Predicting Mortgage Default using XGBoost for Classification

Final Project - Machine Learning, Section INT 2

Morgan Waddington, Divya Srinivasan, Pranav Raj Narayanan Kavitha

December 11, 2024

Abstract

This project explored the challenges and insights involved in predicting loan defaults using machine learning. The complexity of the dataset demanded extensive preprocessing, including feature elimination, anomaly recalibration, and synthetic data generation to mitigate bias. We compared a baseline decision tree model to the more sophisticated XGBoost, employing techniques like GridSearchCV for parameter optimization and oversampling to equalize classes and lower bias as well as undersampling to improve recall for defaulted loans. XGBoost demonstrated improvements, with accuracy increasing from 85% to 87% and recall for defaulted loans rising from 0.07 to 0.27—a 20% absolute improvement. Further experimentation with undersampling reduced accuracy to 75% but significantly boosted recall to 66%, correctly identifying 2,825 out of 4,309 defaults. These findings highlight the critical role of iterative learning and advanced regularization. Our work underscores the importance of clean, complete data for better model performance and illustrates how industry practices like archiving defaulted loan records can limit predictive insights. Improved data collection policies could substantially enhance outcomes in banking and related fields.

1 Introduction

As the world increasingly turns to artificial intelligence, the financial sector leads this transformation. A recent survey of UK Finance members revealed that 90 percent have already deployed AI technologies [1]. Indeed, institutions have begun leveraging machine learning for applications such as trading, risk assessment, fraud detection, and more. However, predicting whether a loan will default or remain current continues to be a critical problem in the financial industry, with significant implications for banks, REITs, hedge funds, and other market participants. [2] Accurate predictions can improve investment decisions, enhance risk management, and optimize asset pricing. For instance, banks could avoid overpaying for risky loans or paying excessive premiums for safer ones. Additionally, predictive models can inform repo lending decisions, helping financial institutions mitigate risks by offering less credit on loans with a higher likelihood of default.[3] This project aims to address this challenge using a machine learning algorithm called XGBoost, to classify mortgage loan outcomes based on borrower characteristics and loan performance data. Beyond practical applications, this project also provides a deeper understanding of the factors influencing mortgage performance and equips the average reader with valuable experience in developing predictive models.

2 Literature Review

Chen and Guestrin [4] introduced XGBoost, a scalable machine learning system designed for gradient tree boosting, widely recognized for its superior performance in numerous data science competitions, including Kaggle and KDDCup. The authors proposed several innovations, such as a sparsity-aware algorithm for handling missing values, a weighted quantile sketch for approximate learning, and cache-aware and out-of-core computation techniques. These optimizations enabled XGBoost to scale efficiently to datasets with billions of examples, outperforming traditional tree boosting systems in speed and memory usage. The study highlighted XGBoost's adaptability across diverse machine learning tasks, including classification, ranking, and regression, demonstrating benchmark results such as the Higgs boson event classification and Yahoo's learning-to-rank dataset. A comprehensive evaluation revealed that XGBoost not only accelerates training significantly but also reduces overfitting through techniques like column subsampling and regularized objective functions. These contributions establish XGBoost as a leading tool in scalable and accurate predictive modeling. Ouyang [5] investigated the comparative efficacy of logistic regression and XGBoost in predicting loan defaults, focusing on key financial indicators such as loan grade, annual income, debt-to-income ratio, and past delinquencies. Their methodology included rigorous data preprocessing—handling missing values, encoding categorical variables, and selecting optimal features. Logistic regression, a classical statistical method, was juxtaposed with XGBoost, an advanced ensemble learning model. The study found that XGBoost outperformed logistic regression in terms of predictive accuracy, as evidenced by a higher AUC value from ROC curve analysis. Feature importance analysis revealed pivotal determinants of loan default, providing actionable insights for financial institutions. The findings underscore the potential of machine learning approaches like XGBoost in enhancing financial risk assessment, offering improved tools for mitigating loan default risks.

3 Models

3.1 Baseline Model: Decision Tree

One requirement of this project is to use a baseline model, which makes sense as it provides a valuable reference point to measure and compare the performance of the main model, XGBoost. We chose a decision tree as our baseline model for several reasons.

First, decision trees are simple and widely used as a standard reference model in machine learning projects. Second, one of their key benefits is their interpretability: they produce a visual representation of the decision-making process, making it easy to identify the most important criteria for classifying loans as performing or nonperforming. This makes them particularly engaging for presentations, as they effectively "tell a story" about the factors influencing loan performance.

In addition, decision trees simplify complex technical features, allowing us to explain and analyze their correlations more effectively, which is essential for this industry-specific project. Lastly, XGBoost itself is an ensemble method built using multiple decision trees, making this baseline a natural complement to our main model.

Unlike XGBoost, we will not delve deeply into the inner workings of the decision tree model, since it serves as a baseline. However, in broad terms, the model operates as follows:

As the name suggests, a decision tree builds a tree-like structure where each node represents a threshold based on a feature of the loans (e.g., Loan-to-Value ratio, LTV).[6] The model splits the loans into two child nodes depending on whether they meet the threshold (true) or not (false). But how does it determine which feature provides the best split for classification? It does so by measuring the resulting impurity of the child nodes.

The impurity quantifies how mixed the classes are within a node. There are several methods to measure impurity, but the most common one -used in our baseline model - is the Gini index. The process begins by calculating the Gini impurity for the potential leaf nodes. For binary (non-numeric) features, the formula is:

$$\text{Gini} = 1 - \left(\frac{\#\text{defaults}}{\#\text{loans in leaf}}\right)^2 - \left(\frac{\#\text{non-defaults}}{\#\text{loans in leaf}}\right)^2$$

Next, the weighted sum of the Gini values for the child nodes is calculated to determine the impurity of the parent node. For numeric features, the process is slightly different: the data is sorted, and adjacent values are averaged to create potential thresholds. The Gini impurity is then calculated for each threshold, using the same formula as above, by testing whether a loan is above or below the average.

The model selects the feature and threshold that produce the lowest Gini index and splits the data accordingly. It continues splitting unless the Gini index reaches 0 (indicating a pure node, where all loans are either default or current) or the tree reaches its maximum depth. Once the tree is complete, the category with the majority of instances in each leaf becomes its output.

3.2 XGBoost

As part of this project, we explored the inner workings of XGBoost to better understand its functionality and advantages. In the following section, we will dive into the finer details—such as initial predictions, pruning techniques, and key calculations—to set the stage for the paper's results. We believe that a solid grasp of these foundations will help contextualize the observations discussed later.

3.2.1 XGBoost – Initial Prediction

The first step in fitting this ML model to the training data is to make an initial prediction. The default prediction, whether for classification or regression, is typically set to 0.5.[7]

This means we initially assume a 50% probability of default. If we plotted loan data, this 0.5 prediction would appear as a horizontal line across the plot, where the x-axis represents the Loan-to-Value ratio (LTV), and the y-axis shows the probability of loan default. The line separates loans labeled as defaulted from those still current, with each category on opposite sides of the line (see Figure 1).

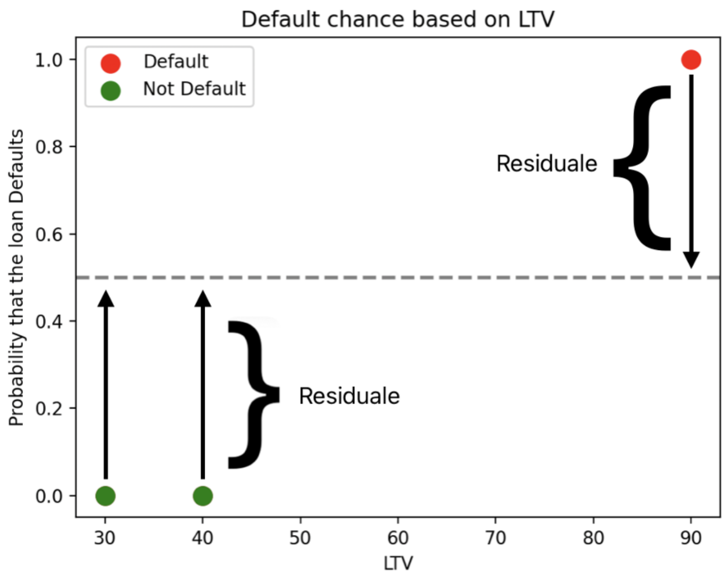


Figure 1: Initial prediction of default probability based on Loan-to-Value ratio (LTV). The horizontal line at 0.5 represents the default assumption of 50% probability.

The difference between the actual value and the prediction is called the residual. Residuals measure how accurate the prediction is. Next, the model fits a regression tree to these residuals.

3.2.2 Similarity Score

Every tree begins as a single leaf (root), which includes all the residuals. Using these residuals, we calculate a similarity score to determine the best way to split the tree. The formula for the similarity score differs depending on whether XGBoost is used for regression or classification.

For regression:

$$\text{Similarity score} = \frac{(\sum \text{residuals})^2}{\text{number of residuals} + \lambda}$$

For classification:

$$\text{Similarity score} = \frac{(\sum \text{residuals})^2}{\sum[\text{previousProbability} \times (1 - \text{previousProbability})] + \lambda}$$

The numerator, $(\sum \text{residuals})^2$, represents the sum of all errors squared, emphasizing larger errors. In regression, the denominator is simply the number of instances in the leaf. For classification, the denominator incorporates *previousProbability*, which is the probability predicted for each instance in the last tree. For the first tree, *previousProbability* defaults to the initial prediction of 0.5.

3.2.3 Regularization

Lambda (λ) is a regularization parameter designed to reduce the model's sensitivity to individual instances, helping to prevent overfitting. It does so by penalizing high similarity scores, ensuring the model doesn't focus too heavily on small sample sizes. A higher λ reduces the similarity score, particularly in leaves with fewer residuals, because the penalty is more pronounced when the count of instances is small.

3.2.4 Gain

Once the similarity score for the root node is calculated, the model attempts to split instances into separate leaves based on thresholds (e.g., LTV > 35) and calculates the similarity score for each split. When residuals in a leaf vary significantly, they can cancel each other out, resulting in a smaller similarity score.

The model evaluates the effectiveness of these splits by calculating the gain metric:

$$\text{Gain} = \text{SimilarityScore}_{\text{LeftLeaf}} + \text{SimilarityScore}_{\text{RightLeaf}} - \text{SimilarityScore}_{\text{Root}}$$

After calculating the Gain for a threshold like LTV > 30, the model calculates the Gain for other thresholds, such as LTV > 40. Once the Gain metric is calculated for all possible thresholds (i.e., leaf combinations), the threshold with the highest Gain is selected for the next split. A higher Gain indicates that the threshold more effectively grouped instances with similar residuals.

This process is repeated for each newly created leaf, testing new thresholds, and calculating similarity scores. The Gains are compared, and the best split is chosen. This continues until the tree reaches the specified number of levels (default is typically 6) or when each leaf contains only a single residual, preventing further splitting.

3.2.5 Pruning via Cover

XGBoost also enforces a minimum number of residuals per leaf, referred to as Cover. Cover ensures that splits are meaningful and avoids overfitting. For classification, Cover is calculated as the denominator of the similarity score formula (minus λ):

$$\text{Cover} = \sum[\text{previousProbability} \times (1 - \text{previousProbability})]$$

In XGBoost, the Cover typically defaults to 1. Cover is important because if its value for a leaf falls below 1, the model will not allow that leaf to be created.

For example, in Figure 1, if the splitting threshold were LTV < 65, and a single instance (in red) that defaulted ended up alone in the leaf, its Cover would be calculated as:

$$\text{Cover} = \text{previousProbability} \times (1 - \text{previousProbability})$$

$$\text{Cover} = 0.5 \times 0.5 = 0.25$$

Since 0.25 is less than 1, that leaf would not be generated. This is a form of pruning, which prevents overfitting by avoiding overly small or uninformative splits.

3.2.6 Pruning via Gamma

When running the model, you need to specify a parameter called Gamma (γ). The model compares this Gamma value to the Gain of the lowest-level splits in the tree. If Gamma is larger than the Gain for a split, that branch is pruned from the tree. However, splits at higher levels will not be removed if their lower-level splits remain intact.

As mentioned earlier, the regularization parameter (λ) reduces the Gain metric, so a higher λ increases the likelihood of pruning. It's also worth noting that splits can result in a negative Gain (e.g., splitting similar values unnecessarily). Therefore, even if Gamma is set to zero, pruning may still occur.

3.2.7 Output for Regression

Although our project focuses on classification, we believe it is important to briefly discuss how the model handles regression. When the tree is complete, the machine learning model evaluates each leaf and the residuals within it. The output value for each leaf is calculated using the formula:

$$\text{Output Value} = \frac{\sum \text{residuals}}{\text{number of residuals} + \lambda}$$

This formula is almost identical to the Similarity Score formula, except that the sum of the residuals is not squared. If λ is not zero, it reduces the model's sensitivity to residuals, helping to prevent overfitting. The smaller the number of residuals in a leaf, the greater the effect of regularization. Without regularization ($\lambda = 0$), the output value of each leaf is simply the average residual.

After calculating the output value for each leaf, the model updates its predictions as follows:

$$\text{New prediction} = \text{Prior prediction} + \eta \times \text{CorrespondingLeafOutput}$$

In XGBoost, η (eta) represents the learning rate, which scales the new prediction. The default learning rate is typically 0.3, but it can be adjusted. Choosing a smaller value, such as 0.1, may yield more accurate results; however, this comes at the cost of requiring more iterations to converge.

3.2.8 Output for Classification

In XGBoost for classification, the output at each leaf is calculated using the formula:

$$\text{Output} = \frac{\sum \text{residuals}}{\sum[\text{previousProbability} \times (1 - \text{previousProbability})] + \lambda}$$

As in regression, this formula is nearly identical to the one used for the similarity score in classification, with the only difference being that the numerator ($\sum$ residuals) is not squared. Interestingly, this formula is also consistent with the one used in regular gradient boosting.

Probabilities in XGBoost must be converted into a log(odds) format. This involves applying the Logit transformation:

$$\text{logit}(P) = \log\left(\frac{P}{1 - P}\right)$$

This transformation scales extreme probabilities into more manageable values for computation. When XGBoost begins, the initial prediction is typically 0.5. Calculating the log(odds) for 0.5 gives 0, since log(0.5/0.5) = 0. From there, like regression, the new prediction is updated using the formula:

$$\text{New prediction} = \text{Prior prediction} + \eta \times \text{CorrespondingLeafOutput}$$

The resulting value from the formula will be in log(odds) format and must be converted back into a probability:

$$\text{Probability} = \frac{e^{\text{new prediction}}}{1 + e^{\text{new prediction}}}$$

At this point, each instance will have an updated probability prediction, closer to 0 or 1, depending on its true binary label. This process continues until the maximum number of trees is reached or the residuals become too small to provide meaningful improvement.

4 Data

For this project, we needed a large dataset of loan information, including labels indicating whether each loan remained current or entered a state of default. Following our professor's recommendation to explore Kaggle for potential datasets, we selected the "Loan Default Dataset – Loan Default Classification Problem" [8].

This dataset is extensive and includes various deterministic factors, such as borrower income, gender, loan purpose, and more. The poster of the dataset also noted the presence of strong multicollinearity, meaning many features are highly correlated and often point to the same outcome. This makes sense intuitively, as multiple aspects of a loan's characteristics typically align with its performance. For instance, it would be unusual for a borrower to simultaneously have an LTV > 100% and a good credit score.

Another notable aspect of the dataset is the presence of a significant number of empty cells. While this reduces the amount of usable information for the model, we also discovered during preprocessing that missing values could inadvertently provide the model with an unfair advantage when classifying loans.

We'd like to take this opportunity to clarify some key data points (or features) of the loans to ensure a common understanding. In the dataset, each row represents a loan, and each column corresponds to a specific feature of that loan. Below, we explain some of the less intuitive features:

4.1 Loan Limit

Represented as either `cf` or `ncf`, which stand for "conforming" or "non-conforming." This feature indicates whether the loan meets the size guidelines set by government-sponsored enterprises (GSEs) like Fannie Mae and Freddie Mac. Conforming loans (`cf`) adhere to these limits, while non-conforming loans (`ncf`) generally exceed the size restrictions, often referred to as jumbo loans. This classification is important because it affects the risk profile and liquidity of the loan.

4.2 Loan Type

Represented by `type1`, `type2`, or `type3`. This feature refers to the way the mortgage's interest rate changes over time, which influences the structure of the loan. The three common types are:

- **Fixed-Rate Mortgage (`type1`):** The interest rate remains constant throughout the life of the loan, providing stability for borrowers.
- **Adjustable-Rate Mortgage (ARM, `type2`):** The interest rate adjusts periodically based on an underlying index, such as LIBOR, plus a margin set by the bank. As a side note, ARMs were a significant contributing factor to the 2008 financial crisis. Borrowers were enticed by low "teaser" rates that would later adjust significantly higher after a honeymoon period. When housing prices began to fall, borrowers could no longer refinance into lower rates, leaving them with unaffordable payments, which triggered widespread defaults.
- **STEP Loan (`type3`):** The interest rate "steps" up or down (typically up) by a predetermined amount on scheduled dates, usually every few years. This structure offers predictability but may lead to increased payments over time.

This feature is critical for understanding loan performance, as the type of interest rate impacts a borrower's ability to keep up with payments.

4.3 Negative Amortization (`Neg_Amortization`)

Represented as either `neg_amm` or `not_neg`, this feature indicates whether the loan is subject to negative amortization. When a loan is originated, the terms specify the monthly payment required to fully pay off the loan by its maturity date (typically 30 years). However, some loans are structured as "interest-only," meaning the borrower's payments cover only the interest due, leaving the loan balance unchanged. In more extreme cases, some loans allow payments that don't even cover the full interest amount. The unpaid interest is added to the loan's principal, causing the balance to grow over time—this process is called negative amortization.

For example, consider a $200,000 loan with a 5% interest rate. If the borrower pays only half the monthly interest for 10 years, the loan balance would grow to $256,000. This type of loan structure increases the borrower's debt and can lead to financial instability. As far as we know, negative amortization is no longer legal for residential loans in most jurisdictions.

4.4 Construction Type

Represented by `sb` (Site-Built) or `mh` (Manufactured House). A Site-Built home is fully constructed on the plot of land where it stands. In contrast, a Manufactured House is built in a factory and later placed on the land. This distinction is important because manufactured houses are generally worth less than site-built homes. They are typically smaller, made with less durable materials, and more prone to damage.

4.5 Occupancy Type

Represented by `pr` (Primary Residence), `sr` (Secondary Residence), or `ir` (Investment Property). This indicates how the property is used—whether the borrower lives there full-time, uses it as a vacation home, or holds it as an investment.

4.6 Secured By

In the case of secured loans, the debt is backed by collateral that the bank can seize if the borrower defaults. This field specifies the type of collateral involved. Since this field is always listed as `home`, it confirms that the loans in this dataset are mortgages secured by the borrower's property.

4.7 Lump Sum Payment

Represented by `lpsm` (Lump Sum Payment) or `not_lpsm`. This feature indicates whether the borrower is required to make a balloon payment at the loan's maturity date. A balloon payment is a large, one-time payment of the remaining balance due at the end of the loan term. This often occurs due to specific loan terms or if the borrower has fallen behind on payments. The presence of a lump sum payment is usually a warning sign of potential financial distress.

4.8 Loan-to-Value (LTV)

This ratio compares the remaining loan balance to the value of the house. For example, if a borrower takes out a mortgage and puts 30% down, their LTV is 70%. However, if no payments are made the following month and the home's value drops by half, the LTV would increase to 140%.

From a bank's perspective, lower LTV ratios are better, while anything above 80% is considered risky. LTV is one of the strongest predictors of loan default. If a borrower owes significantly more than their home is worth (e.g., owing \$100k on a home valued at \$50k), they may be much less motivated to continue making payments.

4.9 Debt-to-Income Ratio (DTI, `dtir1`)

This metric compares the borrower's monthly income to their monthly debt obligations. It provides insight into how much of the borrower's income is already committed to debt payments. DTI can complement the LTV ratio as a predictor of loan performance. For example, a borrower with a low LTV (indicating significant equity in their home) might still struggle to stay current on their mortgage. A high DTI could explain this, as it suggests the borrower is burdened by other non-mortgage debts, leaving less income available for mortgage payments.

4.10 Status

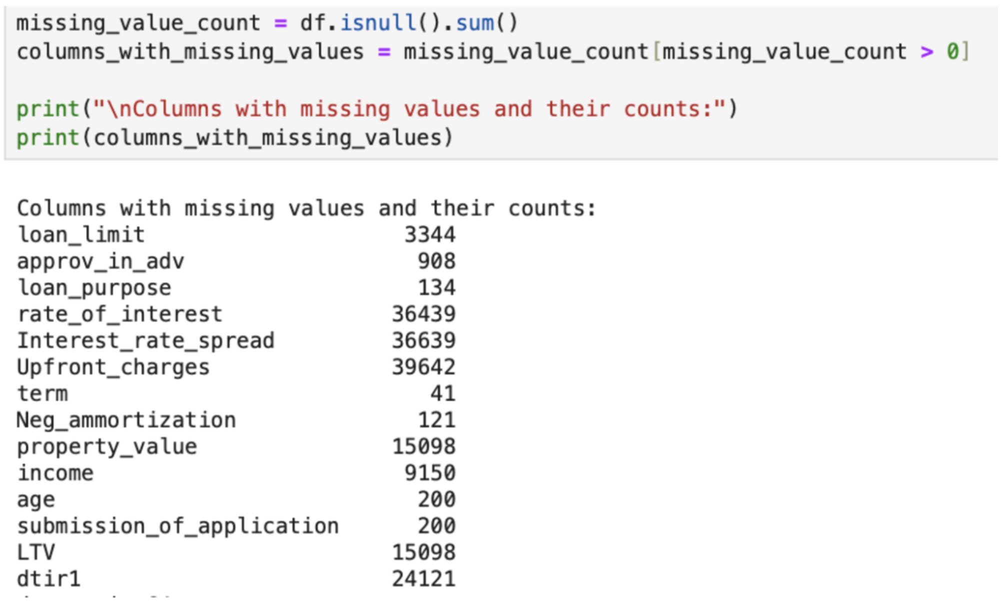
Represented as `1` or `0`, this feature indicates the loan's performance. A value of `1` means the loan has defaulted, while a value of `0` means the loan is current and not in default.

5 Data Preprocessing

As outlined in our draft proposal, we aim to enhance the models' performance through thorough and carefully planned preprocessing. Financial datasets, such as the one sourced from Kaggle, often contain errors or inconsistencies, and this dataset is no exception. Preprocessing the data will not only improve model accuracy but also enhance the interpretability of the results. Given that we are working with Decision Trees and XGBoost, interpretability is a key focus of this project. Decision Trees provide visual representations of the factors driving predictions, making it easier to interpret and present results in a meaningful way. We have identified several key preprocessing steps, which will be discussed in the following section.

5.1 Blank values

Our first major concern was the number of blank cells, as shown in Figure 2



```
missing_value_count = df.isnull().sum()
columns_with_missing_values = missing_value_count[missing_value_count > 0]

print("\nColumns with missing values and their counts:")
print(columns_with_missing_values)
```

```
Columns with missing values and their counts:
loan_limit                    3344
approv_in_adv                  908
loan_purpose                   134
rate_of_interest             36439
Interest_rate_spread         36639
Upfront_charges              39642
term                            41
Neg_ammortization              121
property_value               15098
income                        9150
age                            200
submission_of_application      200
LTV                          15098
dtir1                        24121
```

Figure 2: Python code for imputing missing values in `loan_limit`.

5.1.1 Filling Missing Data

Being reluctant to erase data for fear of reducing the model's predictive power and introducing biases, the first option we considered was to fill in the empty cells with data. The first problematic feature in our dataset was the loan limit feature. It contains categorical data indicating whether a loan is classified as conforming (cf) or non-conforming (ncf). We decided to explore the option of artificially filling in the missing data.

How this step was performed

To impute the missing values in *loan limit*, we first calculated the observed distribution of the values in the feature. This distribution provides the likelihood of each value (*cf* and *ncf*) based on the non-missing data.

Using this distribution, we assigned missing values proportionally by randomly sampling values according to their observed probabilities. This approach ensures that the imputed values reflect the dataset's original structure without introducing additional biases.

Recap of the Process

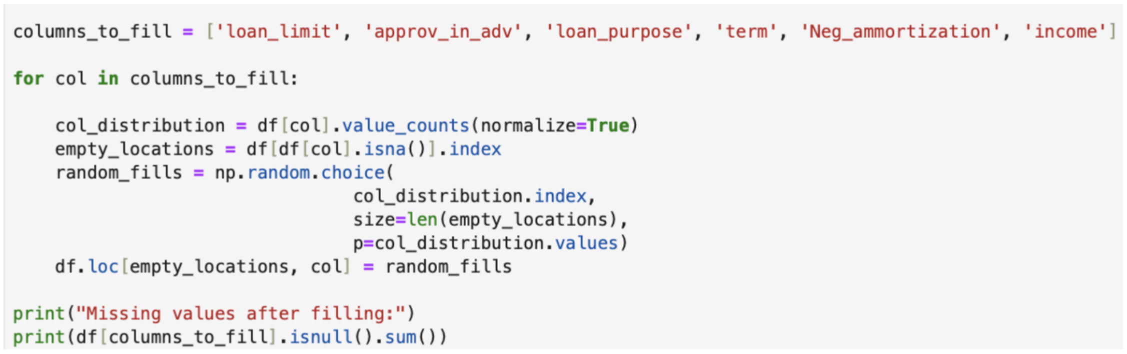
Calculating category proportions: The proportion of *cf* and *ncf* in the non-missing loan limit data was computed.

Identifying missing values: Rows with missing loan limit values were located.

Random sampling: Missing values were replaced by *cf* or *ncf* according to the calculated proportions.

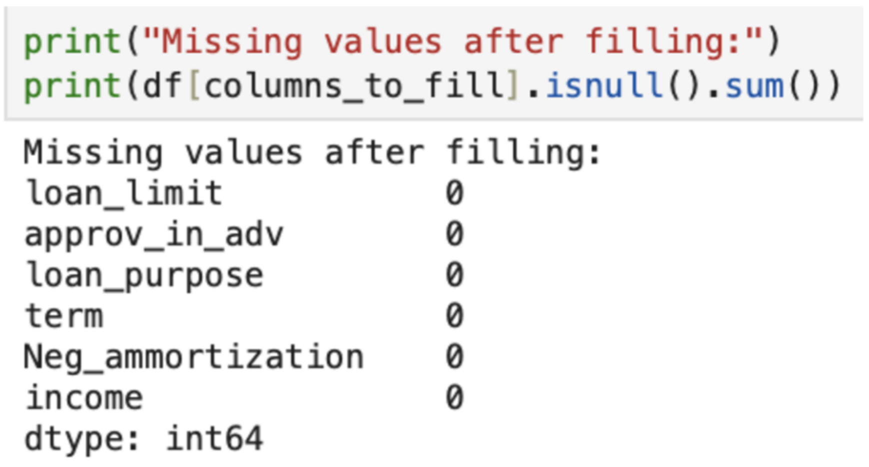
Verification: After imputation, the dataset was checked to confirm that no missing values remained in the loan limit column.

The Python code used for this process is shown in Figure 3 . The output confirming that no missing values remained is displayed in Figure 4



```python
columns_to_fill = ['loan_limit', 'approv_in_adv', 'loan_purpose', 'term', 'Neg_ammortization', 'income']

for col in columns_to_fill:

    col_distribution = df[col].value_counts(normalize=True)
    empty_locations = df[df[col].isna()].index
    random_fills = np.random.choice(
                            col_distribution.index,
                            size=len(empty_locations),
                            p=col_distribution.values)
    df.loc[empty_locations, col] = random_fills

print("Missing values after filling:")
print(df[columns_to_fill].isnull().sum())
```

Figure 3: Python code for imputing missing values



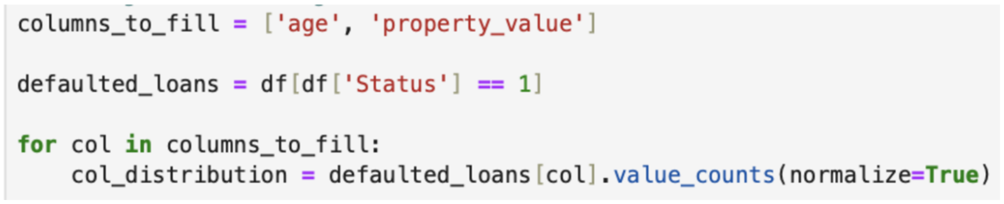
```
print("Missing values after filling:")
print(df[columns_to_fill].isnull().sum())
```

```
Missing values after filling:
loan_limit           0
approv_in_adv        0
loan_purpose         0
term                 0
Neg_ammortization    0
income               0
dtype: int64
```

Figure 4: Output confirming no missing values in loan limit.

Outcome: As shown in the code above, this method was applied not only to the loan_limit column but also to approv_in_adv, loan_purpose, term, neg_amortization, and income. This approach preserved the original distribution of categories in the loan_limit column, ensuring consistency and preventing data loss. After completing this step, all missing values in the loan_limit column were successfully filled, as verified by the absence of null values (see Figure 4).

For fields such as age and property_value, we observed a one-to-one correlation between missing values and defaulted loans. In other words, whenever these features had missing information, they were always associated with loans marked as defaulted. Consequently, when analyzing the distribution of age values, we focused solely on defaulted loans [refer 5], as we are imputing missing data only for defaulted loans. As we did earlier, we used the resulting distribution to randomly fill the missing data.



```
columns_to_fill = ['age', 'property_value']

defaulted_loans = df[df['Status'] == 1]

for col in columns_to_fill:
    col_distribution = defaulted_loans[col].value_counts(normalize=True)
```

Figure 5: Code sample to show that we are not sampling from just defaulted loans

5.1.2 Removing Invalid LTV Values

Why this step was necessary

During the initial runs of the Decision Tree model, we observed nonsensical results in the splits generated by the algorithm. For example, as shown in Figure 6 , one of the nodes split loans based on an LTV value of 3831 %—a completely unrealistic figure. This prompted an investigation into the source of the issue.

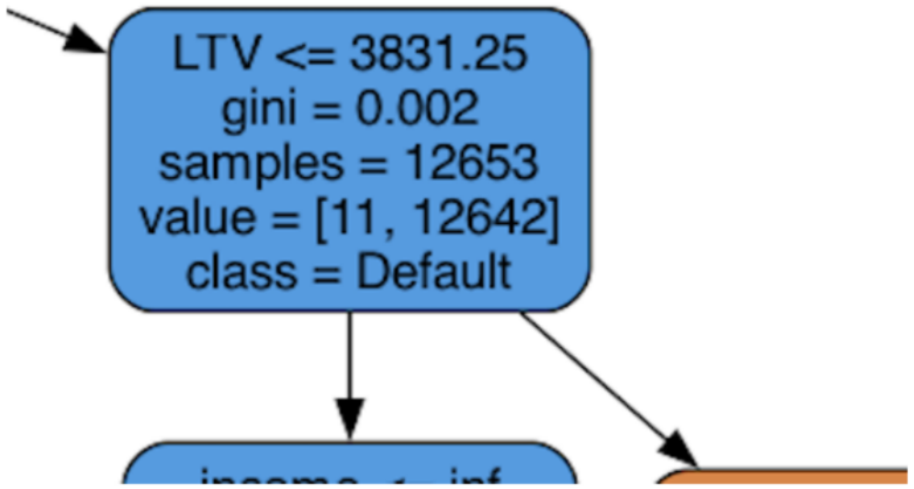


Figure 6: Example of a Decision Tree node splitting on an erroneous LTV value.

To identify the root cause, we recalculated the LTV values from the dataset by dividing the loan amount column by the property value column. This revealed several problems:

- A handful of loans had LTV values in the thousands, while the next highest LTV was 263%—already an unusually high value.

- For the loans with LTV values in the thousands, the property value column was always exactly \$8,000—a highly suspicious and likely erroneous figure. This value was both too small and too coincidental to be realistic.
- Adjusting these values (e.g., dividing by 10) would still produce unrealistic results, suggesting deeper issues with the underlying data. After analyzing the dataset, we concluded that these erroneous loans contributed no value to the model and negatively impacted the quality of results. Therefore, we decided to remove these loans entirely

What this step involved This preprocessing step involved the following actions:

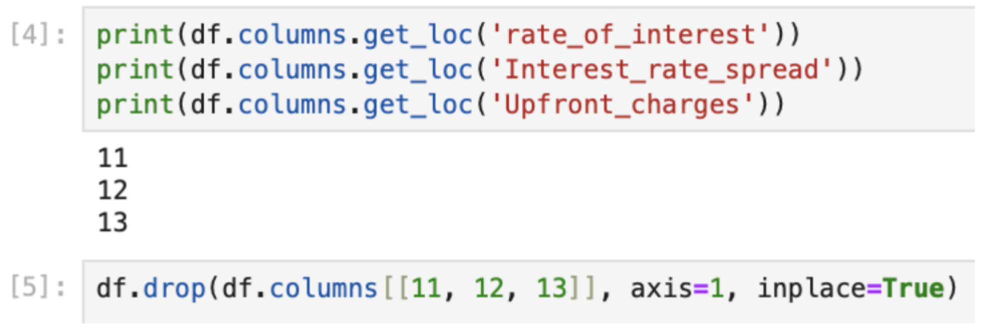
- Loans with LTV values greater than 264 were dropped, as such values are unrealistic in practice.
- Loans with missing LTV values (often due to missing property value data) were also removed, as these missing values could introduce bias.

This highlights poor record-keeping practices in the banking sector, an issue that is widespread and presents a significant challenge to achieving meaningful progress in machine learning (ML) modeling within the industry. Accurate and robust results are unattainable without reliable data.

5.1.3 Dropping Certain features

During the initial iterations of our models, we noticed that they were returning 100% accurate predictions on the uncleaned training data. Initially, we suspected that the classification labels might have inadvertently been included as features in the dataset. However, closer inspection confirmed this was not the case. Our next hypothesis was that a feature in the dataset might be perfectly mimicking the loan's status labels. This proved correct: we observed that defaulted loans consistently lacked interest rate data, while current loans included it. As a result, the model relied on this feature to make overly simplistic classifications. Essentially, the model's logic boiled down to: "if the interest rate is missing, the loan is in default." The same flaw, extend to 3 columns in total:

To address this , we chose to drop them: The code for doing so is shown in Figure 7



```
[4]: print(df.columns.get_loc('rate_of_interest'))
     print(df.columns.get_loc('Interest_rate_spread'))
     print(df.columns.get_loc('Upfront_charges'))

     11
     12
     13

[5]: df.drop(df.columns[[11, 12, 13]], axis=1, inplace=True)
```

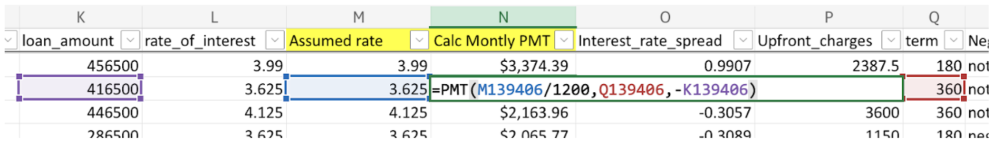
Figure 7: Dropping columns

An alternative approach could have been to input the missing interest rate data using the mean or median of the available data. However, this method would introduce an artificial concentration of values around the imputed number, creating a pattern inconsistent with real-world data. The model could then exploit this imputation for similarly easy, unrealistic classifications, defeating the purpose of building a robust predictive model.

Another issue we identified was that many loans were missing income data, and this correlated strongly with a lack of DTI (Debt-to-Income) ratio for those same loans. This relationship is logical, as the DTI ratio depends partly on income. Upon closer inspection, we also found that every loan missing a DTI value but with available income data was labeled as defaulted. This pattern is problematic for the same reasons previously discussed—it risks introducing bias into the model.

The situation became even more delicate when we realized that loans missing DTI but with income data accounted for nearly 70% of the remaining defaulted loans. Removing these loans was not an option, as it would eliminate a significant portion of critical data and further imbalance the dataset. Instead, we considered removing the DTI feature altogether.

To evaluate this approach, we performed a sanity check on the existing DTI values. We exported the dataset to Excel and used the PMT formula to estimate the monthly mortgage payment, using available data such as loan duration, loan amount, and interest rate. This allowed us to verify whether the DTI numbers were consistent and reliable for inclusion in the model.



K	L	M	N	O	P	Q	
loan_amount	rate_of_interest	Assumed rate	Calc Montly PMT	Interest_rate_spread	Upfront_charges	term	Ne
456500	3.99	3.99	$3,374.39	0.9907	2387.5	180	not
416500	3.625	3.625	=PMT(M139406/1200,Q139406,-K139406)			360	not
446500	4.125	4.125	$2,163.96	-0.3057	3600	360	not
286500	3.625	3.625	$2,065.77	-0.3089	1150	180	ne

Figure 8:

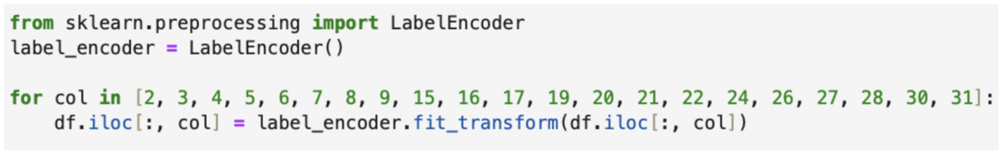
Using the monthly mortgage payment we calculated, we divided it by the income data to establish a baseline DTI. This baseline DTI should logically be lower than or equal to the DTI provided in the dataset, as our calculation only accounts for mortgage payments, while the reported DTI would typically include additional debts such as car loans or credit card payments.

However, upon comparison, we frequently found that our baseline DTI was higher than the provided values, which is nonsensical and a clear indication of data inaccuracies. This realization led us to conclude that the DTI feature was unreliable and likely to introduce noise into the model. As a result, we decided to drop this feature entirely.

5.2 Encoding Certain Features

Since we are using XGBoost and Decision Tree models, the features must be in numeric format for the models to function properly. Many features in the dataset were in text format and required encoding into numerical values. For example, the ”Occupancy Type” column contains entries such as “pr,” “sr,” and “ir” (representing primary residence, secondary residence, and investment residence).

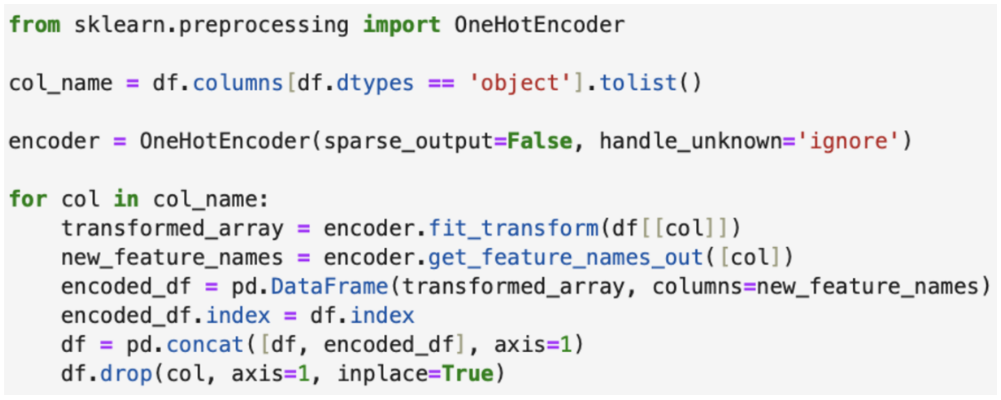
Our original attempt at doing this was incorrect:



```
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()

for col in [2, 3, 4, 5, 6, 7, 8, 9, 15, 16, 17, 19, 20, 21, 22, 24, 26, 27, 28, 30, 31]:
    df.iloc[:, col] = label_encoder.fit_transform(df.iloc[:, col])
```

Figure 9: Label Encoding

As shown in Figure 9, we identified every column in the dataset containing text-based values and transformed them into numeric data using LabelEncoder. However, it wasn't until our lecture on One-Hot Encoding that we realized this approach was flawed. By applying LabelEncoder to nominal data, we inadvertently introduced a sense of ordinality where none existed. For example, if "pr" (primary residence) was encoded as 1 and "sr" (secondary residence) as 2, the model might interpret this as implying that secondary residences are more important than primary residences. This introduced bias and misrepresented the data.



```
from sklearn.preprocessing import OneHotEncoder

col_name = df.columns[df.dtypes == 'object'].tolist()

encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')

for col in col_name:
    transformed_array = encoder.fit_transform(df[[col]])
    new_feature_names = encoder.get_feature_names_out([col])
    encoded_df = pd.DataFrame(transformed_array, columns=new_feature_names)
    encoded_df.index = df.index
    df = pd.concat([df, encoded_df], axis=1)
    df.drop(col, axis=1, inplace=True)
```

Figure 10: One-Hot Encoding

As mentioned earlier, we switched to One-Hot Encoding,(show in 10) which transformed each categorical column into multiple new columns—one for each unique category in the original column. The resulting dataset became binary, with each new column indicating whether a specific condition was met. For example, instead of a single column for "Occupancy Type," we now had separate columns showing whether a loan was for a primary residence, secondary residence, or investment property. This approach eliminated any unintended ordinality and better represented the data.

6 Oversampling and the Role of SMOTE

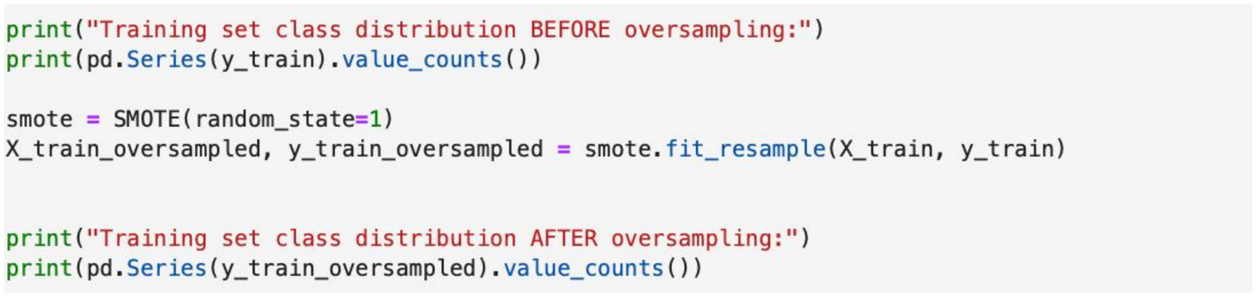
Oversampling is a data preprocessing technique used to address class imbalance in datasets, where one class (usually the minority class) has significantly fewer instances compared to the other. Class imbalance can adversely affect the performance of machine learning models, as they tend to be biased toward the majority class, leading to poor predictive performance for the minority class. Oversampling mitigates this issue by artificially increasing the number of instances in the minority class, ensuring a more balanced distribution.

In this study, the data set exhibited a substantial imbalance between the ‘Default‘ (minority) and ‘Non-Default ‘ (majority) classes. Without oversampling, the model would likely fail to accurately predict ‘Default‘ instances, reducing its effectiveness in detecting loan defaults.

> *SMOTE first selects a minority class instance* ***a*** *at random and finds its k nearest minority class neighbors. The synthetic instance is then created by choosing one of the k nearest neighbors* ***b*** *at random and connecting* ***a*** *and* ***b*** *to form a line segment in the feature space. The synthetic instances are generated as a convex combination of the two chosen instances* ***a*** *and* ***b****.*
>
> — Page 47, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

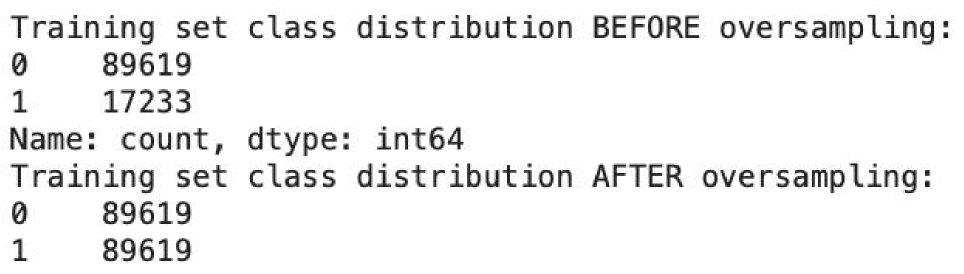
To address this, the **Synthetic Minority Oversampling Technique (SMOTE)** was applied. SMOTE generates synthetic examples for the minority class by interpolating between existing minority class instances. It selects two or more nearest neighbors of a minority instance in feature space and creates new instances along the line segments joining the original instance and its neighbors. This process maintains the feature distribution of the minority class while increasing its representation in the dataset. By using SMOTE, the class distribution was balanced, which enhanced the model’s ability to identify defaults and improved overall predictive performance.

The Python code snippet implementing SMOTE is shown in Figure 11, and the output confirming the balanced class distribution is displayed in Figure 12.



```python
print("Training set class distribution BEFORE oversampling:")
print(pd.Series(y_train).value_counts())

smote = SMOTE(random_state=1)
X_train_oversampled, y_train_oversampled = smote.fit_resample(X_train, y_train)


print("Training set class distribution AFTER oversampling:")
print(pd.Series(y_train_oversampled).value_counts())
```

Figure 11: Code Snippet for Implementing SMOTE to Balance the Dataset



```
Training set class distribution BEFORE oversampling:
0    89619
1    17233
Name: count, dtype: int64
Training set class distribution AFTER oversampling:
0    89619
1    89619
```

Figure 12: Output Verifying the Balanced Class Distribution After SMOTE

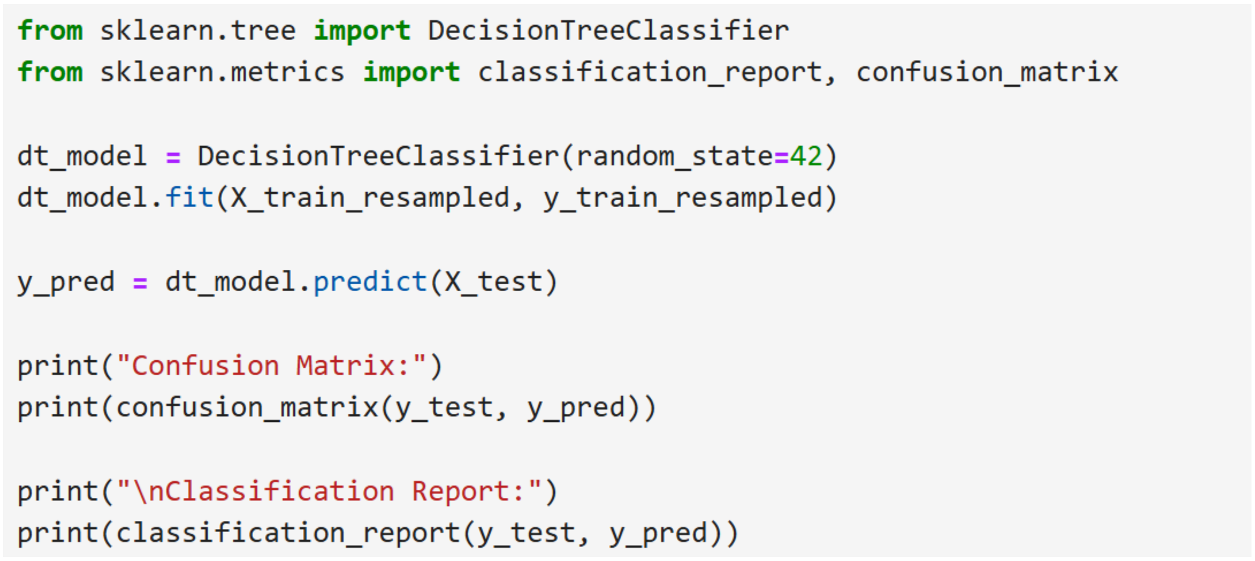
The output in Figure 12 demonstrates that the training set has been successfully balanced, with both non-default (class 0) and default (class 1) now containing 89,619 samples. This ensures the model receives equal representation of both classes, reducing the likelihood of biased predictions.

7 Model Development

This section outlines the methodology adopted for developing two classification models: the Decision Tree Classifier and the XGBoost Classifier. These models were implemented to predict loan default using the resampled training dataset created via SMOTE.

7.1 Decision Tree

The first model developed was the **Decision Tree Classifier**, a simple yet interpretable tree-based algorithm. The model was initialized with a random state to ensure reproducibility and was trained on the resampled data. The Python code snippet for implementing the Decision Tree Classifier is presented in Figure 13.



```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, confusion_matrix

dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train_resampled, y_train_resampled)

y_pred = dt_model.predict(X_test)

print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

Figure 13: Code Snippet for Training and Evaluating the Decision Tree Classifier

When choosing the depth of the tree for our baseline model, we ran decision trees with depths from 1 to 10 and plotted their accuracy and precision. The Precision metric was for the default class specifically, as it's the smaller, more challenging, and critical class to classify.

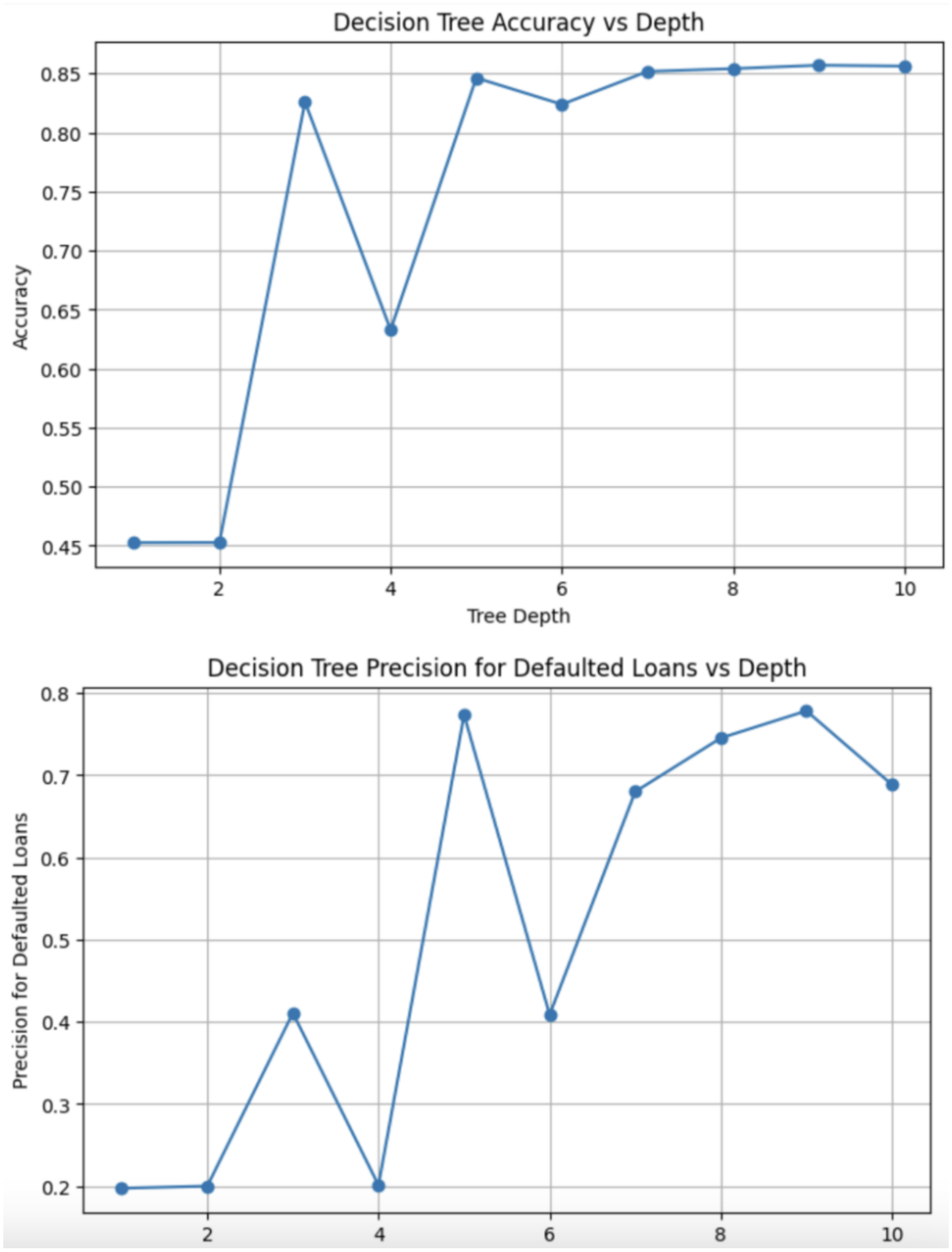


Figure 14: Decision Trees plotted for depths 1 to 10

Inferences: The graph showed that depth 3 had good accuracy (83%) but poor precision for defaulted loans (41%). Therefore, we chose depth 5, which had slightly higher accuracy (85%) and significantly better precision (77%), making it a balanced choice.

Visualizing the best tree : Gini Depth 5

The primary area needing improvement is the recall metric for defaulted loans. The baseline model struggles significantly to identify loans that have defaulted. For instance, with a depth of 5, the recall rate is a poor 7%, meaning the model correctly identifies only 7 out of every 100 defaulted loans.

Table 1: **Gini - Depth 3 Results**

Confusion Matrix		
	Predicted 0	Predicted 1
Actual 0	21283	1122
Actual 1	3527	782

	Precision	**Recall**	**F1-score**	**Support**
Class 0	0.86	0.95	0.90	22405
Class 1	0.41	0.18	0.25	4309
Accuracy			0.83	
Macro Avg	0.63	0.57	0.58	26714
Weighted Avg	0.79	0.83	0.80	26714

Table 2: **Gini - Depth 5 Results**

Confusion Matrix		
	Predicted 0	Predicted 1
Actual 0	22320	85
Actual 1	4019	290

	Precision	**Recall**	**F1-score**	**Support**
Class 0	0.85	1.00	0.92	22405
Class 1	0.77	0.07	0.12	4309
Accuracy			0.85	
Macro Avg	0.81	0.53	0.52	26714
Weighted Avg	0.84	0.85	0.79	26714

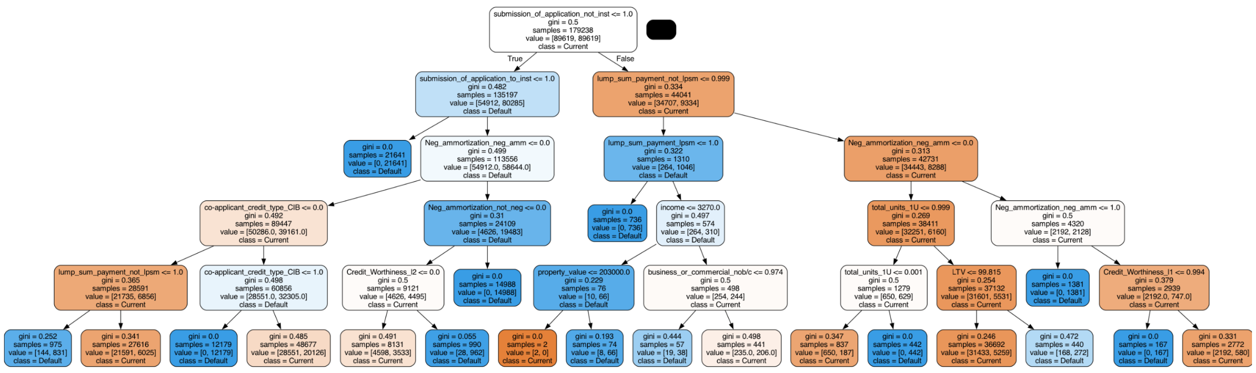

submission_of_application_not_inst <= 1.0 gini = 0.5 samples = 179238 value = [89619, 89619] class = Current
True
False
submission_of_application_to_inst <= 1.0 gini = 0.482 samples = 135197 value = [54912, 80285] class = Default
lump_sum_payment_not_lpsm <= 0.999 gini = 0.334 samples = 44041 value = [34707, 9334] class = Current
gini = 0.0 samples = 21641 value = [0, 21641] class = Default
Neg_ammortization_neg_amm <= 0.0 gini = 0.499 samples = 113556 value = [54912.0, 58644.0] class = Default
lump_sum_payment_lpsm <= 1.0 gini = 0.322 samples = 1310 value = [264, 1046] class = Default
Neg_ammortization_neg_amm <= 0.0 gini = 0.313 samples = 42731 value = [34443, 8288] class = Current
co-applicant_credit_type_CIB <= 0.0 gini = 0.492 samples = 89447 value = [50286.0, 39161.0] class = Current
Neg_ammortization_not_neg <= 0.0 gini = 0.31 samples = 24109 value = [4626, 19483] class = Default
gini = 0.0 samples = 736 value = [0, 736] class = Default
income <= 3270.0 gini = 0.497 samples = 574 value = [264, 310] class = Default
total_units_1U <= 0.999 gini = 0.269 samples = 38411 value = [32251, 6160] class = Current
Neg_ammortization_neg_amm <= 1.0 gini = 0.5 samples = 4320 value = [2192, 2128] class = Current
lump_sum_payment_not_lpsm <= 1.0 gini = 0.365 samples = 28591 value = [21735, 6856] class = Current
co-applicant_credit_type_CIB <= 1.0 gini = 0.498 samples = 60856 value = [28551.0, 32305.0] class = Default
Credit_Worthiness_l2 <= 0.0 gini = 0.5 samples = 9121 value = [4626, 4495] class = Current
gini = 0.0 samples = 14988 value = [0, 14988] class = Default
property_value <= 203000.0 gini = 0.229 samples = 76 value = [10, 66] class = Default
business_or_commercial_nob/c <= 0.974 gini = 0.5 samples = 498 value = [254, 244] class = Current
total_units_1U <= 0.001 gini = 0.5 samples = 1279 value = [650, 629] class = Current
LTV <= 99.815 gini = 0.254 samples = 37132 value = [31601, 5531] class = Current
gini = 0.0 samples = 1381 value = [0, 1381] class = Default
Credit_Worthiness_l1 <= 0.994 gini = 0.379 samples = 2939 value = [2192.0, 747.0] class = Current
gini = 0.252 samples = 975 value = [144, 831] class = Default
gini = 0.341 samples = 27616 value = [21591, 6025] class = Current
gini = 0.0 samples = 12179 value = [0, 12179] class = Default
gini = 0.485 samples = 48677 value = [28551, 20126] class = Current
gini = 0.491 samples = 8131 value = [4598, 3533] class = Current
gini = 0.055 samples = 990 value = [28, 962] class = Default
gini = 0.0 samples = 2 value = [2, 0] class = Current
gini = 0.193 samples = 74 value = [8, 66] class = Default
gini = 0.444 samples = 57 value = [19, 38] class = Default
gini = 0.498 samples = 441 value = [235.0, 206.0] class = Current
gini = 0.347 samples = 837 value = [650, 187] class = Current
gini = 0.0 samples = 442 value = [0, 442] class = Default
gini = 0.246 samples = 36692 value = [31433, 5259] class = Current
gini = 0.472 samples = 440 value = [168, 272] class = Default
gini = 0.0 samples = 167 value = [0, 167] class = Default
gini = 0.331 samples = 2772 value = [2192, 580] class = Current

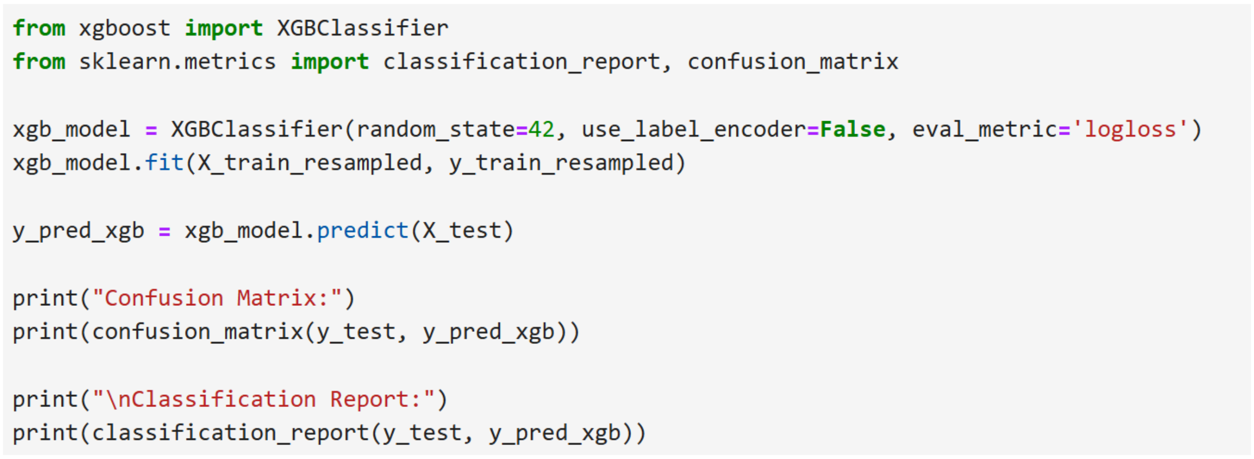
Figure 15: Code Snippet for Training and Evaluating the XGBoost Classifier

7.2 XGBoost Classifier

The second model developed was the **XGBoost Classifier**, a highly efficient gradient boosting framework specifically designed for performance on structured datasets. Our understanding was that log loss was the standard evaluation for XGBoost's performance as it does a good job predicting outcomes, especially for classification problems (like predicting "default" or "current"). It penalizes wrong predictions more heavily. Log loss during training allows the model to learn better probabilities for each class.

If the model predicts 0.9 when the label is 1, the log loss is small because the prediction is close. Vis – versa if the prediction is 0.1, the log loss is large, meaning that the model must make improvements.

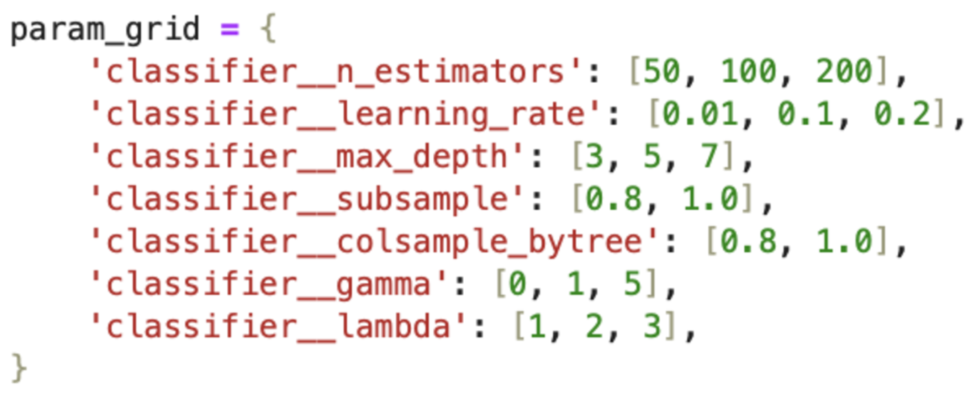
The model was configured with parameters such as `random_state` to maintain consistency across runs and `eval_metric` set to `logloss` for effective evaluation. A general way to implement the XGBoost is shown in Figure 16.



```python
from xgboost import XGBClassifier
from sklearn.metrics import classification_report, confusion_matrix

xgb_model = XGBClassifier(random_state=42, use_label_encoder=False, eval_metric='logloss')
xgb_model.fit(X_train_resampled, y_train_resampled)

y_pred_xgb = xgb_model.predict(X_test)

print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_xgb))

print("\nClassification Report:")
print(classification_report(y_test, y_pred_xgb))
```

Figure 16: Example code snippet for XGBoost Classifier(not included in the code)

XGBoost is more complex than a simple decision tree, and finding the best parameters is crucial. One tool for this is GridSearchCV — You provide a list of parameters to test, specifying the possible values for each. It creates a "grid" of all parameter combinations and tests each one to find the best combination based on a metric like accuracy or F1-score. For example, if you test learning rate [0.1, 0.2] and max depth [3, 4], GridSearchCV tests combinations like (0.1, 3), (0.1, 4), etc., and picks the optimal pair. See below for the params that were used:



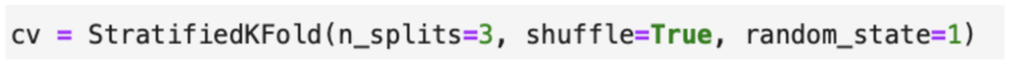
```
param_grid = {
    'classifier__n_estimators': [50, 100, 200],
    'classifier__learning_rate': [0.01, 0.1, 0.2],
    'classifier__max_depth': [3, 5, 7],
    'classifier__subsample': [0.8, 1.0],
    'classifier__colsample_bytree': [0.8, 1.0],
    'classifier__gamma': [0, 1, 5],
    'classifier__lambda': [1, 2, 3],
}
```

Figure 17: Grid Search Parameters

- **N_estimators**: This represents the number of small trees (or stumps) in the ensemble model. Too few can lead to underfitting, while too many can cause overfitting.
- **Learning rate**: This adjusts how much each tree contributes to the final model. Smaller values make learning slower but more precise; larger values can cause the model to miss the best solution.
- **Max depth**: This controls the maximum number of splits a tree can make.
- **Subsample**: This refers to the percentage of training data used to grow each tree. Lower values reduce overfitting but may underfit if too low.
- **Colsample by tree**: This specifies the number of features to consider when building each tree, helping to fight overfitting.
- **Gamma**: A regularization parameter dictating the amount of improvement required before splitting.
- **Lambda**: Helps with pruning by reducing large leaf weights, preventing any single leaf from dominating the predictions.

7.2.1 Cross-Validation and its issues

Alongside GridSearchCV, we use cross-validation. For each combination of parameters that the model tests, cross-validation splits the data into three “folds” or parts. It trains the XGBoost model on two folds and tests it on the third to evaluate performance. The data in these folds is shuffled during each iteration while preserving the distribution.



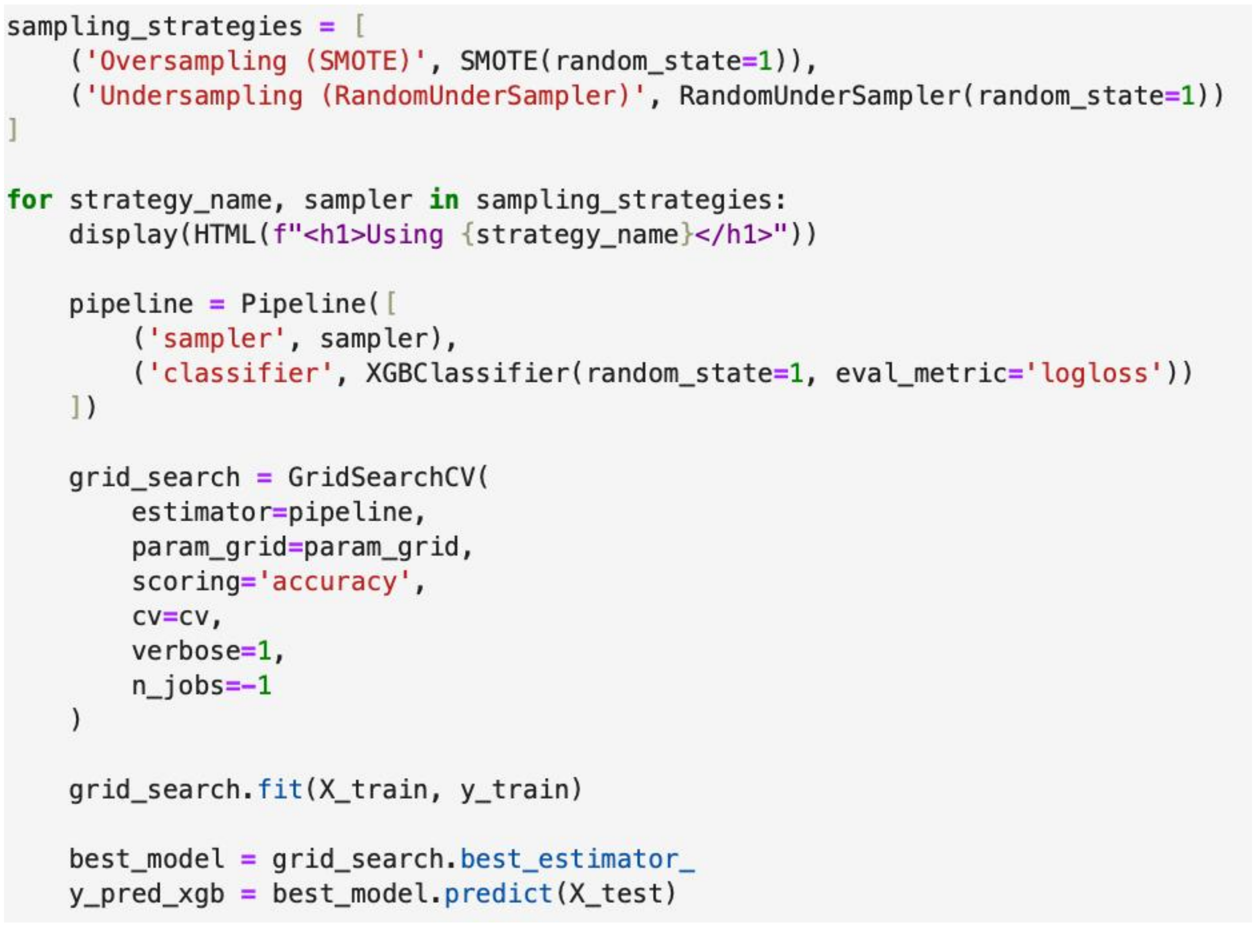
```
cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=1)
```

Figure 18:

When we first ran GridSearchCV, we mistakenly used data that was already oversampled. This caused inconsistencies between the predictions and accuracies reported by GridSearchCV and the results when we tried to replicate them. After investigating, we discovered that oversampling data before using GridSearchCV was the issue.

The problem arises because, during cross-validation, the model trains on two folds and tests on the third. For this process to work correctly, the folds must be independent. Oversampling beforehand creates synthetic data based on the original data, causing the testing fold to overlap with the training fold. This overlap leads to overly optimistic results since the model inadvertently trains on data similar to its test set.

The solution was to use a pipeline. The pipeline ensured that oversampling occurred within each fold during cross-validation. This means the data was partitioned into folds first, and then oversampling was applied only to the training data of each fold, preserving the independence of the test fold. This approach provided more accurate and reliable evaluation metrics.



```python
sampling_strategies = [
    ('Oversampling (SMOTE)', SMOTE(random_state=1)),
    ('Undersampling (RandomUnderSampler)', RandomUnderSampler(random_state=1))
]

for strategy_name, sampler in sampling_strategies:
    display(HTML(f"<h1>Using {strategy_name}</h1>"))

    pipeline = Pipeline([
        ('sampler', sampler),
        ('classifier', XGBClassifier(random_state=1, eval_metric='logloss'))
    ])

    grid_search = GridSearchCV(
        estimator=pipeline,
        param_grid=param_grid,
        scoring='accuracy',
        cv=cv,
        verbose=1,
        n_jobs=-1
    )

    grid_search.fit(X_train, y_train)

    best_model = grid_search.best_estimator_
    y_pred_xgb = best_model.predict(X_test)
```

Figure 19: Grid search with pipeline for running XGBoost

7.2.2 XGBoost Results

As mentioned earlier, one of the parameters that GridSearchCV takes is the evaluation metric it should focus on improving. Our group agreed that "accuracy" is the most appropriate metric, as it provides a clear and general indication of the model's performance. When we ran GridSearchCV with the objective of maximizing accuracy, the following parameters yielded the best results:

- 'classifier colsample_bytree': 0.8
- 'classifier gamma': 0
- 'classifier lambda': 3
- 'classifier learning_rate': 0.1
- 'classifier max_depth': 7
- 'classifier n_estimators': 200
- 'classifier subsample': 1.0

And when XGBoost is ran with these optimal parameters the accuracy score returned is 87.3%. The final confusion matrices and classification report for both the baseline and final model are presented below:

Table 3: **Baseline Model - Decision Tree (Depth 5)**

Confusion Matrix

	Predicted 0	Predicted 1
Actual 0	22320	85
Actual 1	4019	290

	Precision	**Recall**	**F1-score**	**Support**
Class 0	0.85	1.00	0.92	22405
Class 1	0.77	0.07	0.12	4309
Accuracy		**0.85**		
Macro Avg	0.81	0.53	0.52	26714
Weighted Avg	0.84	0.85	0.79	26714

Table 4: **Final Model - XGBoost (Depth 7)**

Confusion Matrix

	Predicted 0	Predicted 1
Actual 0	22117	288
Actual 1	3139	1170

	Precision	**Recall**	**F1-score**	**Support**
Class 0	0.88	0.99	0.93	22405
Class 1	0.80	0.27	0.41	4309
Accuracy		**0.87**		
Macro Avg	0.84	0.63	0.67	26714
Weighted Avg	0.86	0.87	0.84	26714

As shown, the overall accuracy has only slightly improved, from 85% to 87%. However, the highlight is the significant improvement in recall for the minority class (defaulted loans), which increased from 0.07 to 0.27—a 20% absolute improvement. This means the model now identifies 1,170 defaulted loans compared to only 290 previously. While there's still room for improvement, this marks a meaningful step forward in capturing defaults more effectively.

For the sake of diligence and curiosity, we re-ran GridSearch and cross validation with Undersampling to see if this could increase our recall metric for defaulted loans. These were the results:

Table 5: **XGBoost Results with undersampling**

Confusion Matrix

	Predicted 0	Predicted 1
Actual 0	17145	5260
Actual 1	1484	2825

	Precision	**Recall**	**F1-score**	**Support**
Class 0	0.92	0.77	0.84	22405
Class 1	0.35	0.66	0.46	4309
Accuracy		**0.75**		
Macro Avg	0.63	0.71	0.65	26714
Weighted Avg	0.83	0.75	0.77	26714

As you can see, the accuracy decreased to 75%. However, the recall improved significantly to 66%, making this the first iteration to correctly identify the majority of defaulted loans. Out of 4,309 defaulted loans in the test set, the model successfully identified 2,825 of them. This represents a substantial improvement in addressing the challenge of detecting defaults.

8 Conclusion

Overall, our project may have been less flashy than others, but we approached it with a simple idea and goal. However, the complexity of the data quickly turned it into a challenging endeavor. A significant portion of our time went into preprocessing and applying various techniques to make the project viable. This included deleting features and instances, recalculating anomalies, cross-referencing missing data with loan statuses, and even generating synthetic data based on existing distributions. We could have taken an easier route by filling in the missing data with the median and plugging that in for all the blanks, but we were adamant about introducing the least amount of bias into our data.

We faced our share of challenges, such as issues with label encoding and oversampling before cross-validation, but we learned from our mistakes and took corrective actions. We evaluated a baseline decision tree model, rigorously testing it at various depths, and pitted it against XGBoost, using GridSearchCV to optimize its parameters. Even when results weren't stellar, we explored undersampling and successfully improved the recall metric for defaulted loans.

This process taught us the value of persistence, attention to detail, and iterative improvement. While the journey wasn't easy, it was rewarding to see measurable progress in our efforts.

As it should come as no surprise, XGBoost performed better than the baseline decision tree. This is normal since decision trees are simple models that look for the best threshold to split a leaf. They have few standard parameters to adjust, and we mainly focused on the depth. However, XGBoost has a lot more depth to it (pun intended). We experimented with seven parameters, including various levels of regularization, which decision trees cannot do. Since we discussed lambda earlier, we added it as a hyperparameter so GridSearch could test for different pruning levels. The list of parameters continues, illustrating the complexity of XGBoost and its better performance. The model continually builds new stumps and learns from each previous iteration.

I believe that for machine learning models to perform better, we need better data. Based on our experience in banking, it is common for banks to delete loan information from their SQL databases when a loan defaults. At the very least, they may move the data to a separate archived table. Given the significant gaps in data for defaulted loans in this dataset, I suspect this is exactly what happened here.

This industry practice is unfortunate, as it prevents banks from learning from their mistakes. The issue is not limited to banks but also extends to REITs and third-party data providers. Data in this industry is often far from clean, and if better data collection and storing policies were implemented, banks would have more information to work with—ultimately leading to better results.

Thank you for reading,
Morgan, Divya & Pranav

References

[1] UK Finance. The impact of ai in financial services. `https://www.ukfinance.org.uk/`, 2023. Accessed: [date you accessed].

[2] X. Ma, J. Sha, D. Wang, Y. Yu, Q. Yang, and X. Niu. Study on a prediction of p2p network loan default based on the machine learning lightgbm and xgboost algorithms according to different high dimensional data cleaning. *Electronic Commerce Research and Applications*, 31:24–39, 2018.

[3] E. Ravina. Beauty, personal characteristics, and trust: Evidence from online peer-to-peer lending. Working paper, Columbia Business School, 2007.

[4] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*, pages 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[5] Y. Ouyang. Loan default prediction based on logistic regression and xgboost modeling. In *2024 IEEE 2nd International Conference on Control, Electronics and Computer Technology (ICCECT)*, pages 1145–1149, Jilin, China, 2024.

[6] StatQuest with Josh Starmer. Decision and classification trees, clearly explained!!! [video].

[7] StatQuest with Josh Starmer. Xgboost part 2 (of 4): Classification [video].

[8] Y Hossin. Loan default dataset [data set]. kaggle.