Infrastructure and Recommendation System for Banking Products

Vicente Côxo Rocha

Thesis to obtain the Master Degree in

Telecommunications and Informatics Engineering

Supervisors: Prof. Dr. Manuel Fernando Cabido Peres Lopes
Dr. Ricardo Portela

Examination Committee

Chairperson: Prof. Dr. Ricardo Jorge Fernandes Chaves
Supervisor: Prof. Dr. Manuel Fernando Cabido Peres Lopes
Member of the Committee: Prof. Dr. Pável Pereira Calado

October 2018
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.
Acknowledgments

I would like to thank my parents, my little sister and my girlfriend for all the support.
I would like to thank my advisors at INM, Ricardo Portela and Francisco Chaves, for the opportunity, and Manuel Lopes at IST.
I would also like to thank André Nogueira, Isabel Costa, Karan Balu, Lucas Silva and Simão Figueiredo, for the help and crucial insights.
I also would like to thank all my friends who followed this journey with me.
A special thanks to Francisco Caixeiro, who walked this journey with me, without whom this project would not have been possible.
Abstract

Banks have a wide range of products to offer, and with that comes an overwhelmingness of choice for the client. Our challenge is to mitigate that and help both clients - to find the best offers - and banks - to increase product sales. We propose a Recommender System using an XGBoost model and the Multi-armed Bandit. The infrastructure of the Recommender System will be supported by Apache Kafka, which will be responsible to receive the bank’s database events and send them pre-processed to the database so that the recommendation system can use them in almost real-time. The evaluation of the Recommender System showed that it learned to give meaningful recommendations. The infrastructure built helped improve the development of applications that want to consume the bank’s database.

Keywords: Banking, Recommendation System, Apache Kafka, Decision Tree, Random Forest, XGBoost, Multi-Armed Bandit, Data Mining
Resumo

Os bancos têm uma ampla gama de produtos para oferecer, e isso torna a escolha difícil para o cliente. O nosso desafio é mitigar esse aspecto e ajudar os clientes - a encontrar as melhores ofertas - e os bancos - aumentando as vendas de produtos. Propomos um Sistema de Recomendação usando um modelo XGBoost e o Multi-Armed Bandit. A infraestrutura do Sistema de Recomendação é suportada pelo Apache Kafka, que é responsável por receber os eventos da base de dados do banco e trazê-los até ao nosso sistema de recomendação em tempo real. A avaliação do Sistema de Recomendação mostrou que ele aprendeu a dar recomendações significativas. A infraestrutura que construimos ajudou a facilitar o desenvolvimento de aplicação que desejam ler da base de dados do banco.

Bancos, Sistema Recomendação, Apache Kafka, Árvore de Decisão, Random Forest, XGBoost, Multi-Armed Bandit, Extração de Dados
# Contents

**List of Tables** xi  
**List of Figures** xiii

1 Introduction 1  
1.1 Motivation 1  
1.2 Aim and Objectives 2  
1.3 Structure 3

2 Background 5  
2.1 Infrastructure 5  
2.1.1 Event-Driven Architecture 5  
2.1.2 Microservice Architecture 5  
2.1.3 Databases 5  
2.2 Machine Learning 6  
2.2.1 Decision Trees 7  
2.2.2 Random Forest 9  
2.2.3 XGBoost 9  
2.2.4 Multi-armed Bandit 10  
2.3 Summary 11

3 Related Work 13  
3.1 Infrastructure 13  
3.1.1 Event processing 13  
3.1.2 Cloud solutions 14  
3.2 Recommender System 14  
3.2.1 Tree based solutions 15  
3.2.2 Reinforcement Learning solutions 16  
3.3 Summary 17

4 Solution 19  
4.1 Overview 19  
4.2 Infrastructure 20  
4.2.1 Source Database 20  
4.2.2 Apache Kafka 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.3 MongoDB</td>
<td>25</td>
</tr>
<tr>
<td>4.2.4 Data flow</td>
<td>25</td>
</tr>
<tr>
<td>4.3 Recommender System</td>
<td>26</td>
</tr>
<tr>
<td>4.3.1 Gradient Boosting model</td>
<td>27</td>
</tr>
<tr>
<td>4.3.1.1 Build the training dataset</td>
<td>28</td>
</tr>
<tr>
<td>4.3.1.2 Feature Transformation</td>
<td>31</td>
</tr>
<tr>
<td>4.3.1.3 Feature Selection</td>
<td>32</td>
</tr>
<tr>
<td>4.3.1.4 Sampling</td>
<td>32</td>
</tr>
<tr>
<td>4.3.1.5 Model Training</td>
<td>32</td>
</tr>
<tr>
<td>4.3.1.6 Gradient Boost model in action</td>
<td>33</td>
</tr>
<tr>
<td>4.3.2 Multi Armed Bandit</td>
<td>33</td>
</tr>
<tr>
<td>4.3.2.1 Multi Armed Bandit in action</td>
<td>34</td>
</tr>
<tr>
<td>4.4 Summary</td>
<td>34</td>
</tr>
<tr>
<td>5 Evaluation</td>
<td>37</td>
</tr>
<tr>
<td>5.1 Infrastructure</td>
<td>37</td>
</tr>
<tr>
<td>5.2 Recommender System</td>
<td>39</td>
</tr>
<tr>
<td>5.2.1 Overall Comparison</td>
<td>40</td>
</tr>
<tr>
<td>5.2.2 Impact of Feature Transformation</td>
<td>41</td>
</tr>
<tr>
<td>5.2.3 Impact of Sampling</td>
<td>42</td>
</tr>
<tr>
<td>5.2.4 Impact of Feature Selection</td>
<td>43</td>
</tr>
<tr>
<td>5.3 Summary</td>
<td>44</td>
</tr>
<tr>
<td>6 Conclusion</td>
<td>45</td>
</tr>
<tr>
<td>6.1 Achievements</td>
<td>45</td>
</tr>
<tr>
<td>6.2 Future Work</td>
<td>45</td>
</tr>
<tr>
<td>Bibliography</td>
<td>46</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Example of the training dataset ........................................ 29
4.2 Frequency of products in the Dataset ................................. 31

5.1 Overall performance evaluation using the F1 score. The Raw row are the tests on the raw dataset. The FT are the tests after applying Feature Transformation, the SP are the ones after the Sampling and FS after the Feature Selection. In bold we can see the best F1 score for each product ................................. 41
5.2 Sampling F1 scores. US stands for undersampling and OS for oversampling. Bold scores are the best for each product ........................................ 43
## List of Figures

1.1 Project vision. The C/P are consumers or producers of data to Kafka. The bold lines represent the data pipeline we built for this project. The light lines represent the future vision and what we want to support. ........................................ 3

4.1 Overview of the infrastructure ..................................................... 20

4.2 Kafka Connect in more detail. On the left we have \( n \) tables which correspond to \( n \) source connectors. The connectors are populating the respective Kafka topics. On the right we have \( n \) sink connectors consuming those topics and inserting them into their respective MongoDB collections .......................................... 24

4.3 Infrastructure: Flow of data generated by the client .......................... 26

4.4 Entity, source client and account relationship example ...................... 27

4.5 Development cycle for improving the supervised learning algorithm ........ 30

4.6 XGBoost Recommender System architecture .................................. 33

4.7 Multi Armed Bandit Recommender System architecture .................... 34

5.1 RFECV for product 0 and 7. ......................................................... 43
Chapter 1

Introduction

1.1 Motivation

Banks have a wide variety of products/solutions for their clients, which can be both an advantage and a disadvantage regarding offering the best products to the clients. The advantage is that with a wide variety of products, the bank can reach the heterogeneity of clients needs. However, we can also see a clear disadvantage here. How does the bank know which product is the best fit for a particular client? How does the client choose a product accordingly to his needs? There can not be a specialist looking into the profile of every client and analyzing their features to find what products best suit their needs. At best, clients can go to the bank and request for a specialist’s advice. However, people want convenience and flexibility [18].

Financial technology, or fintech for short, can be described as financial solutions supported by technology, where financial services and information technology are tightly coupled. Digitalization of finance is not new, it has been happening since the late twentieth-century [4]. What is new is the way financial products are delivered. Nowadays, many powerful financial institutions are transforming themselves into a digital platform-based banking ecosystem [26]. Almost every bank nowadays offers multichannel solutions. Branches, internet-banking, ATMs, all share the customer information and can create a seamless experience. With that banks also have the advantage of a panoramic view over the clients’ activity [22].

Banks generates data from clients every second. Credit card movements, bills payments, interbank transfers, car leasings, and much more. There is a massive opportunity of transforming that data into information we can use. We can build a system that automatically learns from that data and knows which products best suits any given client at any given time. The financial area is a very secretive area. Banks have for a while now been using financial technology. However, the techniques and results are usually not revealed due to the very competitive and private environment associated with this domain.

The big problem to tackle here is the tailor-made solution the client wants. Clients want solutions tailored to their specific context. The client today might not be the same as the client tomorrow. Take for example a client who recently got a raise, or at the other end of the scope, a client who just got fired. Both their contexts suddenly change and most likely so do the products that should be recommend for them. How can the bank adapt to their clients’ dynamic context?

The impact and development of e-commerce are noticeable in a worldwide scale [40]. It is
important to note some fundamental techniques used to sell products online. Recommendation systems often use two fundamental selling techniques. Cross-selling and Up-selling. Cross-selling being when the seller tries to sell an item to a client based on a previously purchased item or an ongoing purchase. This happens for example in online retail, when we were browsing a shopping website, and it suggests that users that bought the camera we are looking at, also bought some memory cards. Here, the seller is trying to cross-sell the memory card. Up-selling is when the seller tries to sell a premium or a higher cost product to the client [40]. Following the same example, the retailer could suggest that for just a bit more we could have a better camera, that for example has more pixels, or has Bluetooth. These techniques should be taken into account when building a recommender system.

So, what can we do with the information the bank has on its clients. For now, we will focus on building a pipeline of that information so that we can build a product recommendation system. While at the same time building a foundation to will allow the bank to increasingly add new micro-services that consume that pipeline and progressively create a tailor-made experience for each client. This project is being developed at Innovation Makers, which gave us the opportunity to explore the financial technology world.

As a note, for privacy reasons, the name of the bank will be omitted, and so will the products’ descriptions.

1.2 Aim and Objectives

One of our main objectives is to find which product is the client more likely to acquire at any given time. For that, we propose a machine learning recommender system using the XGBoost and Multi-Armed Bandit algorithms. A simple approach would be just to create an application that would read the bank’s database, do the necessary data processing, train the algorithm and create a model ready to be invoked whenever a client logs in on the application. But one of our other objectives is to create a real-time pipeline of the bank’s client data so that many applications can consume that data. Not only that but we want to be ready for new sources of data, like for example receiving the client’s app behavior in real time or the shopping details of the supermarket visits. The long-term goal is to allow n producers and n consumers of data, and even allow to play both roles. For example, have an application that reads supermarket visits and is categorizing that data for a data visualization tool to read that categorized data and display it for the analysis team to interpret. Our goal for the infrastructure is to achieve what is depicted in figure [1.1]. The bold lines represent what was built for this project, whereas the lighter arrows represent what we want to allow to be possible.

When building our infrastructure one of the main focuses was to try to build an infrastructure that is scalable, efficient and allowed us to integrate future applications easily. Our infrastructure is a crucial part of the project since it will not only define how easy it is for our recommender system to consume and make use of the client’s data but also how easy it is for us to integrate more applications in the future. We aim to develop a scalable, micro-service based, near-real-time infrastructure that supports our product recommendation system. It should be capable of effectively recommending the bank’s products to their clients, which consist in common financial products such as savings accounts, interest deposits, and others.
Figure 1.1: Project vision. The C/P are consumers or producers of data to Kafka. The bold lines represent the data pipeline we built for this project. The light lines represent the future vision and what we want to support.

In order for us to achieve that, we set some crucial objectives.

Objectives

- Analyze the bank’s database to conclude if it’s possible to build a recommender system from it;
- Build a pipeline system that receives the events from the bank’s database (inserts/updates) and process them in near real-time;
- Build the pipeline to scale and facilitate the integration of applications that want to consume data from the bank’s database;
- Populate a database with the processed events;
- Build a recommendation system that will learn from that normalized database and is capable of recommending the banks products;
- Have the recommendation system respond to real-time requests.

In summary, we propose to build a Recommender System for the bank’s accounts (products) using a XGBoost and a Multi-Armed Bandit model. The Recommender System is supported by an infrastructure that simplifies and improves the process of adding new applications to consume data from the bank’s database.

1.3 Structure

The remaining of the document is organized as follows:
• Chapter 2 introduces some basic concepts related to both recommender systems and to the infrastructure that supports it.

• Chapter 3 first analyzes work done on recommender systems that are relevant to the scope of our project and then explores technologies relevant to build the infrastructure.

• Chapter 4 details the solution developed.

• Chapter 5 details the evaluation of the Recommender System and critically evaluates the infrastructure we built.

• Chapter 6 states the conclusion and the contributions and future work of this project.
Chapter 2

Background

In this section we present some basic concepts that are relevant to the context of our project. Section 2.1 explains some basic concepts that are relevant in order to take a recommender system to production. Section 2.2 explains some machine learning basic concepts and introduces the algorithms we will be using in our project. Section We end with section 2.3 which summarizes the background section.

2.1 Infrastructure

2.1.1 Event-Driven Architecture

In an event-driven architecture, when something relevant happens the interested parties are informed. Those parties will decide how to act upon the arrival of that event. Such event-driven action could be for example the invocation of a service or further propagation of the event [39]. In this project, the events are a direct or indirect consequence of clients actions.

2.1.2 Microservice Architecture

In a microservice architecture, software functionality is composed of lightweight services. Together they are designed to deliver a complete and robust set of business functionalities [36]. In this architecture, there is a possibility for the same service to be reused by another one [21].

2.1.3 Databases

When it comes to databases, there are two types of data structures, relational databases and non-relational databases usually called NoSQL databases.

Relational databases use a table structure to store information. Those tables are well defined and can have relations between them [17]. The standard application language used is Structured Query Language (SQL). Relational databases follow a predefined schema that determines the structure of the data. Thus it can be challenging to try to make changes to that schema.

Non-relational databases, on the other hand, are more flexible and dynamic than relational databases. They are not organized in tables and usually do not use SQL when interacting with the database. The data is unstructured and has a dynamic schema. It allows data to be stored in several ways, like documents, graph, key-value store or even columns. Using non-relational
It is possible to create documents without needing to define a structure. It also allows us to add fields dynamically, which means each document can have its own structure.

2.2 Machine Learning

Machine learning is the subfield of artificial intelligence (AI) that allows computers to learn. It can be defined as computational methods that use a sufficient amount of data to improve metrics like performance or prediction accuracy \[41\]. Unlike the majority of computer science areas, which mainly follow a deductive line of thought, ML instead follows an inductive line of thought. We could say it learns empirically, that is, it makes assumptions based on previous observations. Imagine for a second that we are a receptionist at a bank. We see that the last ten clients making a deposit have asked us how much time does it take before their balance is updated. We could assume that the next person making a deposit will also ask us the same question, and we could tell them that before they ask the question. The success of a learning algorithm is inherently dependent on the data from which it learns. With that being said, we cannot discard the importance of data analysis and statistics, since learning algorithms are data-driven methods that join computer science with statistics and probability \[41\].

ML algorithms can be divided into three main types, *Supervised, Unsupervised and Reinforcement Learning*. Supervised learning learns from labeled datasets. It relates the features with the given label \[13\]. The example above can be seen as supervised learning, where the feature is the action of making a deposit, and the target we are trying to model is whether or not the client will ask the question. Supervised learning is mostly used in algorithms of *classification* - predict a target class, e.g. win or lose football game - and *regression* - predict a response, e.g. movie ratings stars \[12\]. Unsupervised Learning tries to learn from datasets where there are no labels. The algorithm has the job of suitably organizing the data. Take for example a set of scientific papers. The algorithm tries to infer the main subject addressed in each paper. A naive approach would be to infer that the most frequent word categorizes the paper. Unsupervised learning is mostly used in *clustering* - grouping of similar items - and *dimensionality reduction* - reduce the number of random variables to consider \[42\]. Reinforcement Learning can be seen as the problem of learning through trial-and-error with interactions with an environment which is dynamic by nature. It can be applied into building autonomous systems that can learn and improve themselves with experience \[24\].

Recommender Systems (RS) can be seen as software tools and techniques that provide suggestions to a given recommendation problem \[47\]. That problem can vary from what news to read, what product to buy, what movie to watch, and more. RSs are usually focused on one particular area. In our case, we will focus on banking products recommendations. There are usually two main reasons behind a recommender system. Either to help users who are not familiar with the area in question or who lack the necessary personal experience to make a sustained decision. For instance, a banking solution/product can be tough to evaluate without some financial knowledge, or when there is an overwhelming number of alternatives. An online bookstore, where there is a huge amount of book choices, is an example where RCs can thrive and improve customer experience. This is also related to the long tail problem. The first part of the tail is composed of a small number of popular items, with a high rate of hits and the long
part of the tail composed of a vast number of unpopular items, with an accordingly minimum rate of hits \[14, 58\].

Recommender systems methods are mainly divided into three categories. *Content-Based*, where the item recommendations are based on the similarity of previously liked items. *Collaborative*, where either the users or items similarity is used to base the recommendation. When the focus is on the items, they are recommended based on the user’s evaluations of similar items \[28\]. When the focus is on the users, items are recommended based on users with similar tastes \[47\]. Finally *Hybrid* systems, which as the name suggests combine multiple approaches, where the main focus is to mitigate the disadvantages of some approaches with the advantages of other.

In a real-world scenario, recommender systems need to collect feedback from which to learn. These can be done using two types of feedback, *explicit* and *implicit*. Explicit feedback is the type of feedback where the user has some mechanism to give feedback to the system. For example in video sharing websites, there is usually an up or down vote button, which the system can use to find out if the user liked or not the video, and thus build a recommender system. Implicit feedback, on the other hand, is harder to collect. If we take the same example, where the system wants to find if the user liked or not a particular video, but there is no up or down vote button, the feedback is much harder to collect. A straightforward strategy could be if the user saw the entire video, they might have liked the video, and if they saw another video from the same author, we might increase the probability of that user having liked the video.

The next subsections will describe the three algorithms that will be explored throughout this project: Decision Trees, Random Forests, XGBoost, and Multi-armed bandit.

### 2.2.1 Decision Trees

Decision Trees can be categorized as a *supervised learning* method. It can be used for both regression and classification. The main objective is to predict the values of a given target variable. It can be achieved by building a model that learns a set of decision rules from the data features. The prediction is made by classifying each observation as belonging to the most frequent class of training observations in the region to which it belongs. The interpretation of the results is generally focused on two aspects. The class proportions among the training observations that fall into a particular terminal node region. And in the class prediction corresponding to that region.

The growth of the tree is made using recursive binary splitting. To make the binary splits, the classification error rate is used. Our objective is to take an observation of a given region and assign it to the most frequent class on the training observations in that region. Which mean that the classification error rate will be the fraction of the training observations in that region that do not belong to the most frequent class:

\[
E = 1 - \max_k \hat{p}_{mk}. \tag{2.1}
\]

Where \(\hat{p}_{mk}\) means the proportion of training observations in the \(m\)th region that are from the \(k\)th class. Despite the theoretical applications of classification error, in practice, it is shown that it is not sensitive enough to allow for the tree to grow. There are two alternative measures which achieve better results. The Gini index and entropy.
We will first explore the Gini index, given by

\[ G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}). \tag{2.2} \]

It measures the total variance across the K classes. If the values of all \( \hat{p}_{mk} \) are either close to one or zero, the Gini index’s value will be almost zero. Which is helpful because if we see that the Gini index is small, we can conclude that the given node contains mostly observations from a particular class.

The other alternative to the classification error is entropy, defined by

\[ D = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}. \tag{2.3} \]

We know that \( 0 \leq \hat{p}_{mk} \leq 1 \), which gives us \( 0 \leq -\hat{p}_{mk} \log \hat{p}_{mk} \). Just like before, if the value of all \( \hat{p}_{mk} \) is either close to one or zero, the entropy value will be almost zero. Also, just like the Gini index, the entropy value will also be close to zero if the \( m \)th node is pure. We can see the similarity between the Gini index and entropy. When evaluating a quality of a particular split, for classification trees, either the Gini index or entropy should be used.

An important part of building a decision tree is pruning. It consists of letting the tree fully grow and then prune it back to obtain a subtree. Given a subtree, a tedious and cumbersome way of determining the best subtree would be to estimate its test error using for example cross-validation for every possible subtree. A smart way to do this is to use weakest link pruning. We need only to consider a sequence of trees indexed by a nonnegative tuning parameter \( \alpha \). For each value of \( \alpha \) there corresponds a subtree \( T \subset T_0 \) such that,

\[ \sum_{m=1}^{[T]} \sum_{i : x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T|, \tag{2.4} \]

is as small as possible. The number of terminal nodes of the tree is indicated by \( T \). \( R_m \) is the subset of predictor space corresponding to the \( m \)th terminal node. \( \hat{y}_{R_m} \) indicates the predicted response associated with \( R_m \). The tuning parameter \( \alpha \) controls a trade-off between the subtree’s complexity and its fit to the training data [13]. When increasing the number of terminal nodes there is a potential chance of overfitting, that is, the model could tightly fit the training data but could fail to generalize well and be completely useless when classifying unseen data.

When \( \alpha = 0 \), then the subtree \( T \) will simply equal \( T_0 \). This happens due to equation 2.4 property of only measuring the training error. A good side effect of \( \alpha \) increasing is that the tree branches get pruned in a nested and predictable way. Which makes it easy to obtain the entire sequence of subtrees as a function of \( \alpha \). Using either a validation set or cross-validation we can then select a value of \( \alpha \). After that, we can go back to the full data-set to obtain subtree corresponding to \( \alpha \).

Decision trees can be handy for interpretation as we can see the decision tree generated by the algorithm [27]. One negative aspect of decision trees is the sensibility to data changes, that is, a small change in data can generate a completely different tree. As previously stated, another common problem for decision trees is the chance of overfitting the model when we increase the
number of terminal nodes.

The next section describes an algorithm that tries to improve the Decisions Tree algorithm.

### 2.2.2 Random Forest

In the previous section, we saw the weaknesses of using a single decision tree. Breiman et al. [8] was not the first to note those weaknesses but was the first to develop a solution, which he called Random Forests (RF). The main idea behind Random Forests is to instead of using just one tree, is to use several trees and combine the results to achieve a better and more accurate model.

One of the main concepts behind RFs is the Ensemble Method, which consists of combining many outputs generated by distinct predictors in order to achieve better prediction results. By combining many models, we expect to achieve better results using an ensemble than when using them individually. The result of the prediction will be a combination of all the predictions from the different models. If the problem we are trying to model is a regression, the final prediction value will be an average of all the predictions combined, if it is a classification, then the final prediction value will be the majority of the predictions [27]. As a consequence of the combinations of the predictions, the variance is lowered, and the overfitting probability is also reduced [27].

There are several ensemble methods, but for now let us focus on the one that is used when training RFs, called Bagging. It defines the selection strategy chosen to generate the many Decision Trees used. The strategy consists of three parts. First randomly select multiple subsets of the training data. Second, train a model for each generated subset. Finally, aggregate the predictions.

RFs use another strategy to increase randomness and mitigate variance, often called feature bagging [53], where each tree will train with a random subsample of features. This increases the diversity of the generated trees which further reduces the variance, but as a consequence increases the bias.

Along with reducing variance and the overfitting chance, a significant gain from RFs is the ability to determine feature importance, as important features will tend to be at the top of the trees and consequently less important features will tend to be at the bottom.

### 2.2.3 XGBoost

XGBoost is an implementation of gradient boosting technique, which is another ensemble method. Contrary to bagging, instead of using independent models, they are built sequentially. Here the main idea is to learn from past mistakes [27]. Each iteration takes into account the previous one and will focus more on observations that lead to a mistake. Contrarily to bagging, where the probability of using a training observation is random, here the observations with more errors will appear more.

So the main idea here is instead of building a massive tree prone to overfitting or to use many independent trees and combine the results, here we build a slow learner based on the residuals of previous models.
So the algorithm starts by creating a tree and then by fitting another decision tree to the residuals from the model. That is, instead of learning to fit $Y$, this decision tree will learn to fit the difference between the predicted value and the actual value of $Y$ (residual).

If we call our model $F$, let:

$$\hat{y} = F(x),$$  \hspace{1cm} (2.5)

be the equation that defines the value our model is trying to predict. Let $h(x)$ be the estimator for the residuals. We can now define the improved model:

$$F_{m+1}(x) = F_m(x) + h(x) = y,$$  \hspace{1cm} (2.6)

where, $y - F_m(x)$ is the residual. Each $F_{m+1}$ tries to reduce the errors of $F_m$.

When using gradient boosting algorithms there three parameters that can be tuned to maximize its performance. First, the number of decision trees, which unlike random forests, can cause the model to overfit. Second, a shrinkage parameter, which controls the learning rate of the model. Finally, the number of splits in each tree.

### 2.2.4 Multi-armed Bandit

The Multi-armed Bandit problem, defined by Herbert Robins in 1952 [48], has been a vastly researched problem. [32, 5, 19]. Here we will focus on contextual Multi-armed Bandit problems. We will use the typical slot machine problem for an introductory explanation. Imagine there are $i \in [1, k]$ slot machines arms to choose from. Each with an unknown probabilistic distribution of giving a reward $r_1, ..., r_k \in [0, 1]$. The main objective is to determine which arm-pull will give the highest reward. The player observes contextual information $x$ before choosing which arm to pull. It is important to formally define both the contextual bandit problem and algorithm. Citing Langford et al., 2007, [33]:

**Contextual Bandit Problem:** *In a contextual bandit problem, there is a distribution $P$ over $(x, r_1, ..., r_k)$. Where $x$ is the context and $a \in 1, ..., k$ is one of the $k$ arms to be pulled. The reward of an arm $a$ is given by $r_a \in [0, 1]$. The contextual bandit problem is a cycle. In each iteration a sample $(x, r_1, ..., r_k)$ is drawn from $P$, where the context $x$ is revealed, there is a choice of an arm $a$ and the the reward $r_a$ is consequently revealed.*

**Contextual Bandit Algorithm:** *A contextual bandit algorithm $\beta$ determines an arm $a \in 1, ..., k$ to pull at each time step $t$, based on the previous observation sequence $(x_1, a_1, r_{a_1}, ..., (x_{t-1}, a_{t-1}, r_{a_{t-1}})$, and the current context $x_t$.*

The main focus of the MAB algorithm is to find the balance between two fundamental techniques. Exploitation and exploration. Exploitation is the part where the algorithm exploits past events to choose the arm he thinks has the best probability to give the best reward at a given time. Exploration is when the algorithm tries another arm, that appears not to be the optimal one, in order for it to increase its knowledge on that arm. This is important because the algorithm may think the arm it had been exploiting was the one that had the best outcome
when in fact it was not. Exploration can be crucial to obtain a short-term regret by theoretically reducing the number of suboptimal arm choices. So, the trade-off between exploration and exploitation at each step is key to an optimal Multi-armed Bandit algorithm.

There are two important performance measures we should note [31]. First, the optimally expected payoff. It can be seen as,

\[ \mathbb{E}\left[ \sum_{t=1}^{T} r_{t,a_t^*} \right], \]  

(2.7)

where \( a_t^* \) is the arm that has the maximum expected payoff at a given time step, \( t \). Another important measure is the total expected regret,

\[ R_T = T \mu^* - \sum_{t=1}^{T} \mu_{j(t)}, \]  

(2.8)

where \( \mu^* = \max_{i=1,\ldots,k} \mu_i \) is the expected reward from the best arm.

### 2.3 Summary

This section presented some basic concepts that are relevant when considering the development and planning of this project. We introduced the event driven, microservice architecture and SQL and NoSQL database. We explained the tree based methods relevant to the development of our recommendation systems. We also described the Multi-Armed Bandit, which tries to find a balance between exploitation of the best arm and the exploration to increase the probabilities of exploit the best arm. To build the infrastructure of the recommender system we need to be familiar with the event-driven architecture which provides a flexible and scalable solution to dealing with client’s actions. Micro-services architecture characteristics facilitates the development of a robust and flexible solution.
Chapter 3

Related Work

In this section we introduce concepts and technologies that are pertinent to the context of our work. In order to develop a recommender system we need an infrastructure to support the machine learning algorithm. So we divided this section into two parts. First, in section 3.2 we begin by analyzing the machine learning solutions. In section 3.1 we analyze the technologies that can be used to support a recommender system, and take it to production.

3.1 Infrastructure

To make the recommender system scale to production and fill the requirements of scalability, micro-service architecture and near-real time, we need to build a robust infrastructure. We want to build a pipeline that brings the banks events to our recommender system in a seamless fashion. One of our goals is to facilitate the training process, make it as easy and as automated as possible. In order to do that, we need to read data from the bank’s database, process that data, and finally feed that data to the recommender system. In this section, we will analyze some technologies and solutions that could make it possible.

3.1.1 Event processing

In the scope of event processing, we will explore three widely adopted solutions [29].

Apache Kafka is a streaming platform that can act as a distributed resilient log. It allows applications to store, read and transform data. In Kafka, that data is called a record. It stores streams of records grouped in topics. There are four APIs available: Producer, Consumer, Streams and Connector. They are all relevant to the context of this project and allow applications to do different jobs. The producer makes it possible to publish a stream of records to Kafka. The Consumer makes it possible to subscribe and consume a topic. The Streaming makes it possible to efficiently consume and transform input streams and publish them into output streams. The Connector facilitates the communication with external applications and systems like databases [30].

Apache Spark is a robust solution when it comes to fast and cluster computing. It allows
applications to work on top of it, like MLlib, a library for machine learning, GraphX, a graph processing, or Spark Streaming, a streaming framework which is very similar to Kafka Streams [59]. Spark architecture is divided into four parts. Spark Core, which is, as its name points out, the core of Spark, it is responsible for maintaining everything in order. Spark SQL works on top of Spark Core and allows the data to be efficiently queried using SQL syntax. Spark Streaming allows to stream and to transform data in real time. MLlib is a machine learning framework which allows building machine learning applications that run on top of Spark. Finally, GraphX is a distributed graph processing framework.

Spark can work on top of Kafka, by having spark work as a subscriber to a given Kafka topic, processing it and publishing the processed data onto Kafka again so other applications can use it. Spark has been widely adopted in the technology industry. For example, Guavus used it to create an operational intelligence platform [12]. Graphflow used it to build a real-time customer intelligence platform [46]. Another case is Zoomdata using Spark to build an intelligent analytics platform [34].

Apache Hadoop, unlike the above solution, instead of streams, it uses batch processing to power its system. Its primary focus is offline data processing. It stores, processes and analysis data. Its core objective is to solve the problem of big data analysis, by separating and distributing data so that it can be analyzed concurrently. As all the above solutions, Hadoop is open-source, which is a huge benefit. Its default storage layer is HDFS (Hadoop Distributed File System). Hadoop offers a MapReduce framework specifically designed to tackle map-reduce problems. It also works well with NoSQL, due to its advantages in operational tasks and interactive workloads.

3.1.2 Cloud solutions

Other solutions help us build an infrastructure for our recommender system. There are Software as a Service (SaaS) cloud solutions like Amazon’s AWS Machine Learning, Microsoft’s Azure ML or IBM Watson. Which offer ready to use solutions, that are efficient and scalable. Moreover, integrate well with other solutions from the providers. For example, AWS Machine Learning service can integrate solutions related to querying processed data or real-time data analysis. The big downside of using this “all-in-one” solutions is data lock-in. Another limitation is the lack of control in adjusting the machine learning algorithms. An also critical issue is privacy because the bank’s data would need to be sent off-premise and it is typically not an option in the banking industry.

3.2 Recommender System

When building a recommender system, there are a wide variety of approaches one can take. Depending on the objectives and requirements. The usual approaches reside within this three categories, Content-Based, collaborative or hybrid. Collaborative filtering recommender systems are the most widely adopted strategies when building a recommender system. In a nutshell they assume that people who have shared the same opinion/behavior will share it in the future. For example, given two people who acquired the same item, let us say a Savings Account, they
are more likely to acquire one in the future that comparing to a random person. When following a content-based approach, recommender systems try to find similarities between items that a person has previously acquired. For example, if a client has signed up for a savings account in the past, they might be more inclined to acquire a new one, when the current one expires, than they would be to acquired a credit card. Hybrid recommender systems appear when we want to combine two or more approaches, where the main idea behind is that we want to mitigate the weaknesses and improve the strengths of all the approaches involved.

Here, we present some solutions related to the core of our recommender system. We analyze and compare them. There is not much information on banking recommender systems, due to its secretive environment, so we will explore recommender systems from other domains but in which their work can be useful to help us decided our approach.

3.2.1 Tree based solutions

Tree based solution can be beneficial when building recommender system because even though they are relatively simple and easy to implement, they can achieve great results [16]. The first tree based approach recommender systems started to use were Decision trees (DT), which are particularly efficient and easy to interpret [61].

In the domain of movie recommendation systems Li and Yamada [45] developed a movie recommender system in which they used the decision tree algorithm. The DT was used to construct both the user preferences and the recommendation. In the test scenario, the users were asked to evaluate from 1 to 5 stars the list of movies they were given based on their profile. Using the DT to make recommendations they achieved good results with a Mean Absolute Error (MAE) of around 0.7, which means that most of the times the algorithm makes good predictions, with at most one rank of deviation in average. Utku et al. [55] also decided to use DTs for movie recommendations, but instead of explicit feedback, they used an implicit feedback strategy. The DT used the Gini algorithm. They obtained user behavior from users clicks to feed the DT. For testing, their setup used the help of 200 student volunteers. The results where promising as they achieved a recall, precision, and accuracy of around 90%. Gershman et al. [23] proposed a more general approach for a DT recommender system. They introduce two exciting improvements to a traditional DT recommender system. To improve scaling when producing a list of recommendation the leaf nodes of the DT produced a list of recommended items opposed to single items. The second improvement was the change of the splitting method to a new criterion, the Least Probable Intersection Size, meaning it computes the intersection probability for each potential split in a random split and selects the one with a smaller probability. The evaluation was done using movie ratings. It shows it to outperform the quality of well-known recommendations that use another splitting criterion. Sheth and Deshpande [50] proposed a probabilistic record linkage using DTs. It classifies the matching entities by linkage. The record linkage results then are used in the recommender system domain to produce a list of recommendations. The domain they tested in was book recommendations, which showed good results and proved to be good for cold start problems.

More DTs recommendations were successful in improving the results, like Bouza et al. [7] which uses an ontology approach to generalize item features and therefore improve the results of
the recommendations. Amin et al. [2] used DTs to improve backpackers experience by providing more precise accommodations suggestions. Sun et al. [52] showed improved results for cold start recommendations using DTs.

Following the Decision Tree algorithm, Random Forests appear as an improvement by combining many trees to arrive to a prediction decision. Zhang and Min [60] propose a framework for a recommender system using Random Forests to provide a three-way recommendation. The three-way recommendation either rejects the recommendation, accepts it or waits for a condition [57], for example in a medical decision making scenario. The algorithm uses the Random Forests model. The tests were done using the MovieLens dataset. The model proved to outperform other commonly used baseline algorithms. Brommund and Skeppstedt [9] developed an hybrid recommender system for movies using Random Forests and k-Nearest Neighbours (kNN). The Random Forest model focused on content-based filtering and the kNN focused on a collaborative filtering approach. Each model generates a list of recommendations and then the recommendation is done using five of the commonly generated recommendations. If there are not five, the list is completed with the overall top rated movies. The tests were also done using the commonly used MovieLens dataset and using the F1 metric. In average the results showed a seven percent decrease of performance comparing to other models. Ajesh et al. [1] build a recommender system using KMeans and Random Forests to predict recommendations based on users ratings. The KMeans algorithm is used to cluster users based on their ratings given to each movie. The objective of the clustering is to try to cluster users with similar tastes. The Random Forest algorithm is used to predict the rating the users give to movies. The metric used to evaluate the results was the Root Mean Square Error (RMSE), which was around 0.9%. The results were very similar to other approaches that use kNN for user clustering, such as of Bystrom [10], and Bao et al. [6].

Gradient Boosting has been developed as an improvement of Random Forests, as it generally can achieve better results and mitigate the overfitting problem [27]. To the best of our knowledge there are no papers about recommender systems using Gradient Boosting, we will try to show that it can be used in a Recommender System.

3.2.2 Reinforcement Learning solutions

The multi-armed bandit can be very useful for exploring products that are not very often bought due to the lack of knowledge about them, but they can have the opportunity to be very lucrative. The MAB algorithms try to explore those lesser known products and bring them to the client’s eye. There is a wide range of domains where solutions for the multi-armed bandit product have been developed, we will explore some of them.

Ek and Stigsson [51] propose a solution for garment-based e-commerce systems. They use a Contextual Multi-Armed Bandit Algorithm to make product recommendations. They used the Apache framework Mahout and LensKit to develop the recommendation algorithm. The evaluation relied on the measuring of the percentage of successful recommendations. They also experimented with different privacy-preserving techniques, where increasing privacy resulted in worst results. With the full algorithm they achieved an average prediction rate between 19 and 21 percent. In their paper Mary et al. [38] explore the MAB for online recommendation
systems focusing on the cold-start problem. They used a combination of matrix factorization and the contextual bandit with the objective of balancing the trade-off between exploration and exploitation. They used several large openly available datasets like Netflix movie ratings and MovieLens. They achieved promising results, outperforming common strategies like greedy or Upper Confidence Bound and mitigating the cold-start problem. Felício et al. [20] also focused on the critical cold-start problem. The core here was to develop an algorithm for model selection during the cold-start phase. It consisted of three phases. First, compute the prediction models, then sort the predictions and finally recommend. Even though it was a preliminary experiment, it shows motivating results with 86% of accuracy. Caron and Bhagat [11] made another cold-start mitigation improvement. They tried to improve cold-start recommendations in a social network, using two strategies that leverage the bandit neighbors. MixPair that extends UCB1, and MixNeigh based on consensus in the neighborhood of a user. The results showed an improvement of 2 to 5 times in relation to the common UCB1. Li et al. [35] used the contextual bandit problem to tackle personalized news recommendations. To improve the trade-off between exploration and exploitation, they proposed LinUCB, as in Linear Upper Confidence Bound. It can be adjusted with an input parameter. The news recommendation uses the mean and standard deviation metrics to choose which news articles to recommend. The evaluation was done offline with results from Yahoo! data. They achieved a 12.5% click prediction increase.

3.3 Summary

In this section we presented Apache Kafka and other technologies useful to build our recommender system. We also presented a brief discussion on cloud solutions. We then described some solutions that used decision trees, random forests and the multi-armed bandit. We saw the potential of building a recommender system with those algorithms.
Chapter 4

Solution

This chapter describes the solution developed for the near real-time Recommender System. We decided to separate this chapter into five sections. First we start with an overview of the project, where we give a brief description of the four main components involved. The next four sections will dive into the details of those four components. Section 4.2.1 describes the organization of our source of data, the bank’s database. Section 4.2.2 describes the Apache Kafka implementation details and the challenges we faced during the implementation. Section 4.2.3 describes the database we chose to store the data that is useful for our Recommender System, which is detailed in section 4.3.

4.1 Overview

In this section, we will give an overview of the project’s components and how they interact with each other. In figure 4.1 we can see the main four components. The arrows represent the flow of data. From left to right, the starting point is the bank’s database, our main source of data. It all begins with the data that the bank generates, whether it is a newly created account, an update to an existing account, an electric bill paid or a credit card transaction, all that data is stored in the bank. We want to consume that data as soon as it arrives at the bank’s database. Not only that but we want to process and organize the data in a way that is useful for us to use. For that job, we decided to use Apache Kafka. It is responsible for consuming the bank’s data and processing it. Even though Kafka can be used to store data, we use it more as a transient data store and processing platform. The next step is for data to reach our Recommender System. For that, we need a database where data will be permanently stored for our Recommender System to consume. We chose the NoSQL database MongoDB. The main reason we chose it is because our data will not have a defined structure, as we will store data for our machine learning algorithm to use it. Every new feature we add, or change to our training dataset, the structure will vary during the continuous development of the Recommender System. Having a flexible database like MongoDB - which does not need a predefined data schema - makes the machine learning development process much more manageable. The last component is the Recommender System itself. It will consume data from the MongoDB database and at the same time, it will also produce data to the database. The main workflow when testing or improving the recommender system algorithm will be to consume data from the database, pre-process that
4.2 Infrastructure

In this section we will start by describing the tables in the bank’s database that are relevant to our project. We then dive into the details of the Apache Kafka architecture, describing the several components and how they interact. We follow with a description of the MongoDB. Finally we describe the data flow in our infrastructure.

4.2.1 Source Database

As we previously stated, from the infrastructure point of view, it all starts with the bank generating data. Since that data is a consequence of the client’s actions, it all starts with the client, the moment the client interacts with the bank. For the context of our project, that is the crucial moment for us, the interaction between the client and the bank, even if the interaction is not direct. For example when the client is paying using the bank’s card at the supermarket. We want to take that interaction data and transform it into information. In other words, we want to take the series of interactions the client has with the bank and generate information useful for the bank. In our case, we take those interactions data and transform them into information that allows us to know which products are relevant to which clients, at any given time. All those interactions are stored in the bank’s database.

For the context of our project only some of those tables are relevant:

**Entity** - The Entities table contains information about the physical person. For example their name, birthday or place of work.

**Source-Client** - The Source-Clients table has the purpose of establishing the relation between Entities and Accounts.

**Entity & Source-Client Connection** - This table is used to connect a Source-Client to its Entities.
**Accounts** - There is one Account table per product. Each table contains information about the actual banking account. It can be for example a savings account or a loan account.

**Accounts Transitions** - The Account Transactions tables, which registers the transactions associated with accounts. As before, each type of account has its transactions table.

**Accumulated Monthly Balance** - The accumulated monthly balance table contains the accumulated monthly balance per account. Its a rolling mean of the monthly average balance per year.

To better understand the concept of a client to the bank lets take the following hierarchy into consideration. An entity, a client and an account. An account can only belong to a client. A client can have several accounts. An entity can belong to several clients. A client can contain many entities. To not further complicate things, when talking about the aforementioned client - not the physical client - we will refer to it as source-client.

As we mentioned, all the sources come from a database, and that database has inserts and updates. So to consume those data sources we need an application that can be capable of managing these two aspects. It needs to be capable of detecting both inserts and updates to our data sources.

At this point, we needed to evaluate if we wanted all future applications to consume directly from the bank’s database, or if we wanted to only have one application responsible for that job. One advantage of having just one application consuming the bank’s database was the reduction of its load. Another advantage was that if some reprocess was made to that data, all the consumers in the pipeline could take advantage of that, and would not have to reprocess that information.

### 4.2.2 Apache Kafka

Apache Kafka is an application that acts as a distributed log and follows the publisher-subscriber pattern. So we can have a consumer that is receiving data from the bank’s database and have multiple applications consuming that information. We could have for example an external application consuming the bank’s database events and publishing them into Kafka. With all that in mind we will dive into the details of Kafka and how we used it.

**Apache Kafka**

Kafka makes it possible for us to receive events in real time. They can be processed and manipulated as soon as they arrive. Inside Kafka each event is a message, where each message is published to a topic. In our case, these events are table inserts and updates to the bank’s database. Each table’s messages are published to a topic specifically for that table. For example, when a client uses their debit card in a supermarket, a message containing information like transaction value, terminal location, and date, is published into Kafka to the debit card topic.

**Kafka Connect**

As we need to mirror the databases events to Kafka, we needed an application that, for example, would fetch the bank’s database periodically and would publish the new messages to
Kafka. This is a very common use case for Kafka, and that is why the team behind Apache Kafka developed a framework specifically for these particular cases, called Kafka Connect. With it, we can build an application that continually polls from some database source into Kafka. The same way we can do that, we can also use Kafka Connect framework to build an application that consumes from Kafka and publishes its topics to a database. It can work as a sync between databases. Since we have a source database and a sink database, it fits our needs perfectly. Kafka Connect is launched as an application and is divided into connectors, which are processes managed by Kafka Connect itself. Here, we have two types of connectors, source connectors, and sink connectors. The source connectors are the ones reading the bank’s database and publishing the events to Kafka. Each connector is responsible for a table and publishes in only one topic. There is a connector per table. The sink connectors are the ones subscribed to a topic and are publishing their messages into MongoDB. We can see this in figure 4.2. Each source connector reads a table, publishes the table events to the respective Kafka topic. Each sink connector reads the incoming messages arriving at Kafka and publishes them to the respective MongoDB collection.

Most of the critical and time-consuming challenges that arose were due to some of the design choices on the bank’s database and because it is a IBM DB2 database.

One of our requirements was to support inserts and updates. Which means that we want to receive an event in Kafka when an insert or an update is made to a table in the bank’s database. For that, Kafka Connect requires those tables to have an auto-increment and a timestamp column, which they do not have. That was a huge setback, but we decided to focus on trying to solve the lack of auto-increment and timestamp columns in the bank’s database, since the alternative seemed to develop a solution similar to Kafka Connect itself. We started by analyzing the tables we needed. Not all tables had updates, some just had inserts. We noticed that for the first case, the tables that had updates, there was a corresponding history table. That table had three useful columns, one with the date, another with the time and another one with an operation ID (unique per day). We decided to create a view which created two columns. The first was the combination of those three and was structured as a timestamp. The second was simply the row number, by invoking the \( RRN \) function. With this, the Kafka Connect finally worked.

It is possible to have several Kafka Connect instances in several machines. This allows Kafka to distribute the work of the connectors through the machines, increasing the performance of the system. In case of a machine fault, Kafka can distribute the tasks of that machine through the other ones. This process is called rebalance. We tried to test this setup, but we could not get it to work. The several Kafka Connect instances would not stabilize, they were constantly rebalancing.

Kafka REST

The consumption and production of messages in Kafka is made using an API, which is managed by a component called Kafka REST. By building an application, we can use the API to produce and consume messages. All the applications that consume or produce messages to

Kafka use this API. The applications built using Kafka Connect do just that. It is also useful to access the internal state of the topics. We used it during the development phase to monitor the cluster. It is handy to confirm that topics are created when we expect them to be or to reset a topic in order for Kafka Connect to reprocess a particular source.

**Schema Registry**

Every message is published using Avro (similar to JSON, but allows schemas). When a connector creates a topic, it needs to store its message structure somewhere - the avro schema - in order for consumers to know how to consume that particular topic. For that, we needed a way to register the schema somewhere and allow consumers to query it when we want to consume a certain topic. That is the job for a component designed to do just that, the Schema Registry. It is responsible for managing the topic schemas. This includes adding a schema, deleting a schema and reading a schema. This facilitates the process of adding a consumer to a particular topic, we only need to point the schema registry location, and the consumer will fetch the schema so it can read the topic.

**Kafka Streams**

Kafka Streams is a library that allows to produce and consume messages from Kafka. We can build applications - for example in Java - that allow us to consume a topic, in real time, process it, and publish the newly generated data to another topic. For example we can consume the accounts transactions topic and calculate the transactions categorization on the fly, thus having a topic with the transactions categorized ready for any consumer to read in real time.

We developed several Kafka Streams applications. The main idea for these applications is to reduce the pre-processing from applications that want to use the data stored in Kafka, and thus facilitating its integration with the Kafka pipeline. Using the monthly average balance example, each application that wants to use the transactions categorizations can simply consume the topic generated by the Kafka Streams application.

The development of Kafka Streams applications will be a continuous process, focused on better improving the integration and quality of the integration in the Kafka pipeline. For the context of our project, the Streams applications developed focused on facilitating the integration and performance of our Recommender System.

Every Streams application can be running on any machine that supports Java and has access to the machines where the other components are running.

**Apache Zookeeper**

Each component described is a java application. The communication between the applications is done using TCP. There can be many instances of each component. There are two reasons for that. The first is useful to increase the performance. For example, if there is only one subscriber for a topic which the production rate is higher than the consuming rate, and we want to increase the consuming rate one can add a consumer for the same topic to try to balance things out. The second reason is redundancy. If one consumer dies, we can have $n$ backup ones.

---

2https://avro.apache.org/
3https://www.json.org/
Figure 4.2: Kafka Connect in more detail. On the left we have $n$ tables which correspond to $n$ source connectors. The connectors are populating the respective Kafka topics. On the right we have $n$ sink connectors consuming those topics and inserting them into their respective MongoDB collections to take its job.

The last component, Apache Zookeeper, is the component responsible for precisely that, to manage the whole ecosystem. It has many responsibilities, like the situation mentioned above where it needs to detect that a node has died and needed to redirect its job to another node. If a component is replicated it will need a leader election, that process is managed by the zookeeper. The zookeeper itself can be replicated. Another responsibility is to manage clients access to resources, for example, to indicate to a Kafka Connect application, that is trying to connect to Apache Kafka, its corresponding location.

All these components are part of a solution distributed by Confluent[^1]. They offer an enterprise solution for Apache Kafka that includes all these components. One of the advantages of this bundle is that it comes with several pre-built Kafka Connect applications for well-known databases like MongoDB, mySQL or MicrosoftSQL. Which facilitate the process of integrating Kafka with a database, as it is in our case.

**Kafka Management**

During the development of this project, we built a bash script to facilitate the management of the Kafka infrastructure. The script started as a way to automate the deployment process, but grew to provide an easier access to the context of the deployment. Here are some of the most relevant commands:

- The script allows to start, stop and destroy the project. The start command initiates the components and likewise, the stop command stops the components. The destroy command is similar to the stop command but it resets the internal state and all the data. The start and stop commands can be used to start/stop all the components or even just a subset.

- There are a subset of commands focused on the management of the Kafka Connect. This includes starting and stopping particular connectors. Also, it is possible to check the status of the connectors, such as how many messages each connector as produced or consumed. It is also possible to update the configuration of a connector.

[^1]: https://www.confluent.io/
- The last set of commands is focused on easily accessing the log files. It allows us to easily `tail` or `tac` the log files of each component.

4.2.3 MongoDB

At this moment of the infrastructure, we have the components working together to bring data from the bank to Kafka. The next part is to store the data in a way that is easy for our Recommender System to use. We chose MongoDB as it is a robust and widely adopted NoSQL solution [56].

The MongoDB database’s main purpose is to help the Recommender System, so that it can read data directly from it. There are two ways we write to MongoDB. The first is directly from Kafka, using a connector for each table topic that is publishing the messages to MongoDB. This connector is Kafka Connect MongoDB [7], a connector developed by the community using the Kafka Connect framework. The second is from the Recommender System, during the training of the model phase. The several phases of preprocessing are stored in MongoDB, including the final dataset that is used to train the model and to fetch the clients features upon a recommendation request.

4.2.4 Data flow

The events that are propagated throughout our infrastructure are a consequence of clients actions. To better understand how and in what way they interact with our infrastructure we can follow figure 4.3. It shows the data flow through the infrastructure when a client makes an action - like a purchase with a debit card or an acquisition of a product.

The data flow starts when a client makes an action. That action can be either an account manipulation or the usage of that account. An account manipulation can be either a creation of an account, a modification to an account or the deletion of an account.

The usage of an account has much more cases, which include usage of a credit card, deposit to a savings account or loan payment. There are some cases in which the bank starts the action, like for example when changing the settings of an account.

After the client’s action, the bank receives that information and stores it on the database. Each action is stored in a different table depending on its category. For example, the acquisition of a deposit account is stored in a table, and the transactions associated with that account are stored in another.

The source connectors of the Kafka Connect application that is reading the table in which the client action was stored reads the newly arrived data and publishes it to the respective topic.

Kafka receives the message and stores it internally. Kafka does not inform the consumers that there is a new message. They need to be continually asking Kafka if there are new messages.

Some topics might need to be processed before going to the database. For that job, we use the Kafka Streams applications. There are multiple Streams applications, each responsible for a topic. The one that is processing the topic where a new message arrives will read it and uses it to

---

5http://man7.org/linux/man-pages/man1/tail.1.html
6https://linux.die.net/man/1/tac
7https://github.com/hpgrahsl/kafka-connect-mongodb

25
do some pre-processing. After the processing, it will then publish a newly generated message to Kafka. For example, there can be a streams application responsible for calculating the monthly balance of a deposit account.

The sinks connectors of the Kafka Connect application, detect that a new message was published. They consume it and insert it in MongoDB.

A new message is inserted in MongoDB in the respective collection and is available for our recommender system to consume it.

### 4.3 Recommender System

The Recommender System is the component responsible for trying to know what product is a client more likely to acquire on a given moment. For that we decided to create an algorithm that would learn from the history of client data and learn to model product acquisitions. The model uses a gradient boosting technique to be able to recommend products that clients we more likely to acquire.

This approach requires clients to have a couple of months at least of interactions with the bank, otherwise, the algorithm will not have much or nothing to learn from. That is why we decided to use another strategy for the cases where client data is insufficient for us to learn to give a meaningful recommendation. In this cases, where data is insufficient, we will use a Multi Armed Bandit algorithm to decide which recommendations will be shown. This approach consists of a reinforcement strategy in which the algorithm tries to find which advertisement is more likely to be clicked based on the times it showed them and the number of times clients clicked on them.

The next two subsections will detail the two approaches we implemented. Subsection [4.3.1](#) will detail the first approach, which uses a gradient boosting model to learn from the client’s historical data. Subsection [4.3.2](#) we detail the second approach which uses the Multi Armed Bandit algorithm.
Bandit algorithm, where clicks are the focus of the recommendations.

4.3.1 Gradient Boosting model

As previously stated this approach focuses on creating a gradient boosting model that learns from the client’s history. One of the reasons for choosing Gradient Boosting for our solution is that they come from Decision Trees. Which unlike Gradient Boosting, are highly interpretable\cite{55}. Interpretability is a desired factor for the bank, so it can have better insight into clients product acquisitions. Since Gradient Boosting achieves better results but lacks interpretability, we can still train Decision Trees in order to get an interpretable model. Even though Gradient Boosting models are not interpretable, in the sense that we do not have a single tree to see the decision points, we can see the most important features using other approaches. The main intuition behind this idea is that from all the decision trees built, the features that appear more on the top of the trees are more likely to be the ones that have more discriminatory decision power. The development scenario is depicted in figure 4.5 following this steps:

1. Read the MongoDB database.
2. Do the preprocessing necessary to create our main dataset.
3. Choose the model to be trained on that dataset.
4. Train the model and test the model.
5. Here we evaluate which one of the four steps we want to go to.
   (a) If we want to alter the collections, we read from the database and continue the development steps from step 2.
   (b) Alternatively, if we want to change something on the preprocessing phase like generating a new feature we had not generated before.
   (c) Another option is to change the model we are using and test with a different one.
   (d) The last option is to tweak the model parameters in order to try to improve the test results.

Let us revisit the entity, source client, account relationship. Take the example from figure 4.4. We have two persons, Jane and John. Each one of them is represented in the bank as an

![Figure 4.4](image)

**Figure 4.4:** Entity, source client and account relationship example
entity. In this example, the entity Jane belongs to Client A that has a Credit Card Account. The entities Jane and John belong to Client B which has two accounts, a Loan Account, and a Deposit Account. Now if we take a moment to analyze the creation of Client B, we can see that both persons might have influenced the shared client. So when we are recommending products, or better, when we are building the recommended models, we need to take into account these situations.

Many variables influence the acquisition of a product, from a conversation with a friend that led us to create a savings account, to a car advertisement we saw on the television that influenced us to get a loan. However, here we can only focus on the variables we can get, which are the interactions from the client and the bank.

The same way that it is not trivial to understand which entities of a given client influenced that account acquisition, it is also not trivial to decide to whom should the recommendation be shown. Imagine the previous scenario. If we train our algorithm to focus on the clients, we expect him to understand what products does Client B need based on the features of both entities combined. We also expect the algorithm to understand what products does Client A need based on Jane’s entity features alone. However, maybe Client A is also influenced by John.

In our solution, the models are trained with a focus on the source client and entity. The dataset has an entry per entity. So product acquisitions will appear more than once. Following the same example, we have a source client (Client B) that has two entities (Jane and John) and an account (Credit Card Account). The acquisition of that account will appear two times. The first entry will contain the personal features - date of birth, place of residence, and so forth - of Jane and the second entry will contain the personal features of John. Both entries will also contain the combination of the financial features - average balance, standard deviation, and so forth - of the two entities. So in each entry, the personal features will be different but the financial features will be the same. The last set of features are the products acquired by the clients.

The decision to whom should the recommendation be shown will also be focused on the source client and entity. From the previous example, if Jane accessed the internet banking using Client A, she would see recommendations tailored to Jane and Client A combined. If she accessed using Client B, she would see recommendations tailored to Jane and Client B combined. Likewise, when John accesses the Internet Banking, he will be recommendations tailored to John and Client B combined.

The decision of the product to recommend to a client is made using a combination of all the models. There is a list of five products generated from the five models with the highest confidence for the prediction. So we invoke all the models, In case of a tie, a random product is selected.

4.3.1.1 **Build the training dataset**

Steps 1 and 2 in figure 4.5 represent the building of our training dataset, which is critical for the performance of our Recommender System. The steps to build our training dataset are the following:

1. The first step is to read the database. The first dataset we built had personal information
about the clients. In order to do that we had to join the entities table with the source-clients table. The join was done using a third table that contains the connections between entities and source-clients. The personal information of the clients include details like place of residence or date of birth.

2. After the personal details of clients, the next step is to add information about which products each client has. There is a table in the bank’s database that contains the information about products acquisitions. We read its corresponding MongoDB collection and apply the necessary processing to clean it. We now have a dataset of product acquisitions per source-client.

3. The next step is to take the product acquisitions dataset and transform it into having the accumulated history of products. In other words, each entry has now the information of all the previous products that the client already had. So when we look at an entry of a product acquisition, we can see all the products that the client had when he made that acquisition. This dataset is focused on being a snapshot of the product acquisitions of a given client. After this dataset, the next step is to combine the first dataset with this. We combine the two using the source-client as a key. Since the number of entries for a source-client is the number of entities the source-client has, this third dataset will have an entry for the same product acquisition for each entity of a source client.

4. At this point, we have to add features that are calculated based on the financial state of the client. We read the accumulated average balance per month collection, which has the accumulated mean for the year and we convert it to the average balance per month. From that, we generate features like average balance in the last six month, standard deviation over the last six month or the percentage of increase/decrease in the last 12 months. The last step is to add these features to the previous dataset so that for each product acquisition we have this features for the date of the acquisition

Table 4.1 shows an example of the structure of the training dataset. Each row is identified with the ID of the client. There is an entry per product acquisition. The previously acquired products are also represented. As we can see in the table there are two entries for client 1. The first shows the acquisition of Product 1 on October 23th of 2015, the second shows the acquisition of Product 2 about one year later. In the second entry, we can see that the first product is also represented. This way the algorithm can learn which products the client had when acquired the new one. Each model will only learn to recommend one product. So only one column will be the target column, the other products will be used as features.

For privacy reasons the name of the products and the features will be changed to numbers.

Table 4.1: Example of the training dataset

<table>
<thead>
<tr>
<th>Client ID</th>
<th>Date</th>
<th>Feature 1</th>
<th>Feature 2</th>
<th>...</th>
<th>Feature n</th>
<th>Product 1</th>
<th>Product 2</th>
<th>...</th>
<th>Product n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23/10/2015</td>
<td>4</td>
<td>1 032</td>
<td>...</td>
<td>10/10/1960</td>
<td>yes</td>
<td>no</td>
<td>...</td>
<td>no</td>
</tr>
<tr>
<td>1</td>
<td>02/11/2016</td>
<td>4</td>
<td>3 119</td>
<td>...</td>
<td>10/10/1960</td>
<td>yes</td>
<td>yes</td>
<td>...</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>50.000</td>
<td>11/02/2018</td>
<td>2</td>
<td>10 118</td>
<td>...</td>
<td>1/02/1990</td>
<td>yes</td>
<td>yes</td>
<td>...</td>
<td>no</td>
</tr>
</tbody>
</table>

29
The final dataset has 27,074 entries, corresponding to 20,956 entities, 18,558 source clients and 21,912 accounts. The number of entities is higher than the number of source clients because source clients can contain more than one entity. The number of accounts is higher than the number of source clients because one source client can have multiple accounts.

**Products**

Table 4.2 describes the distribution of product acquisitions. The first column indicates the name of the product. The second column indicates the total number of entries on the dataset. The third column indicates the number of entries that are not relative to the acquisition of that product. The forth column indicates the number of entries that are relative to the acquisition of that product. The last column indicates the percentage of entries that are relative to the acquisition of that product in relation with the total number of entries.

We can see the 17 products present in the dataset. From those products we decided to focus on the ones that had more than 1% of frequency. Which left us with 8 products.

**Features**

We used three categories of features. The first are personal details features, which are relative to the entity (physical client). For example, date of birth, place of residency, level of education, and more. The second category of features are financial details features, which are relative to the accounts of the source client. For example the average balance of the deposit accounts in the last 12 months, or the percentage of increase/decrease of monthly average salary between the last two periods of three months. The last category of features are the products themselves, that is, if the client has or not that product. Here we use all the products, even those with less than 1% of occurrences. This are examples of the features we use. For privacy reasons the

![Figure 4.5: Development cycle for improving the supervised learning algorithm](image-url)
name of the features will be omitted and simply numerated. We use a total of 992 features, being 966 categorical features and 26 numerical.

4.3.1.2 Feature Transformation

There are two types of features datatypes in our dataset, numerical and categorical. Numerical features are continuous real numbers, from features like age or average balance in the past two months. Categorical features are label features, for example place of birth or employer company. The numerical features are ready to be fed to our model, but the categorical features are not, they need to be transformed in order for our model to use them.

To clean the categorical features we applied text cleaning techniques to the text data:

1. We applied One Hot Encoding (OHE) to this features. With OHE, a new column is created for each value of the feature, for example the sex feature, three columns are created, $SEX_F$, $SEX_M$ and $SEX_O$ column. For each data point, a 0 or 1 will be attributed to the value of each columns. For example, a client whose sex feature was M would have $SEX_F = 0$, $SEX_M = 1$ and $SEX_O = 0$.

2. To reduce the number of features generated by the OHE we performed some data cleaning:

   (a) Removed trailing spaces.

   (b) Converted the entries to uppercase.

   (c) Transformed accented characters into simple characters.

3. We also simplified some features in order to improve the model’s performance, for example if the employer company feature contained the word bank, we converted it to bank.

Table 4.2: Frequency of products in the Dataset

<table>
<thead>
<tr>
<th>Product Name</th>
<th>Total</th>
<th># No</th>
<th># Yes</th>
<th>% Yes of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product 0</td>
<td>27074</td>
<td>1016</td>
<td>26058</td>
<td>96.24732</td>
</tr>
<tr>
<td>Product 1</td>
<td>27074</td>
<td>6384</td>
<td>20690</td>
<td>76.42018</td>
</tr>
<tr>
<td>Product 2</td>
<td>27074</td>
<td>12819</td>
<td>14255</td>
<td>52.65199</td>
</tr>
<tr>
<td>Product 3</td>
<td>27074</td>
<td>21191</td>
<td>5883</td>
<td>21.72933</td>
</tr>
<tr>
<td>Product 4</td>
<td>27074</td>
<td>22100</td>
<td>4974</td>
<td>18.37187</td>
</tr>
<tr>
<td>Product 5</td>
<td>27074</td>
<td>25446</td>
<td>1628</td>
<td>6.01315</td>
</tr>
<tr>
<td>Product 6</td>
<td>27074</td>
<td>25499</td>
<td>1575</td>
<td>5.81739</td>
</tr>
<tr>
<td>Product 7</td>
<td>27074</td>
<td>25641</td>
<td>1433</td>
<td>5.29290</td>
</tr>
<tr>
<td>Product 8</td>
<td>27074</td>
<td>26798</td>
<td>276</td>
<td>1.01943</td>
</tr>
<tr>
<td>Product 9</td>
<td>27074</td>
<td>26936</td>
<td>138</td>
<td>0.50971</td>
</tr>
<tr>
<td>Product 10</td>
<td>27074</td>
<td>26941</td>
<td>133</td>
<td>0.49125</td>
</tr>
<tr>
<td>Product 11</td>
<td>27074</td>
<td>26953</td>
<td>121</td>
<td>0.44692</td>
</tr>
<tr>
<td>Product 12</td>
<td>27074</td>
<td>26993</td>
<td>81</td>
<td>0.29918</td>
</tr>
<tr>
<td>Product 13</td>
<td>27074</td>
<td>26999</td>
<td>75</td>
<td>0.27702</td>
</tr>
<tr>
<td>Product 14</td>
<td>27074</td>
<td>27038</td>
<td>36</td>
<td>0.13297</td>
</tr>
<tr>
<td>Product 15</td>
<td>27074</td>
<td>27053</td>
<td>20</td>
<td>0.07757</td>
</tr>
<tr>
<td>Product 16</td>
<td>27074</td>
<td>27058</td>
<td>16</td>
<td>0.05910</td>
</tr>
<tr>
<td>Product 17</td>
<td>27074</td>
<td>27069</td>
<td>5</td>
<td>0.01847</td>
</tr>
<tr>
<td>Product 18</td>
<td>27074</td>
<td>27073</td>
<td>1</td>
<td>0.00004</td>
</tr>
</tbody>
</table>
4. Finally we removed the features that had less than 5 occurrences.

4.3.1.3 Feature Selection

The feature transformation, more precisely the One Hot Encoding created almost a thousand new features, so we performed a feature selection process in order to improve the model efficiency and reduce the chance of overfitting [37]. The feature selection was done using a technique called Recursive Feature Elimination with Cross Validation (RFECV). It consists in trying different combinations of features and testing the chosen feature subset using Cross Validation [54]. The subset of features chosen is the one with the best Cross Validation score.

4.3.1.4 Sampling

In our dataset we can see that the product acquisitions are very imbalanced. In other words, for most products, the number of clients that have acquired that product is much lower than the number of clients that have not acquired that product. In this cases, it is common to use techniques like oversampling or undersampling [49].

• Oversampling consists on replicating some of the datapoints that belong to the least occurring class until they match the number of datapoints that belong to the most occurring class [49]. It basically means to duplicate some datapoints until both classes have almost the same number of occurrences. When doing oversampling is critical to do it only after splitting the dataset into the training and testing sets, and applying it to the training set. Otherwise we duplicated entries can ”leak” to the testing set, and cause overfitting.

• Undersampling consists in removing datapoints of the most frequent class, thus balancing the datapoints for each class. When doing undersampling we are trying to increase the sensitivity of the model [49].

4.3.1.5 Model Training

During the training process, we performed a grid search to optimize the models’s parameters and cross-validation to evaluate our model. We used 10 fold cross-validation, as it is a commonly used fold [3]. Cross-validation consists in training and testing the model in different subsets of the dataset. It generally follows this steps:

1. Split the dataset into  \( k \) groups.

2. For each group:

   (a) Use the group as the testing dataset.

   (b) Train the algorithm using the remain groups as one dataset.

   (c) Evaluate the model using the test dataset.

   (d) Save the evaluation results.

3. Aggregate the evaluation results.
4.3.1.6 Gradient Boost model in action

When using the Gradient Boost model, the Recommender System behaves as figured in Figure 4.6.

It starts with a login of a user in the internet banking application, which triggers a recommendation request to the application middleware. The middleware will then request our Recommender System for a recommendation for that client. That request arrives through the REST API of our Recommender System, which will invoke our XGBoost models. Each model will return a probability on recommending or not a product to the client, the probabilities are sorted and a list of five products is send to the application middleware, which will choose to display the products according to which device is the client accessing from. Products that the client has are not shown. Periodically the models are retrained and updated. The tests described in chapter 5 are done to ensure only models that have better scores, and thus are likely to perform well, are deployed.

4.3.2 Multi Armed Bandit

As previously stated, the approach mentioned above, where we learn from the clients’ history with the bank, has an obvious downside known as the cold start problem [20]. This happens when we do not have enough information on new clients, which have few or no interactions with the bank, and thus can not give a confident product recommendation. To overcome this problem, we decided to take a different approach with new clients.

We decided to use an algorithm that tries to solve the Multi Armed Bandit. In which there are n arms to be pulled, each with an unknown probability of giving a reward. The bandit has to decide which arm to pull in order to maximize the expected reward.

The main idea here is to maximize the clicks on the advertisement showed to the client. This is done by displaying with a higher probability the advertisement that has had the most clicks so far and with a lower probability a random advertisement. This way we have two phases. The first is the exploitation phase, where we show the product we think has the higher chance of

Figure 4.6: XGBoost Recommender System architecture
Grabbing the client’s attention and thus generating a click. The second is the exploration phase, where we try a different product to see if it achieves better results.

In a more formal way, the algorithm used to implement the Multi Armed Bandit is called e-greedy. It is widely used [31], and it is straightforward to implement. It behaves as follows:

- In a series of rounds \( t \in [1,..n] \), the algorithm selects an arm \( k \). Where \( k \) is the number of arms, and an arm is a product to recommend.
- With probability \((1-\epsilon)+(\epsilon/k)\) the algorithm will choose the arm with the highest expected payout. The payout is the click on the advertisement.
- With probability \(\epsilon/k\) the algorithm will choose a random arm.

In practice, a random number \( p \), between 0..1 is generate. When \( p \) is smaller than \( \epsilon \), a random number is once again generated but this time between 1..\( k \). When \( p \) is higher than \( \epsilon \), the expected best arm is chosen.

### 4.3.2.1 Multi Armed Bandit in action

If we follow figure 4.7, it starts with a client logging in the Internet Banking application. It will trigger a request to the middleware responsible for the application, which will direct the request to the API of our Recommender System. The Multi Armed Bandit will choose with probability \((1-\epsilon)+(\epsilon/k)\) the product with the highest expected chance of a click. And with probability \(\epsilon/k\) it will choose a random product. When there is a click, the expected reward of the clicked product is increased.

### 4.4 Summary

In this chapter we detailed the components of the solution of our project. We started with an overview of the four main components involved. The Bank’s database, which is the main
source of data. Apache Kafka, that consumes that data, processes it and sends it to the third component, the NoSQL database, MongoDB. The last component is the Recommender System, divided into two approaches. The first one focused on learning from the clients history of interactions with the bank. The second one focused on dynamically understand what are the products that captivate more interest from clients by analysing the clicks on advertisements.
Chapter 5

Evaluation

In this chapter we will describe the evaluation process that lead us to our solution. There are two main focuses of our evaluation. The first is focused on the infrastructure and is an overall critical evaluation to the project. The second is focused on the Recommender System, where we detail the tests done during the machine learning part of the project.

5.1 Infrastructure

The infrastructure we built allows us to have many applications consuming data from the bank in real time directly from Kafka. Not only that but we can have endless sources of information all producing to Kafka. This way Kafka becomes a centralized source of information. Before we built this infrastructure, in order to create for example a machine learning application that consumed data from the bank, there were a lot of limitations:

1. Developers were required to have an in-depth business logic knowledge to extract relevant information from the database.

2. The only way to retrieve data was through SQL queries.

3. The extraction of data was very time consuming. Developers needed relatively complex queries to retrieve relevant information. The database is behind a restricted VPN with a low bandwidth, so the retrieving of data was very slow as it implied downloading a .csv file. Even simply querying the database is slow.

4. Queries are custom made for each problem at hand. So reusing the result of the queries was not simple.

5. Preprocessing needs to be done after extracting the data from the database thus this process is repeated for every different project.

Our infrastructure mitigates the above stated limitations. Most of the business logic that is needed to retrieve data from the database was already applied to build the Kafka infrastructure, by selecting the useful tables from the database and creating the pre-processing that abstracts some of the business logic.
Kafka allows the consumers to be created in several very used programming languages, like Python, Scala or Java. This has two major advantages. First, it allows a developer to use its language of expertise, which most likely increases the efficiency of the development process and probability of success of the project. The second advantage is that it broadens the possibility of using many types of libraries, removing the limitation caused by the programming languages it supports. For example the H2O\(^1\) machine learning library, which can be used using Java, allows a developer who is an expert in Java but does not know Python, to prefer it instead of scikit-learn\(^2\) for example, which is only available in Python.

Having the data directly in Kafka, on the company’s premises, avoids the overhead of accessing the data through the downloading csv process. Another improvement done was the configuration of a Machine Learning virtual machine, that can be used by every developer as a development environment. The machine is configured with Python and Jupyter\(^3\). Jupyter is a notebook for developing Python code. It can be accessed through the browser or using and Python IDE. By using Jupyter the developer only needs a browser and can start developing right away.

During the development of the Recommender System, we created a Python library. This library contains a set of functions that facilitate the consuming and processing of data from the bank. The library allows three types of actions. The first is to read collections from the MongoDB into pandas\(^4\) dataframes. Those dataframes are like manipulable tables in Python. The second type of actions allows us to create base dataframes, that for the context of analyzing the bank are the foundation for many approaches. For example, revisiting our Recommender System, on the bank’s database there is a table for the entities, another for the source-clients and another for the relation between the two. The entity table contains the personal information about the physical client, like date and place of birth. The client table has the purpose of establishing the connection between the entities and the accounts. So to find which accounts belong to which entities, we need first to read those three tables. Connecting the entity and source-client tables using the relation table. This creates the clients’ snapshot dataframe. This dataframe is the starting point for analyzing the client, with it we can find which accounts each client has and proceed with the rest of the development process. So this dataframe is an example of a dataframe that is crucial for many future machine learning applications. The third type of action that the library functions allow are related to the second type, in the sense that they are used when creating the base dataframes. They are pre-processing functions that allow for example to calculate the age based on two columns - for example in our case date of birth and date of product acquisition.

The pre-processing is being migrated to Kafka Streams applications in order to improve the development of machine learning applications. The idea is to have the pre-processing done in real time, thus reducing the time spent in pre-processing. There are several advantages in having the pre-processing done in real time using Kafka Streams. For example, many applications can reuse it without having to execute the preprocessing individually by themselves. Another advantage

\(^1\)https://www.h2o.ai/
\(^2\)http://scikit-learn.org/
\(^3\)http://jupyter.org/
\(^4\)https://pandas.pydata.org/
is that as soon as we want to train a model it is ready to run, we don’t need to worry about the pre-processing part. On top of that the data is always recent.

Along with the advantages related with machine learning applications, with this infrastructure, we can easily create a consumer application that receives data in real time. It is straightforward to create a consumer application, which can be a consumer and at the same time a producer. This is all possible by using the Kafka APIs through the Kafka Streams library and allows us to centralize the information in Kafka and create communication pipelines between applications.

This infrastructure opens the possibilities for endless applications. Let us follow an example scenario. We can have an application that is monitoring fraud detection, and when it suspects that there is a fraud, it publishes that information to a topic. An application that is responsible for processing that topic and creating an email from every message send that information to the bank’s security team. In the future, we decided to create a new application to create statistics from several alert systems. It reads several alert topics, like the fraud detection topic, but instead of creating an email, it processes those messages in order to extract statistical information. To escalate things further, let us say that the statistics generated by that application are published to another topic, and that is being consumed by an application responsible for displaying those statistics on a webpage, as a series of interactive graphs. To push a little further, the graphs relative to the fraud detection system could allow the security team to flag false positives and send that information to a topic that the fraud detection application was reading and from which it could use that false positive information to improve the fraud detection algorithm.

Until now the only source of information we talked about was the bank’s database, but we can create an application to that consumes the interaction that the client has with the mobile applications in order to, for example, improve the user experience by analyzing those interactions.

With this infrastructure there are two applications there are already in development. The first application is a transactions categorization application that uses the Kafka Streams library to categorize the transactions of Deposit Accounts. For now, it consumes the Deposit Accounts topic and a Categorizations topic. For every message/transaction it checks the operation code of the transaction and finds a match in the Categorization topic. In the future, we want to add some artificial intelligence to the categorization. Due to the modularity of our infrastructure, to improve the categorization, we can develop a parallel application using Kafka Streams that consumes the transactions but categorizes them using machine learning techniques. When we are satisfied with that application, we can easily swap the Kafka streams applications. The second application is a notifications application that is reading a couple of topics and generating notifications accordingly to the topic. Those notifications are sent to an application responsible for propagating them to the client.

5.2 Recommender System

When evaluating a recommender system, a simple way is to analyze whether or not clients are buying the recommended products or whether or not there is an increase in sales consequent to the deployment of the recommender system. That evaluation is possible when the system is
in production, with real time data. But the system is no in production yet, so need to test it offline.

We have two components to evaluate, the XGBoost Model and the Multi-Armed Bandit. The Multi-Armed Bandit was not evaluated, due to the fact that it needs to learns from clicks on advertisements, it only makes sense to evaluate its behavior in a production enviroment. The rest of the evaluation section focuses on the evaluation process to arrive to the XGBoost model.

To do that we will evaluate the results using some known metrics when evaluating recommender systems: Precision, recall and F1 [25].

The precision being

$$\frac{#tp}{#tp + #fp},$$  \hspace{1cm} (5.1)

where \(#tp\) represents the number of true positives and \(#fp\) the number of false positives.

Following the same notation, the recall is expressed as

$$\frac{#tp}{#tp + #fn},$$  \hspace{1cm} (5.2)

where \(#fp\) represents the false negatives.

Finally F1,

$$2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$ \hspace{1cm} (5.3)

The ideal is for the false positives and false negatives values to be low, meaning the algorithm is making good predictions. In this context the positives are the the client having the product and the negatives are the client not having the problem. So false positives for example, are when the algorithm predicts that the client has the product, when in fact they have not.

5.2.1 Overall Comparison

Our first idea for the model to use in our Recommender System was a Decision Tree. The main reasons were because it is a simple to implement model, very interpretable can achieve good results [35]. From the that, Random Forests are a better algorithm, but less interpretable, and XGBoost even better and even less interpretable. Even though we can not see the exact tree in the last two algorithms, we can still access features importance, which is very useful for the bank. So we decided to train the three models, and use and improve the model with a better performance in this initial tests.

Table 5.1 describes the overall results of the evaluation process of the decision tree models. The table contains the tests for the 8 products. In product we have three different columns for the three models. We performed a series of techniques to try to improve each model’s performance, the results can be seen in different rows. The techniques were the following:

1. FT, which stands for Feature Transformation, was the first technique we applied, consisting in transformations to clean the categorical data. It proved to be very effective in improving the results. Section 5.2.4 dives into the details of the Feature Transformation evaluation process.

2. SP, which stands for Sampling, consists in duplicating or reducing a given class in order to
allow the algorithm to better retrieve relevant information from the data. Sampling was also effective, but only in products where the minority class was yes and in product 2, whose classes are balanced. Table 5.1 only shows the best score from the two techniques. Section 5.2.3 dives into the details of the Sampling evaluation process.

3. FS, which stands for Feature Selection, consists in finding which subset of features better improves the performance of the model. It is useful to eliminate features that mislead the algorithm and also to improve the training time efficiency. Section 5.2.4 dives into the details of the Feature Selection evaluation process.

In table 5.1, the + sign represents the combination of two techniques. We can see that for product 0 and 1, the Feature Transformation and Feature selection achieve the best results. For the remaining products we can see that by adding Sampling to those two techniques we achieve the best results.

### 5.2.2 Impact of Feature Transformation

The feature transformation help to greatly improve the performance of the models. Since our categorical features are not ready to be fed to our models, we needed to convert them to numerical features. One solution was to apply Label Encording, in which we take all the cases in a feature and transform them to numbers. The problem is that decision tree based methods in sci-kit learn interpret numerical features as having a numerical relationship [44], so Label Encoding would damage the data and result in a bad model. Another, more appropriate solution is to use OHE. This time, the problem is the huge number of new features it can generate if there are a lot of cases in our features, which there are. For example in the employer company feature. We decided to apply OHE and analyze the results.

When we applied the OHE the dataset grew from 50 features to 17 214 features. A high
number of features creates two problems, it becomes harder to compute and it can lead to overfitting [43].

With this high number of features we could not compute the dataset in feasible time, so the next step was to reduce the number of features. The first approach was to clean this categorical features by using a series of text data cleaning techniques:

- We started with removing the trailing spaces, converting the text to uppercase and removing accentuation from characters. After OHE with we reduced the number of features to 16 233.

- The next step was to better analyze the content of the features before OHE. We decided to try to group features by converting multiple words to one. For example if a client had the employer company feature of Bank A and another had Bank B, they both would be converted to Bank reducing the number of features in 1. After applying OHE we reduced the number of generated features to 12 088.

- 12 088 is still a large number to compute, so another effort was done to reduce the number of features. The last step was to remove entries with frequency lower than 5. With this we reduced the number of features to 992.

With 992 features it was possible to train the algorithm and continue testing different approaches to improve the models’ performance.

5.2.3 Impact of Sampling

Due to our dataset being imbalanced, we decided to apply two sampling techniques in order to improve the performance of our model.

We first tried undersampling hoping that the model became more sensitive to the minority class, thus improving the prediction results. As we can see in table 5.2, undersampling had a positive impact in products whose yes class is the minority. In product 7 we see the greatest improvement of all the products, with an increase of the F1 score of 80.43 %. Product 2 shows a lower improvement, of only 4 %. The undersample removed noise from the majority class datapoints and allow the algorithm to extract more relevant rules that improved the prediction score. In product 0 and product 1, where no is the minority class, we can see that it actually worsens the results. This means that there is relevant information present in the majority class, and removing those entries leads the algorithm to a worst performance.

We then processed to test oversampling to see if increasing the minority class would improve the F1 score. We have to take into account that oversampling can cause overfitting due to the duplication of samples [19], so we used Cross Validation and oversampled only the training data. Looking at table 5.2 we can see that there are improvements only in product 0 and product 1. This can mean two things, either the minority classes, the no datapoints, contained information useful but the algorithm was not sensible to it, or the majority classes contains features that are misleading the algorithm. In the Feature Selection phase, section 5.2.4, we try to understand which one is the case and proceed accordingly.
Table 5.2: Sampling F1 scores. US stands for undersampling and OS for oversampling. Bold scores are the best for each product.

Feature Selection was done using RFECV in order to improve the model’s performance and training efficiency. In table 5.1 we can see the positive impact for feature selection in the models’ performance. In every case the it improved the F1 score.

In figure 5.1 we can see the evolution of feature selection for product 0 and product 7. From figure 5.1a we can see that using more than 50 features does not greatly improve the model for product 0, which leads us to the conclusion that there is not much information the algorithm can take from those extra features. The improvement is only of 1%.

The same does not happen with product 7. In figure 5.1b we can see that the model’s performance has an improvement at around the 100th feature, and continues to improve. There is an improvement of 80% from the 100th feature to the 779th.

The improvement of using feature selection is not huge, but still helps improve the model’s F1 score. Since increasing the number of features did not appear to make a significant difference in the models’ performance, we can run a Grid Search, which is very intensive, using a lower number of features. We can then perform a feature selection again and then run the other tests using the optimal number of features. So the big advantage we took from feature selection was the efficiency in training.
5.3 Summary

In this chapter we described the evaluation of our infrastructure and recommender system. Starting with a critical view of the infrastructure, we detailed the improvements we gain from it, regarding the previous development scenarios. We then proceed to detail the evaluation results of the several techniques used to improve the models. We saw that the XGBoost model performs best. And the sampling techniques were useful when the minority class was the *yes*. 
Chapter 6

Conclusion

A bank can offer its clients a wide variety of banking solutions. Ranging from deposit accounts, to insurances or even car leasing. The wide range of options can be attractive because it can reach the heterogeneity of clients’ needs. With the data the bank has on its clients, we can build a system to find the best solutions for the clients. Here we propose a Recommender System that learns from the client’s data in two ways. The first is from feeding the history of clients interactions with the bank, to an XGBoost algorithm in order to learn to recommend the bank’s products. The second is through the Multi-Armed Bandit (MAB), in which the objective is to maximize the clients’ advertisement click rate. Our Recommender System performed well during the evaluation, with an average F1 score of 89. To support our Recommender System we built an infrastructure using Apache Kafka that is capable of consuming the bank’s database in real time. Also, it allows applications to consume data from Kafka in real time, this way facilitating the integration of applications that want to consume the bank’s database in real time. The infrastructure proved to improve the work flow of integrating applications that consume data from the bank, particularly machine learning ones.

We strongly believe that it can have a huge impact on both the clients and the bank. The clients will have a sort of personal advisor, given them recommendations based on their profile. The bank will benefit from the increase of clients’ product acquisitions.

6.1 Achievements

With this work we were able to develop an infrastructure for our Recommender System. The Recommender System allows the bank to have a machine learning powered recommendation platform, that is capable of making meaningful recommendations.

The infrastructure greatly improves the process of creating applications that use the bank’s database in real-time.

6.2 Future Work

The following tasks are proposed as future work:

- The greatest improvements to our models’ performance were when we applied cleaning
techniques to the dataset, so it would be relevant to improve this. Using for example Natural Language Processing to clean the categorical features.

- There is a lot to explore from the financial features of clients. It would be potentially beneficial to further explore and analyze the clients transactions.

- For now, the two recommendation approaches are working independently. It would be interesting to have the Multi-Armed Bandit using the XGBoost models to decide which product to recommend.
Bibliography


[34] J. Langseth and F. Aref. BI Style Analytics with Spark. 2014.


