FA 590 Statistical Learning in Finance

Assignment 3: Loan Default and Return Prediction using Nonlinear Models

Federica Malamisura

2024 Fall

Abstract

This assignment focuses on predicting loan defaults and returns using nonlinear models within a financial dataset from LendingClub, spanning 2008 to 2017. The dataset includes detailed information on loan and borrower characteristics. The analysis involves several key steps: Exploratory Data Analysis (EDA), data preprocessing, feature engineering, and model building. We employ advanced regression techniques, including Decision Trees, XGBoost, CatBoost, and LightGBM, to predict loan returns. Feature importance is analyzed using SHAP values to interpret model predictions. Models are evaluated using a time-series cross-validation approach to ensure robustness and reliability. This comprehensive analysis aims to enhance predictive accuracy and provide insights into factors influencing loan outcomes in the financial domain.

Contents

1 Introduction

In the realm of finance, accurately predicting loan outcomes is crucial for risk management and strategic decision-making. This assignment leverages statistical learning techniques to analyze a comprehensive dataset from LendingClub, encompassing over one million personal loans originated between 2008 and 2017. The dataset provides rich insights into borrower profiles, loan characteristics, and subsequent loan performance.

The primary objective of this analysis is to develop predictive models for loan defaults and returns using nonlinear methodologies. By employing advanced machine learning algorithms such as Decision Trees, XGBoost, CatBoost, and LightGBM, we aim to enhance the precision of predictions. An integral part of the process involves Exploratory Data Analysis (EDA) and meticulous data preprocessing to handle missing values and engineer meaningful features.

Additionally, the use of SHAP (SHapley Additive exPlanations) for feature importance analysis allows us to interpret the contributions of different features towards model predictions. This assignment not only seeks to improve predictive accuracy but also to provide actionable insights into the factors influencing loan outcomes, thereby supporting better financial decision-making.

2 Importing Libraries and Loading Data

2.1 Import Libraries

```python
import pandas as pd
import numpy as np
import matplotlib as mpl
import shap
from sklearn.model_selection import RandomizedSearchCV, TimeSeriesSplit,
    StratifiedKFold
from sklearn.metrics import mean_squared_error, r2_score, accuracy_score,
    f1_score, roc_auc_score, make_scorer
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier
import xgboost as xgb
import catboost as cb
import lightgbm as lgb
```

Listing 1: Importing Libraries

2.2 Loading Data

```python
# Load training data
loan_df = pd.read_csv('lc_loan.csv')

# Load test data
loan_test_df = pd.read_csv('lc_loan_test.csv')

# Load submission template
```

```python
submission_df = pd.read_csv('submission_test.csv')
```

Listing 2: Loading the Dataset

We start by loading the required libraries for data manipulation, visualization, modeling, and interpretability. The dataset is loaded into pandas DataFrames for both the training and test sets. We also load the submission template which will be used later to prepare our predictions for submission.

3 Data Preprocessing

3.1 Sampling Strategy: TimeSeriesSplit

In time series data, it is essential to consider the temporal order of observations when splitting data into training and testing sets. I chose to implement **TimeSeriesSplit** for the following reasons:

1. **Order Preservation**: TimeSeriesSplit maintains the chronological order of data, which is crucial for time series analysis.
2. **Incremental Training**: With each split, more recent data is used for testing, mimicking real-world scenarios where future data is unknown.
3. **Avoiding Data Leakage**: Ensures that future data is not used in training, maintaining the integrity of model evaluation.
4. **Efficient Resource Use**: Unlike expanding or rolling windows, TimeSeriesSplit focuses on efficient use of available data without unnecessary computational overhead.

This approach is particularly suited for financial data, where maintaining temporal order is key for accurate predictions.

```python
# Sort data by origination date to ensure temporal ordering
loan_df['year'] = pd.to_datetime(loan_df['year'], format='%Y')
loan_df = loan_df.sort_values('year')

# Define number of splits for expanding window
n_splits = 5
tscv = TimeSeriesSplit(n_splits=n_splits)
print(f'Number of splits: {n_splits}')
```

Listing 3: Defining Expanding Window Splits

3.2 Handling Missing Values

I handled missing values to prevent biases and inaccuracies in our models. For numerical features, we use the median value for imputation, and for categorical features, we use the mode.

```
# Summarize missing values
missing_values = loan_df.isnull().sum()
missing_values = missing_values[missing_values > 0].sort_values(ascending=
    False)
print('Missing values in training data:')
print(missing_values)

# Impute missing values
for column in missing_values.index:
    if loan_df[column].dtype in ['float64', 'int64']:
        median = loan_df[column].median()
        loan_df[column].fillna(median, inplace=True)
    else:
        mode = loan_df[column].mode()[0]
        loan_df[column].fillna(mode, inplace=True)

# Verify no missing values remain
print('\nMissing values after imputation:')
print(loan_df.isnull().sum().sum())
```

Listing 4: Handling Missing Values

3.3 Feature Engineering

I performed feature engineering to enhance the predictive power of our models. This includes creating new features and transforming existing ones.

```
# Create debt-to-income ratio
loan_df['debt_to_income'] = loan_df['loan_amnt'] / loan_df['annual_inc']
loan_test_df['debt_to_income'] = loan_test_df['loan_amnt'] / loan_test_df['
    annual_inc']

# Log transformation for skewed features
loan_df['log_loan_amount'] = np.log1p(loan_df['loan_amnt'])
loan_test_df['log_loan_amount'] = np.log1p(loan_test_df['loan_amnt'])

# Encode categorical variables
categorical_cols = loan_df.select_dtypes(include=['object']).columns

for col in categorical_cols:
    if col in loan_test_df.columns:
        loan_df[col] = loan_df[col].astype('category').cat.codes
        loan_test_df[col] = loan_test_df[col].astype('category').cat.codes
    else:
        loan_df[col] = loan_df[col].astype('category').cat.codes

# Define feature list
feature_cols = [col for col in loan_df.columns if col not in ['id', '
    early_default', 'return', 'loan_date', 'loan_status', 'year']]

# Update feature_cols to include only columns present in both training and
    test data
train_features = set(loan_df.columns)
```

```
test_features = set(loan_test_df.columns)
common_features = train_features.intersection(test_features)
feature_cols = [col for col in feature_cols if col in common_features]
```

Listing 5: Feature Engineering

4 Model Building

4.1 Regression: Predicting Loan Returns

I implemented and evaluated several linear regression models:

- **Decision Tree Regressor**: Captures nonlinear relationships and interactions without requiring feature scaling.
- **XGBoost Regressor**: An efficient and scalable gradient boosting framework.
- **CatBoost Regressor**: Handles categorical features automatically and reduces the need for extensive preprocessing.
- **LightGBM Regressor**: Optimized for speed and performance, particularly with large datasets.

```

# Define target and features for regression
X = loan_df[feature_cols]
y_return = loan_df['return']

# Standardize features
X = X.select_dtypes(include=[np.number])  # Ensure only numeric data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split data using TimeSeriesSplit
tscv = TimeSeriesSplit(n_splits=5)
best_regression_model = None
best_mse = float('inf')

for train_index, val_index in tscv.split(X_scaled):
    X_train, X_val = X_scaled[train_index], X_scaled[val_index]
    y_train, y_val = y_return.iloc[train_index], y_return.iloc[val_index]

    # Define models
    decision_tree = DecisionTreeRegressor(random_state=42)
    xgboost_reg = xgb.XGBRegressor(random_state=42)
    catboost_reg = cb.CatBoostRegressor(verbose=0, random_state=42)
    lightgbm_reg = lgb.LGBMRegressor(random_state=42)

    # Define reduced parameter grids
    param_dist_tree = {'model__max_depth': [3, 5, 7], '
        model__min_samples_split': [2, 5]}
    param_dist_xgb = {'model__n_estimators': [100], 'model__max_depth': [3,
        5], 'model__learning_rate': [0.01, 0.1]}
```

```
param_dist_cb = {'model__iterations': [100], 'model__depth': [3, 5], '
    model__learning_rate': [0.01, 0.1]}
param_dist_lgb = {'model__n_estimators': [100], 'model__max_depth': [3,
    5], 'model__learning_rate': [0.01, 0.1]}

# Create pipelines
pipeline_tree = Pipeline([('scaler', StandardScaler()), ('model',
    decision_tree)])
pipeline_xgb = Pipeline([('scaler', StandardScaler()), ('model',
    xgboost_reg)])
pipeline_cb = Pipeline([('scaler', StandardScaler()), ('model',
    catboost_reg)])
pipeline_lgb = Pipeline([('scaler', StandardScaler()), ('model',
    lightgbm_reg)])

# Use RandomizedSearchCV
randomized_tree = RandomizedSearchCV(pipeline_tree, param_distributions=
    param_dist_tree, cv=3, scoring='neg_mean_squared_error', n_iter=5,
    n_jobs=-1, random_state=42)
randomized_xgb = RandomizedSearchCV(pipeline_xgb, param_distributions=
    param_dist_xgb, cv=3, scoring='neg_mean_squared_error', n_iter=5,
    n_jobs=-1, random_state=42)
randomized_cb = RandomizedSearchCV(pipeline_cb, param_distributions=
    param_dist_cb, cv=3, scoring='neg_mean_squared_error', n_iter=5,
    n_jobs=-1, random_state=42)
randomized_lgb = RandomizedSearchCV(pipeline_lgb, param_distributions=
    param_dist_lgb, cv=3, scoring='neg_mean_squared_error', n_iter=5,
    n_jobs=-1, random_state=42)

# Fit models and collect results
models = {'DecisionTree': randomized_tree,
           'XGBoost': randomized_xgb,
           'CatBoost': randomized_cb,
           'LightGBM': randomized_lgb
           }

validation_results = {}
for name, random_search in models.items():
    random_search.fit(X_train, y_train)
    best_model = random_search.best_estimator_
    y_pred = best_model.predict(X_val)
    r2 = r2_score(y_val, y_pred)
    mse = mean_squared_error(y_val, y_pred)
    print(f'{name} on validation set:')
    print(f' R-squared: {r2:.4f}')
    print(f' MSE: {mse:.4f}')
    # Store R-squared and MSE for model selection
    validation_results[name] = {'model': best_model, 'r2': r2, 'mse':
        mse}

# Select the best model based on validation performance (lowest MSE)
for name, result in validation_results.items():
mse = result['mse']
if mse < best_mse:
    best_mse = mse
    best_regression_model = result['model']

    print(f'Best regression model: {best_regression_model} with MSE: {
        best_mse:.4f}')
```

Listing 6: Regression Modeling

4.1.1 Advantages of Using MSE for This Dataset

A. Absolute Measure of Prediction Error:

- **Direct Interpretation:** MSE provides an absolute measure of the average squared prediction error, directly indicating how far the model's predictions deviate from the actual values.
- **Same Units (Squared):** Although in squared units, MSE is interpretable in the context of the target variable's units, making it useful for understanding the magnitude of errors.

B. Sensitivity to Large Errors:

- **Penalizes Large Deviations:** By squaring the errors, MSE places a higher penalty on larger errors, which is crucial in financial contexts where large prediction errors can have significant consequences.
- **Risk Management:** Prioritizing models with lower MSE can help in selecting models that minimize the risk of large losses.

C. Model Comparison Across Different Models:

- **Consistency:** MSE allows for consistent comparison across different models, even if the models use different transformations or have different complexities.
- **Hyperparameter Tuning:** Using MSE aligns the evaluation metric with the loss function optimized during model training (e.g., in Ridge, Lasso, and ElasticNet regression), facilitating more effective hyperparameter tuning.

D. Focus on Prediction Accuracy:

- **Practical Relevance:** In financial applications, the practical goal is often to minimize prediction errors rather than maximizing the explained variance.
- **Actionable Insights:** Lower MSE directly translates to more accurate predictions, which is critical for decision-making processes like loan pricing, risk assessment, and investment strategies.

4.1.2 Specific Justifications for This Loan Return Prediction Task

A. Financial Impact of Prediction Errors:

- **Loan Returns:** Predicting loan returns accurately is vital for assessing profitability and risk. Large errors in prediction can lead to incorrect investment decisions.
- **Emphasis on Error Magnitude:** Using MSE ensures that the model selection process emphasizes minimizing large prediction errors that could have significant financial repercussions.

B. Alignment with Business Objectives:

- **Risk-Averse Strategies:** Financial institutions often adopt risk-averse strategies, preferring models that minimize the potential for large losses.
- **Model Evaluation Criteria:** Selecting models based on MSE aligns with the goal of minimizing financial risk, as it prioritizes models with smaller average squared errors.

C. Suitability for Imbalanced and Noisy Data:

- **Handling Outliers:** Financial datasets may contain outliers due to extreme market movements or anomalies.
- **Robustness:** While MSE is sensitive to outliers, in this context, we want to be aware of models' performance concerning extreme values. This sensitivity encourages the selection of models that perform better even in the presence of outliers.

4.2 Classification: Predicting Early Defaults

I used non-linear models as well to predict early defaults. By performing hyperparameter tuning and threshold optimization to maximize the F1 score.

```python
# Prepare data for classification

# Define target and features for classification
X_default = loan_df[feature_cols]
X_default = X_default.select_dtypes(include=[np.number])  # Ensure numerical
    columns
y_default = loan_df['early_default']

# Handle class imbalance if necessary
class_balance = y_default.value_counts(normalize=True)
print('Class distribution:', class_balance)

# Standardize features
scaler_default = StandardScaler()
X_default_scaled = scaler_default.fit_transform(X_default)

# Use StratifiedKFold for cross-validation
```

```python
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Define models
decision_tree_clf = DecisionTreeClassifier(random_state=42)
xgboost_clf = xgb.XGBClassifier(random_state=42, use_label_encoder=False,
    eval_metric='logloss')
catboost_clf = cb.CatBoostClassifier(verbose=0, random_state=42)
lightgbm_clf = lgb.LGBMClassifier(random_state=42)

# Define parameter grids
param_dist_tree_clf = {'model__max_depth': [3, 5, 7], '
    model__min_samples_split': [2, 5]}
param_dist_xgb_clf = {'model__n_estimators': [100], 'model__max_depth': [3,
    5], 'model__learning_rate': [0.01, 0.1]}
param_dist_cb_clf = {'model__iterations': [100], 'model__depth': [3, 5], '
    model__learning_rate': [0.01, 0.1]}
param_dist_lgb_clf = {'model__n_estimators': [100], 'model__max_depth': [3,
    5], 'model__learning_rate': [0.01, 0.1]}

# Create pipelines
pipeline_tree_clf = Pipeline([('scaler', StandardScaler()), ('model',
    decision_tree_clf)])
pipeline_xgb_clf = Pipeline([('scaler', StandardScaler()), ('model',
    xgboost_clf)])
pipeline_cb_clf = Pipeline([('scaler', StandardScaler()), ('model',
    catboost_clf)])
pipeline_lgb_clf = Pipeline([('scaler', StandardScaler()), ('model',
    lightgbm_clf)])

# Define RandomizedSearchCV
models_clf = {
    'DecisionTree': (pipeline_tree_clf, param_dist_tree_clf),
    'XGBoost': (pipeline_xgb_clf, param_dist_xgb_clf),
    'CatBoost': (pipeline_cb_clf, param_dist_cb_clf),
    'LightGBM': (pipeline_lgb_clf, param_dist_lgb_clf)
}

# Use f1_score as scoring metric
f1_scorer = make_scorer(f1_score, average='binary')

validation_results_clf = {}

for name, (pipeline, param_dist) in models_clf.items():
    random_search = RandomizedSearchCV(pipeline, param_distributions=
        param_dist, cv=skf, scoring=f1_scorer, n_iter=5, n_jobs=-1,
        random_state=42)
    random_search.fit(X_default_scaled, y_default)
    best_model = random_search.best_estimator_
    y_pred = best_model.predict(X_default_scaled)
    f1 = f1_score(y_default, y_pred)
    print(f'{name}:')
    print(f'  F1 Score (training data): {f1:.4f}')
    validation_results_clf[name] = {'model': best_model, 'f1': f1}

```

```python
# Select best model based on highest F1 score
best_model_name_clf = None
best_f1_clf = 0

for name, result in validation_results_clf.items():
    f1 = result['f1']
    if f1 > best_f1_clf:
        best_f1_clf = f1
        best_model_name_clf = name
        best_classifier = result['model']

print(f'Best classification model: {best_model_name_clf} with F1 Score: {
    best_f1_clf:.4f}')

# Predict probabilities on training data
y_pred_prob = best_classifier.predict_proba(X_default_scaled)[:, 1]

# Tune threshold to maximize F1 score
thresholds = np.arange(0.1, 0.9, 0.1)
best_threshold = 0.5
best_f1_score = 0

for thresh in thresholds:
    y_pred_thresh = (y_pred_prob >= thresh).astype(int)
    f1 = f1_score(y_default, y_pred_thresh)
    if f1 > best_f1_score:
        best_f1_score = f1
        best_threshold = thresh

print(f'Final threshold set to: {best_threshold:.2f} with F1 score: {
    best_f1_score:.4f}')

# Define function to apply threshold
def apply_threshold(probs, threshold):
    return (probs >= threshold).astype(int)

# Evaluate on the training set
y_train_pred = apply_threshold(y_pred_prob, best_threshold)
train_accuracy = accuracy_score(y_default, y_train_pred)
train_auc = roc_auc_score(y_default, y_pred_prob)

print('\nIn-sample Performance for Early Default Prediction:')
print(f'Accuracy: {train_accuracy:.4f}')
print(f'F1 Score: {best_f1_score:.4f}')
print(f'AUC Score: {train_auc:.4f}')
```

Listing 7: Classification Modeling

5 Predicting on the Test Set

5.1 Preparing the Test Data

I ensured that the test data undergoes the same preprocessing and feature engineering steps as the training data.

```python
# Handle missing values in test data
missing_values_test = loan_test_df.isnull().sum()
missing_values_test = missing_values_test[missing_values_test > 0].
    sort_values(ascending=False)
print('Missing values in test data:')
print(missing_values_test)

# Impute missing values in test data
for column in missing_values_test.index:
    if loan_test_df[column].dtype in ['float64', 'int64']:
        if column in loan_df.columns:
            median = loan_df[column].median()
        else:
            median = loan_test_df[column].median()
        loan_test_df[column].fillna(median, inplace=True)
    else:
        if column in loan_df.columns:
            mode = loan_df[column].mode()[0]
        else:
            mode = loan_test_df[column].mode()[0]
        loan_test_df[column].fillna(mode, inplace=True)

# Verify no missing values remain in test data
print('\nMissing values in test data after imputation:')
print(loan_test_df.isnull().sum().sum())

# Feature engineering on test data
loan_test_df['debt_to_income'] = loan_test_df['loan_amnt'] / loan_test_df['
    annual_inc']
loan_test_df['log_loan_amount'] = np.log1p(loan_test_df['loan_amnt'])

# Encode categorical variables
for col in categorical_cols:
    if col in loan_test_df.columns:
        combined_categories = pd.concat([loan_df[col], loan_test_df[col]],
            axis=0)
        combined_categories = combined_categories.astype('category').cat.
            categories
        loan_df[col] = pd.Categorical(loan_df[col], categories=
            combined_categories).codes
        loan_test_df[col] = pd.Categorical(loan_test_df[col], categories=
            combined_categories).codes
    else:
        loan_test_df[col] = 0  # Or an appropriate default value

# Prepare test data
```

```python
X_test = loan_test_df[feature_cols]
X_test = X_test.select_dtypes(include=[np.number])
X_test = X_test.reindex(columns=feature_cols)

# Apply the same scaling
X_test_scaled = scaler.transform(X_test)
```

Listing 8: Preparing Test Data

5.2 Making Predictions

I used the trained models to make predictions on the test data.

```python
# Predict loan returns using the regression model
test_returns = best_model.predict(X_test_scaled)

# Predict early defaults using the classification model
test_default_prob = best_logreg.predict_proba(X_test_scaled)[:, 1]
test_defaults = apply_threshold(test_default_prob, best_thresh)
test_defaults = test_defaults.astype(int)
```

Listing 9: Making Predictions

5.3 Preparing the Submission File

I prepared the submission file by combining the predictions with the test set IDs.

```python
# Prepare Submission DataFrame
if 'id' in loan_test_df.columns:
    submission_df = pd.DataFrame({'id': loan_test_df['id']})
else:
    submission_df = pd.DataFrame({'id': loan_test_df.index})

# Add predictions to submission dataframe
submission_df['early_default'] = test_defaults
submission_df['return'] = test_returns

# Reorder columns to match submission requirements
submission_df = submission_df[['id', 'early_default', 'return']]

# Save predictions to CSV
submission_df.to_csv('submission_test.csv', index=False)
print('Predictions saved to submission_test.csv')
```

Listing 10: Preparing Submission File

6 Generative AI Implementation

In this assignement, Generative AI, in particular o1-mini from OpenAI and Claude Sonnet 3.5 from Anthropics were used to get a more precise view of the parameters to be used for training the various Gradient Boost models, in particular the strengths and weaknesses of XGBoost, CatBoost, and LightGBM, in order to use for testing and predictive purposes the most effective model based on the available dataset

7 Conclusion

In this assignment, I performed a comprehensive analysis of LendingClub loan data to predict loan returns and early defaults using non-linear models. Through exploratory data analysis, we gained insights into the data structure and relationships between variables. We addressed missing values and performed feature engineering to enhance our models.

Machine Learning Gradient-Boost models were employed for predicting loan returns, with hyperparameter tuning to select the best model. They were used as well for predicting early defaults, with careful threshold tuning to optimize the F1 score.

The goal of the research is to highlight how linear and nonlinear models, even of different types from each other, can give different results depending on the specific dataset. This makes it clear that there is no absolute best model, but that careful and meticulous research is needed on which model is actually the best fit for the specific dataset and the variables that we want to predict