Improving Bank Loan Decisions: Tailored Loan Offer Recommendations

Barbara Pacetta[1], Denise Tampieri[2]

[1]barbara.pacetta@studenti.unitn.it

[2]denise.tampieri@studenti.unitn.it

***Abstract*— This project presents a novel algorithmic approach to improve risk assessment in banks by leveraging client behavioral profiles and loan repayment history. The objective is to develop an algorithm that can advise banks on whether to accept or decline new loan requests, while also providing tailored offers in case of acceptance. By employing machine learning techniques such as clustering and decision trees, the proposed algorithm provides reliable loan request evaluations. The system architecture incorporates data creation, storage, loan processing, and ongoing loan management. The implementation utilizes SQL, Redis, and Apache Spark technologies. This approach offers banks an effective tool to enhance risk mitigation, streamline loan management, and deliver personalized loan offers.**

***Keywords*— Risk assessment, loan requests management, SQL, clustering**

I. Introduction

The aim of our project is to develop a robust solution that enables banks to recognize fraud and identify individuals likely to become bad payers, leading to a reduction in the number of charged off loans. To accomplish this, we faced notable challenges, particularly in the creation of synthetic data that faithfully represents real-world situations due to privacy concerns and the lack of available online data. In particular, the interest rate, which is often determined by banks through undisclosed criteria. Lastly, given our background, we invested considerable effort in understanding the task from an economic perspective, ensuring a comprehensive approach to our solution.

II. System Model

A. System architecture

The system architecture consists of a well-defined pipeline with multiple components working together to handle loan requests and ongoing loan repayments.

1. Data Creation: Initially, data was synthetically created to simulate past and real customer loan requests. However, in practice, the system would rely on bank-provided database

2. SQL Database: The data is stored in an SQL database. The database is structured with three tables: *loan_records, ongoing,* and *fully_paid*. Each table includes a unique loan ID (a 32-character alphanumeric code) column for identification purposes.

3. Loan Request Processing: When a new loan request is received, it undergoes analysis through three machine learning algorithms: k-NN, Hierarchical Clustering (HC), and Decision Tree (DT). These algorithms compare the input request with existing loans in the *loan_records* table, identifying similar loans and their corresponding statuses.

4. Loan Decision: Based on the analysis, the system determines whether to accept or reject the loan request. If the request is declined, the simulation ends. If it is accepted, the system proceeds to compute a tailored offer for the customer.

5. Ongoing Loans Pipeline: Concurrently, a pipeline is implemented to handle ongoing loans, and update the SQL tables.

6. Redis Ingestion: Dynamic information about ongoing loans is stored in a Redis hash, with loan IDs serving as the keys. The hash fields include the months left to the end of the repayment and the number of late payments made by the client.

7. Monthly Transactions: Using Apache Spark, the system simulates monthly transactions made by customers to repay their loans. Each month, it checks whether a transaction occurs. If a transaction is made, the months left are reduced by 1. If a transaction occurs after the 27th of each month, the field of late payments is incremented by 1.

8. Loan Repayment and Update: Once a loan is fully repaid, it is removed from the Redis hash and subsequently deleted from the *ongoing* table in the SQL database. The loan details are then stored in the *loan_records* and *fully_paid* tables for historical tracking.

By following this pipeline, the system effectively handles loan requests, performs real-time updates on ongoing loan repayments, and maintains comprehensive records in the SQL database for analysis and future requests.

B. Technologies

We have strategically chosen several technologies to support our system architecture, namely SQL, Redis,

and Apache Spark, each serving a specific purpose within our pipeline.

SQL played a critical role in the first pipeline, enabling us to create, modify, manage, and visualize the data stored in our database. The relational database support offered by SQL was particularly advantageous, as our tables are highly interconnected. With SQL, we could easily handle data manipulation tasks and leverage its ability to quickly retrieve data and execute complex queries.

In the ongoing loans management section, which can be considered as the second pipeline, our focus shifted to Redis and Apache Spark. For efficient data handling during transaction simulations, we opted to utilize Redis, specifically utilizing its hash data structure. By assigning the loan ID as the key and associating it with fields such as *months_left* and *delays*, we prioritized the need for rapid data reading and writing operations. The choice of a hash structure proved beneficial when dealing with structured data that required key-value associations.

Redis was also selected due to its capability to handle large volumes of data. While our simulation involved a small number of transactions, we anticipated that a real-world banking scenario would involve a much larger dataset. Redis' robust data handling capabilities ensured scalability and performance.

Apache Spark emerged as the ideal technology for transaction simulation due to its exceptional scalability. Spark seamlessly distributes workloads across multiple nodes, enabling efficient processing of large datasets. Moreover, Spark's ability to store data in memory significantly reduces disk I/O operations, further enhancing processing speed and performance.

In summary, the selection of SQL, Redis, and Apache Spark aligns with the building blocks of our system architecture. SQL empowers the pipeline with robust data management capabilities, while Redis and Apache Spark enhance the transactions pipeline's data handling and scalability, respectively. These technologies combine to provide efficient and effective support for our system's functionalities.

III. Implementation

GitHub: https://github.com/DeniseGH/BDT_project_RiskProfiling.git

The project consists of two parallel pipelines simulating different phases of loan management in a bank. The first pipeline focuses on the decision-making process, evaluating loan applications based on historical data. The second pipeline simulates loan repayment and maintains a repayment history. The code is structured with dedicated modules for each pipeline, and execution is performed through a main script. Configuration parameters were chosen to optimize performance and adapt to the requirements of Redis and Apache Spark, ensuring efficient processing and resource utilization.

1. Synthetic data creation

The *main.py* file serves as the entry point, where data creation and processing are initiated. Upon execution, the code first sets up the database (called *CREDIT_RECORDS.db*) and creates the necessary tables with predefined columns to store the loan records.

The data ingestion process begins with the *insert_loan_records(..)* function, which takes the desired number of loan records as input. These records are instances of the *Loan* class, which contains attributes related to the loan itself, such as loan ID, client information, loan amount, purpose, term, and status. The client information is stored in the *Client* class, which includes details like name, age[1], education level, net monthly income, reports from the Centrale dei Rischi (CR) of late other payments, reports from the Centrale dei Rischi Finanziari (CRIF) of bank non-performing loans (i.e. suffering), co-applicant information, dependents, savings, and co-applicant's net monthly salary. The creation of loan and client attributes is handled by the *create_random_loan(..)* and *create_random_client(..)* functions within their respective classes.

As mentioned earlier, the *Loan* class encompasses the "status" attribute, which can take on values such as *Fully_Paid, Declined, Charged_off, Ongoing,* and *Unknown*. A status of *Fully_Paid* indicates that the loan has been completely repaid. *Declined* signifies that the loan application has been rejected. *Charged_off* refers to loans that have not been repaid according to the agreed terms and have become problematic or risky for the lending institution. *Ongoing* denotes loans that are still in the process of repayment, and *Unknown* is the status assigned to new loan requests (and the one we aim to predict later).

After inserting the loan records into the primary table, the code proceeds to extract the fully paid loans and adds them to the *Fully_Paid* table. The *Fully_Paid* class includes additional details like monthly installment amount, interest rate, duration, late payments (reported to the Central Financial Risk Office), and the final installment for loan closure.

The creation of attributes for the various classes is facilitated by a set of functions (in the *functions.py* file), as explained in Table 1.

[1] We deliberately used age as a parameter instead of the date of birth, as our focus is to track the information at the time of loan acceptance

TABLE I
DATA CREATING FUNCTIONS

Function	Description	Range
IDs	Random alphanumeric 32-characters code	
Name Surname	Random Italian names and surnames	
Age (at the moment of the loan request)	Random integer (in years)	25 - 60
Education level [1]	Boolean value chosen following ISTAT report on education level	Graduated/ Not graduated
Applicant net monthly income [4]	Random net monthly income based on an approximate ISTAT division of the Italian population (€)	1000 - 6000
CR Delay	Random number of delays in other payments (provided by CR)	0 - 2
CRiF Suffering	Boolean variable indicating non-performing loan	Yes/No
Co-Applicant	Boolean variable specifying the presence of a co-applicant	Yes/No
Co-Applicant Income	Same calculation as Applicant net monthly income (€)	1000 - 6000
Personal Savings	Random integer (€)	200 - 10000
Number of Dependents	Random number of dependents based on the income of the applicant(s)	0 - 3
Total Income	Estimated net income by removing life expenses of dependents	
Monthly Repayment [2]	Monthly installment calculated on 30% of total income	
Loan Amount [2]	Amount granted (€)	0 - 80000
Purpose	Reason for the loan calculated according to the amount requested	5 different categories[2]
Interest Rate	Random float number for the interest rate[3]	7% - 8.8%
Duration [3]	Loan duration in month based on the amount, interest rate and monthly fee	
Term	Boolean value based on the duration	Short Term/ Long Term
Last Fee	Final loan installment	
Status	Final status of the Loan	declined charged off fully paid

[2]These are: debt consolidation, home improvement, vacation or travel, wedding expenses, education or tuition fees

[3]In the case of loans with a debt consolidation purpose, it is higher, exceeding 8%. For other types of loans, the interest rate falls within the range of 7% to 8%.

It is essential to emphasize that the assignment of loan status in the system is initially randomized, but is then influenced by some factors; If there are more than two CR delays in previous payments, if there is a record of previous non-performing loans, or if the savings are insufficient to cover at least the amount of the first two monthly installments, we classify the loan status as *declined*.

Notably, we intentionally excluded attributes like gender and citizenship from the *client* class, as they were deemed insignificant for client profiling purposes. Indeed, including sex and ethnicity in loan eligibility algorithms can perpetuate discrimination and reinforce biases, leading to unfair treatment and limiting opportunities for marginalized groups.

Lastly, the *main.py* file calls the *create_ongoing_loans(..)* function, which generates additional simulated loans in the *ongoing* SQL table by incorporating suitable loan acceptance parameters.

The configuration parameters for the technologies used, such as SQL database, have been chosen to optimize performance and accommodate the specific requirements of the loan management system. The table structure and column types are designed to accurately represent the loan and client information.

2. New Loan Ingestion

By running the *simulation_loan_application.py* file the user input their own loan request (implemented in the *input_data.py* file). This request is then transformed by the system into an instance of the *Loan* class and then ingested by the machine learning algorithms. However, in the above mentioned file, it is also possible to disable this mode and activate the creation of random loans for testing purposes.

3. Decision Process and Tailored Offer through Machine Learning Techniques

In our project, we utilized machine learning techniques to support the decision-making process based on previous loan request outcomes. Because real data was not available for testing the accuracy of the models, we incorporated three algorithms in the code: Hierarchical Clustering (HC), K-Nearest Neighbors (KNN), and Decision Tree (DT). Our objective was to provide a user-friendly approach that could be easily understood and interpreted. Therefore, we avoided complex algorithms like Random Forest and XGBoost, which can be perceived as black boxes.

Hierarchical Clustering was chosen because it can generate clusters of different sizes, allowing us to identify a variable number of loans from the records that closely match the input loan request. We set the

number of loans to be retrieved (n) to a value greater than 5. **K-Nearest Neighbor** was selected due to its fixed number of neighbors (k), ensuring that the algorithm always returns the k closest loans (k=5 in the code). This simplicity facilitates decision-making and enhances understanding.

As a careful reader may have pondered, these two methods rely on a distance metric. However, in our case, we encounter variables with differing scaling conditions (e.g., age and loan amount) and categorical variables. To address this challenge, we have chosen to utilize the **Gower distance**. Briefly, the Gower distance assigns a value of 1 when categorical variables differ and 0 when they are the same. For numerical variables,it calculates the normalized absolute difference between the values and divides it by the range of the variable. We deemed this distance measure to be the most suitable for our dataset. However, we acknowledge the inherent weightage present in categorical variables compared to numerical variables.

Lastly, the **Decision Tree** algorithm was utilized as it provides a clear and simple representation of the decision process. As we expected, the fitted tree depicts patterns that we established in our synthetic data. For instance, we observed that in Italy, having two or more delays recorded at Centrale Rischi or being classified as "sofferenza" at CRIF typically results in loan denial. These decisions are reflected in the decision tree shown in Figure 1.

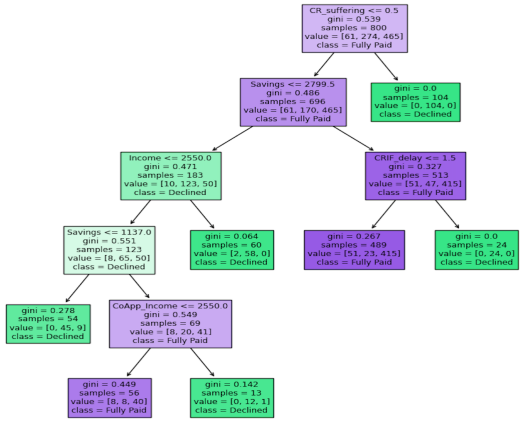


Fig. 1 Plot of the trained Decision Tree on the loan records (simplified version)

All three strategies mentioned above are implemented in the code (in the *clustering.py* and *tree.py* file), as we consider them to be effective approaches for decision-making in loan requests.

Let's now discuss how we derived a decision from the machine learning results. For the KNN and HC strategies, we counted the number of fully paid, charged off, and declined loans within the same cluster (for HC) or among the k closest loans (for KNN). Based on these counts, we made a decision regarding the loan request.

To accept a loan, we required the number of fully paid loans to be higher than the sum of the declined loans and six times the number of charged off loans. This decision rule assigned significant weight to the charged off loans, ensuring a cautious approach to mitigate potential risks. Different weights can be assigned to the decision criteria based on the bank's policy and specific considerations. These weights can be adjusted to reflect the bank's risk appetite, the desired number of loans to be approved, and the relative importance of predicting charged off loans versus fully paid loans. For example, if the bank wants to prioritize minimizing the risk of predicting a charged off loan when it would have been fully paid, they can assign a higher weight to the charged off category in the decision-making process. This would make the algorithm more cautious in approving loans to mitigate potential losses.

On the other hand, for the decision tree strategy, we input a new loan request (with the categorical variables binarized) into the trained decision tree model, and we get the resulting status.

In most cases, the three strategies yield consistent results. However, occasional disagreements can occur among them. This is precisely why we included all three strategies in our implementation. By considering the majority decision, we ensure a more robust and reliable determination regarding the loan request.

Additionally, the tailored offer, which is based on clustering, is only displayed when utilizing the first two machine learning strategies. We first identify the most similar loans, belonging to the same cluster. The customized offer is derived by selecting just the loans with status *Fully Paid* and with less *delays* in the installments, and by averaging their interest rates. Subsequently, the monthly repayment amount is calculated as approximately 30%[4] of the total income, considering both the applicant's and co-applicant's income if applicable. An adjustment of 400 euros is made for each dependent, if applicable.

Furthermore, the loan duration is mathematically computed using the monthly repayment amount and the interest rate, assuming a simple interest rate calculation. It is worth noting that banks have the flexibility to opt for a compound interest rate, and the code can be easily updated to accommodate such preferences.

4. Real-Time Loan Tracking and Dynamic Updates: Enhancing Decision-Making and Monitoring in a Loan Management System

Previously, we demonstrated a static part of the project where decision making was based on past loan records. These records were initialized at the beginning.

[4] This approach is widely used by banks, as it assumes that individuals need to allocate around 60-70% of their income for various expenses and financial obligations to maintain a sustainable financial situation.

Now, we will introduce a dynamic component that showcases how we update the loan records table. Specifically, we will illustrate how we track when a loan is fully repaid and how we add it to the loan records table, potentially marking it as "fully paid". This dynamic update allows us to maintain an up-to-date and accurate representation of the loan repayment status within our system, therefore up-to-date the algorithm.

First of all, by running the *simulation_ongoing_loans.py* the user will start a simulation of transactions per month. The transactions consist of loan_id, customer_id, amount, date, and time, and they correspond to the customer_ids present in the ongoing table in SQL. In the simulation, the analysis and storage of transactions occur at the end of each month, based on the number of months determined by the user. The user will find the corresponding number of folders, each containing the transaction files. The transactions are stored in JSON files, typically four files per month. These JSON files are ingested using PySpark, enabling efficient processing and analysis of the transaction data (see the *spark.py* file).

Simultaneously, a hash is created in Redis with the loan id as the key and two fields: *months_left* and *delays*. As the program processes the transactions, it updates the corresponding loan id field in Redis by subtracting one from the *months_left* field. For simplicity, we assume that all the repayments are due by the 27th of each month. If the program detects a payment made after the 27th, it increments the *delays* field by one. When the *months_left* field reaches zero, the loan id is removed from Redis. Additionally, the loan is moved from the *ongoing* table in SQL to both the *fully_paid* and *loan_records* table (see the key functions *check_loan_ids(..)* and *cancel_loan(..)* in the *analyzing_transaction.py* file).

IV. Results

The system presented can be effective in aiding the financial institution's loan acceptance decision-making process. Through the utilization of ML algorithms, a customer's risk profile can be accurately determined, taking into account various input restrictions.

The decision to employ three distinct algorithms stems from the recognition that each algorithm operates with different parameters, thereby enabling calibration to different decision-making processes. This diversity enhances the likelihood of complete loan repayment and mitigates the risk of loans being classified as *Charged off*, ultimately minimizing potential financial losses for the bank.

Furthermore, the system's ability to manage loans over time facilitates the continuous monitoring of loan performance and the enrichment of the loan records database. As a result, the algorithm is constantly improving and refining its predictive capabilities. In fact, with the adoption of the proposed ongoing loan managing system, there is no longer a need for manual initialization of the loan records. The system automatically populates the *loan_records* and fully_paid tables.

Through this project, several valuable lessons have been learned. Firstly, the importance of utilizing diverse algorithms with distinct parameters to enhance decision-making accuracy. Secondly, the significance of continuously updating and enriching the loan records database for improved predictive performance. Lastly, the benefits of integrating ongoing loan tracking systems to streamline data management and automate the population of relevant tables.

V. Conclusions

The project provides insights into a customer's risk profile based on accessible parameters for financial institutions. To enhance the system, utilizing real customer data would be more beneficial compared to synthetic data. Real data enables machine learning algorithms to identify potential fraud or non-performing payers using extensive databases. Implementing a daily update system for data analysis and storage would ensure the availability of the most recent information. Furthermore, for tailored offers, relying solely on previous interest rates is not ideal. Incorporating the inflation rate to calculate the interest rate would ensure alignment with current economic conditions and borrowing costs. The transaction simulation system implemented is approximate, suggesting the introduction of a streaming transaction update system that considers the financial availability of branches and the bank as a whole. This system would dynamically update the bank's capital when payments are made or new loans are granted.

References

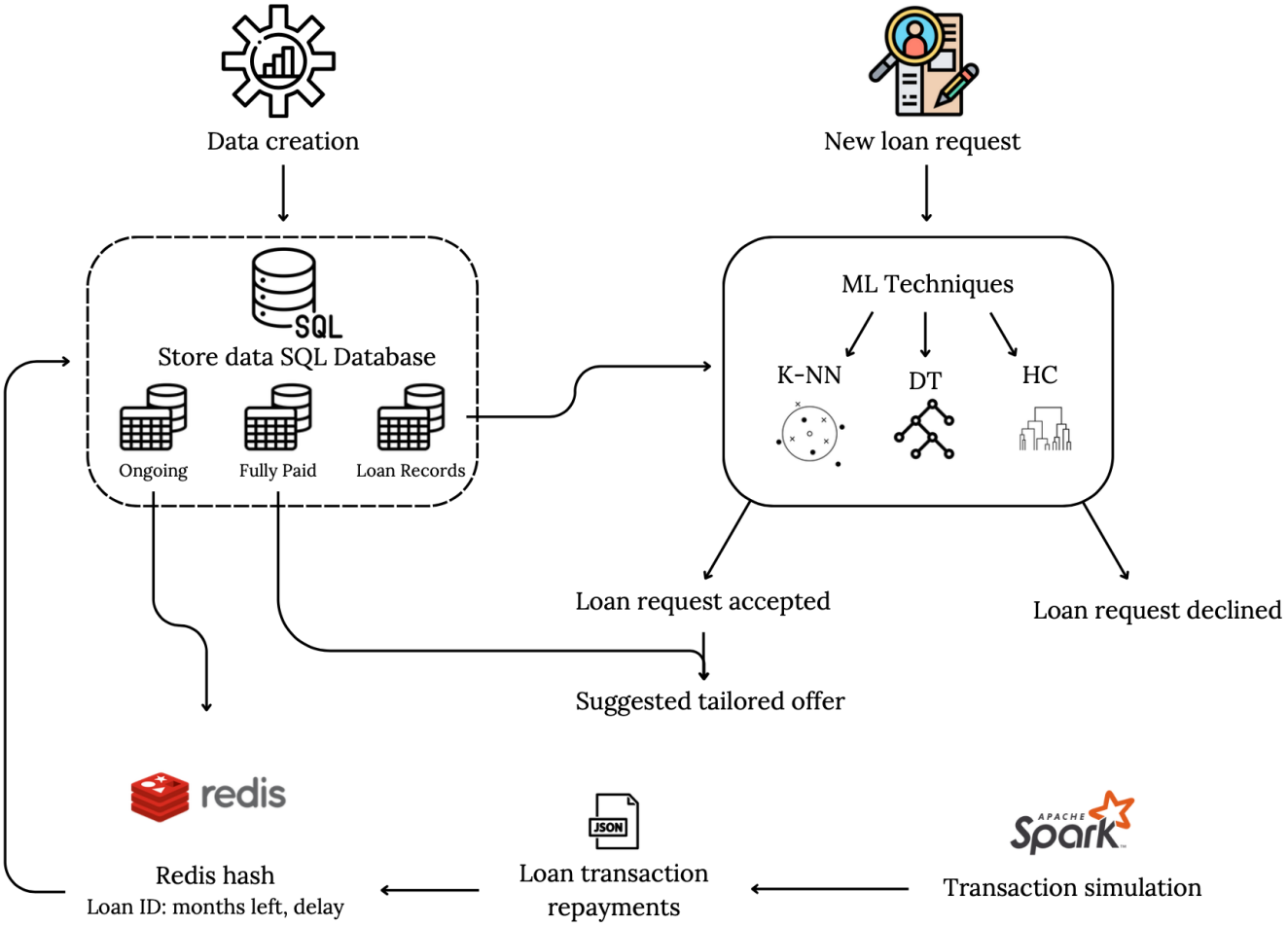
[1] ISTAT (2021). *Livelli di istruzione e ritorni occupazionali 2021*. Retrieved from: Livelli di istruzione e ritorni occupazionali - Anno 2021

[2] Tipresto.it. *Durata, Rata e Importo Massimo Prestito Personale: Quali Sono e Come Scoprirli*. Retrieved from: Durata, rata e importo massimo prestito personale: quali sono e come scoprirli

[3] Facile.it. *Prestiti Bancari a Termine: Come Funziona.* Retrieved from: Cosa sono i prestiti bancari a breve termine: parola all'esperto di Facile.it

[4] ISTAT (2020). *Reddito Netto 2020*. Retrieved from: Reddito netto

PIPELINE


Data creation
New loan request
SQL
Store data SQL Database
Ongoing
Fully Paid
Loan Records
ML Techniques
K-NN
DT
HC
Loan request accepted
Loan request declined
Suggested tailored offer
redis
Redis hash
Loan ID: months left, delay
JSON
Loan transaction repayments
APACHE Spark
Transaction simulation