FA 590 Statistical Learning in Finance

Assignment 2: Loan Default and Return Prediction Using Linear Models

Federica Malamisura

2024 Fall

Abstract

In this assignment, we analyze a dataset containing over one million personal loans originated between 2008 and 2017 on LendingClub. The dataset includes detailed loan characteristics, borrower characteristics, and loan outcomes. The tasks include Exploratory Data Analysis (EDA), Data Preprocessing, Building Regression and Classification Models using Linear Models, Feature Importance Analysis using SHAP, and Making Predictions on the Test Set. We utilize linear regression and logistic regression models to predict loan returns and early defaults, respectively, and assess feature importance to understand the factors influencing these loan outcomes. The models are evaluated using appropriate metrics, and predictions are made on the test set for submission.

Contents

1 Introduction

In this assignment, we focus on analyzing and predicting loan defaults and returns using data from LendingClub. Accurately predicting these outcomes is crucial for financial institutions to manage risk effectively and make informed lending decisions. The dataset comprises over one million personal loans issued between 2008 and 2017, providing a rich source of information for analysis.

We perform Exploratory Data Analysis to understand the data structure and identify key relationships. Data preprocessing steps include handling missing values, feature engineering, and encoding categorical variables. We employ linear regression models to predict loan returns and logistic regression models for predicting early defaults. Feature importance is assessed using SHAP values, allowing us to interpret the models and identify the most influential features. Finally, we make predictions on the test set and prepare the results for submission.

2 Importing Libraries and Loading Data

2.1 Import Libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, GridSearchCV,
    TimeSeriesSplit
from sklearn.linear_model import LinearRegression, Lasso, Ridge, ElasticNet,
    LogisticRegression
from sklearn.metrics import mean_squared_error, r2_score, accuracy_score,
    f1_score, roc_auc_score, classification_report, confusion_matrix
import shap
from sklearn.preprocessing import StandardScaler
```

Listing 1: Importing Libraries

2.2 Loading Data

```
# 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
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 Exploratory Data Analysis

3.1 Understanding the Data Structure

```
# Display summary statistics for key variables
key_vars = ['loan_amnt', 'int_rate', 'fico_range_high', 'annual_inc']
loan_df[key_vars].describe()
```

Listing 3: Exploring the Dataset

We explore the dataset to understand its structure, the types of variables it contains, and the presence of missing values.

```
# Check data types and missing values
loan_df.info()
print('\nMissing values in training data:')
print(loan_df.isnull().sum())
```

Listing 4: Data Information and Missing Values

3.2 Distribution of Target Variables

```
# Distribution of 'early_default'
sns.countplot(x='early_default', data=loan_df)
plt.title('Distribution of Early Default')
plt.show()
```

Listing 5: Distribution of Early Default

```
# Distribution of 'return'
sns.histplot(loan_df['return'], kde=True)
plt.title('Distribution of Loan Returns')
plt.show()
```

Listing 6: Distribution of Loan Returns

We examine the distribution of the target variables `early_default` and `return` to gain insights into their characteristics.

3.3 Correlation Analysis

```
# Correlation matrix of numerical features
corr = loan_df.select_dtypes(include=['float64', 'int64']).corr()

# Heatmap of the correlation matrix
plt.figure(figsize=(12, 10))
sns.heatmap(corr, annot=True, fmt=".2f", cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```

Listing 7: Correlation Matrix

We generate a correlation matrix for numerical features to identify potential relationships between variables.

3.4 Additional Insights

```
# Relationship between FICO score and interest rate
plt.figure(figsize=(8,6))
sns.scatterplot(x='fico_range_high', y='int_rate', data=loan_df, alpha=0.3)
plt.title('FICO Score vs Interest Rate')
plt.show()
```

Listing 8: FICO Score vs. Interest Rate

An exploratory analysis of the relationship between the borrower's FICO score and the interest rate of the loan is conducted.

4 Data Preprocessing

4.1 Sampling Strategy: Expanding-Window Approach

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 an **expanding window approach** for the following reasons:

1. **Data Utilization**: The expanding window approach utilizes all available historical data up to the current point for training. This means that as time progresses, the model is trained on an increasing amount of data, potentially capturing more comprehensive patterns.

2. **Real-World Scenario Representation**: In financial contexts, especially in credit risk modeling, it is realistic to assume that models are updated as more data becomes available. The expanding window mirrors this practice.

3. **Model Stability**: Using more data for training can lead to more stable and robust models, reducing variance and improving generalization.

4. **Avoiding Data Leakage**: Since future data is not used in training past predictions, the expanding window prevents data leakage, maintaining the integrity of model evaluation.

In contrast, a rolling window approach uses a fixed-size window that moves forward in time, potentially discarding valuable historical data. While rolling windows are useful for non-stationary data where recent observations are more relevant, the expanding window is more appropriate when historical data remains informative over time.

```
# 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 9: Defining Expanding Window Splits

4.2 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 10: Handling Missing Values

4.3 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 11: Feature Engineering

5 Model Building

5.1 Regression: Predicting Loan Returns

I implemented and evaluated several linear regression models:

- **Linear Regression**: Serves as a baseline model.
- **Ridge Regression**: Addresses multicollinearity by introducing L2 regularization.
- **Lasso Regression**: Performs feature selection through L1 regularization.
- **ElasticNet**: Combines L1 and L2 regularization.

```
# 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 expanding-window approach
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]

    # Hyperparameter tuning with GridSearchCV
    ridge = Ridge()
    lasso = Lasso()
    elastic = ElasticNet()

    # Define parameter grid
    param_grid = {'alpha': [0.01, 0.1, 1, 10, 100]}

    # Ridge Regression
    grid_ridge = GridSearchCV(ridge, param_grid, cv=5, scoring='
        neg_mean_squared_error', n_jobs=-1)
    grid_ridge.fit(X_train, y_train)
    best_ridge = grid_ridge.best_estimator_
```

```
27
28      # Lasso Regression
29      grid_lasso = GridSearchCV(lasso, param_grid, cv=5, scoring='
            neg_mean_squared_error', n_jobs=-1)
30      grid_lasso.fit(X_train, y_train)
31      best_lasso = grid_lasso.best_estimator_
32
33      # ElasticNet
34      param_grid_elastic = {'alpha': [0.01, 0.1, 1, 10], 'l1_ratio': [0.2, 0.5,
            0.8]}
35      grid_elastic = GridSearchCV(elastic, param_grid_elastic, cv=5, scoring='
            neg_mean_squared_error', n_jobs=-1)
36      grid_elastic.fit(X_train, y_train)
37      best_elastic = grid_elastic.best_estimator_
38
39      # Evaluate models
40      models = {'Ridge': best_ridge, 'Lasso': best_lasso, 'ElasticNet':
            best_elastic}
41      validation_results = {}  # To store results for each model
42
43      for name, model in models.items():
44          y_pred = model.predict(X_val)
45          r2 = r2_score(y_val, y_pred)
46          mse = mean_squared_error(y_val, y_pred)
47          print(f'{name} on validation set:')
48          print(f'  R-squared: {r2:.4f}')
49          print(f'  MSE: {mse:.4f}')
50          # Store MSE for model selection
51          validation_results[name] = {'model': model, 'mse': mse}
52
53  # Select the best model based on validation performance (lowest MSE)
54  best_model = None
55  best_mse = float('inf')
56
57  for name, result in validation_results.items():
58      mse = result['mse']
59      if mse < best_mse:
60          best_mse = mse
61          best_model = result['model']
62
63  print(f'Best model: {best_model} with MSE: {best_mse:.4f}')
```

Listing 12: Regression Modeling

5.1.1 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.

5.1.2 4. 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.

5.2 Classification: Predicting Early Defaults

I used logistic regression to predict early defaults. Due to computational considerations, we sample 10% of the dataset. By performing hyperparameter tuning and threshold optimization to maximize the F1 score.

```
# Sample 10% of the dataset to speed up computation
loan_df_sampled = loan_df.sample(frac=0.1, random_state=42)

# Define target and features for default prediction using the sampled data
X_default = loan_df_sampled[feature_cols]
X_default = X_default.select_dtypes(include=[np.number])
y_default = loan_df_sampled['early_default']

# Standardize features using the same scaler
X_default_scaled = scaler.transform(X_default)

# Define the model with a reduced parameter grid
logreg = LogisticRegression()
param_grid = {
    'C': [0.1, 1, 10],        # Reduced number of C values
    'penalty': ['l1', 'l2'],  # Penalty types
    'solver': ['liblinear']    # 'liblinear' solver supports 'l1' and 'l2'
}

# Define custom scorer to maximize F1 score
def custom_scorer(estimator, X, y):
    y_pred_prob = estimator.predict_proba(X)[:, 1]
    thresholds = np.arange(0.1, 0.9, 0.1)
    best_f1 = -np.inf
    for thresh in thresholds:
        y_pred = (y_pred_prob >= thresh).astype(int)
        f1 = f1_score(y, y_pred)
        if f1 > best_f1:
            best_f1 = f1
    return best_f1

f1_scorer = make_scorer(custom_scorer, needs_proba=False, greater_is_better=
    True)

# Perform GridSearchCV with fewer folds (e.g., cv=3)
grid_search = GridSearchCV(logreg, param_grid, cv=3, scoring=f1_scorer, n_jobs
    =-1, verbose=1)
grid_search.fit(X_default_scaled, y_default)

# Get the best model
best_logreg = grid_search.best_estimator_

# Predict probabilities on the entire sampled dataset
y_pred_prob = best_logreg.predict_proba(X_default_scaled)[:, 1]

# Tune threshold to maximize F1 score on the sampled data
thresholds = np.arange(0.1, 0.9, 0.1)
best_f1 = -np.inf
```

```
best_thresh = 0.5

for thresh in thresholds:
    y_pred = (y_pred_prob >= thresh).astype(int)
    f1 = f1_score(y_default, y_pred)
    if f1 > best_f1:
        best_f1 = f1
        best_thresh = thresh

print(f'Final threshold set to: {best_thresh:.2f} with F1 score: {best_f1:.4f}')

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

# Evaluate on the sampled training set
y_train_pred = apply_threshold(y_pred_prob, best_thresh)
train_accuracy = accuracy_score(y_default, y_train_pred)
train_f1 = f1_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: {train_f1:.4f}')
print(f'AUC Score: {train_auc:.4f}')
```

Listing 13: Classification Modeling

6 Feature Importance

6.1 Feature Importance Using SHAP Values

We use SHAP (SHapley Additive exPlanations) values to interpret our models and identify the most significant features influencing the predictions.

```
# Convert scaled arrays back to DataFrames with column names
X_scaled_df = pd.DataFrame(X_scaled, columns=X.columns)
X_default_scaled_df = pd.DataFrame(X_default_scaled, columns=X_default.columns)

# SHAP for Regression Model
explainer_reg = shap.Explainer(best_model, X_scaled_df)
shap_values_reg = explainer_reg(X_scaled_df)

# Plot feature importance for regression
shap.summary_plot(shap_values_reg, features=X_scaled_df, plot_type='bar', max_display=10)

# Get top 10 features for regression
shap_importance_reg = pd.DataFrame({
    'feature': X_scaled_df.columns,
```

```
    'importance': np.abs(shap_values_reg.values).mean(axis=0)
})
shap_importance_reg = shap_importance_reg.sort_values(by='importance',
    ascending=False).head(10)
print('Top 10 Important Features for Regression:')
print(shap_importance_reg)

# SHAP for Classification Model
explainer_clf = shap.Explainer(best_logreg, X_default_scaled_df)
shap_values_clf = explainer_clf(X_default_scaled_df)

# Plot feature importance for classification
shap.summary_plot(shap_values_clf, features=X_default_scaled_df, plot_type='
    bar', max_display=10)

# Get top 10 features for classification
shap_importance_clf = pd.DataFrame({
    'feature': X_default_scaled_df.columns,
    'importance': np.abs(shap_values_clf.values).mean(axis=0)
})
shap_importance_clf = shap_importance_clf.sort_values(by='importance',
    ascending=False).head(10)
print('Top 10 Important Features for Classification:')
print(shap_importance_clf)
```

Listing 14: Feature Importance with SHAP

7 Predicting on the Test Set

7.1 Preparing the Test Data

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

```
# 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
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 15: Preparing Test Data

7.2 Making Predictions

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

```
# 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 16: Making Predictions

7.3 Preparing the Submission File

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

```
# 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 17: Preparing Submission File

8 Conclusion

In this assignment, I performed a comprehensive analysis of LendingClub loan data to predict loan returns and early defaults using 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.

Linear regression models were employed for predicting loan returns, with hyperparameter tuning to select the best model. Logistic regression was used for predicting early defaults, with careful threshold tuning to optimize the F1 score. Feature importance analysis using SHAP values provided interpretability and insights into the significant factors influencing loan outcomes.

While linear models offer simplicity and interpretability, they may not capture complex non-linear relationships inherent in financial data. Future work could involve exploring non-linear models, ensemble methods, and advanced feature engineering to improve predictive performance. Overall, this assignment highlights the challenges and considerations in building predictive models in finance using linear approaches.