Infrastructure and Recommendation System for Banking Products

Vicente Côxo Rocha
Instituto Superior Técnico, Universidade de Lisboa
Advisors: Professor Manuel Cabido Lopes and
Ricardo Jorge Gonalves Portela

***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 a 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

I. Introduction

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? [1].

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 [2]. 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 [3].

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.

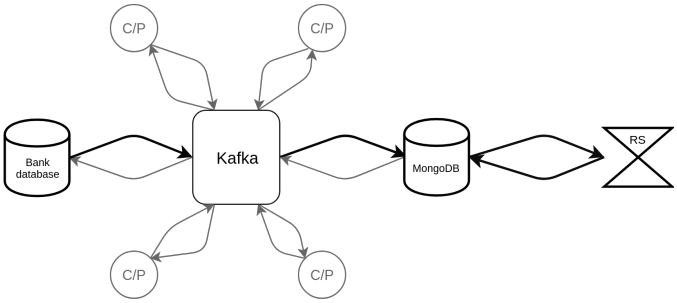

Fig. 1: Project vision. The C/P are consumers or producers of data to Kafka. The bold lines represent the data pipeline we built for this project. The light lines represent the future vision and what we want to support.

Clients want solutions tailored to their specific context. The client today might not be the same as the client tomorrow. How can the bank adapt to their clients' dynamic context?

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.

A. Objectives

We prupose 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.

Our goal for the infrastructure is to achieve what is depicted in figure 1. The bold lines represent what was built for this project, whereas the lighter arrows represent what we want to allow to be possible.

B. Structure

The remaining of the document is organized as follows: Section II 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. Section IV details the solution developed. Section VI details the evaluation of the Recommender System and critically evaluates the infrastructure we built. Section VII states the conclusion and the contibutions and future work of this project.

II. 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 III we begin by analyzing the machine learning solutions. In section II-A we analyze the technologies that can be used to support a recommender system, and take it to production.

A. 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.

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 [4].

III. 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 cite. 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.

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.

A. Tree based solutions

Three 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 [5]. The first tree based approach recommender systems started to use were Decision trees (DT), which are particularly efficient and easy to interpret [6].

In the domain of movie recommendation systems Li and Yamada [7] 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. Using the DT to make recommendations they achieved good results with a Mean Absolute Error of around 0.7. Utku et al. [8] also decided to use DTs for movie recommendations, but instead of explicit feedback, they used an implicit feedback strategy. For testing, they setup used the help of 200 student volunteers. The results where promising as they achieved a recall, precision, and accuracy of around 90%.

Following the Decision Tree algorithm, Random Forests appear as an improvement by combining many trees to arrive to a prediction decision. Zhang and Min [9] purpose 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 [10]. The model proved to outperform other commonly used baseline algorithms. Brommund and Skeppstedt [11] 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. In average the results showed a seven percent decrease of performance comparing to other models.

Gradient Boosting has been developed as an improvement of Random Forests, as it generaly can achieve better results and mitigate the overfitting problem [12]. To the best of our knowledge there are no papers about recommender systems using Gradient Boosting, we will try to show that it ca be used in a Recommender System. analisar com mais paciencia

B. Reiforcement 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 [13] propose a solution for garment-based e-commerce systems. They use a Contextual Multi-Armed Bandit Algorithm to make product recommendations. The evaluation relied on the measuring of the percentage of successful recommendations. They achieved an average prediction rate between 19 and 21 percent. In their paper Mary et al. [14] explore the MAB for online recommendation systems focusing on the cold-start problem. They achieved promising

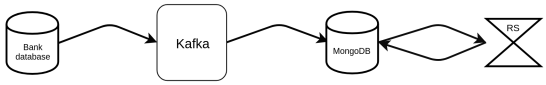

Fig. 2: Overview of the infrastructure

results, outperforming common strategies like greedy or Upper Confidence Bound and mitigating the cold-start problem.

C. Summary

In this section we described some solutions that used tree based methods, and the multi-armed bandit. We saw the potential of building a recommender system with those algorithms. Then we presented Apache Kafka, which is used to build our recommender system.

IV. Solution

A. Overview

In this section, we will give an overview of the project's components and how they interact with each other. In figure 2 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 data on the bank's database. We want to consume, 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. 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. 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 Recommender System uses an XGBoost model and a Multi Armed Bandit to perform recommendations.

Now that we have an overview of the project let us dive into the details of each component.

B. 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:

- The Entities table contains information about the physical person. For example their name, birthday or place of work.
- The Source-Clients table has the purpose of establishing the relation between Entities and Accounts.
- This table is used to connect a Source-Client to its Entities.
- 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.
- The Account Transactions tables, which registers the transactions associated with accounts. As before, each type of account has its transactions table.
- 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.

C. 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.

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.

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.

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 3. 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.

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.

Every message is published using Avro [1] (similar to JSON[2] , 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. Schemas are managed by the Schema Registry.

Kafka Streams is a library that allows to produce and consume messages from Kafka. We can build applications that allow us to consume a topic, in real time, process it, and publish the newly generated data to another topic.

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.

The last component, Apache Zookeeper, is the component responsible for precisely to manage the whole ecosystem. It has many responsibilities. For example detecting 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.

During the development of this project, we built a bash script to facilitate the management of the Kafka infrastructure.

[1]https://avro.apache.org/

[2]https://www.json.org/

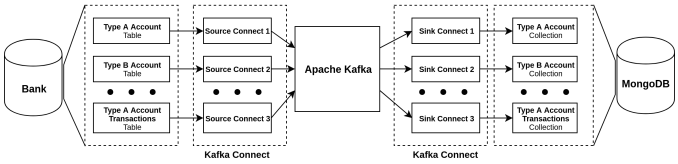

Fig. 3: Kafka Connect in more detail. On the left we have *n* tables which correspond to *n* source connectors. The connectors are populating the respective Kafka topics. On the right we have *n* sink connectors consuming those topics and inserting them into their respective MongoDB collections

The script started as a way to automate the deployment process, but grew to provide an easier access to the context of the deployment.

D. 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 [15].

The MongoDB database's main purpose is to help the Recommender System, so that it can read data directly from it instead of directly from Kafka.

E. 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. 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.

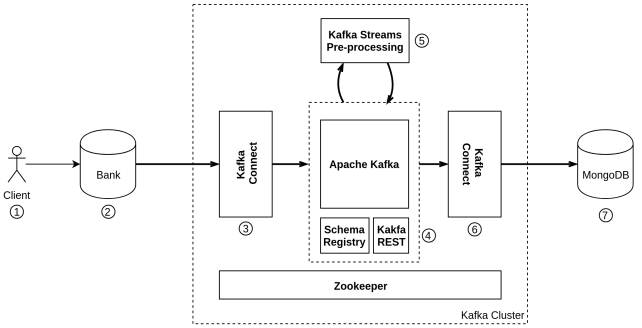

Fig. 4: Infrastructure: Flow of data generated by the client

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 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.

V. 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.

The next two subsections will detail the two approaches we implemented. Subsection V-A will detail the first approach, which uses a gradient boosting model to learn from the client's historical data. Subsection V-B we detail the second approach which uses the Multi Armed Bandit algorithm, where clicks are the focus of the recommendations.

A. 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 [8]. 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.

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. The decision to whom should the recommendation be shown too will also be focused on the source client and entity. 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.

1) Build the training dataset: 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 I 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 I, 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.

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 II 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 name of the features will be omitted and simply numerated. We use a total of 992 features, being 966 categorical features and 26 numerical.

TABLE I: Example of the training dataset

Client ID	Date	Feature 1	Feature 2	...	Feature n	Product 1	Product 2	...	Product n
1	23/10/2015	4	1 032	...	10/10/1960	yes	no	...	no
1	02/11/2016	4	3 119	...	10/10/1960	yes	yes	...	no
...	...	...	...	...	...	...	...	...	...
50.000	11/02/2018	2	10 118	...	1/02/1990	yes	yes	...	no

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 we applied *One Hot Enconding* (OHE) to this features. With OHE, a new column is created for each value of the feature. To reduce the number of features generated by the OHE we performed some data cleaning: Removed trailing spaces, converted the entries to uppercase and transformed accented characters into simple characters. We also simplified some features in order to improve the model's performance. Finally we removed the features that had less than 5 occurrences.

3) Feature Selection: The feature transformation, 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 [16]. The feature selection was done using a technique called Recursive Feature Elimination with Cross Validation (RFECV).

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 [17].

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 [18].

TABLE II: Frequency of products in the Dataset

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

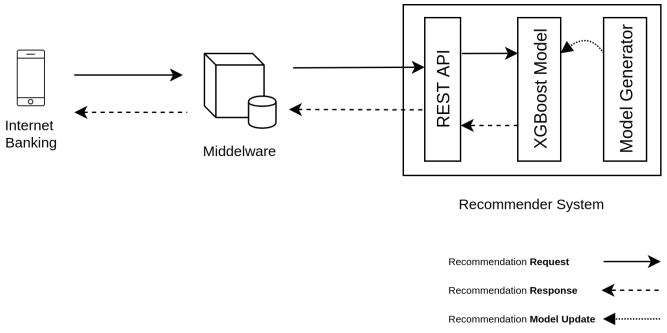

Fig. 5: XGBoost Recommender System architecture

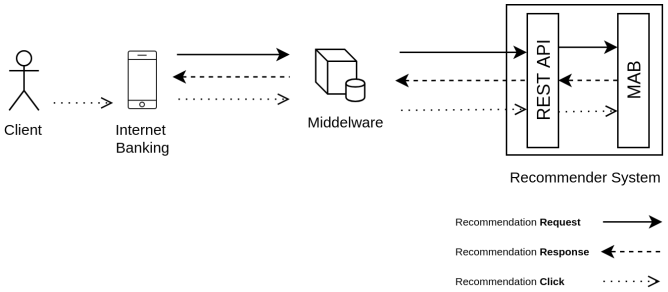

Fig. 6: Multi Armed Bandit Recommender System architecture

6) Gradient Boost model in action: When using the Gradient Boost model, the Recommender System behaves as figured in 5.

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 VI are done to ensure only models that have better scores, and thus are likely to perform well, are deployed.

B. 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 [19].

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 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.

If we follow figure 6. 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 (ϵ/k) it will choose a random product. When there is a click, the expected reward of the clicked product is increased.

C. 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 analyzing the clicks on advertisements.

VI. 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.

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: The only way to retrieve data was through SQL queries; The extraction of data was very time consuming; The database is behind a restricted VPN with a low bandwidh, so the data access is very slow; Queries are custom made for each problem at hand. So reusing the result of the queries was not simple; 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 retreive 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, allowing developers to use its language of expertise

Having the data directly in Kafka, on the company's permises, avoids the overhead of accessing the data through the downloading csv process.

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 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.

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.

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. The second application is a notifications application that reads a couple of topics, generates and sends notifications to the clients.

A. Recommender System

When evaluating a recommender system, a good way to evaluate the performance of the system is by using the Precision, recall and F1 metrics [20].

The precision being

$$\frac{\#tp}{\#tp + \#fp}, \tag{1}$$

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

Following the same notation, the recall is expressed as

$$\frac{\#tp}{\#tp + \#fn}, \tag{2}$$

where $\#fp$ represents the false negatives.

Finally F1,

$$2 \times \frac{precision \times recall}{precision + recall} \tag{3}$$

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 de product, when in fact they have not.

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 [7]. 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 III 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. FT, which stands for Feature Transformation. SP, which stands for Sampling. FS, which stands for Feature Selection. 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.

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 the to numerical features.

When we applied the OHE the dataset grew from 50 features to 17 214 features. A high number of features creates two problems, it becames harder to computate and it can lead to overfitting [21]. With this high number of features we could not compute the dataset in feaseble 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. 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.

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 IV, 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,

	Product 0			Product 1			Product 2			Product 3		
	DT	*RF*	*XGB*	*DT*	*RF*	*XGB*	*DT*	*RF*	*XGB*	*DT*	*RF*	*XGB*
Raw	0.5263	0.5323	0.697	0.4071	0.4313	0.5033	0.4243	0.4923	0.533	0.3644	0.4075	0.4542
FT	0.7783	0.7689	0.981	0.7638	0.793	0.8287	0.6928	0.7639	0.7764	0.4706	0.475	0.7419
$FT + SP$	0.7692	0.7221	0.9011	0.7031	0.7612	0.8146	0.7677	0.7474	0.8112	0.5833	0.5839	0.9101
$FT + FS$	0.829	0.7732	**0.9912**	0.8931	0.801	**0.8274**	0.7211	0.7991	0.7862	0.5983	0.5733	0.7778
$FT + FS + SP$	0.7881	0.8234	0.9547	0.7331	0.8021	0.8181	0.7833	0.8018	**0.8512**	0.6231	0.5981	**0.9224**

	Product 4			Product 5			Product 6			Product 7		
	DT	*RF*	*XGB*	*DT*	*RF*	*XGB*	*DT*	*RF*	*XGB*	*DT*	*RF*	*XGB*
Raw	0.3449	0.3389	0.3166	0.3332	0.4189	0.4677	0.1655	0.2693	0.4977	0.1002	0.2022	0.3311
FT	0.5569	0.4772	0.6306	0.7737	0.8234	0.8496	0.6179	0.6254	0.7738	0.4337	0.4476	0.5074
$FT + SP$	0.5963	0.6231	0.7796	0.8034	0.8410	0.916	0.7041	0.7544	0.9122	0.5448	0.5734	0.9155
$FT + FS$	0.5833	0.4813	0.6099	0.7979	0.8338	0.8348	0.6601	0.5088	0.8478	0.4803	0.4978	0.6044
$FT + FS + SP$	0.6031	0.7041	**0.7995**	0.8343	0.8891	**0.9362**	0.7731	0.8504	**0.9181**	0.6122	0.7303	**0.9231**

TABLE III: 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

	Product 0			Product 1		
	DT	*RF*	*XGB*	*DT*	*RF*	*XGB*
FT	0.7783	0.7689	0.981	0.4337	0.4476	0.5074
$FT + US$	0.6914	0.7022	0.7889	0.5448	0.5734	**0.9155**
$FT + OS$	0.7692	0.7221	**0.9011**	0.3339	0.3821	0.4025

TABLE IV: Sampling F1 scores for product 0 and 1. US stands for undersampling and OS for oversampling. Bold scores are the best for each product.

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 [17], so we used Cross Validation and oversampled only the training data. Looking at table IV 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 VI-A, we try to understand which one is the case and proceed accordingly.

Impact of Feature Selection

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

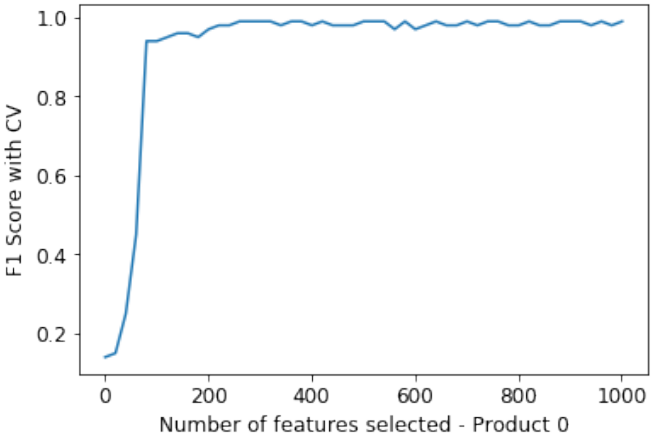

Fig. 7: XGBoost Recommender System architecture

In figure 7 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 8 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.

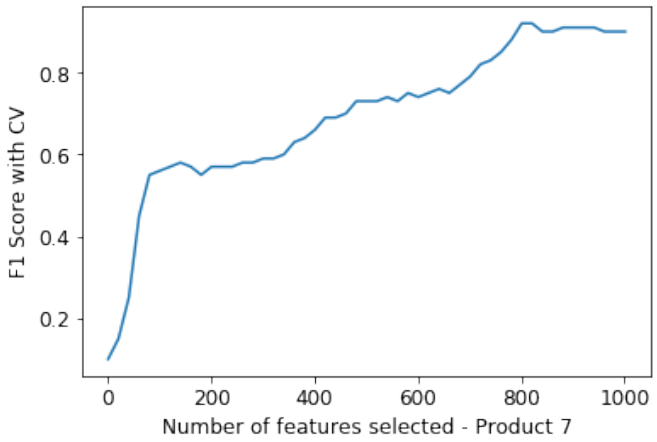

Fig. 8: XGBoost Recommender System architecture

B. Summary

In this section we described the evaluation of our infrastructure and recommender system. Starting with a critical view of the infrastructure. We then proceed to the several techniques used to improve the models.

VII. Conclusion

A bank can offer its clients a wide variety of banking solution. Ranging from deposit accounts, to insurances or even car leasings. 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 built 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 banks products. The second is through the Multi-Armed Bandit (MAB), in which the objective is to maximize the clients advertisement click rate. Our Recomender System performed well during the evaluation, with an average F1 score of from all the product models of 0.89. 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 that 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 products acquisitions.

References

[1] N. H. Erik Brieva, Victoria Yasinetskaya, "Personal financial management (pfm)," strands, 2015.

[2] D. W. Arner, J. Barberis, and B. P. Buckley, "The Evolution of Fintech: A New Post-Crisis Paradigm?" *University of Hong Kong Faculty of Law Research Paper*, vol. 2015/047, pp. 1689–1699, 2015.

[3] C. Issues, "Fintech_reloaded__Traditional_banks_as_digital_ec," pp. 1–27, 2015.

[4] J. Kreps, N. Narkhede, and J. Rao, "Kafka: a Distributed Messaging System for Log Processing," *ACM SIGMOD Workshop on Networking Meets Databases*, p. 6, 2011.

[5] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," 2016.

[6] T. Zhang and V. S. Iyengar, "Recommender Systems Using Linear Classifiers," *Journal of Machine Learning Research*, vol. 2, no. Feb, pp. 313–334, 2002.

[7] Peng Li and S. Yamada, "A movie recommender system based on inductive learning," *IEEE Conference on Cybernetics and Intelligent Systems, 2004.*, vol. 1, pp. 318–323, 2004.

[8] A. Utku, H. Karacan, O. Yldz, and M. Ali Akcayol, "Implementation of a New Recommendation System Based on Decision Tree Using Implicit Relevance Feedback," *Journal of Software*, vol. 10, no. 12, pp. 1367–1374, 2015.

[9] H. R. Zhang and F. Min, "Three-way recommender systems based on random forests," *Knowledge-Based Systems*, vol. 91, no. July, pp. 275–286, 2016.

[10] X. Yang and J. Yao, "Modelling multi-agent three-way decisions with decision-theoretic rough sets," *Fundamenta Informaticae*, vol. 115, no. 2-3, pp. 157–171, 2012.

[11] A. Brommund, "A Hybrid Film Recommender System Based on Random A Hybrid Film Recommender System Based on Random," 2017.

[12] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2017.

[13] R. Stigsson, "Recommender Systems ; Contextual Multi-Armed Bandit Algorithms for the purpose of targeted advertisement within Fredrik Ek," p. 63, 2015.

[14] J. Mary, R. Gaudel, P. Preux, J. Mary, R. Gaudel, P. P. Bandits, and R. S. F. Interna, "Bandits and Recommender Systems," 2016.

[15] Z. Wei-ping, L. Ming-xin, and C. Huan, "Using mongodb to implement textbook management system instead of mysql," pp. 303–305, May 2011.

[16] H. Liu, E. R. Dougherty, J. G. Dy, K. Torkkola, E. Tuv, H. Peng, C. Ding, F. Long, M. Berens, H. Liu, L. Parsons, Z. Zhao, L. Yu, and G. Forman, "Evolving feature selection," *IEEE Intelligent Systems*, vol. 20, no. 6, pp. 64–76, Nov 2005.

[17] B. B. T. H. M. M. D. J. A. Sallam Osman Fageeri, Rohiza Ahmad, *Proceedings of the First International Conference on Advanced Data and Information Engineering (DaEng-2013)*, 1st ed., ser. Lecture Notes in Electrical Engineering 285. Springer-Verlag Singapur, 2014.

[18] S. Arlot and A. Celisse, "A survey of cross-validation procedures for model selection," vol. 4, pp. 40–79, 2009.

[19] C. Z. Felício, K. V. R. Paixão, C. A. Z. Barcelos, and P. Preux, "Multi-Armed Bandits to Recommend for Cold-Start User," pp. 3–6, 2016.

[20] T. Saito and M. Rehmsmeier, "The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets," *PLoS ONE*, vol. 10, no. 3, pp. 1–21, 2015.

[21] N. J. Nilsson, "Introduction to Machine Learning," *Machine Learning*, vol. 56, no. 2, pp. 387–99, 2005.