- 1. Loading Data ---

Train shape: (593994, 13), Test shape: (254569, 12)

--- 2. Defining Preprocessing & Feature Engineering Pipelines ---

--- 3. Testing Feature Sets and Scalers (Points 1 & 2) ---

Testing V4 Features with different scalers...

[illegible]

Testing V10 Experimental Features with different scalers...

--- Scaler and Feature Validation Results ---

	features	scaler	auc
0	V4_Baseline	Standard	0.917880
2	V4_Baseline	Quantile	0.917829
5	V10_Experimental	Quantile	0.917771
1	V4_Baseline	Robust	0.917642
4	V10_Experimental	Robust	0.917559
3	V10_Experimental	Standard	0.917432

Decision (Point 1): V4 features performed best. Sticking with V4 features.

Selected Features: V4\_Baseline

Selected Scaler (Point 2): StandardScaler

--- 4. OOF Sanity Check (5-Folds) ---

Running OOF Fold 1/5...

Running OOF Fold 2/5...

Running OOF Fold 3/5...

Running OOF Fold 4/5...

Running OOF Fold 5/5...

Sanity Check (Point 4) OOF AUC: 0.918349

(This confirms the 80/20 split is not unusually lucky)

--- 5. AUC-Driven Optuna Tuning (with 5-Fold CV) ---

Tuning LGBM for CV AUC (50 trials)...

Best LGBM CV AUC: 0.871992

Tuning XGB for CV AUC (50 trials)...

Best XGB CV AUC: 0.919968

Tuning CatBoost for CV AUC (30 trials)...

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

Default metric period is 5 because AUC is/are not implemented for GPU  
AUC is not implemented on GPU. Will use CPU for metric computation, this could significantly affect learning time  
Default metric period is 5 because AUC is/are not implemented for GPU  
AUC is not implemented on GPU. Will use CPU for metric computation, this could significantly affect learning time  
Default metric period is 5 because AUC is/are not implemented for GPU  
AUC is not implemented on GPU. Will use CPU for metric computation, this could significantly affect learning time  
Default metric period is 5 because AUC is/are not implemented for GPU  
AUC is not implemented on GPU. Will use CPU for metric computation, this could significantly affect learning time  
Default metric period is 5 because AUC is/are not implemented for GPU  
AUC is not implemented on GPU. Will use CPU for metric computation, this could significantly affect learning time  
Default metric period is 5 because AUC is/are not implemented for GPU  
AUC is not implemented on GPU. Will use CPU for metric computation, this could significantly affect learning time  
Default metric period is 5 because AUC is/are not implemented for GPU  
AUC is not implemented on GPU. Will use CPU for metric computation, this could significantly affect learning time  
Best CATBOOST CV AUC: 0.918521

--- 6. Final Training, Blending, and Submission ---

Calculating AUC-based blend weights...

LGBM Weight: 0.3217 (CV AUC: 0.871992)

XGB Weight: 0.3394 (CV AUC: 0.919968)

CAT Weight: 0.3389 (CV AUC: 0.918521)

Retraining models on 100% data...

Default metric period is 5 because AUC is/are not implemented for GPU  
AUC is not implemented on GPU. Will use CPU for metric computation, this could significantly affect learning time

Preprocessing test data...

Generating predictions from all 3 models...

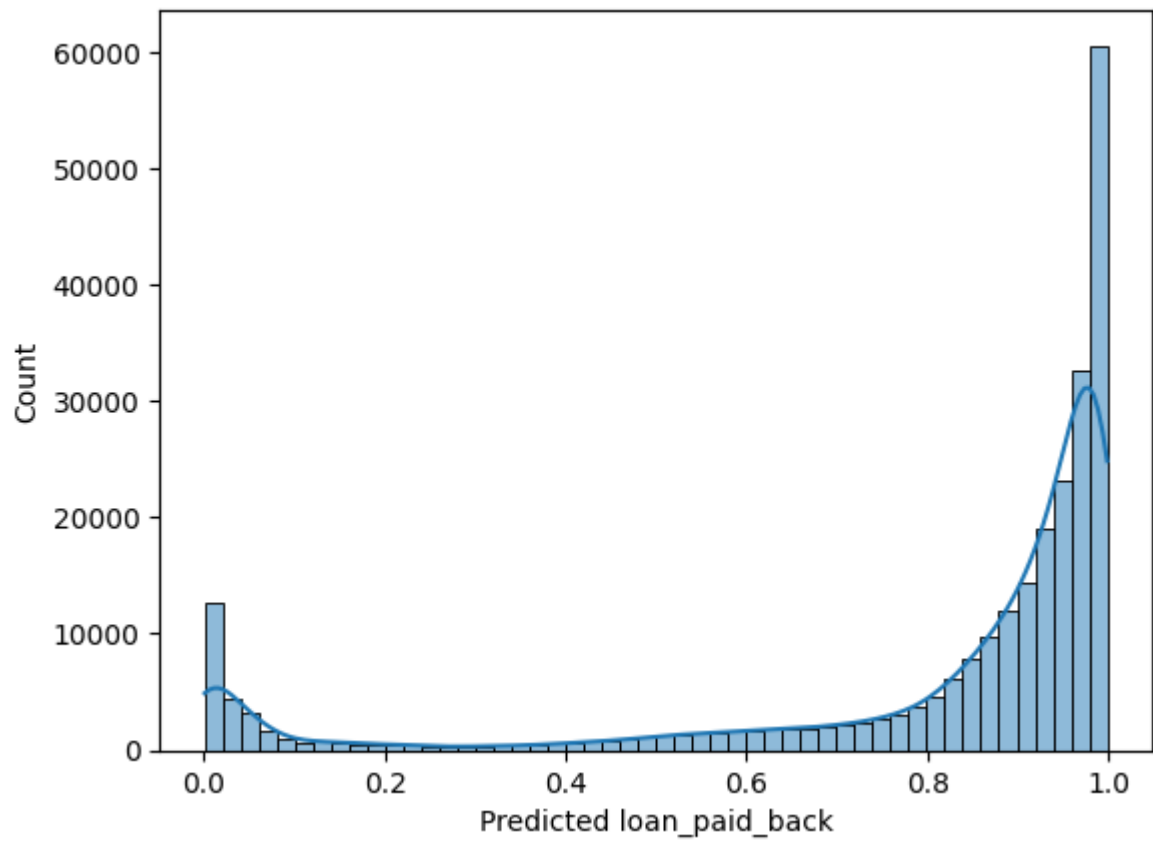
Creating submission.csv...

✔ Version 9 (Upgraded) Complete. Submission file generated.

	id	loan_paid_back
0	593994	0.961607
1	593995	0.975820
2	593996	0.411689
3	593997	0.922231
4	593998	0.968179

Final Prediction Distribution:

Version 9 Final Blended Prediction Distribution



In [ ]: