Building a Custom Logistic Regression Model with Class Imbalance

30-December-2024

Team:
Rizvan Nahif,
David Abad,
Navpreet Kaur Dusanje,
Darshil Shah

Introduction

Overview

In today's digital marketplace, understanding and mitigating the financial risks associated with online transactions is crucial for businesses. This project focuses on developing a predictive machine learning model to assess and classify the risk of payment defaults for an online trader's orders. Each transaction in the provided dataset includes details that help identify risk patterns, allowing us to classify orders as either “high-risk” or “low-risk” based on the probability of payment default. This classification model aims to assist the online trader in managing and mitigating financial risk, by flagging potentially high-risk orders before they are processed.

This problem naturally aligns with binary classification techniques, where we categorize each order based on its potential risk level. For this project, we propose to implement a logistic regression model, given its suitability for binary classification problems and interpretability, which can be valuable for business stakeholders. However, a significant challenge lies in the asymmetry of misclassification costs: the cost of predicting a high-risk customer as low-risk is substantially greater than the reverse. To address this, our logistic regression model will require modifications to account for a custom cost matrix, ensuring that the model prioritizes minimizing high-cost misclassifications.

Goal

Develop a robust, cost-sensitive classification model that accurately predicts the likelihood of payment default for each online order. The model will be tailored to prioritize high-risk orders, minimizing the financial impact of misclassification errors, and empowering the client to make proactive, data-driven decisions to manage payment risks more effectively.

Dataset Description

Data Overview

The dataset provided by the online trader contains 30,000 records of individual purchase orders. Each order is described by 44 attributes, which cover various aspects of the transaction, customer details, and payment-related information. This comprehensive set of features includes indicators for email and phone verification, customer demographics, order amount and frequency, and different validation checks, among others. These attributes serve as potential predictors for the risk of default payment on an order.

The Class attribute is the target variable in this dataset, indicating whether an order is high-risk or low-risk. It is a binary classification label where:

- **yes** - represents high-risk orders likely to result in payment defaults
- **no** - represents low-risk orders where payments are likely to be successful

This target variable allows the machine learning model to learn patterns in the features that differentiate high-risk orders from low-risk ones. Additionally, the class imbalance and cost matrix associated with this variable underscore the importance of accurately predicting high-risk orders to minimize potential losses.

Data Preprocessing Steps

Handling Missing Values

1. **Initial Missing Value Check**
   A quick check was conducted to identify columns with missing values. This allowed us to understand the extent of missing data across each feature.
2. **Threshold-Based Column Removal**
   Columns with a high proportion of missing values were removed based on a predefined threshold. Specifically, columns were dropped if they had missing values exceeding 50% of the total number of rows in the dataset. This decision was made to prevent the model from being influenced by sparse or unreliable features.
3. **Selective Feature Removal**

Certain columns, such as Last_Name, were removed manually due to limited relevance or potential data privacy considerations. This step also reduced noise in the dataset.

4. **Imputation of Remaining Missing Values**
    - **Categorical Features**: For features with object data types, missing values were replaced with the mode (most frequently occurring value) to preserve the dominant category.
    - **Numerical Features**: For numeric columns, missing values were imputed with the median, as this approach is less sensitive to outliers compared to mean imputation.

This method ensured that the dataset was both complete and relevant, with missing values systematically handled to maximize data integrity. By retaining essential features and filling in missing values thoughtfully, the dataset was prepared for robust model training and analysis.

Feature Engineering

In this project, several data transformation techniques were applied to prepare the dataset for modeling. The following transformations were executed:

1. **Categorical Column Check**: A function was created to check the number of unique categories in each categorical column. This helps identify columns with a single category, which may not be useful for modeling
2. **Binary Conversion of Yes/No Columns**: Several columns containing binary responses (yes/no) were converted into integer format (1 for 'yes' and 0 for 'no')
3. **Feature Creation for Ordered Hours**: A new feature called Ordered_hours was created by converting the Order_Time column to datetime format and extracting the hour and minute values:
4. **Encoding Categorical Variables:** Categorical variables, specifically Payment_Method and Order_Weekday, were encoded using one-hot encoding to convert them into numerical format suitable for machine learning algorithms
5. **Converting Boolean Columns to Integer**: The resulting binary columns from one-hot encoding were converted to integer type to ensure consistency.
6. **Standardization of Features**: The features matrix was standardized using the StandardScaler from scikit-learn. Standardization ensures that each feature contributes equally to the analysis by centering the data around zero and scaling it to have unit variance.

Classification Method and Model Selection

This section outlines the rationale behind the model selection and explains the customization required to address the cost-sensitive nature of the classification problem. The approach involves a custom implementation of logistic regression, specifically designed to accommodate the provided cost matrix. Rather than relying on an off-the-shelf library, the logistic regression model was implemented from scratch to ensure full control over its behavior and facilitate the seamless integration of custom cost-sensitive logic..

Building A Logistic Regression Algorithm.

Creating a logistic regression model from scratch requires several essential components. Class imbalance was addressed by incorporating weights directly into the gradient descent algorithm. Two different methods were implemented to integrate these weights into the gradient descent process. Logistic regression typically includes the following:

- **Sigmoid Function**: This function transforms the output of the model into a probability score, ranging from 0 to 1, representing the likelihood of an order belonging to the high-risk class.

```
# This the sigmnoid function for the logistic regression
def sigmoid(self, z):
    return 1 / (1 + np.exp(-z))
```

- **Gradient Descent A**:The gradient_descent_with_class_weights function incorporates class-specific weights directly into the gradient descent process. It uses a predefined dictionary (class_weights) to assign a weight to each class. These weights are applied to the error term, ensuring that the contribution of misclassified examples is scaled according to their class importance. The weighted errors are then used to compute the gradient, which updates the model parameters (self.__theta). This method provides flexibility for assigning arbitrary weights to classes, making it well-suited for addressing imbalanced datasets.

```python
# This is the gradient decent with the weights
def gradient_descent_with_class_weights(self, x, y, class_weights):
    self.__theta = np.zeros((x.shape[1], 1))
    m = x.shape[0]
    j_all = []

    for _ in range(self.num_epochs):
        # Compute predictions
        h_x = self.sigmoid(x @ self.__theta)

        # Calculate weighted error term
        weighted_errors = (h_x - y) * np.array(
            [class_weights[int(label)] for label in y.flatten()]
        ).reshape(-1, 1)

        # Compute the gradient using the weighted error term
        gradient = (1 / m) * (x.T @ weighted_errors)

        # Update self.__theta
        self.__theta = self.__theta - self.learning_rate * gradient

        # Track cost for analysis
        cost = self.cost_function(x, y, self.__theta)
        j_all.append(cost)

    return self.__theta, j_all
```

- **Gradient Descent B**:The alternate_gradient_descent function applies misclassification costs dynamically based on the predictions and ground truth labels. Instead of using a pre-defined dictionary for weights, it calculates costs using conditional logic. A higher cost is assigned when a high-risk example is misclassified as low risk and vice versa. This approach directly ties the weighting scheme to the model's predictive behavior during each iteration. Unlike the gradient_descent_with_class_weights method, this approach focuses on penalizing specific types of misclassification errors, allowing for more tailored cost adjustments based on the problem's requirements.

```python
def alternate_gradient_descent(self, x, y):
    self.__theta = np.zeros((x.shape[1], 1))
    m = x.shape[0]
    J_all = []

    high_risk_cost = 50  # Cost for predicting low risk when actual is high risk
    low_risk_cost = 5  # Cost for predicting high risk when actual is low risk

    for _ in range(self.num_epochs):
        h_x = self.sigmoid(x @ self.__theta)  # Predicted probabilities
        error = h_x - y  # Error term

        # Apply misclassification costs
        weighted_error = error * np.where(
            (y == 1) & (h_x < 0.5),
            high_risk_cost,
            np.where((y == 0) & (h_x >= 0.5), low_risk_cost, 1),
        )

        # Calculate the gradient with weighted error
        gradient = (1 / m) * (x.T @ weighted_error)
        self.__theta = self.__theta - self.learning_rate * gradient

        # Track cost without misclassification for reference
        J_all.append(self.cost_function(x, y, self.__theta))

    return self.__theta, J_all
```

- **Cost Function (Cross-Entropy Loss)**: The cross entropy loss function which is given below.

$$L = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(h_i) + (1 - y_i)\log(1 - h_i)]$$

The following function is used to calculate the cross loss.

```
# This calculates the cost function
def cost_function(self, x, y, theta):
    h = self.sigmoid(x @ theta)
    one = np.ones((y.shape[0], 1))
    cost = -((y.T @ np.log(h)) + (one - y).T @ np.log(one - h)) / (y.shape[0])
    return cost
```

  - `x @ theta` computes the linear combination of features and model parameters.
  - `self.sigmoid(x @ theta)` applies the sigmoid function to convert the linear outputs into probabilities.
  - Logarithmic terms (**`np.log(h)`** and **`np.log(1 - h)`**) represent the log-probabilities for predicted values when the label is either 1 or 0, respectively.
  - `y.T @ np.log(h)` calculates the contribution of correctly predicting positive class examples.
  - `(1 - y).T @ np.log(1 - h)` calculates the contribution of correctly predicting negative class examples.

- **Fit Function**: This function is used to train the model with the provided training data (x_train and y_train). It iteratively applies the gradient descent algorithm to optimize the model's parameters based on the training data, minimizing the error, and improving the model's accuracy over time.

- **Predict Function**: Generates the final classification (high-risk or low-risk) for each order based on the probability output from the sigmoid function based on the trained weight and bias.

Hyperparameter Optimization

We have implemented a simple function that performs a grid search over three hyperparameters: learning_rate, num_epochs, and test_size. This would help us evaluate the performance of different hyperparameter configurations.
We found the best values to use in the following parameters:

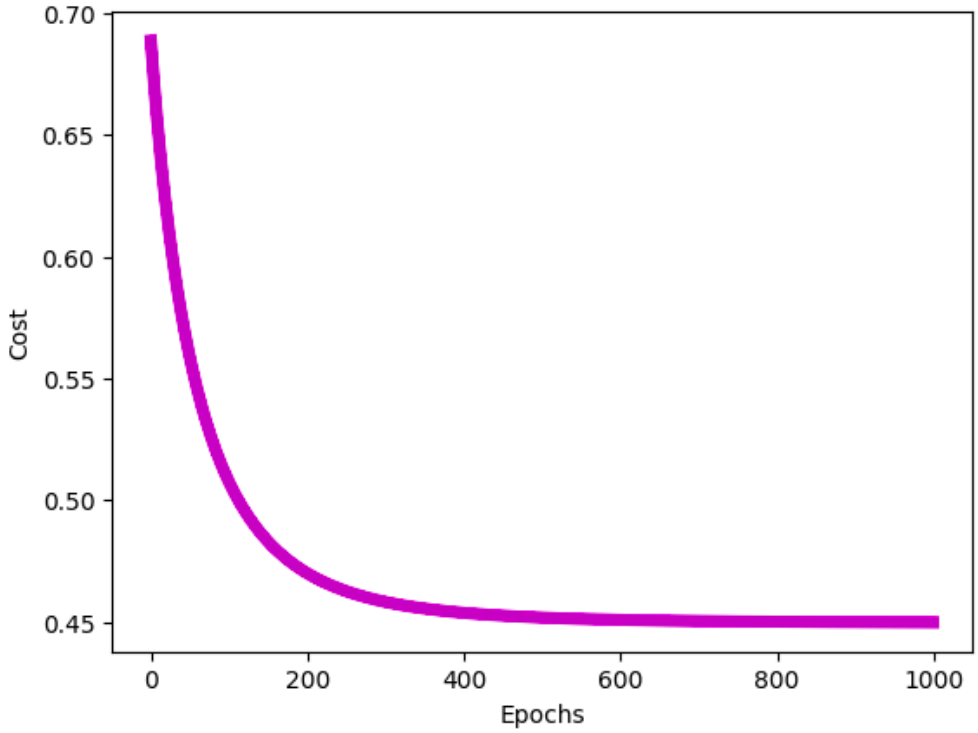
- test_size: 0.25
- learning_rate: 0.01
- num_iterations: 1000

Model Performance

The performance of both the gradient descent solution is given below. We will look at the cost vs iteration as well as the confusion matrix of each solution to compare the results.

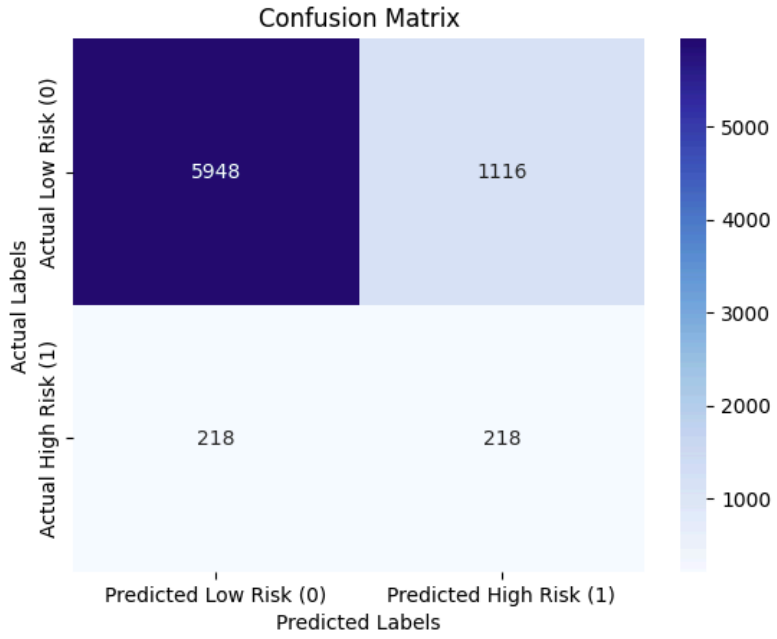
1. Gradient Descent A:

The following is the cost vs iteration of the Gradient Descent A solution



The cost vs. epoch graph demonstrates the model's convergence during training:The cost decreases rapidly in the early epochs, indicating effective learning and significant reduction in error.After approximately 400 epochs, the cost reduction slows down and

plateaus, suggesting that the model has nearly converged to its optimal parameters.This is a typical learning curve for logistic regression, where the model minimizes the cross-entropy loss effectively.



The confusion matrix provides insight into the model's performance:

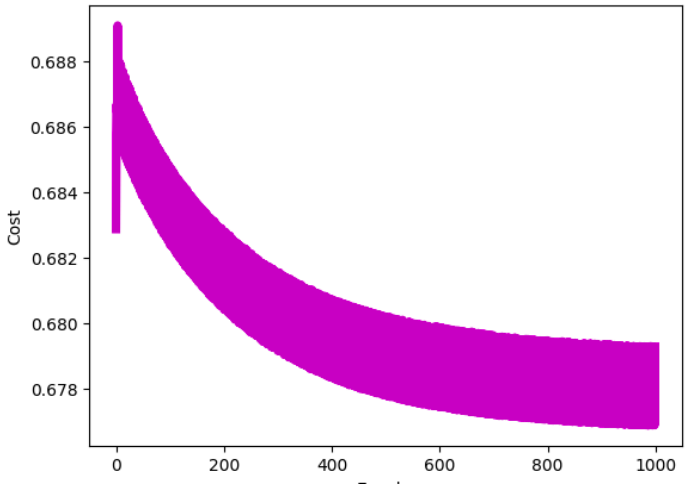
- **True Negatives (5948)**: The model correctly classified 5948 instances of low risk.
- **False Positives (1116)**: 1116 low-risk cases were incorrectly classified as high risk.
- **False Negatives (218)**: 218 high-risk cases were incorrectly classified as low risk.
- **True Positives (218)**: The model correctly classified 218 instances of high risk.

Key Findings:

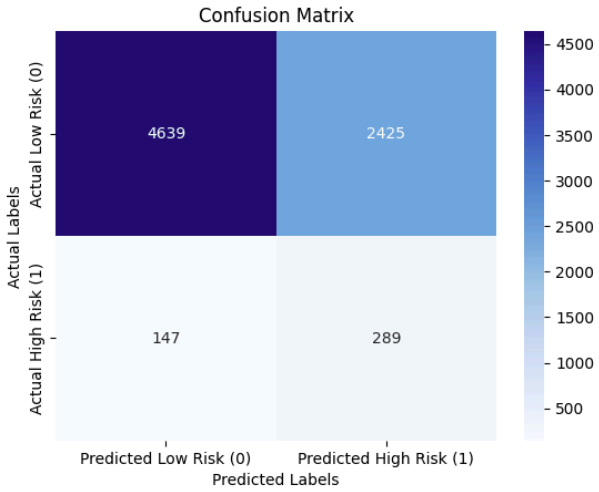
- **Class Imbalance**: The matrix highlights the class imbalance, with many more low-risk examples than high-risk ones.
- **Sensitivity Issue**: The number of false negatives (218) is equal to the number of true positives (218), which might be concerning if correctly identifying high-risk cases is critical.
- **Cost-Sensitive Learning**: While the model has incorporated class weights, it still struggles with high-risk identification, suggesting potential room for improvement, such as further tuning the cost weights or trying alternative approaches.

2. Gradient Descent B:

For this solution we get a chaotic cost vs iteration plot.



The cost shows fluctuations all the way through, which may indicate instability in learning due to the way the weights or gradient updates are applied. However, it is clear that the cost steadily decreases with iterations.



The confusion matrix provides insight into the model's performance:

- **True Negatives (4639)**: The model correctly classified 4639 instances of low risk.
- **False Positives (2425)**: 2425 low-risk cases were incorrectly classified as high risk.
- **False Negatives (147)**: 147 high-risk cases were incorrectly classified as low risk.
- **True Positives (289)**: The model correctly classified 289 instances of high risk.

Key Findings:

- **Class Imbalance**: The matrix highlights the class imbalance, with many more low-risk examples than high-risk ones.
- **Accuracy:** The overall accuracy fell drastically in this solution from a previous 82.2% to 65.7%.
- **Recall:** We get higher recall in this solution which increased from 50% to 66.3%. This is an important improvement because we want the model to correctly predict the high risk.

Interpretation and Conclusion:

The results show that the model is struggling to accurately classify the minority class (high-risk orders) despite applying custom class weights. The confusion matrix highlights that. Setting custom class weights is a reasonable attempt to handle the class imbalance. It penalizes the misclassification of high-risk orders more heavily than low-risk ones, theoretically encouraging the model to pay more attention to the high-risk class.

Nevertheless, the results indicate that the chosen weights did not fully solve the imbalance issue, as the model still struggles to achieve high recall for the high-risk class. This may suggest that logistic regression, even with class weighting, might not be complex enough to capture the distinctions between high-risk and low-risk orders in this dataset.

Recommendations:

1. **Regularization**: Incorporate regularization techniques (e.g., L1 or L2 regularization) in both methods to prevent overfitting and improve generalization.
2. **Optimization Techniques**: Explore advanced optimizers like Adam or RMSprop, which could stabilize early epochs and enhance convergence for both approaches.
3. **Data Balancing**: Complement the cost-sensitive approaches by using data-level techniques, such as oversampling the minority class or undersampling the majority class, to reduce the imbalance effect.
4. **Different Algorithms**: We should try to solve the problem using other algorithms like Random Forest or Bayes Classifier to see if we get better results.

References:

[1] Scikit-Learn Developers. (n.d.). sklearn.decomposition.PCA. Scikit-Learn. Retrieved from https://scikit-learn.org/dev/modules/generated/sklearn.decomposition.PCA.html

[2] Kumar, A. (2021, January 15). Coding logistic regression from scratch. Towards Data Science. Retrieved from https://towardsdatascience.com/coding-logistic-regression-from-scratch-d18b4fbfca8a

[3] Fraud Detection Handbook. (n.d.). Chapter 6: Imbalanced Learning - Cost Sensitive Learning. Retrieved from https://fraud-detection-handbook.github.io/fraud-detection-handbook/Chapter_6_ImbalancedLearning/CostSensitive.html

[4] Tantai, H. (2023, February 23). Use weighted loss function to solve imbalanced data classification problems. Medium. Retrieved from https://medium.com/@zergtant/use-weighted-loss-function-to-solve-imbalanced-data-classification-problems-749237f38b75