Recommender Systems for Grocery Retail - A Machine Learning Approach

XAVIER DOS SANTOS SILVA

Outubro de 2020

Instituto Superior de Engenharia do Porto

Recommender Systems for Grocery Retail
[A Machine Learning Approach]

Xavier dos Santos Silva

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Software Engineering

Supervisor: Professor Carlos Ferreira
Co-Supervisor: Eng. Sílvio Macedo

Porto, October 14, 2020

Dedicatory

To my parents, who have supported me like no one else during these journey; to my brother, who has been there since day one; to my grandmother, who has once told me she would walk me to my first day in school, but passed away before. Thank you.

Abstract

Recommender systems are present in our daily activities in different moments, such as when choosing a song to listen to or when doing online shopping. It is an everyday reality for people to have the help of computer systems in order to simplify regular decision activities.

Grocery shopping is an essential part of people's life and a frequent activity. Despite being a common habit, each customer has unique routines, needs and preferences regarding products and brands. This information is valuable for grocery retailers to know their customers better and to improve their marketing and operational activities.

This dissertation aims to apply machine learning algorithms to the development of a recommender system capable of preparing personalized grocery shopping lists. The proposed architecture is designed to allow integration with different grocery retailers and support distinct TensorFlow algorithms.

The process of extracting information from the dataset as features was explored, as well as the tuning of the model hyperparameters, to obtain better results. The recommendation engine is exposed via a distributed software architecture designed to allow retailers to integrate the recommender system with different existing solutions (e.g., websites or mobile applications).

A case study to validate the implemented solution was performed, integrating it with a public dataset provided by Instacart. A comparison study between different machine learning algorithms over the adopted dataset has lead to the choice of the *gradient boosted trees* algorithm.

The solution developed in the case study was compared against two non-machine learning approaches at predicting the last purchase of 360 arbitrary test customers. A pattern mining-based solution and a SQL-based heuristic were used. Different evaluation metrics (namely, the average accuracy, precision, recall, and f1-score) were registered. The way association rules with different strengths were reflected in the predictions of the developed solution was also analyzed.

The gradient boosted trees-based implementation from the case study was capable of outperforming the compared solutions as far as evaluation metrics are concerned, and has shown a higher capability of predicting at least one correct item per customer. Also, it became evident that the strictest association rules were frequently found in the recommendations.

The adopted solution and algorithm have shown promising results and a remarkable capability to provide meaningful predictions to the different customers, evidencing its capability to add value to grocery retail. Nevertheless, there is still potential for further expansion.

Keywords: recommender systems; grocery retail; machine learning; gradient boosted trees

Resumo

Os sistemas de recomendação estão presentes no nosso quotidiano, em momentos como a escolha da música a ouvir ou a preparação de compras *online*. Estamos acostumados a contar com a ajuda de sistemas computacionais para simplificar tarefas habituais que envolvem decisões.

Realizar compras de retalho alimentar é uma parte importante e frequente da nossa vida. Apesar de ser um hábito comum, cada um de nós tem as suas próprias rotinas, necessidades e preferências no que toca a produtos e marcas. Esta informação é valiosa para que os retalhistas alimentares consigam conhecer melhor os seus clientes e melhorar atividades operacionais e de *marketing*.

Esta dissertação tem como objetivo a aplicação de algoritmos de *machine learning* na criação de um sistema de recomendação capaz de preparar listas de compras personalizadas. A arquitetura proposta é desenhada com o objetivo de permitir a integração com diferentes retalhistas e a utilização de diferentes algoritmos em TensorFlow.

O processo de extração de informação na forma de *features* foi explorado, tal como a afinação dos hiperparâmetros do modelo, para obter melhores resultados. O motor de recomendações é exposto através de uma arquitetura de *software* distribuída, com o propósito de permitir que os retalhistas alimentares possam integrar este sistema com diferentes soluções existentes (e.g., *websites* ou aplicações móveis).

Foi realizado um caso de estudo para validar a solução implementada, através da integração da solução com os dados públicos disponibilizados pelo retalhista Instacart. Uma comparação entre a aplicação de diferentes algoritmos de *machine learning* aos dados utilizados, levou à adoção do algoritmo *gradient boosted trees*.

A solução desenvolvida no caso de estudo foi comparada com duas abordagens não baseadas em *machine learning* para a previsão da última compra de 360 clientes arbitrários. Foi usada uma abordagem baseada em *pattern mining* e uma abordagem baseada em SQL. Diferentes métricas de avaliação (nomeadamente *accuracy, precision, recall* e *f1-score* médios) foram registadas. Foi também analisada a forma como diferentes regras de associação se encontraram refletidas nas recomendações da solução desenvolvida.

A implementação baseada em *gradient boosted trees* do caso de estudo superou as soluções com as quais foi comparada quanto às métricas de avaliação, e mostrou uma maior capacidade de recomendar pelo menos um produto correto por cliente. Verificou-se também que as regras de associação mais fortes estão frequentemente refletidas nas recomendações.

A abordagem adotada e o algoritmo aprofundado mostraram resultados promissores e uma capacidade notável de fornecer recomendações úteis aos diferentes clientes, evidenciando a sua aptidão para adicionar valor ao retalho alimentar. Ainda assim, este sistema apresenta um elevado potencial para expansão.

Acknowledgement

I want to thank both my supervisors, Professor Carlos Ferreira and Eng. Sílvio Macedo, for providing me this opportunity and for the crucial discussions we have had. My parents, for helping me keeping the motivation and supporting me every time. Carla Miguel, for showing me the brighter side of life and supporting me throughout the different situations. My brother, for always having a smile and the calm in life. Santiago, for showing me how tiny things can have huge meanings. João Martins, for all the discussions we have had and the support we have transmitted to each other. All my family, family, and teachers across this amazing journey, because each one of you has left a mark in my life. Thank you.

This dissertation was carried out at and financed by the proponent company Xarevision SA.

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

AI	Artificial Intelligence.
ANN	Artificial Neural Network.
API	Application Programming Interface.
CPU	Central Processing Unit.
DB	Database.
DNN	Deep Neural Network.
FMCG	Fast-Moving Consumer Goods.
FN	False Negatives.
FP	False Positives.
GPU	Graphics Processing Unit.
JSON	JavaScript Object Notation.
JWT	JSON Web Token.
kNN	K Nearest Neighbor.
LSTM	Long Short-Term Memory.
MAE	Mean Absolute Error.
ML	Machine Learning.
MSE	Mean Square Error.
NCD	New Concept Development.
QFD	Quality Function Deployment.
REST	REpresentational State Transfer.
RMSE	Root Mean Square Error.
RNN	Recurrent Neural Network.
SQL	Structured Query Language.
SVM	Support Vector Machines.
TN	True Negatives.
TP	True Positives.
UC	Use Case.

Chapter 1

Introduction

This chapter discusses the problem that motivated the present dissertation, its purpose, the results (expected and achieved), the methodology adopted, existing contributions, and the document's organization.

1.1 Context

Recommender systems have gathered a lot of research interest since the mid-90s (Adomavicius and Tuzhilin 2005), focusing on providing additional value to many different areas such as music, cinema, or retail. These systems faced some additional problems due to the capabilities of the time (e.g., amount and quality of existing data or computational power), but the evolution of technology was able to spark its progress.

A recommendation problem consists, traditionally, of assigning a rate to an item or a set of items, and sorting it accordingly (Adomavicius and Tuzhilin 2005). This way, it is possible for recommender systems to generate recommendations regarding similar items or the *top-n* items for a specific context (Guidotti et al. 2017).

Grocery retail covers the biggest percentage of the revenue associated with all the retail subfields worldwide (Deloitte 2019), and it is characterized by a large number of unique products and customer segments. These customers have unique shopping habits, making generating recommendations for grocery retail a problem of its own. Unlike in other retail areas, grocery customers buy the same items frequently and tend to manifest clear preferences on certain items or brands, conditioning their receptivity to new items.

This dissertation aims at covering the topic of recommender systems applied to grocery retail, from both theoretical and practical stand-points, culminating with the development of a solution capable of generating accurate grocery retail predictions.

1.2 Problem

Grocery retailers understand the importance of investing in unique customer experience, both in-store and online stores (Deloitte 2018). A way of offering personalized service and, thus, increasing the proximity with customers, is by facilitating the process of creating and managing the shopping list for the next purchase (Guidotti et al. 2017).

Recommending items based on an item the customer is viewing at a specific time is a common reality in e-commerce platforms. Recommending shopping lists for the next purchase (i.e., the *top-n* next items) is, however, a distinct and more complex task because of the correlation between users and their habits (Guidotti et al. 2017). Each customer's unique shopping habits make grocery retailers face difficulties when trying to specifically target individual customers with product recommendations that they desire at a specific moment (Sun, Gao, and Xi 2014).

Recommending products for grocery retail further becomes a unique and complex task because of the wide variety of different products, its seasonality, its promotional character, its evolution in time, the huge number of customers, and the reduced proportion of items they buy (Sano et al. 2015). For grocery retailers to achieve this proximity level with their customers, information regarding their products and customer's shopping history is needed, which is not always a reality.

1.3 Purpose

This dissertation's main goal is to develop a machine learning-based recommender system for grocery retail, capable of generating shopping list predictions for the next purchase. These predictions should match the customer's needs at that specific moment.

With this solution, retailers can get closer to their customers and benefit directly from individual marketing advantages, as it gets easier to promote different products, while enhancing, at the same time, the trust relationship between retailer and customer (Sun, Gao, and Xi 2014). Besides, sales can be maximized, and stock management optimized.

To develop the recommender system, a study on state of the art is carried out, aiming to understand the existing approaches and their characteristics, as well as the most promising technologies and machine learning approaches to use. A case study integrating the developed recommender system with a public grocery retail dataset allows a validation of the solution.

The recommender system needs to be open to expansion and customization by the possibility of training the model using data from different retailers. Recommendations should be available through a REST API, capable of abstracting the internal logic. This way, the solution is decoupled and integrated with retailers and existing systems in the proponent organization.

Both the predictions' quality and solution's performance are important factors to the final result, and thus they are measured and analyzed. To compare the quality of the recommender system, two non-machine learning approaches provided by the organization are used for comparison. The performance is evaluated using a simulated workload.

1.4 Methodology

This dissertation has the following two significant phases: investigation and development. Both phases were paired with meetings with both the supervisor and co-supervisor, where the work was presented, and the next steps were discussed· Both phases were also organized using a task board, where the progress and next-steps were tracked.

The investigation was conducted to design a generic architecture to allow integrating the recommender system with different retailers and identify the more adequate technologies to implement it. A case study using a grocery retail dataset and machine learning algorithm that better performs under this data was developed to validate the solution.

The development of a machine learning-based recommender system followed the investigation, and the whole process was documented. The case study was then compared against other implementations under test circumstances. Additional quality and performance tests were also performed and discussed.

All meetings and board organization follow the Kanban methodology - an agile methodology that aims at periodic deliveries and progressive and straightforward ways of following the project status (Raut, Wakode, and Talmale 2015). Based on this methodology, tasks were divided into smaller ones, allowing better track of the project status. Then, tasks were organized in a board according to its status: backlog, to-do, in progress, blocked, in a test, and done.

1.5 Contributions

The contributions provided by this dissertation to the proponent organization can be synthesized with the following artifacts:

- the study performed on the state of the art of machine learning algorithms and its application to recommender systems for grocery retail;
- the development of a case study, where the implemented architecture is integrated with a public grocery retail dataset, useful features are identified and extracted, and the processes behind tuning/optimizing a machine learning algorithm are explored;
- the study comparing a machine learning-based recommender system with traditional approaches;
- the development of a generic and modular architecture, designed for the integration with different grocery retailers.

This dissertation has also contributed to the student's personal and professional development by providing a chance to explore the machine learning field and exploring alternatives to overtake the challenges of recommender systems for grocery retail.

1.6 Document Organization

This dissertation is composed of 7 chapters: introduction, context, state of the art, solution description, solution implementation, evaluation and results, and conclusion.

The introduction chapter provides a brief contextualization of the matter, presenting the problem and the primary purpose and analysis of the results, methodology, and organization.

The context chapter is organized under two main areas: the context of the dissertation and value analysis. The first part presents recommender systems, grocery retail, and its uniqueness, some essential business concepts, processes and actors, and existing restrictions.

The second part analyzes the value and the quality provided by this dissertation and presents a business model analysis.

The state of the art chapter starts with an analysis of recommender systems, including its architectures, challenges, and approaches. It is followed by a study on machine learning applied to recommendations and a discussion on the different approaches, metrics, and comparisons. An analysis of other approaches to the same problem is presented thereafter, followed by a comparison of the results achieved by different methodologies on public studies. It is concluded by a study of the leading technologies for machine learning-based solutions.

The solution description chapter starts by analyzing the functional and non-functional requirements of the problem and essential domain concepts. It is followed up by the solution design, describing the architecture and the journey to its adoption, a use case realization view of the use cases, a deployment view, the modeling methodology, and the adopted technologies. The solution implementation contains details on the different architectural components and their development, the different machine learning-related processes involved, and the software tests developed to support the implementation.

The evaluation and results chapter starts with an analysis of experimentation and evaluation, test hypothesis, metrics, and methodology. It is followed by a case study, where the implemented architecture is validated against a public dataset, using a promising machine learning algorithm. This case study solution is then optimized as far as hyperparameters are concerned, and compared against other recommender systems. An evaluation of the inference of association rules and performance tests are also performed. The last chapter presents the conclusions obtained in this dissertation, limitations, and future work.

Chapter 2

Context

This chapter aims to contextualize this dissertation and to understand its value. Both the processes and the actors are explained, and the existing restrictions are discussed. Value analysis is also presented, including details on the innovation process, a value proposition, a business model canvas, an analysis on the value network, and the quality function deployment view.

2.1 Context of the dissertation

This section contextualizes the work done in this dissertation regarding recommender systems and grocery retail. Some important business concepts are also presented, as well as the actors evolved. The restrictions on the development of the solution close this section.

2.1.1 Recommender Systems

Since the mid-90s, recommender systems have gathered much interest, as far as research is concerned (Adomavicius and Tuzhilin 2005). Authors describe a recommendation problem, in its purest and more basic form, as estimating ratings for items that users have not yet demonstrated interest on, based on their interest in other items or other user's interests, and then recommending the most highly-rated ones.

The utility of an item is defined by a rating - which can be specifically provided (e.g., historical data) or calculated by a function. Besides, users are commonly cataloged according to various characteristics into profiles that represent their user-space (Adomavicius and Tuzhilin 2005; Jariha and S. K. Jain 2018).

According to the literature (Adomavicius and Tuzhilin 2005), papers on collaborative filtering have sparked the interest of both academy and industry, which have responded by exploring and developing techniques that enable the generation of customized real-world recommendations on increasingly bigger datasets. Despite the progress, due to both the embryonic phase of the matter and hardware capabilities of the time, the work done in this area faced some difficulties: the specific representation of users' behavior was ambiguous, contextual information (i.e., data) was sometimes lacking, evaluating multi-criteria ratings and performance was not clear and generating predictions was computationally demanding. Noticeable signs of progress have been achieved, on the matter, since day one.

Despite the common objectives, recommender systems differ across domains. Some business areas, such as the fashion or music industries, thanks to its domain, deal with a smaller set of both items and customers (Hanke et al. 2018). Other areas, like retail, have a complex product hierarchy, deep on both depth and width, and a huge and distinct amount of customers (Sano et al. 2015).

2.1.2 Grocery Retail

Grocery retail is an increasing vector when it comes to revenue, covering the most significant percentage associated with retail worldwide (Deloitte 2019). It is mainly characterized by a large number of unique products, sparse customer segments, and seasonality (Sano et al. 2015).

Traditionally, grocery retail was done in local stores owned either by big or small players (commonly referred to as bricks and mortar retailers). Many types of grocery retailers use loyalty management systems to target their customers more efficiently and improve the way their stocks are managed. Technology advances have made people more comfortable while shopping online but have also increased the bar when it comes to in-store technology, either for customers or for employees' usage.

Despite the evolution in grocery retail, the most significant percentage of purchases corresponds to in-store purchases (Deloitte 2018; Mitova 2020). Retailers are investing in creating a more omnichannel (strategy applied across the different channels) experience by providing their customers with the ability to have a complete shopping experience from home to stores. Online stores, loyalty mobile apps, shopping-list management mobile apps, price-checking software (either in-store or via mobile apps), and self-scanning mobile apps are examples of retailers' effort to improve their footprint.

The pre-shopping experience has gained interest in recent years, with grocery retailers investing in shopping list recommendations and personalized prices (Deloitte 2019). Most customers still show a preference for preparing shopping lists before shopping (around 69% of women and 52% of men have shown this habit (Mitova 2020)), providing retailers with a way of improving the technology and assisting their customers in the whole shopping experience.

Unlike some other retail areas, where suggesting new items is mostly good and suggesting duplicated items may not be considered (e.g., recommending the same shirt may not be ideal for fashion retail), in grocery retail, customers' habits are unique. People like the same product and repeat some purchases frequently (e.g., people may have a favorite brand for an item and always buy that same product). Also, products have hierarchical categories, and a product of a category is not necessarily a replacement of another (e.g., a solid yogurt is different from a liquid one).

Recommending customized shopping lists for grocery retail is a singular task, and it is attracting attention and investments (Deloitte 2019), making it valuable to explore.

2.1.3 Business Concepts

Some retail-related concepts are important and referred over this dissertation.

- **customer** - a person with unique preferences, opinions and needs regarding items;

- **shopper** - a person who goes shopping. A shopper can represent an individual customer or a number of customers (e.g., a person that goes shopping for the items her family needs);
- **item** - a product that can be bought by a customer, with specific characteristics;
- **shopping list** - a list of the items that a customer wants to buy;
- **shopping cart/basket/order** - all the items bought in a single purchase.

2.1.4 Processes and Actors

The main actors in the tasks are customers, grocery retailers, and the recommender system.

Four main processes are on the foundation of the flow represented by this dissertation:

1. Customers go shopping;
2. Retailers acquire, process, and store loyalty and transactional data;
3. Recommender systems are trained against historical information and generate recommendations;
4. The tailored recommendations are shown to the customer.

2.1.5 Existing Restrictions

The recommendations' success can be affected by restrictions regarding data, technology, and literature. In order to maximize it, it is important for retailers to have solid historical data and to choose a proper recommendation method.

Different grocery retailers have different technology footprints (e.g., traditionally, brick and mortar retailers are less evolved technologically than retailers with online solutions), causing differences in the information they acquire. The lack of loyalty information on some retailers or the insufficient depth can make generating predictions a difficult or impossible task.

The specificities of grocery retail can turn scaling into a problem, as the number of customers and products may cause some recommendation methodologies to have poor or non-practical performance when using big amounts of data (Sun, Gao, and Xi 2014). Data sparsity, in addition, can cause some of the algorithms to fail. The uniqueness of the grocery retail domain may demand high computational resources to accomplish good results (Sano et al. 2015; Sun, Gao, and Xi 2014).

Besides technological restrictions, the lack of literature and deployed recommender systems for grocery retail makes managing expectations on the predictions and comparing results two difficult tasks.

2.2 Value Analysis

This section aims to analyze the value of the dissertation. First, the front-end of innovation (using the New Concept Development model) is detailed via the five front-end elements. A value proposition is presented, discussing the advantages of generating customized product recommendations. A business model canvas is presented after synthesizing the business model, and the quality function deployment is studied. The section is concluded with an analysis of the value network.

2.2.1 Innovation Process

The New Concept Development (NCD) model, proposed by Peter Koen (P. A. Koen et al. 2002), has made it possible to use both a common language and insights on the process of innovating (P. Koen et al. 2001). The proposed model includes three key-parts, shown in Figure 2.1: five front-end elements, which comprise the front-end of innovation; an engine that drives these elements; and external environment features, which influence the engine and the front-end elements.

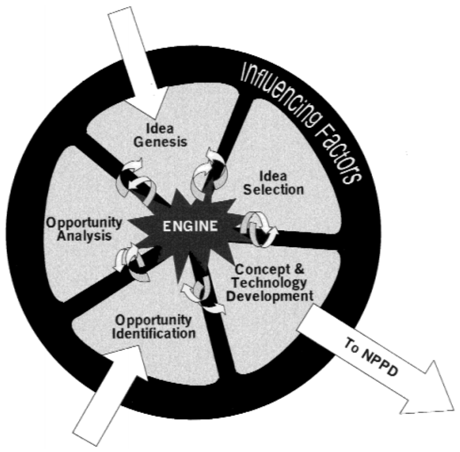


Figure 2.1: NCD model as a circular representation (P. Koen et al. 2001)

The five front-end elements, shown within a circular shape, mean that the ideas are expected to flow, circulate and iterate, freely, between them (P. Koen et al. 2001). The Opportunity Identification element represents the moment where potentially interesting opportunities are identified; the Opportunity Analysis element corresponds to the process of using information from Opportunity Identification, in order to validate if the opportunity is viable; Idea Genesis is the building and maturation of an opportunity into concrete ideas; the Idea Selection element consists of selecting the most promising ideas, as far as business value is concerned; Concept and Technology Development is the final element, and involves the development of a business case, according to factors such as market potential, customer needs, competition, investment, and project risk. The Opportunity Identification and Idea Genesis are typically the two existing starting points on the innovation process (P. A. Koen et al. 2002).

This section details the application of the five elements of the NCD engine to the current dissertation.

Opportunity Identification

Customers have unique shopping habits when it comes to shopping for groceries. Demographic, social or economic motifs, personal preferences, and habits make people have different needs. Besides, they wish frictionless processes, at hand, within the smallest amount of time possible.

Grocery retail stores have a more difficult time when it comes to attending to customer preferences because of the huge amount of products they offer, the number of customers they serve, and the many factors associated with the process of choosing items to buy (Sano et al. 2015). They are working on getting closer to their customers in many ways, such as increasing their online footprint - which is being welcomed by customers, mostly younger ones (Group 2018).

The revenue associated with Fast-Moving Consumer Goods (FMCG) is increasing over the years, representing around 66% of the revenue associated with retail (Deloitte 2019). The footprint of online grocery shopping keeps also increasing (Deloitte 2018). In addition, customers understand the evolution of technology, being it related to payments, in-store shopping, or loyalty management. The competitiveness between grocery retailers is a reality, and it has an important role in the way they target their customers.

Approximating grocery retailers and customers, through technology, taking advantage of the growing market is seen as an opportunity window. A retailer who can target their specific customer's needs is a retailer with a higher chance of succeeding in a competitive market.

Opportunity Analysis

By having a chance to use technology as part of the grocery shopping experience, people have gained comfort in dealing with a wider variety of items. Subscription plans are also conquering new customers each day, especially among younger customer profiles. Frictions associated with the process of returning items are also being minimized (Group 2018). This information could be perceived as higher confidence for grocery retailers to suggest new items to customers in a more traditional way (e.g., through advertising) or by improving subscription plans to include product suggestions.

Researchers have concluded that customers are comfortable with using retailer-specific or common grocery apps in order to manage shopping lists, and they are welcoming customization achieved by the use of previous shopping information (e.g., when building their lists) (Group 2018). Also, studies have shown that around 69% of women and 52% of men prepare shopping lists prior to go shopping (Mitova 2020).

Thus, personalized retail technology can be used to promote sales and make customers feel a personalized experience when shopping.

Idea Genesis

The confidence that customers deposit on technology can be taken advantage of, and the need for comfort that customers seek can be attended if retailers have the opportunity to suggest personalized shopping lists based on the historical data they have from their customers. Customer's individual and household needs can, this way, be met more efficiently, and retailers can improve sales and improve their marketing strategy into a more individual format.

Idea Selection

Using recommender systems to generate a personalized shopping list, retailers can target the specific needs of their customers. The historical data can be used to develop a model that can propose shopping lists for a specific moment.

The recommender system can be integrated with different systems, having the potential to enhance the shopping experience in many vectors. Grocery retailers can use these recommendations to save customer's time while promoting sales (e.g., using subscription plans to ship items into customer's houses), facilitating the pre-shopping experience (e.g., when building shopping lists), and increasing basket sizes.

Concept and Technology Development

The developed solution is expected to be a low-coupled system, with a clear communication interface to make the integration with external systems a simple task. Public data should be used when developing a case study.

2.2.2 Value

Value is an ambiguous concept since it is distinct per stakeholder (Wolfgang and Andreas 2006). As far as this dissertation is concerned, benefits are distributed mainly between retailers and customers. Some minor sacrifices exist, however, between parts, in order to make it a reality.

The following subsections present a value proposition of the solution and the value perception and existing trade-offs.

Value Proposition

The value proposition describes what products and services offer value to customer segments (Petrovic and Kittl 2003). This analysis is particularly useful when presenting the project to both customers and investors in order to be clear about how it differs from its competition (Wolfgang and Andreas 2006).

The solution presented by the current dissertation is designed to provide personalized grocery shopping cart suggestions to customers, making the shopping experience easier while giving grocery retailers a chance to promote new products and boost sales. These recommender systems make grocery retailers closer to customers while giving them the potential to boost their footprint.

Value Perception And Trade-offs

Grocery retailers benefit from being closer to customers by understanding them in a better way and from being able to switch to an individual marketing strategy. By predicting shopping lists, retailers can target their needs more efficiently and effectively, to both parts.

It is possible for grocery retailers who adopt the solution to optimize stock management, sell new products, and improve the trust-relationship with customers. They receive the chance to integrate the recommendations engine with different technologies they provide their customers with (e.g., mobile apps or websites), benefitting from the advantages associated with an omnichannel experience.

By including these technologies, grocery retailers also make a statement as far as technology use in their business is concerned. Nevertheless, a front-end interface with its customers has to be developed or adapted, so they can interact with the predictions.

Customers will benefit from a smoother experience while preparing shopping lists and shopping, especially while online shopping, because of the simplification of time-consuming traditional processes. They will also have a bigger chance of discovering interesting products for their profile.

The solution gets better (learns) with time, being as good as the shopping history size. For grocery retailers to benefit from customized recommendations, loyalty context data has to exist. It is of paramount importance for customers to understand that to benefit from a tailored experience, their shopping history has to be known and analyzed by computer systems.

Besides, since machine learning and recommender systems are currently a hot topic, showing an application of a recommender system for grocery retail, and evaluating its results, can be valuable for both the academy and enterprise world.

2.2.3 Business Model

The solution proposed by this dissertation is intended to be used by grocery retailers, giving them the ability to boost sales and enhancing their customer's experience. This way, the total available market (commonly referred to as TAM) comprises all the grocery retailers in the world.

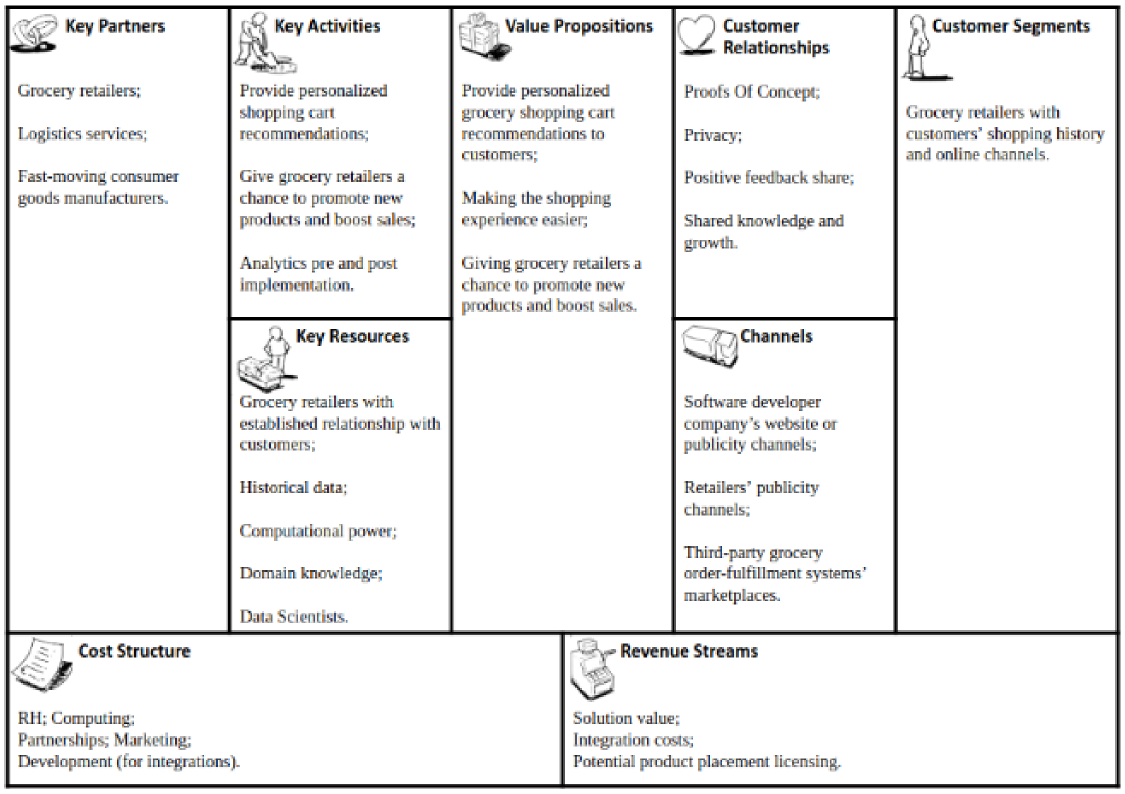
Despite the similarities between retailers, there are differences in the products they sell and how they target their customers, and historical data is crucial to generate good predictions. This way, the serviceable available market (commonly referred to as SAM) represents all the grocery retailers with accessible historical data regarding their customer's shopping habits.

This solution can be easily and intuitively communicated using a business model canvas (Qastharin 2016). This canvas captures nine building blocks: key partners, key activities, key resources, cost structure, value propositions, customer relationships, customer segments, channels, and revenue streams. The first four elements are commonly referred to as business-related and the remaining ones as customer-related.

Grocery retailers, logistic services, and manufacturers are identified as the main partners, and they are of paramount importance to provide personalized recommendations. Being a customized solution, data, and knowledge associated with it are the most important resources.

Grocery retailers with accessible customer's shopping history are the target market of the solution. Proof of concepts and shared knowledge with customers are identified as keys to success. The visibility is mainly foreseen as associated with the publicity channels of customers and with partnerships with order-fulfillment systems.

A business model canvas for this business-to-business solution is presented in figure 2.2.



Key Partners

Grocery retailers;

Logistics services;

Fast-moving consumer goods manufacturers.

Key Activities

Provide personalized shopping cart recommendations;

Give grocery retailers a chance to promote new products and boost sales;

Analytics pre and post implementation.

Key Resources

Grocery retailers with established relationship with customers;

Historical data;

Computational power;

Domain knowledge;

Data Scientists.

Value Propositions

Provide personalized grocery shopping cart recommendations to customers;

Making the shopping experience easier;

Giving grocery retailers a chance to promote new products and boost sales.

Customer Relationships

Proofs Of Concept;

Privacy;

Positive feedback share;

Shared knowledge and growth.

Channels

Software developer company's website or publicity channels;

Retailers' publicity channels;

Third-party grocery order-fulfillment systems' marketplaces.

Customer Segments

Grocery retailers with customers' shopping history and online channels.

Cost Structure

RH; Computing;
Partnerships; Marketing;
Development (for integrations).

Revenue Streams

Solution value;
Integration costs;
Potential product placement licensing.

Figure 2.2: Business model canvas diagram

2.2.4 Value Network

Value network analysis is a methodology for business modeling, where business activities and relationships are visualized from a dynamic standpoint of the whole system (Allee 2006). Verna Allee, the author, argues a modeling method centered on the networking principle, where activities enhance the value of the business as a whole. The business should favor this characteristic, favoring connections between activities through the whole ecosystem.

Figure 2.3 represents the value network diagram, where the important elements to the project idea and its tangible and intangible connections are shown. One can understand the value of the idea and how the different parties interact/benefit within the flow.

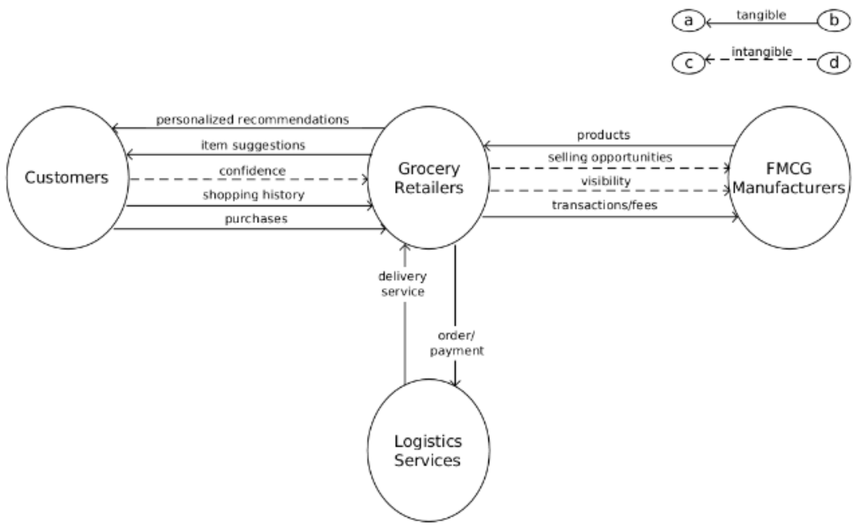


Figure 2.3: Value network analysis diagram

Since the solution proposed in the current dissertation consists of a system for enabling grocery retailers to predict their customer shopping needs, the symbiosis is clear: retailers and customers benefit each other. In addition, FMCG manufacturers and logistics services also have an important role in supporting the activity (by providing items and by offering services, respectively).

2.2.5 Quality Function Deployment

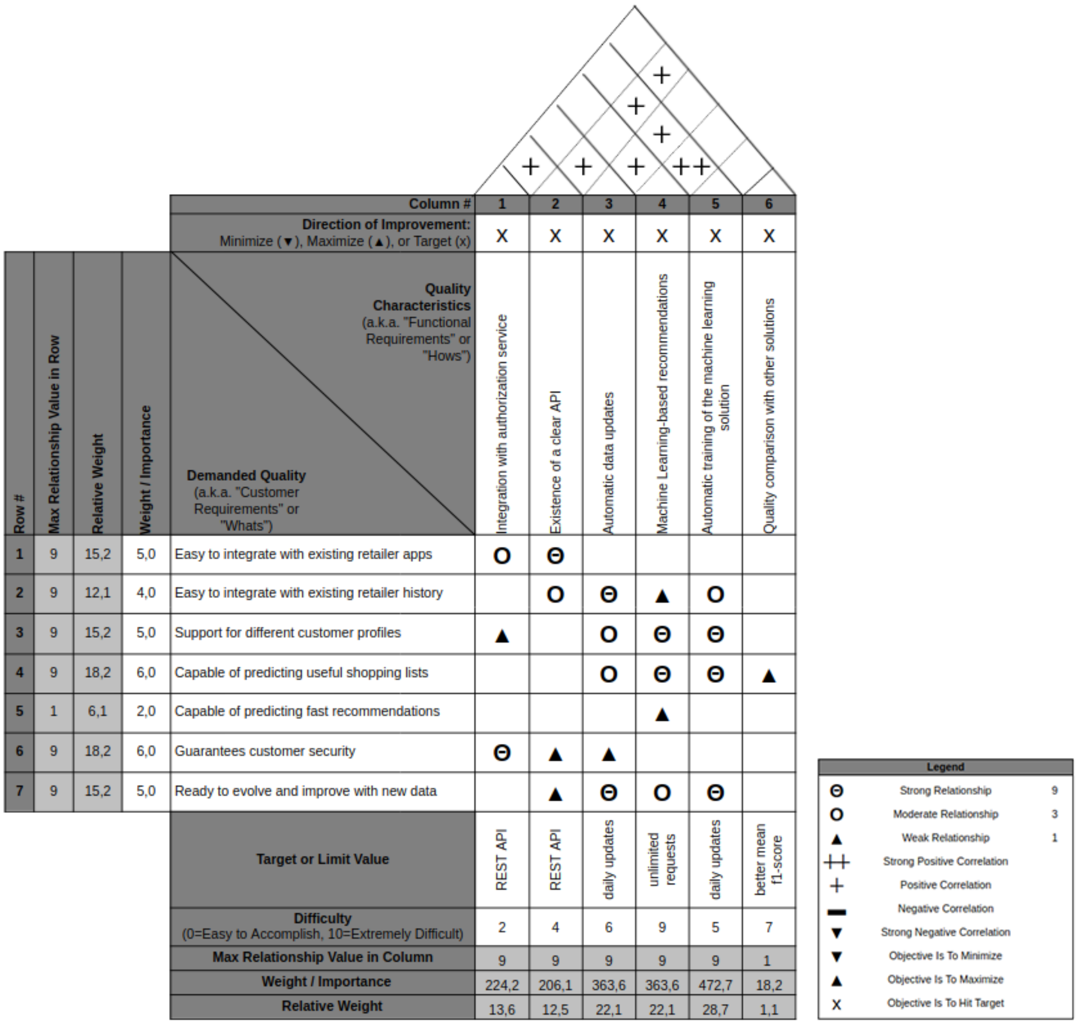
Quality Function Deployment (QFD) is a methodology for designing a product or a service, based on the customer demands, considering both the producer and supplier chains (Manufacturing Group 2007). This technique allows the product or service to have a better quality in a smaller amount of time, makes sure that the design is customer-driven, and provides a tracking system for future design or process improvements. Besides, it encourages the involvement of the different parts of the organization.

The *house of quality* is the primary design tool of the QFD methodology, and it consists of a map with a large amount of information in one place that can be used by the different teams involved in order to find and follow design priorities (Hauser and Clausing 1988). The name has its origins in its shape.

This tool maps the customer quality demands (rows), according to a predefined weight, with the quality characteristics of the product (columns), the difficulty associated with it, and the target quality value per technical attribute (Hauser and Clausing 1988; Manufacturing Group 2007). Also, it describes the correlation between attributes (gabled roof).

The central matrix contains symbols representing the relationship between the demanded quality and product characteristics. This diagram targets competitors as well, by comparing the solution against the service provided by similar companies. However, since no public information regarding competitors is known, this part cannot be approached.

The figure 2.4 presents a house of quality diagram for the current dissertation. It describes the main quality attributes for the customers (i.e., grocery retailers) and the characteristics of the solution. The relationships between both these entities and between the attributes itself can be found.



				Column #	1	2	3	4	5	6
				Direction of Improvement: Minimize (▼), Maximize (▲), or Target (x)	x	x	x	x	x	x
Row #	**Max Relationship Value in Row**	**Relative Weight**	**Weight / Importance**	**Quality Characteristics** (a.k.a. "Functional Requirements" or "Hows") / **Demanded Quality** (a.k.a. "Customer Requirements" or "Whats")	Integration with authorization service	Existence of a clear API	Automatic data updates	Machine Learning-based recommendations	Automatic training of the machine learning solution	Quality comparison with other solutions
1	9	15,2	5,0	Easy to integrate with existing retailer apps	O	Θ				
2	9	12,1	4,0	Easy to integrate with existing retailer history		O	Θ	▲	O	
3	9	15,2	5,0	Support for different customer profiles	▲		O	Θ	Θ	
4	9	18,2	6,0	Capable of predicting useful shopping lists			O	Θ	Θ	▲
5	1	6,1	2,0	Capable of predicting fast recommendations				▲		
6	9	18,2	6,0	Guarantees customer security	Θ	▲	▲			
7	9	15,2	5,0	Ready to evolve and improve with new data		▲	Θ	O	Θ	
				Target or Limit Value	REST API	REST API	daily updates	unlimited requests	daily updates	better mean f1-score
				Difficulty (0=Easy to Accomplish, 10=Extremely Difficult)	2	4	6	9	5	7
				Max Relationship Value in Column	9	9	9	9	9	1
				Weight / Importance	224,2	206,1	363,6	363,6	472,7	18,2
				Relative Weight	13,6	12,5	22,1	22,1	28,7	1,1

Legend		
Θ	Strong Relationship	9
O	Moderate Relationship	3
▲	Weak Relationship	1
++	Strong Positive Correlation	
+	Positive Correlation	
▬	Negative Correlation	
▼	Strong Negative Correlation	
▼	Objective Is To Minimize	
▲	Objective Is To Maximize	
x	Objective Is To Hit Target	

Figure 2.4: Quality Function Deployment - House of Quality diagram

The diagram evidences that the most important quality characteristics for the solution are related to the ability to evolve in time by automatically updating the solution and the data it works with, in order to assure an up to date machine learning-based recommender system. These quality characteristics target the most crucial quality demands of customers, namely the quality of recommendations, the support for different customer profiles, and customer security. So they should be pursued in order to keep the customers satisfied and remain a priority throughout the different steps of this dissertation.

Chapter 3

State of The Art

This chapter contextualizes the work done in this dissertation as far as approaches and technologies are concerned. It starts by detailing recommender systems, their architecture, the challenges associated with them, and their possible classifications.

Machine learning is, hereafter, described. Key concepts such as *unsupervised learning*, *supervised learning* and *reinforcement learning* are detailed, and machine learning approaches to recommender systems are presented. Learning methodologies and evaluation metrics are analyzed, followed by a comparison between the algorithms.

Non-machine learning approaches to recommender systems are also analyzed, followed by the most relevant technologies and frameworks. This chapter is concluded by a summary of all the presented topics.

3.1 Recommender Systems

Recommender systems are solutions with the ability to use previously known historical data to generate predictions on a matter (Adomavicius and Tuzhilin 2005). Items are assigned a rating (e.g., a weight or a sequential order), and the most relevant ones, according to the problem, are recommended.

Different recommender systems can be developed according to the problem and the requirements (Adomavicius and Tuzhilin 2005). Simpler ones can be solved using heuristics (which use business rules to generate the predictions upon historical data). More complex scenarios can benefit from data mining and machine learning algorithms. In the last two scenarios, models are created and used to generate future predictions.

A model is said to be created after applying the algorithm to the training data (i.e., the part of the dataset used for training). A model can also be tested with test data (i.e., the part of the dataset used for testing) and having its performance evaluated.

Recommender systems can face problems when generating predictions, depending on the data used in the training process or even the methodology adopted (Jariha and S. K. Jain 2018). Since they predict upon different criteria, different classifications are associated with them.

In this section, the main architecture behind a recommender system is analyzed, as well as the major processes evolved. Classical problems that they tend to face are also discussed. The section is concluded by going through the different ways a recommender system can be classified.

3.1.1 Architecture

A recommender system is composed of two different moments: information collection and prediction (Jariha and S. K. Jain 2018). The first moment consists of gathering meaningful information regarding both users and items from the available options to build a solid knowledge basis; the second one, on the other hand, consists of generating predictions based on the information obtained in a previous moment.

The information collection task is of paramount importance as it is responsible for building the information used as input of the recommender system. The prediction task can consist of the execution of a single technique or multiple techniques combined (Jariha and S. K. Jain 2018).

When recommender systems are more complex and thus represented by a model, success on the predictions depends on a successful training process, where the model was trained with useful information retrieved from the training data. Feedback is provided during this phase, either implicitly or explicitly.

Feedback is said to be implicit when it arises from information existing in the training data (e.g., the purchase history) (Jariha and S. K. Jain 2018). On the other hand, feedback is considered explicit when it is originated from input prompted explicitly to the user (e.g., if a user is asked to classify whether a specific basket would make a good prediction for that specific moment or not). Naturally, both kinds of feedbacks are useful for a recommender system life cycle and success and can be combined to create a more robust model.

3.1.2 Challenges

Recommender systems may face different challenges during training. Data quantity or quality, limitations regarding the approach, or training time are some of the main challenges. The presented challenges are common to many other research areas besides recommender systems.

The sections below present some of the most frequent challenges faced by recommender systems, according to the literature.

The Cold-Start Problem

Adding a new user or a new item into the system can work against some recommendation approaches, as there is no sufficient information for it to predict viable recommendations. Recommender systems such as collaborative filtering, because of its dependency on the relationships between users and items, struggle with this problem. On the other hand, content-based recommender systems do not find this issue when it comes to new users, as they do not rely on previous associations (Jariha and S. K. Jain 2018; Mohamed, Khafagy, and Ibrahim 2019). New items, because of the lack of history, are still prone to this issue.

The Sparsity Problem

There is typically a correlation between data sparsity and the success of the predicted recommendations. Recommender systems that rely on ratings provided by users tend to face this issue, as they may not provide the classification for all items (Jariha and S. K. Jain 2018; Mohamed, Khafagy, and Ibrahim 2019). Also, items may not be considered as they do not meet certain requirements related to the number of classifications provided by users, decreasing the quality of recommendations. Some authors argue demographic similarities, in addition to the existing ratings, as a possible solution for this problem (Jariha and S. K. Jain 2018).

The Overfitting Problem

Overfitting occurs when a model fits too closely to a subset of the items in the training data. This way, the predicted items will not have a significant margin for innovation nor predict an unbiased scenario. Instead, only previously seen/recommended items related to previously existing scenarios are recommended, without reflecting the recommendation moment and conditions (Jariha and S. K. Jain 2018).

The Scalability Problem

Scalability is the ability of a system to handle increasing loads gracefully (Jariha and S. K. Jain 2018; Mohamed, Khafagy, and Ibrahim 2019). The increasing number of both users for which the recommender system has to generate predictions and items being predicted brings scalability problems. The system faces difficulties to remain capable of generating recommendations in a reasonable time. Approaches like collaborative filtering suffer in a critical way, as the computation is directly related to the number of users and items existent (Jariha and S. K. Jain 2018).

The literature argues some approximation mechanisms to increase the capacity of a recommender system to scale (e.g., splitting the data across systems). Despite this fact, most of them sacrifice the accuracy of the results (Jariha and S. K. Jain 2018). Areas such as retail, because of the huge amount of users and items involved, tend to struggle with this problem.

The Long-Tail Problem

The long-tail problem affects approaches that rely on item ratings for items that were newly added or that have fewer recommendations. Similar to what happens in the cold-start problem, these items may find it difficult to be recommended. The literature states that this is an issue faced by most recommendation techniques (Jariha and S. K. Jain 2018), where long-tails are introduced by suggesting only the most popular or classified items.

The Grey Sheep Problem

The grey sheep problem occurs when a recommender system relies on user categories in order to generate recommendations and faces users that do not properly fit any of the existing profiles (Mohamed, Khafagy, and Ibrahim 2019). These users cannot, thus, benefit from the information regarding the profiles, risking its predictions to be of less quality. Approaches like collaborative filtering or demographic-based recommendations tend to suffer from this issue because of its dependency on collaborative information.

The Privacy Problem

Recommender systems need information regarding users or their taste in order to be able to generate accurate and meaningful predictions (Jariha and S. K. Jain 2018; Mohamed, Khafagy, and Ibrahim 2019). Users may find some discomfort with the provided information, which brings the necessity of transparency and trust relationship between both user and recommender system, so users know which information is provided (Mohamed, Khafagy, and Ibrahim 2019) and understand that there may be a risk of personal information being exposed (Jariha and S. K. Jain 2018). This should be fought by the recommender entity.

3.1.3 Classification

Recommender systems can be classified according to the way that recommendations are generated. The literature presents two major approaches: content-based recommendations and collaborative recommendations (Adomavicius and Tuzhilin 2005). However, the evolutions on the field have made some authors consider additional techniques, commonly referred to as hybrid approaches, as they combine at least two methodologies into one (Jariha and S. K. Jain 2018). Some examples of hybrid approaches include utility-based recommendations, knowledge-based recommendations, demographic-based recommendations, interactive recommendations, and context-based recommendations (Burke 2002; Jariha and S. K. Jain 2018).

In addition to the classifications presented by the literature, it is common for the industry to develop their own heuristics, often based on the most popular items or on items frequently bought together.

The sections below present some of the most typical classifications for recommender systems, according to the literature.

Content-Based Recommendations

Content-based recommender systems, also known as cognitive filter recommender systems (Jariha and S. K. Jain 2018), are popular for their simplicity: recommendations are generated based on relationships between item's features (e.g., words in texts (Burke 2002)) and the similarity between user profiles (traced based on the historical information regarding their previous interests). This way, the system learns from the features present in the information regarding what users have seen in the past (Burke 2002; Jariha and S. K. Jain 2018; Lops, Gemmis, and Semeraro 2011).

Different learning methods can be used with content-based recommender systems to produce user profiles. Among the most used methods, one can find neural networks, vector representations, and decision trees (Burke 2002). The learned models are typically updated only when new information about user preferences exists.

Because of the tight relationship between the recommendations and the user history, these systems tend to face some difficulties while recommending new products for which a user has not yet specifically manifested its interest (Jariha and S. K. Jain 2018).

Collaborative Recommendations

Collaborative recommender systems, also known as "social filter" or "collaborative filtering" recommender systems (Jariha and S. K. Jain 2018), generate recommendations based on users' behavior. Information from multiple users is combined (collaboratively) in order to learn patterns that can help to predict the interest of specific users.

Typically, collaborative recommendations are either classified as *item-item* or *user-item* (Jariha and S. K. Jain 2018). Item-item recommendations are based on relationships between different items, which can be used to generate predictions related to items that the user has already demonstrated interest. User-item recommendations, on the other hand, are generated based on the proximity/similarity between users. The interest of a specific user is used in the predictions for similar ones.

The more the historical data and the number of users, the more relevant and accurate the recommendations tend to be, as more similarities exist between user habits (Burke 2002). This characteristic tends to be flipped into a problem: since the recommendations are based on users and items, adding a new element (either a user or a product) can cause poor predictions because of the lack of data (Jariha and S. K. Jain 2018).

Utility-based recommendations

Utility-based recommender systems are hybrid approaches that try to base their predictions on the evaluation of the match between what a user needs and the available options (Burke 2002). These recommendations are, thus, predicted by computing the utility of each item for a specific user, and the systems get to know the user preferences over time.

The utility of an item for a user is calculated by an evaluation function. This function may vary according to the implementation. Different companies have different techniques to reach this result (Burke 2002), according to their needs.

One key benefit of utility-based recommender systems is that the evaluation function can be designed to model business rules or preferences (e.g., exchange price for delivery schedule for a user with an immediate need (Burke 2002)). This is, at the same time, the main difficulty regarding these recommender systems, as the evaluation function is a key component in the solution, but it might get hard to model.

Knowledge-Based Recommendations

Knowledge-based recommender systems are hybrid approaches that attempt to generate recommendations based on inferences regarding user needs and preferences. A recommendation is made upon a reasoning process on the need for an item, using functional knowledge about how an item meets a user's need (Burke 2002).

The knowledge used for the prediction process can be specifically provided by the user (Burke 2002; Jariha and S. K. Jain 2018) or obtained via the analysis of existing data across the domain. The first scenario can be observed by the usage of the data provided in the query on search engines (Burke 2002); an example of the second one is the usage of information regarding items to infer similarities (e.g., use cuisines to infer similar restaurants).

The difficulty in obtaining the knowledge used to generate recommendations is the major challenge these recommendation systems face (Jariha and S. K. Jain 2018) because of its origin and domain specificity.

Demographic-Based Recommendations

Demographic-based recommender systems are hybrid approaches that use user-user relationships in order to generate predictions. Using user data (e.g., age, gender, and location), demographic profiles based on the similarity between users are created (Burke 2002; Jariha and S. K. Jain 2018). The relationship between items and user-groups is calculated and used to recommend items.

The demographic data can be directly obtained from the user for pre-usage surveys or by analyzing actual data regarding users and inferring useful information (Burke 2002). Different solutions may obtain this information in different ways, either by convenience or by design, but always aiming to collect useful information that enables the system to calculate user profiles.

Since the correlations between users are based on the similarity between their data, no history is needed in order for these systems to start recommending items to a new user (Jariha and S. K. Jain 2018), because the system assumes that users with similar demographic information will have similar interests. Common problems such as cold-starts or overfitting 3.1.2 are also avoided by these systems. Users and items are typically represented as vectors. Thus the proximity can be calculated by the scalar product of two vectors, for example.

Interactive Recommendations

Interactive Recommendation Systems are hybrid approaches focused on generating predictions based on the preferences of a user, for that specific moment, by analyzing the answers to questions asked interactively to the user (Jariha and S. K. Jain 2018). This technique is based on the principle that user interests change over time, which would invalidate recommendations obtained by analyzing user history.

Users' current preferences are obtained from the feedback analysis. Users are asked to provide feedback via short/long answer questions, yes/no questions, like/dislike buttons, ratings, etc., and systematic search strategies are used to extract the relevant information. Frequently, predictions are generated after matrix operations or exploratory data analysis executed over the information (Jariha and S. K. Jain 2018).

Context-Based Recommendations

Context-based recommender systems are hybrid approaches that take advantage of contextual information (such as location, time, and social information) at recommendation time to attempt accurate predictions (Jariha and S. K. Jain 2018). This technique has the user-profile at its core and aims to enhance it with the user circumstance and contextual knowledge acquired.

Different implementations of this principle were developed, according to the recommendation business model. Typical implementations of these systems are cultural or entertainment recommendations (e.g., music/movies or travel recommendations). The difficulty in generating knowledge from the contextual information makes this technique complex to implement, despite its promising results and academic interest (Jariha and S. K. Jain 2018).

3.2 Machine Learning Approaches to Recommender Systems

Machine Learning (ML) is a field of Artificial Intelligence (AI), where learning algorithms are used to make inferences from provided data, to learn and execute specific tasks (e.g., generating predictions or making a decision) (Shalev-Shwartz and Ben-David 2014).

The literature describes the learning process as converting experience into knowledge (Shalev-Shwartz and Ben-David 2014), where the training data, used to train the model, represents the experience, and the output represents the knowledge and can be used over unseen data.

Machine learning problems are typically classified according to the nature of the interaction into three main categories: unsupervised learning, supervised learning, and reinforcement learning (Shalev-Shwartz and Ben-David 2014). Details on each of these categories can be found in the following subsections.

3.2.1 Unsupervised Learning

Unsupervised learning is a field of Machine Learning where algorithms are used in order to find previously unknown patterns in unlabeled data (Gama et al. 2015).

Algorithms adapt their behavior based on observations in the data, without specifically being told to. Structures related to datasets' properties can be identified, enhancing their decision-making capability and the ability to learn from uncategorized data (Le et al. 2012). This process consists of the main idea of the self-taught learning framework.

Unsupervised learning can be interesting to different businesses, as they can provide a way to understand datasets in a better way (e.g., understanding customers or providers in a better way). Problems solved by this field of Machine Learning are called descriptive tasks (Gama et al. 2015).

The sections below present some of the most relevant unsupervised learning algorithms to the literature.

Clustering

The literature (Shalev-Shwartz and Ben-David 2014) identifies clustering as one of the most known problems in unsupervised learning. In this technique, the algorithms group unlabeled data into clusters, allowing a bigger understanding of a subject.

The resulting clustered information can, then, be used to execute targeting actions. Retailers, for example, are known to use this information in order to execute targeted marketing campaigns (Shalev-Shwartz and Ben-David 2014).

Autoencoder

Autoencoders are unsupervised learning algorithms that use the encoding and decoding processes to solve problems (Xu et al. 2017). These algorithms compress input data into a code and attempt to generate a representation of the input data.

Because of its simple architecture, autoencoders tend to learn a low-dimensional representation, which can be a better representation of the input data (Xu et al. 2017). Autoencoders are often used as layers of complexity in neural networks trained in unsupervised ways (Shalev-Shwartz and Ben-David 2014).

3.2.2 Supervised Learning

Supervised learning is a field of Machine Learning where, during the training process, both the input and output data of datasets are known. Based on the knowledge built up from a training set, algorithms are able to analyze and predict with high accuracy the outputs for unseen input data in the test set (Gama et al. 2015). These algorithms are said to learn from experience.

In order to train a supervised learning model, n ordered pairs (*xi*, *yi*) are used (Learned-Miller 2014). Each *xi* corresponds to a measurement or set of measurements (often called a vector) of a single example, and *yi* is its label. The algorithm processes this data (trains) and generates a model to predict the output labels of unseen cases - test cases that are structurally similar to the training ones.

Supervised learning tasks can be developed over two distinct types of output labels (Gama et al. 2015): discrete data (e.g., identifying shelves with missing products) and continuous data (e.g., estimating the optimum price of an item). Problems that deal with the first type of data are referred to as classification problems, while the other ones are classified as regression problems.

Classification is a supervised learning variant where cases are assigned with categories. The trained model predicts categories associated with unseen data (Gama et al. 2015). Regression models, on the other hand, generate real or continuous predictions.

In addition to classification and regression problems, the literature refers to deep learning algorithms as supervised implementations with good results (Bengio, Courville, and Vincent 2014). These algorithms take advantage of deep multilayered structures to solve the problems where there is a huge amount of data (Dhumale, Thombare, and Bangare 2019). Deep learning is commonly used in supervised learning (Bengio, Courville, and Vincent 2014), although not strictly (there are unsupervised implementations of deep learning, but in the context of this dissertation, they are not considered).

Grocery retailers can benefit from supervised learning implementations to, among other use cases, identify perishable items while weighting them and calculate the price (Learned-Miller 2014). Problems solved by this field of Machine Learning are called predictive tasks (Gama et al. 2015).

The sections below present some of the most relevant supervised learning algorithms to the literature.

Support Vector Machine

A Support Vector Machines (SVM) is a technique for classification problems solved in supervised learning. These algorithms can be classified as linear and non-linear (Auria and Moro 2011), according to the characteristics of the decision function (i.e., linear and parametric and kernels, respectively).

SVMs use hyperplanes (mathematical functions) to separate the vectorial representations of the training set elements in the space. The distance of the points to the hyperplane is used to chose the best function to perform classifications: the hyperplane with the biggest distance from the closest vector is chosen (He and Chen 2005). If no hyperplane that separates the classes in a clear way can be chosen, the selected hyperplane is the one that minimizes the error (Auria and Moro 2011).

Each vector from the training set has a score associated with it (depending on its distance), which is analyzed during the training phase. Vectors linearly closer to the chosen hyperplane are denominated as support vectors, having more relevance on future predictions (Auria and Moro 2011).

SVM typically provide good recommendations as the model is trained, looking to maximize the generalization capability and performance of the classifier (He and Chen 2005).

K Nearest Neighbor

A K Nearest Neighbor (kNN) is a technique for solving problems using supervised learning, being it classifications or regressions. These algorithms use the closest value in the training set in order to predict a class assigned to an unseen example (Gama et al. 2015).

This technique is associated with expensive computation times, since it requires the entire dataset to be stored and scanned, at test time, to identify the nearest neighbors of an element (Shalev-Shwartz and Ben-David 2014). This way, these algorithms are more performant when trained using datasets composed of elements with a reduced dimension.

K Nearest Neighbour is simple to implement and applicable to problems with different complexities (Gama et al. 2015), making it into one of the most commonly used implementations.

Decision Tree

A decision tree is a technique for solving problems using supervised learning, in both classification and regression techniques. These trees consist of a sequence of linked tests, starting in a root node and continuing through internal nodes until the last leaf node (Anuradha and G. Gupta 2014).

The decision tree is generated during the training phase, where each internal node partitions the instance space according to a function applied to the input value from the training set (Anuradha and G. Gupta 2014). After completing the training process, the tree can generate predictions on unseen data by going across the tree, testing conditions until reaching a confidence point about the input data. This structure enables complex problems to be solved by a set of simpler ones.

It is common to generate ensembles composed of multiple decision trees in order to achieve better results. The principle is to generate a strong learner from a set of weaker ones. The literature presents two predominant techniques: bagging and boosting (Anuradha and G. Gupta 2014).

Bagging is a technique where several subsets are chosen randomly from the training set (Anuradha and G. Gupta 2014). Each of these smaller sets is used to train a decision tree. The average of all the models is used to generate the prediction.

Boosting consists of a set of consecutive weighted classifiers, trained together, aiming to improve the accuracy of the previous one (Anuradha and G. Gupta 2014). Detecting misclassified inputs is translated into an update of the weight to improve future predictions (i.e., the weight of each classifier results of its exactness during the training process). The prediction is generated by a ponderation on a weighted vote of each of the evolved classifiers.

Although decision trees are intuitive prediction algorithms to humans, they are computationally complex to learn. The literature presents multiple heuristics associated with their training processes (Shalev-Shwartz and Ben-David 2014). Also, these algorithms are typically not recommended to scenarios where data is missing, as it will cause some nodes to be empty in the tree (Gama et al. 2015), making it less stable.

Gradient Boosted Trees

Gradient boosted trees are a supervised learning technique used to solve regression and classification problems. Being a boosted technique, it generates predictions after weighing on a set of weighted classifiers trained together (Anuradha and G. Gupta 2014). In addition, this technique is based on the gradient descent algorithm (the algorithm used to find minimums of a function by iteratively taking steps into the negative of the gradient (see Shalev-Shwartz and Ben-David 2014 for more details)).

During the training process, at each step, a decision tree is built in order to fit the residuals of the trees that precede it (Si et al. 2017). Similarly to the gradient descent algorithm, the chosen decision trees are the ones that reduce the error the most (Hastie, Tibshirani, and Friedman 2009). This process stops when the number of iterations matches the value specified for the training or until the number of trees matches the maximum value configured. A prediction is computed using all the decision trees that constitute the ensemble.

Gradient boosted trees are defended by the literature as having promising results because of their high accuracy, training speed, fast prediction time, and reduced memory footprint (Si et al. 2017). This technique is shown as being strong when trained over big datasets. In order to improve further their results with huge amounts of data, modifications can be done, such as the usage of embeddings or even algorithms that extend the classical implementation.

Ada Boosted Tree

Ada boosted trees (from adaptive boosted trees) are a supervised learning technique used to solve regression and classification problems. Similar to gradient boosted trees, they generate predictions after a ponderation on a set of weighted classifiers trained together (Anuradha and G. Gupta 2014).

During the training phase, every time a classifier in the ensemble generates a failed result, its weight is diminished; in addition, it is also reduced on each good result, but on a smaller amount (Rojas 2009). In this technique, a classifier is extracted upon each N iterations (being this value customized before the training process).

This technique deals with the bias induced by weak learners in a natural way, as their weight is reduced in the ponderation (Freund and Schapire 1996). Also, the better the tree behaves, the bigger is the impact of the wrong prediction on the weights. The final prediction is, thus, based on a ponderation between the strongest classifiers.

Artificial Neural Network

An Artificial Neural Network (ANN), commonly referred to as *neural network*, is a supervised learning technique popular in classification or regression problems. Similar to the way a human brain performs, neural networks use communication between simple computing cells (called neurons) to generate a prediction (Haykin et al. 2009).

Neural networks are constituted by three distinct layers: input, output, and hidden layers (Shalev-Shwartz and Ben-David 2014). The first layer is used to process the input and feed the network with initial data; the output layer is responsible for producing the final result; the hidden layer corresponds to the layers located between the initial and the output layers, and it is where most of the communication occurs. When a ANN has a hidden layer composed of two or more layers, it is called a Deep Neural Network (DNN).

Neurons receive as input a weighted sum of the outputs provided by the neurons connected to their incoming edges (Shalev-Shwartz and Ben-David 2014). During the training phase, these synaptic weights of the network are modified in order to reduce the difference between the prediction and the labeled value (Haykin et al. 2009). Commonly, in this phase, the gradient descent algorithm is used to update the network parameters and try to find a global minimum of the cost function.

The network can also modify its own topology, resembling the human brain's neurons, which may die or grow new synapsis (i.e., the principle of plasticity, which corresponds to the need to adapt to its surrounding environment). The adaptation of synaptic weights and the ability to change its topology give neural networks the ability to be easily trained for environments with minor differences.

The literature presents two major classifications for neural networks: feedforward and feedback networks (Haykin et al. 2009; Shalev-Shwartz and Ben-David 2014), differing from the way the communication flows. Feedforward neural networks have information flowing from layer to layer: the input layer feeds the hidden layer, which communicates with the output layer (Gama et al. 2015). Feedback networks can have signals traveling in both directions, using feedback loops (e.g., recurrent neural networks) (Haykin et al. 2009).

Progresses on computational power, data size, and algorithmic advancements have improved the effectiveness of neural networks (Haykin et al. 2009). More complex networks have emerged, including convolutional neural networks, restricted Boltzmann machines, or networks with layers of autoencoders. Because they are out of the scope of this dissertation, these networks are not detailed.

Neural networks are especially valuable because of their parallel and distributed architecture and their capability to learn within very distinct training datasets (Haykin et al. 2009). Their architecture makes ANNs able to provide measurable confidence in the result, in addition to the prediction itself.

Long Short-Term Memory

A Long Short-Term Memory (LSTM) is a deep learning technique used in supervised learning, and it is an application of Recurrent Neural Network (RNN). RNNs are implementations of neural networks with feedback, based on recurrent connections (i.e., signals from previous interactions that are fed back into the network) (Staudemeyer and Morris 2019).

LSTMs can use information from steps far in time (e.g., more than 1000 time steps back) (Staudemeyer and Morris 2019). In order to keep a sense of memory, these networks use memory cells with an internal state (Zhang et al. 2017), which are responsible for carrying relevant information during the flow.

This technique uses three important gates to work: input, output, and forget gates (Staudemeyer and Morris 2019). Input gates control the signals from the network to the memory cell. Output gates are used to control access to the memory cell's content. Forget gates are used to reset the memory cell's internal state in order to clean up unnecessary information.

This technique has shown promising results in dealing with the seasonality of items and the changes in customer's preferences (Staudemeyer and Morris 2019), having the potential to better represent the recurrent nature of grocery shopping patterns.

Linear Regression

Linear regression is a supervised learning technique popular in regression problems. This technique uses a simple mathematical linear algebra function to trace the best line able to separate the elements from the training set (Dhumale, Thombare, and Bangare 2019). Linear regressions are known for being the simplest way to solve a machine learning problem.

Linear regressions include complex loss-functions to dictate how a misclassified value should be penalized since one is dealing with continuous data (i.e., a misclassified value can be measurably close or distant from the training one) (Shalev-Shwartz and Ben-David 2014).

The literature classifies linear regression models into two possible categories: simple linear regressions and multi-linear regressions (Dhumale, Thombare, and Bangare 2019). The first category uses a single dependent-variable in the regression - all other variables are considered independent; The second category uses more than one dependent-variable to generate the regression.

Other Supervised Learning Algorithms

In addition to the previous algorithms, there are other common algorithms used in supervised learning, which are not deeply studied in this dissertation, given its relevance on the matter. In this section, algorithms such as *random forests, naive Bayes, logistic regressions, one-hot encoding, and embeddings* are briefly analyzed.

Random forest is an algorithm used in supervised learning to solve both regression and classification problems. It uses a composition of decision trees in a way that each tree produces its own opinion regarding the input. The final prediction is a ponderation of the output of each composing decision tree (Breiman 2001).

Naive Bayes is a technique for classification problems solved in supervised learning. These algorithms generate a probability of each of the possible outputs and chose the final prediction based on the class with a higher probability (Gama et al. 2015).

Logistic regressions are supervised learning techniques used in regression problems. These techniques generate an estimation of the probability of input matching a class using a multilinear function of the features and convert it into an actual decision (Dhumale, Thombare, and Bangare 2019). This way, logistic regressions are regression analysis that can be applied to classification tasks (Shalev-Shwartz and Ben-David 2014).

One-hot encoding is a technique mainly used in classification problems solved in supervised learning. In this technique, each word from the contextual-vocabulary is encoded into a vector, according to its position. Vectors contain "0"s and "1"s representing the existence or absence, respectively, of each element (Potdar, S., and D. 2017).

Word embedding (or just “embedding”) is a technique mainly used in supervised deep learning techniques, such as deep neural networks, where words are mapped into vectors of real numbers (H.-T. Cheng et al. 2016). This vectorial representation can be generated by classic methods or learned by using neural networks (Grbovic and H. Cheng 2018). Embeddings were originated in natural language processing but have been extended beyond word representations to scenarios such as e-commerce or web searches. Contextual information (e.g., words in a sentence or product details) is often used with embeddings.

3.2.3 Reinforcement Learning

Reinforcement learning is a field of Machine Learning where intelligent agents learn from their own experience and receive signals from an environment in order to help with predictions (Wang and Zhan 2011). These systems include intelligent agents, an environment, environment states, and a notion of reward for actions taken by the agents.

Agents take actions in an environment. The environment feeds back the agent with a reward or punishment signal and an update on its state (Wang and Zhan 2011). This is a continuous process, where actions are chosen according to the feedback provided by the environment. The goal of these systems is to learn an action strategy, which consists of discovering a sequence of actions that maximize the cumulative reward provided by the environment.

Actions are determined according to an algorithm called *policy* (Gron 2017). Depending on the situation, the policy can be a simpler or a more complex algorithm. Common examples used to explore the policy space include neural networks, stochastic algorithms, and genetic algorithms.

Reinforcement learning has gained much interest in 2013, when researchers from DeepMind proved that a computer could play Atari games without prior knowledge of the rules, even outperforming human players (Gron 2017). One of the most important moments of this technique was the victory of the system AlphaGo against Go’s world champion, Lee Sedol, in 2016.

The notion of an environment and the effect it has on the predictions can provide retailers with benefits such as tailoring prices dynamically to each customer (Raju, Narahari, and Ravikumar 2003; Slivkins 2019), which can enhance their competitive power.

The section below details multi-arm bandits, an important reinforcement learning algorithm for recommendations.

Multi-Armed Bandits

A multi-armed bandit (also known as k-armed bandit) is a powerful technique used in reinforcement learning, where the system explores and learns from the environment, according to the pieces of evidence (rewards) it receives (Vermorel and Mohri 2005). Its name and its idea are an analogy to a gambler using multiple slot-machines, which provide different results despite looking identically.

The algorithm has a set of possible actions to chose (called "arms") and a fixed number of rounds to keep training (Slivkins 2019). During the training process, an arm is chosen on each round, and the agent collects a reward from the environment for the action - and the other arms are not evaluated. Also, a reward function is arm-specific.

The process of choosing which arm to explore is the subject of study (Vermorel and Mohri 2005). Multiple algorithms are presented by the literature: e-greedy strategy, where arms are chosen randomly; SoftMax strategy, where a Boltzmann distribution is used to decide; or the interval estimation strategy, where arms are assigned and updated upon each interaction with it.

The algorithm needs to explore the different arms in order to look for the best action strategy (Slivkins 2019). This way, multi-armed bandits learn which arms provide the best balance between rewards and duration.

3.2.4 Learning Methodologies

Machine learning systems can be classified according to their ability to learn incrementally upon new existing data (Gron 2017). In this section, offline and online learning techniques are presented.

Offline Learning

Using offline learning (also known as "batch learning"), models have no capability to learn incrementally (Gron 2017). Models are trained using the existing data and then deployed. After this moment, these models generate predictions based on what they have learned, without further learning anymore. Since deployed models are static instances, they can be horizontally scaled easily.

These algorithms are trained by dividing the dataset into training and test subsets (Vinagre, Jorge, and Gama 2014). The dataset can be split by choosing a portion of shuffled data or according to a moment in time.

Since models are trained using a big set of data, the learning task is usually a slow and computationally expensive process (Gron 2017). It is common to have models training for many hours.

In order to train the model with new data, it is necessary to aggregate both the original and new data and re-train and re-deploy the model (Gron 2017). Since it is a repetitive task, very often this process is automated, and new offline learning solutions are re-trained and re-deployed frequently, automatically.

Offline learning algorithms are typically preferred in controlled scenarios, where data does not vary very often (Vinagre, Jorge, and Gama 2014). Also, since data tends to accumulate with time, there are some situations that may require a decision on filtering part of it.

Depending on the problem requirements and the size of the dataset, using offline learning might not be an option as the training time could be superior to business needs (Vinagre, Jorge, and Gama 2014). Scenarios where using an offline learning model might not be feasible include models for mobile devices and for devices on space missions (Gron 2017).

Online Learning

Using online learning, models have the ability to be trained in an incremental way by feeding data instances individually or in groups (mini-batches) (Gron 2017). This training process typically occurs once the model is already deployed and serving real users (Vinagre, Jorge, and Gama 2014). This characteristic makes online models unable to scale horizontally since learning would be decentralized.

Online learning is of paramount importance in scenarios where the system needs to adapt to the environment in order to change rapidly or autonomously (e.g., stock prices) (Gron 2017). In addition, if there are limitations regarding hardware, this technique is beneficial since the incremental training iterations are computationally inexpensive, and the data can be discarded after the training because the model has already learned it.

Another usage of online learning algorithms is to train models in scenarios where the dataset does not fit in the machine's memory (also known as "out-of-core" learning) (Gron 2017). In these situations, portions of the dataset are iteratively used to train the model until all data is used to feed the model.

The learning rate (i.e., how fast the model should adapt to changing data, has a major role in this type of learning (Gron 2017). High learning rates are translated into a quick change of the results since it adapts quickly to the new data and forgets the old one easier. Low learning rates mean that the system will adapt slower to changes, being, at the same time, less prone to errors due to noise in data (e.g., attacks).

Since these models evolve iteratively, it is possible to perform a more detailed evaluation of their performance by analyzing behaviors that could be unseen in offline learning (Vinagre, Jorge, and Gama 2014). It is common for companies with deployed online learning models to monitor performance and act upon abnormal situations by turning off the learning process (Gron 2017).

3.2.5 Evaluation Metrics

Recommendations predicted by machine learning-based recommender systems are evaluated according to different mathematical metrics in order to evaluate system quality. These metrics are obtained by running the algorithms during the test process.

Evaluating Regression Algorithms

When evaluating regression algorithms, errors can be calculated based on the distance between predicted values and known values from the training set (Gama et al. 2015).

The sections below present some of the most common metrics used when evaluating regression algorithms.

Mean Absolute Error

Mean Absolute Error (MAE) is a popular metric when comparing the predicted rating on an item to the rating that a user has actually provided (Jariha and S. K. Jain 2018). MAE measures the average of the absolute deviance between these two ratings (i.e., the average error on all the predicted ratings) (S. Gupta and Nagpal 2015; Lerato et al. 2016).

MAE is calculated according to the following equation (Jariha and S. K. Jain 2018), where pr_i corresponds to the predicted rating of item i, and ar_i to the actual rating.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|pr_i - ar_i| \tag{3.1}$$

Mean Square Error

Mean Square Error (MSE) is used in replacement of MAE, in order to enhance the scenarios of larger deviance between the predicted rating and the actual one (Lerato et al. 2016).

MSE is calculated according to the following equation (Lerato et al. 2016), where pr_i corresponds to the predicted rating of item i, and ar_i to the actual rating.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(pr_i - ar_i)^2 \tag{3.2}$$

Root Mean Square Error

Root Mean Square Error (RMSE) is a variant of MSE, which gives more importance to larger predictions errors (S. Gupta and Nagpal 2015; Lerato et al. 2016).

RMSE is calculated according to the following equation (Lerato et al. 2016), where pr_i corresponds to the predicted rating of item i, and ar_i to the actual rating.

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(pr_i - ar_i)^2} \tag{3.3}$$

Evaluating Classification Algorithms

When evaluating classification algorithms, errors are based on the correctness and incorrectness of the prediction, when compared to the remaining training set. The performance of these algorithms is based on four indicators (Gama et al. 2015):

- **True Positives (TP)** - number of examples of positive class which were correctly predicted;
- **True Negatives (TN)** - number of examples of negative class which were correctly predicted;
- **False Positives (FP)** - number of examples of negative class wrongly predicted as positive;
- **False Negatives (FN)** - number of examples of positive class wrongly predicted as negative.

The sections below present some of the most relevant measures of classification algorithms, combining the four evaluation indicators.

Accuracy

Accuracy, or success rate, measures the degree of closeness of a prediction (i.e., the number of relevant items, among all the possibilities) (Shalev-Shwartz and Ben-David 2014).

Accuracy is calculated according to the following equation (Jariha and S. K. Jain 2018), where the abbreviations from 3.2.5 are used.

$$Accuracy = \frac{relevant\ predictions}{total\ items} = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.4}$$

Precision

Precision measures the fraction of items in the prediction that is actually relevant (i.e., the number of correctly predicted items in the total number of recommendations) (S. Gupta and Nagpal 2015; Lerato et al. 2016).

Precision is calculated according to the following equation (Jariha and S. K. Jain 2018), where the abbreviations from 3.2.5 are used.

$$Precision = \frac{correct\ predictions}{total\ predictions} = \frac{TP}{TP + FP} \tag{3.5}$$

Recall

Recall measures the fraction of relevant items that were actually recommended over the whole dataset (i.e., the amount of correctly recommended items over the total number of relevant items) (S. Gupta and Nagpal 2015; Lerato et al. 2016).

Recall is calculated according to the following equation (Jariha and S. K. Jain 2018), where the abbreviations from 3.2.5 are used.

$$Recall = \frac{correct\ predictions}{total\ relevant\ predictions} = \frac{TP}{TP + FN} \tag{3.6}$$

F1-Score

F1-score measures the combined effect of both precision and recall on the results, giving equal importance to both, helping to simplify the interpretation of both metrics (S. Gupta and Nagpal 2015; Jariha and S. K. Jain 2018).

F1-score is calculated according to the following equation (Jariha and S. K. Jain 2018).

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{3.7}$$

3.2.6 Comparing Machine Learning Approaches to Recommender Systems

As presented in section 3.1.3, recommender systems can be classified in many different ways depending on the path to the generation of recommendations. Some traditional techniques used in collaborative filtering recommender systems include linear regressions and SVM (Adomavicius and Tuzhilin 2005). Decision trees are often used in both collaborative-filtering and content-based systems. Neural network implementations tend to be used in hybrid approaches. Depending on the context, different implementations can be used in different types of recommender systems.

In addition, machine learning approaches to recommender systems are typically categorized according to the nature of the interactions into unsupervised, supervised and reinforcement learning 3.2. Popular unsupervised learning techniques include clustering and autoencoders; common supervised algorithms include SVM, kNN, decision trees, artificial neural networks, LSTMs, and linear regression models; a powerful reinforcement learning technique in recommender systems is multi-armed bandits. Depending on the problem, recommender systems based on supervised learning can be used to solve regression problems (e.g., predicting a user's rate on an item) or classification problems (e.g., predicting if a user wants an item or not).

Beyond the differences in the characteristics of each algorithm, there are differences in performance. Algorithms perform in different ways depending on many variables (algorithm-related, problem-related, or dataset-related). Most implementations have parameters (also known as "hyperparameters") that can be tuned before the training phase in order to achieve the best results (Olson et al. 2017). These parameters can be as simple as the depth of a decision tree classifier or more complex such as learning rates or settings of the loss function.

Depending on the technique and the implementation, different hyperparameters can be used. The number of trees is a common parameter in decision trees; the number of layers or depth is a typical parameter for neural network implementations; regret is a common parameter in reinforcement learning implementations.

The differences presented on the results of the same algorithms over different datasets and different parameter conditions (Olson et al. 2017, Fernández-Delgado et al. 2014), make it possible to conclude that, although using the proper algorithm for each problem is key, tuning the parameters can lead to the most significant improvements. Spending appropriate amounts of time and effort on this task can lead to better results than testing superficially many different implementations looking for the best results.

Recent developments have been made on the automation of hyperparameter tuning and algorithm selection (often called *automated machine learning* or AutoML) (Komer, Bergstra, and Eliasmith 2014; Thornton et al. 2013). Several approaches exist and target different machine learning frameworks (supporting algorithms often used for recommender systems), but share the common principles of experimenting with different algorithms and hyperparameter configurations to achieve good results. These approaches are capable of achieving good results - sometimes outperforming standard algorithm selection and hyperparameter configuration (Thornton et al. 2013).

3.3 Non-Machine Learning Approaches to Recommender Systems

In addition to Machine Learning, recommender systems can also be developed using tailored heuristics and data mining rules (Adomavicius and Tuzhilin 2005). Depending on the problem and the existing data, some recommender systems can provide satisfactory results with solutions that do not require machine learning algorithms.

Although the main goal of this dissertation is to use a machine learning algorithm to develop a recommender system, these techniques are very valuable. In this section, three non-machine learning implementations are introduced, namely heuristic-based, pattern mining-based, and association rules-based recommender systems.

3.3.1 Heuristic-Based Recommender Systems

Heuristics are customized algorithms designed in order to solve specific problems in a quick, sufficient, and non-optimal way. These algorithms are developed with business rules in mind, and they tend to model knowledge.

If the context in which predictions are generated is simple enough (e.g., recommending the most popular or most recent items), simple heuristics can be used. Heuristics will solve the problem in a quick way and can, probably, be integrated in an easier way with existing technology.

Since grocery retail is a complex field, with many variables and without rational relationships between domain concepts (e.g., between items), a heuristic to generate personalized basket predictions for customers is expected to be complex. Time-related constraints, for instance, need to be addressed (e.g., to prevent suggesting Christmas products in summer). Other examples would include shopping frequency, seasonality, globally favorite items, the time between purchases. Customers with no historical data would need special attention as well.

3.3.2 Pattern Mining-Based Recommender Systems

Pattern mining algorithms can be used to develop recommender systems. Often, periodic and sequential pattern mining algorithms are used to generate basket predictions (Fournier-Viger et al. 2016; Guidotti et al. 2017).

There are multiple periodic pattern mining tasks, aiming at factors such as frequency or utility (Fournier-Viger et al. 2016), and they can be used to build recommender systems for retail. Shopping lists can be generated by applying different periodic pattern miners to shopping history datasets. An effective pattern mining algorithm used in this field is PHM, which aims to efficiently discover periodic high-utility items in a dataset.

Sequential pattern mining implementations rely on discovering recurrent sequences of behavior in the dataset (e.g., sequences of items bought together or recurrency of sequential purchases during a period) (Guidotti et al. 2017). Recommender systems for retail based on this type of pattern mining rely on the principle that customers have individual behaviors that evolve in time.

Frameworks such as XMuSer (Ferreira, Gama, and Santos Costa 2011) use temporal patterns in the form of sequences to discover frequent discriminative patterns using predictive sequence miners. The information perceived as the most important is mapped into a temporal table. This framework also uses an algorithm to learn a way to complete the exploitation of temporal information.

Summing up, recommender systems based on periodic pattern mining algorithms discover temporal patterns within the customer information to predict new items (e.g., preference for certain items during the weekends). Recommender systems based on sequential pattern mining algorithms figure out subsequences within sequences of data (e.g., recurrent purchases of items).

3.3.3 Association Rules-Based Recommender Systems

Association rules represent a branch of data mining with high investigation interest (Jooa, Bangb, and Parka 2016). This technique aims at identifying rules about the relationship between two events in the format of A->B, where the probability of the second part of a rule (i.e., B) existing is higher when the first one (i.e., A) is present. Association rules can be used to develop or improve recommender systems based on relationships between products.

Different algorithms for finding association rules exist and can be applied to grocery retail (Agarwal, Yadav, and Anand 2013). Apriori is a classical implementation of this technique, known for optimizing the way it handles with big amounts of data.

Association rules are filtered according to two traditional criteria: confidence and support (Agarwal, Yadav, and Anand 2013; Jooa, Bangb, and Parka 2016). Some implementations can include other criteria, such as lift, adopted in some implementations such as Apriori (Malik 2020). Confidence measures the likelihood of buying the item sets B when buying the item set A. Support is the ratio of orders with an order on the total number of orders.

The lift measures the number of times an item set B is more likely to be present when item set A is present (and is calculated by the confidence of the rule divided by the support of item set B).

This technique is very flexible when identifying patterns in a transaction dataset. The patterns found on a dataset have different values of the algorithm criteria (e.g., confidence, support, and lift). Also, a certain item set B can be associated with different item sets A.

The ability to find association rules in grocery retail extends the interest of product recommendations. Some rules may have a high interest in business operations, such as aisle sorting or sales. For recommender systems, these rules can be combined with other methodologies to increase the performance of a recommender system (Jooa, Bangb, and Parka 2016), or used by themselves for simple non-personalized suggestion systems (e.g., real-time next-item suggestion).

3.4 Comparing Machine Learning Approaches to Recommender Systems

As observed in section 3.2, different approaches to recommender systems using machine learning exist. In order to understand how they compare when it comes to producing good results, some studies were analyzed.

This section presents the criteria used to evaluate the algorithms and how they differ from each other.

3.4.1 Evaluation Criteria

The criteria for selecting which studies to analyze was the range of algorithms and datasets compared. The main goal was to understand which algorithms achieve the best and the worst results.

Because of the lack of public studies on comparing these algorithms on grocery retail datasets, broader applications were considered. The key concern was to include a notion of user and data associated with it (e.g., movies, e-commerce, or health procedures).

The results achieved by different studies were analyzed in order to understand the most powerful machine learning approaches to recommender systems.

3.4.2 Evaluating Machine Learning Algorithms

Extensive studies on the comparison between the behavior of different machine learning algorithms in multiple fields and problems were performed. In (Fernández-Delgado et al. 2014), 179 classifiers were applied to 121 distinct datasets. These datasets vary from commerce-related data to health-related data. More recently, in (Olson et al. 2017), 13 powerful state-of-the-art algorithms were analyzed on 165 different datasets representing different classification problems.

Despite not being strictly related to retail, these studies have great importance since they include valuable comparisons in multiple algorithms used to create recommender systems and include scenarios with similar data structures and problems. Other studies including valuable information regarding the comparisons between classifiers are also analyzed.

Studies (Olson et al. 2017; Vanschoren et al. 2012) have shown that, with tuned hyperparameters, ensemble-based tree algorithms (namely gradient boosted trees, random forest, and extra trees) tend to show the most promising results. Traditional SVM vectorial implementations have succeeded the first in the results.

Naive Bayes, on the other hand, is shown as achieving the worst results (Olson et al. 2017). The kNN implementations, according to the same study, present middle-range results. Implementations of RNN (namely LSTMs and Gated Recurrent Unit (GRU)s) have also achieved strong comparable results (Liu and Singh 2016; Sheil, Rana, and Reilly 2018).

The studies in (Fernández-Delgado et al. 2014) show a similar trend: they show ensemble-based tree implementations, and SVMs as achieving an average best accuracy. Neural network implementations have also obtained great results. Naive Bayes classifiers and regressions are presented as having obtained the worst results. This study also shows a difference between the behavior of different extensions of the same algorithm (e.g., different extensions of neural networks or different ada boost classifiers).

These studies have confirmed that no algorithm performs the best in all the scenarios nor datasets (Olson et al. 2017). Also, these studies confirm that different classifiers produce different results in the same datasets (Fernández-Delgado et al. 2014). This means that it is of paramount importance to adapt the chosen method to the current problem.

Although some algorithms tend to show better results than others on similar problems, it is common to see the overall less-performing algorithms in a study outperforming the overall best-performing in a specific problem or dataset (Olson et al. 2017).

Summing up, analyzing the results obtained by all these comparisons (Fernández-Delgado et al. 2014; Olson et al. 2017; Vanschoren et al. 2012), one can conclude that boosted tree algorithms, SVMs and neural network implementations present the overall best results. Not only do these algorithms achieve the best average results across the different datasets, but they also achieve lower fluctuations as far as best and worst results are concerned. In addition, kNN and naive Bayes implementations seem to achieve overall lower results.

3.5 Technologies for Machine Learning-Based Recommender Systems

Several technologies or frameworks exist to help developing machine learning-based recommender systems, reducing the effort in the development and deployment of machine learning applications. Depending on the framework, they typically provide implementations for most of the state of the art algorithms.

While some of the existing technologies are paid, the open-source community is big, including software released and supported by big companies (Bloice and Holzinger 2016). Some frameworks provide Application Programming Interface (API) for multiple programming languages and for multiple deployment paradigms (such as traditional servers or mobile phones).

The following subsections present some of the more relevant open-source frameworks to the literature and include a synthetical comparison on them.

3.5.1 TensorFlow

TensorFlow [1] is an open-source library for numerical computation using data flow graphs, created and developed by Google (Nguyen et al. 2019). It is popular in both the academy and industry. TensorFlow was designed for large-scale distributed training and inferences. Since it is based on data flow graphs, it includes nodes and edges: nodes represent mathematical operations; edges represent multidimensional data arrays (also known as “tensors”).

Complementary to the original TensorFlow implementation, TensorFlow Lite, a lightweight distribution, was provided, aiming at mobile and embedded devices (Nguyen et al. 2019). These models have a small size but are known to provide good on-device inferences with low latency. In addition, TensorFlow Lite supports hardware acceleration using Android Neural Networks API.

Despite being written mostly in C++, TensorFlow is available for both Python and C++ (Bloice and Holzinger 2016; Nguyen et al. 2019), including developments for other programming languages ongoing. TensorFlow provides the ability to run models on Central Processing Unit (CPU) and Graphics Processing Unit (GPU) (even in multi-GPU settings), and mobile, and has good scaling capabilities.

TensorFlow is a recent but powerful low-level library, making it possibly harder to use, but capable of many optimizations and control (Bloice and Holzinger 2016).

3.5.2 SciKit-Learn

SciKit-Learn [2] is a general-purpose, open-source tool for Python, containing implementations of the most popular machine learning algorithms (Bloice and Holzinger 2016). It started as a project for a competition, and it is now one of the most popular technologies (especially in the academy) when it comes to developing machine learning models, maintained by INRIA, Telecom ParisTech, and occasionally Google (Nguyen et al. 2019).

SciKit-Learn uses NumPy and SciPy libraries as its core and provides multiple algorithms on top of it as a higher-level tool. This technology is known for being well-updated and supporting many subsets of problems, despite being relatively basic when it comes to neural networks (Nguyen et al. 2019).

This library also provides several convenience utilities for pre-processing tasks (e.g., normalization), as well as built-in sample datasets (Bloice and Holzinger 2016). Despite its multipurpose character, SciKit-Learn has no native support for GPU (Nguyen et al. 2019).

[1]https://github.com/tensorflow/tensorflow
[2]https://github.com/scikit-learn/scikit-learn

3.5.3 PyTorch

PyTorch [3] is an open-source machine learning library, available for Python, based on a dynamic computational graph (Nguyen et al. 2019). It is developed and maintained by the Facebook team, based on Torch, and written mostly in Python and C (Bloice and Holzinger 2016). It is popular in both the academy and industry.

PyTorch supports both CPU and GPU (Nguyen et al. 2019) and edge devices, and it is popular for providing a simple way to build complex solutions. One of its most well-known characteristics is a concept called "reverse-mode auto-differentiation", which consists of enabling a neural network to behave differently by changing configuration without needing to start from scratch.

This library has strengthened up by merging with Caffe2, joining the effort of powerful teams, and the respect of both the academy and industry.

3.5.4 Keras

Keras [4] is an open-source machine learning library, available for Python and popular in both the academy and industry (Nguyen et al. 2019). It works as a wrapper around lower-level technologies, such as TensorFlow, R, or Theano (Bloice and Holzinger 2016). Keras is currently maintained by François Chollet and supported by companies like Google or Microsoft (Nguyen et al. 2019).

In its core, models are represented as a sequence or a graph of stand-alone, adjustable modules that are plugged together (e.g., cost functions or activation functions) (Nguyen et al. 2019). It is possible to add new modules in a simple way.

Despite its design being deep-learning oriented, Keras has the ability to perform other general mathematical computations since it is bound to lower-level frameworks. It is popular for providing simple abstractions to less experienced users and having good documentation - although this comes at the price of being less flexible and modular (Nguyen et al. 2019).

Keras supports both CPU and GPU (Bloice and Holzinger 2016). The support to edge devices depends on the technology Keras is built on top of (e.g., using TensorFlow, it is possible).

3.5.5 Other Popular Technologies

In addition to the previously analyzed technologies, some other important libraries are worth mentioning, either for historical reasons or for its successful results. Some examples include Caffe, Caffe2, Theano, and Weka.

Caffe [5] is an open-source deep learning framework, developed with speed and modularity in mind (Nguyen et al. 2019). It is developed by BAIR and by community contributors. Caffe is very detailed at describing neural networks, using a definition per-layer. It contains multiple utility functions out of the box and provides users with the ability to create their own ones

[3]https://github.com/pytorch/pytorch
[4]https://github.com/keras-team/keras
[5]https://github.com/BVLC/caffe/

using C++. There are multiple forks of Caffe done by big companies, such as Intel Caffe [6] or Nvidia Caffe [7].

Caffe2 [8] is an open-source, lightweight, modular and scalable deep learning framework, based in Caffe, and developed by Facebook (Nguyen et al. 2019). This framework provides support for GPU and edge devices (Caffe2go). This framework was merged with PyTorch, which is analyzed in 3.5.3.

Theano [9] is an open-source, pioneer deep learning framework (Nguyen et al. 2019). It is currently maintained by the University of Montreal, although the end of its development was announced. Theano is available for Python, using NumPy at its core. It is a low-level framework, popular for being very efficient and supporting both CPU and GPU (but not mobile devices).

Weka [10] is an open-source deep learning framework, popular for its graphical user-interface (Nguyen et al. 2019). Weka is developed in Java, and it is maintained by the University of Waikato. It is commonly used in the academic world because of its recognition and simplicity to use (Bloice and Holzinger 2016).

XGBoost [11] is an open-source, distributed a library designed to model flexible and efficient gradient boosted trees (Nguyen et al. 2019). This library can be used in the programming languages C++, Java, Python, R, and Julia. It supports computations on both CPU and GPU, and it is well-known for its performance. Its downside is being restricted to one algorithm.

LibSVM [12] is an open-source library for modeling support vector machines (Nguyen et al. 2019). It can be used with many programming languages such as Java, R, PHP, and Python, and it is popular among the community. Its downsides are problems when scaling and its restrictions to one algorithm.

These technologies are not detailed since, despite their relevance, they are not completely aligned with the purposes of this dissertation, either by not being updated or for not being the most appropriate tool for the studied problem or for the most promising algorithms.

3.5.6 Comparing Technologies for Machine Learning

From the study performed in the previous sub-sections, the table 3.1 was prepared, synthesizing the most relevant technologies according to the literature for the current context. Only open-source technologies were considered in order to guarantee control and the best community support level.

Python is the programming language with a bigger presence among the languages supported by these technologies. All of them are capable of being scaled into production, but not all of them support GPU and edge computations: SciKit-Learn includes support for CPU only

[6]https://github.com/intel/caffe
[7]https://github.com/NVIDIA/caffe
[8]https://github.com/pytorch/pytorch/tree/master/caffe2
[9]https://github.com/Theano/Theano
[10]https://svn.cms.waikato.ac.nz/svn/weka/
[11]https://github.com/dmlc/xgboost
[12]https://github.com/cjlin1/libsvm

and has no support for edge devices; Keras includes support for edge devices depending on the technology it is bound to (e.g., using TensorFlow it is able to support mobile devices).

Recalling the large amounts of data in grocery retail, supporting GPU computations is of paramount importance. In addition, supporting mobile phones can enhance the value of the solution, despite not being a priority.

There are major differences in the way these technologies were developed: TensorFlow is a lower-level library while SciKit-Learn and Keras provide a higher-level abstraction; PyTorch is somewhere in the middle regarding this aspect. Also, all the technologies detailed include good official documentation and powerful support from the community. TensorFlow and Keras occupy the first and last position as far as the repository's activity is concerned, in this order. SciKit-Learn is the technology with less presence when it comes to industrial use.

Some benchmarks (Nguyen et al. 2019) performed on a wide set of machine learning problems and datasets (e.g., IMDb, ImageNet, and MNIST), comparing different frameworks on different algorithms have shown similar accuracy and performance results. The study shows TensorFlow and PyTorch as having obtained slightly better results on average, but not in every scenario.

The slight differences in the results show that there is no clear winner (Nguyen et al. 2019). Although those differences have not shown themselves as significant, the study does not find a clear justification for its origin: it may be related to the framework itself or the implementation of a specific algorithm.

Other benchmarks (V. Kovalev, Kalinovsky, and S. Kovalev 2016) performed using different frameworks to build a deep learning solution for a classification problem (on Digits dataset) have found similar results: there are differences in performance, accuracy, and even training times, but they are residual. Keras and TensorFlow tend to fight for the first place, on average, but it is also not a rule.

In addition, this study has compared the lines of code spent to solve the same problems and concluded that lower-level frameworks (e.g., TensorFlow or Theano) require fewer lines of code. Again, as in the benchmark performed in (Nguyen et al. 2019), the differences may have to do with the framework itself.

This table synthesizing how the different technologies handle different criteria does not produce a winner nor a loser, because all the technologies behave similarly. The benchmarks regarding their behavior do not present major differences but show TensorFlow and Keras as slightly more performant technologies. Also, the lack of support for GPU and mobile computations, and its reduced use in industry, make SciKit-Learn into a vulnerable situation.

TensorFlow, Keras, and PyTorch are, thus, the technologies that are more prepared for the current scenario, according to the literature.

Table 3.1: Comparison between TensorFlow, SciKit-Learn, Keras and PyTorch

	TensorFlow	**SciKit-Learn**	**Keras**	**PyTorch**
What programming languages does it support?	C++, Python	Python	Python, R	Python
Is it able to scale in production?	Yes	Yes	Yes	Yes
Is it open-source?	Yes	Yes	Yes	Yes
Does it support both CPU and GPU?	Yes	No, CPU only	Yes	Yes
Does it include support for edge devices?	Yes	No	Yes, depending on the technology	Yes
Is it a low-lever or high-level technology?	Low-level	High-level	High-level	“Mid”-level
Does it have good documentation?	Yes	Yes	Yes	Yes
Does it have good support from the community?	Yes	Yes	Yes	Yes
Where is it mostly used?	Academy and Industry	Academy	Academy and Industry	Academy and Industry

3.6 Summary

Recommender systems use historical data in order to generate predictions (Adomavicius and Tuzhilin 2005). To build such systems, heuristics, data mining, and machine learning algorithms can be used. Some issues may arise when training a prediction model, mostly related to the data quality or quantity, the chosen approach, or the training methodology (Jariha and S. K. Jain 2018). Typical issues include cold-starts, sparsity, overfitting, scalability, or long-tail. Authors classify recommender systems according to three major categories: content-based recommendations, collaborative recommendations, and hybrid recommendations (Adomavicius and Tuzhilin 2005). Examples of hybrid recommendations are utility-based, knowledge-based, demographic-based, interactive, and context-based recommendations (Burke 2002; Jariha and S. K. Jain 2018).

In machine learning, algorithms make inferences from data to learn and generate predictions (Shalev-Shwartz and Ben-David 2014). Learning is described as converting experience into knowledge, training data corresponds to the experience, and the output of a model over unseen data is considered knowledge.

According to the nature of the interaction, machine learning problems are typically classified into three main categories: unsupervised learning, supervised learning, and reinforcement learning (Shalev-Shwartz and Ben-David 2014). In unsupervised learning, algorithms are used to find patterns in unlabeled data. In supervised learning, models learn from datasets, where both the input and output information is known (Gama et al. 2015). Supervised learning problems can be categorized as classification problems, regression problems, or deep learning (Bengio, Courville, and Vincent 2014). In reinforcement learning, intelligent agents learn from their experience and receive signals from an environment to help with the decisions (Wang and Zhan 2011).

Machine learning systems can also be classified according to their ability to learn upon new data (Gron 2017). When a model cannot learn incrementally when deployed, it is called an "offline" or "batch" algorithm. On the other hand, if the model can learn in an incremental way when deployed, it is called an "online" algorithm.

The recommendations predicted by a machine learning model can be evaluated according to different mathematical metrics obtained. When evaluating regression algorithms, typical metrics include: mean absolute error, mean square error, and root mean square error (S. Gupta and Nagpal 2015; Lerato et al. 2016). When evaluating classification algorithms, typically, four indicators are used (Gama et al. 2015): true positives, true negatives, false positives, and false negatives. These indicators are used to calculate important metrics, such as accuracy, precision, recall, and f1-score (S. Gupta and Nagpal 2015; Shalev-Shwartz and Ben-David 2014).

Studies have shown that, despite being essential to adjust the algorithm to the problem being solved, the performance and results achieved by a machine learning model can be highly improved by manipulating the hyperparameters of the algorithm (Fernández-Delgado et al. 2014; Olson et al. 2017).

Besides the many machine learning approaches existent, there are also recommender systems developed using approaches like tailored heuristics, pattern mining, and association rules (Adomavicius and Tuzhilin 2005; Jooa, Bangb, and Parka 2016). Heuristics are customized algorithms designed to solve a specific problem, being typically fast but non-optimal. Pattern mining models are used to discover temporal patterns within the information or figure out subsequences within sequences of data (e.g., recurrent purchases). Association rules represent relationships between items of a dataset, allowing the identification of items or sets of items that are more likely to be bought when other certain items are bought.

Several studies comparing the results achieved by different machine learning algorithms in multiple problems and datasets with different hyperparameter tunings have achieved similar results. The results obtained in (Olson et al. 2017; Vanschoren et al. 2012) have shown that ensemble-based tree algorithms (namely gradient boosted trees, random forests, and extra trees) tend to obtain the most promising results. Vectorial implementations (namely SVMs) occupy the next place on the podium. Neural networks also present good results. Naive Bayes, on the other hand, is shown as achieving the overall worst results among all the compared techniques (Olson et al. 2017).

Multiple technologies or frameworks can be used to develop machine learning algorithms for recommender systems. The most relevant technologies, according to the literature, include TensorFlow (Nguyen et al. 2019), PyTorch, Keras, and SciKit-Learn (Bloice and Holzinger 2016).

Chapter 4

Solution Description

This chapter describes the solution proposed in this dissertation for a machine learning-based recommender system for grocery retail. It starts with a requirement analysis, where domain concepts, functional and non-functional requirements are analyzed. The solution design is, hereafter, described, where the different architectural proposes are presented and analyzed, use cases are designed, a deployment view is proposed, the modeling methodology is described, and technologies presented. A summary of all the presented topics is present at the end of this chapter.

4.1 Requirement Analysis

This section presents an analysis of the different requirement artifacts. It is started by an analysis of the domain concepts associated with a recommender system for grocery retail, translated into a domain model. Functional requirements are then presented as use cases. The section is concluded by an analysis of the non-functional requirements, based on the FURPS+ model (Eeles 2004).

4.1.1 Domain Concepts

A recommender system for grocery retail deals with different actors, entities, and interactions throughout the different steps to provide custom recommendations to a specific customer.

The most important roles are played by two main actors: customers and the recommender system. Customers are responsible for starting the recommendation request and for viewing the results. The system is responsible for keeping the information up to date, training the model periodically, and providing a way for customers to have tailored recommendations.

Retailers could also be seen as actors, but since they do not execute a specific action in the solution, they are not considered. Retailers are only responsible for providing access to information consumed by the system.

The figure 4.1 represents a possible representation for the domain model of a grocery retail recommender system, with the main domain concepts and the interactions between them.

Customers perform different orders, containing several products. Each product belongs to a certain department and a certain aisle. A shopping history represents several customers and all their orders.

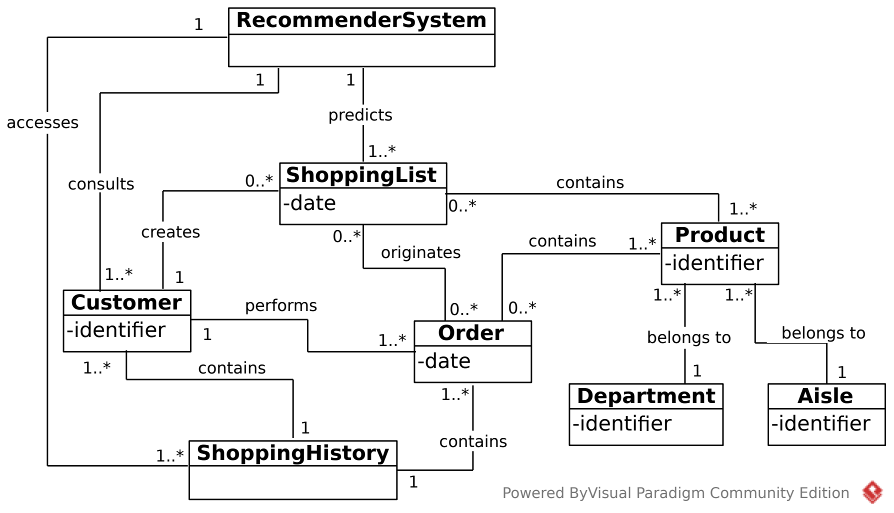


Figure 4.1: Domain model for a recommender system in grocery retail

In addition to orders, customers can also prepare shopping lists containing different products. These shopping lists can also be turned into orders.

The recommender system accesses the shopping history, containing information regarding customers and orders. It is also able to predict shopping lists for the different customers.

4.1.2 Functional Requirements

Functional requirements describe the main characteristics of the system and its core functionalities, and they were identified with the proponent organization. Each functional requirement is represented as a Use Case (UC) and referenced by the first letter of its actor and by a number (e.g., UC_C1 is the first use case and has the customer as the actor, and UC_S3 is the third use case and has the recommender system as the actor).

Customer Use Cases

- UC_C1 - View a shopping list recommendation
- UC_C2 - View an item recommendation

Recommender System Use Cases

- UC_S3 - Update the shopping history
- UC_S4 - Train and update the model

Use case UC_C1 is the main use case in the solution, responsible for generating a recommendation for a complete shopping list for a specific customer, representing its needs at a particular time. Use cases UC_S3 and UC_S4 are crucial for the solution to work, as they represent the support processes of acquiring the data, adapting it to a format feasible to be learned by the machine learning algorithm and the training itself, followed by the update of the model used.

Use case UC_C2, however, represents a functionality that can value the solution but is not the main target of this dissertation. By providing this feature, the recommender system would be stronger as it would suggest shopping lists (target of this dissertation) and the next-items to buy.

The use cases of the solution are presented below, grouped by their actor. They are also synthesized in the use case diagram in figure 4.2. Also, details on the realization of each use case can be found in section 4.2.2.

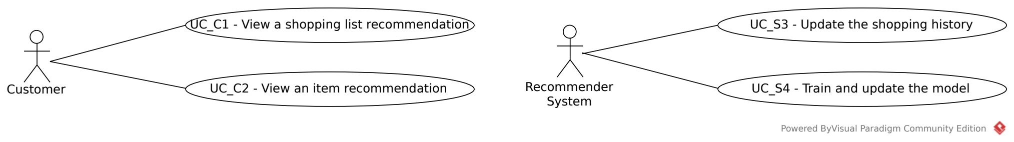

Customer
UC_C1 - View a shopping list recommendation
UC_C2 - View an item recommendation
Recommender System
UC_S3 - Update the shopping history
UC_S4 - Train and update the model
Powered ByVisual Paradigm Community Edition

Figure 4.2: Use case diagram for the dissertation

4.1.3 Non-Functional Requirements

Non-functional requirements are analyzed using the FURPS+ model, which succeeds the FURPS model (Eeles 2004). In addition to the original non-functional requirements (usability, reliability, performance, and supportability), the successor model brings the specification of design, implementation, interface, and physical constraints. Despite the solution being more oriented to a concrete problem and result, non-functional requirements assume an important role in this solution since they capture how the system is expected to behave.

The sections below present the non-functional requirements for this project, presented using the FURPS+ model.

Usability

Usability represents the requirements based on the interface with the user (Eeles 2004). Since the solution is a module capable of being integrated by different stakeholders, its core is not to include a graphical user interface - instead, the interface assumes the form of an API. The usability requirements for this project are presented below.

1. The communication interface should be clearly documented and simple to use;
2. The system should treat errors and unexpected situations smoothly, providing clear details regarding the operations.

Reliability

Reliability represents how a system is said to behave, measuring aspects such as its availability, precision, and failure recovering (Eeles 2004). The reliability requirements for this project are presented below.

1. The system should keep itself available after a previous error;
2. The system should treat errors that may occur inside internal operations/requests, without propagating it to other components.

Performance

Performance represents how a system behaves under different usage loads, including concerns such as response times, load time, and failure recovering (Eeles 2004). The performance requirements for this project are not measurable in a quantifiable way since there are no specific requirements. However, it is valuable for the system to be able to be used without disrupting the user experience on the integrator side. This way, the performance requirements for this project are simplified below.

1. The system should provide recommendations in a feasible amount of time, in order not to deteriorate the user experience;
2. Metrics about the way a model performs against a training set should be captured and evaluated.

Supportability

Supportability represents how a system is concerned about language limitations, testability, adaptability, maintainability, compatibility, and scalability (Eeles 2004). The supportability requirements for this project are presented below.

1. The system should be ready to a replacement of the machine learning model with another one, without affecting other components or integrators;
2. The system should be ready for integration with retailers with different historical data representations.

Design Constraints

Design constraints limit the way a system is designed upfront (Eeles 2004). The design constraints for this project are presented below.

1. The recommendations engine should be developed using a machine learning algorithm;
2. The recommendations should be abstracted behind an API;
3. The architecture should support periodic dataset updates and training of new models;
4. The usage of an online or offline approach should be based on the ease of integrating with retailers.

Implementation Constraints

Implementation constraints affect the way a solution is built, possibly impacting the solution design (Eeles 2004). The implementation constraints for this projects are listed below.

1. No personal information regarding a customer should be used by the solution - only a unique identifier is allowed;
2. Unit, integration and end-to-end tests should be developed with the solution.

Interface Constraints

Interface constraints limit the way a component communicates with another one (Eeles 2004). The interface constraints for this project are presented below.

1. The recommendations API should provide a REpresentational State Transfer (REST) interface.

4.2 Solution Design

This section presents the solution designed after understanding the requirements. It starts by presenting alternatives for the architecture to be used and considerations of its choice. It is followed up by the realization of each use case, an analysis of the deployment view for the chosen architecture, and the modeling methodology adopted. It is concluded by a presentation of the technologies used for each important module.

4.2.1 Architecture

From the requirements, one can conclude that the system needs to take care of two main activities: acquire updated data regarding the customer's shopping history and suggest personalized shopping list recommendations.

To accomplish the data acquisition process, the system needs access to data exposed by retailers, containing their customer's shopping history, either via private APIs or private data storages. This information is then processed within the system itself and used to train a machine learning model. The model learns from this information and uses it to generate predictions upon future requests.

For customers to view personalized recommendations, the model must be exposed by a private API that encapsulates its internal logic. In addition, to provide an extra layer of security, authorization, and, if necessary, request caching, an API should be exposed. This should be the main entry point with which customers interact with.

The core modules of the architectures proposed are, thus:

- **Customer Device** - The device used by the customer to interact with the system - despite being an external module, it is simulated in this dissertation;
- **Grocery Retailer API** - API provided by the retailer to expose their customer's data - despite being an external module, it is also simulated in this dissertation;
- **Customer Loyalty DB** - Retailer's Database (DB), representing its customer's loyalty information;
- **Customer Authorization Service** - Retailer's authorization service, used by the recommender system to make sure the customers are authorized to access the recommendations - despite being an external module, it is simulated in this dissertation;
- **Recommendations Front-end Service** - Main API, responsible for exposing the recommender system to retailers in a secure and clear way;

- **Machine Learning Model** - Component responsible for generating tailored predictions for a customer; trained and used by the Machine Learning Model Interface;
- **Machine Learning Model Interface** - API responsible for training and exposing the Machine Learning Model, and updating the internal Shopping History - accessible only for scheduled support tasks or via the Recommendations Front-end Service;
- **Shopping History** - Internal representations of the shopping history, used to train the model and generate predictions for the customers.

With the different architectural modules identified, different architectures were thought, analyzed, and designed. This section includes three different architectures that were explored in order to solve this problem: an offline learning approach (see section 3.2.4), and two online learning approaches differing in the way they provide data updates (see section 3.2.4).

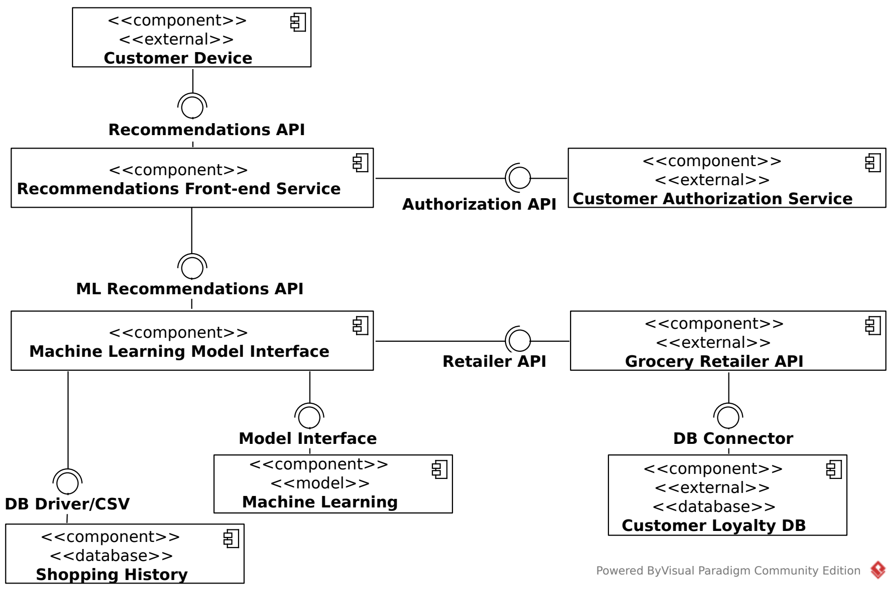
Architectural Proposal 1

This architecture proposal, represented in figure 4.3, represents an offline learning approach (see section 3.2.4). In this alternative, the learning process occurs via a scheduled task responsible for acquiring new data and training (and evaluating) a new version of the model.

Grocery retailers, in this approach, provide an API that exposes loyalty information regarding their customers (component Grocery Retailer API). This module is consumed by the Machine Learning Model Interface, which digests it and stores it in the local Shopping History.

The generated model (represented via the component Machine Learning Model) is exposed via its private API (component Machine Learning Model Interface), and it is consumed by the public Recommendations Front-end Service.

Customers interact with the system by invoking the public Recommendations Front-end Service, which verifies its authorization against the Customer Authorization Service. This service calls, after successful validation, the Machine Learning Model Interface, which then interacts with the Machine Learning Model to generate predictions.



<<component>> <<external>> Customer Device
Recommendations API
<<component>> Recommendations Front-end Service
Authorization API
<<component>> <<external>> Customer Authorization Service
ML Recommendations API
<<component>> Machine Learning Model Interface
Retailer API
<<component>> <<external>> Grocery Retailer API
Model Interface
<<component>> <<model>> Machine Learning
DB Connector
<<component>> <<external>> <<database>> Customer Loyalty DB
DB Driver/CSV
<<component>> <<database>> Shopping History
Powered ByVisual Paradigm Community Edition

Figure 4.3: Architecture proposal number 1 (adopted architecture)

Architectural Proposal 2

This architecture proposal, represented in figure 4.4, represents an online learning approach (see section 3.2.4). In this alternative, the learning process is done after a specific call from the retailer API to the Recommendations Front-end Service, informing about new data. This data is then processed, validated, and sent to the Machine Learning Model Interface, where it is used to train the model. Since this is an online approach, no new model would be generated nor deployed, just updated.

Grocery retailers, in this approach, still provide an API that exposes loyalty information regarding their customers (component Retailer API). This module is consumed by the Machine Learning Model Interface, which digests it and stores it in the local Shopping History. It is mainly used for initial model training.

The generated model (representing via the component Machine Learning Model) is still also exposed via its private API (component Machine Learning Model Interface), and it is consumed by the public Recommendations Front-end Service.

Customers interact with the system by invoking the public Recommendations Front-end Service, which verifies its authorization against the Customer Authorization Service. This service calls, after successful validation, the Machine Learning Model Interface, which then interacts with the Machine Learning Model to generate predictions.

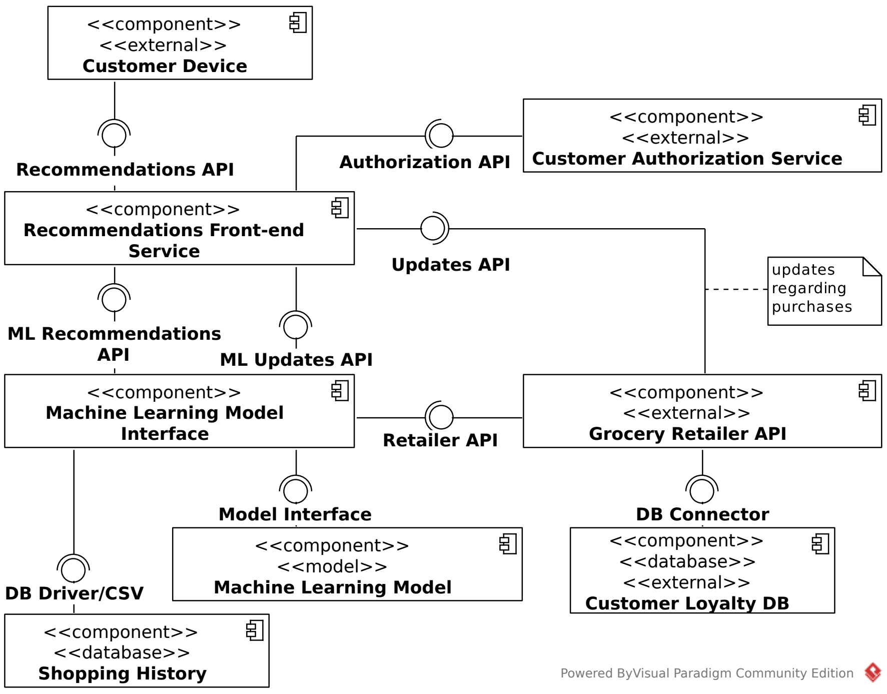

<<component>> <<external>> Customer Device
Recommendations API
Authorization API
<<component>> <<external>> Customer Authorization Service
<<component>> Recommendations Front-end Service
Updates API
updates regarding purchases
ML Recommendations API
ML Updates API
<<component>> Machine Learning Model Interface
Retailer API
<<component>> <<external>> Grocery Retailer API
Model Interface
DB Connector
<<component>> <<model>> Machine Learning Model
<<component>> <<database>> <<external>> Customer Loyalty DB
DB Driver/CSV
<<component>> <<database>> Shopping History
Powered ByVisual Paradigm Community Edition

Figure 4.4: Architecture proposal number 2

Architectural Proposal 3

This architecture proposal, represented in figure 4.5, represents also an online learning approach (see section 3.2.4). In this alternative, the learning process is done on-the-fly, while the customer is interacting with its Customer Device, running software from the retailer.

In this scenario, the software is responsible for informing the Recommendations Front-end Service about the creation of a new purchase. This data is then processed, validated, and sent to the Machine Learning Model Interface, where it is used to train the model, similarly to alternative 2 (figure 4.4). Since this is an online approach, no new model would be generated nor deployed, just updated.

Grocery retailers, in this approach, still provide an API that exposes loyalty information regarding their customers (component Grocery Retailer API). This module is consumed by the Machine Learning Model Interface, which digests it and stores it in the local Shopping History. It is mainly used for initial model training.

The generated model (representing via the component Machine Learning Model) is still also exposed via its private API (component Machine Learning Model Interface), and it is consumed by the public Recommendations Front-end Service.

Customers interact with the system by invoking the public Recommendations Front-end Service, which verifies its authorization against the Customer Authorization Service. This service calls, after successful validation, the Machine Learning Model Interface, which then interacts with the Machine Learning Model to generate predictions.

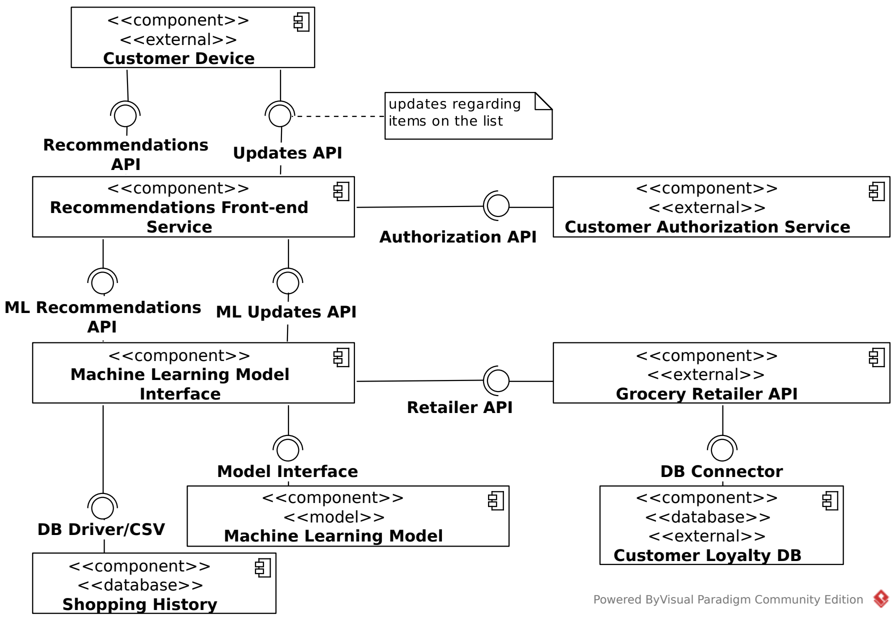


Figure 4.5: Architecture proposal number 3

Adopting an Architectural Proposal

As detailed in the previous sections, the main difference regarding the first proposal (figure 4.3), the second (figure 4.4) and the third one (figure 4.5) is the learning methodology (cf. 3.2.4). The first alternative uses an offline learning approach, while both the second and third alternatives use online approaches.

As studied in section 3.2.4, both learning alternatives provide good results. The frequency of data updates is something retailer-specific, which is important to consider, as it may be translated into a difficulty to take advantage of online approaches.

Also, all the alternatives require an API from the grocery retailer to expose its shopping history. The alternatives 2 and 3 require an additional effort from the retailer to publish updates regarding new shopping transactions. Alternative 1 deals with this in a less-efficient way, but comparably effective, since the Machine Learning Model Interface consumes the Grocery Retailer API to acquire new information, just like it does while training the model.

With this constraints in mind, alternative 1 (see section 4.3) is adopted. The proponent organization supports the decision.

4.2.2 Use Cases Realizations

This section presents the use-cases realization for the use cases identified in section 4.1.2. For each of these uses cases, a sequence diagram is presented, helping the identification of the components and interactions designed to fulfil the requirement (Bittner 2016).

View a Shopping List Recommendation

Obtaining a shopping list recommendation is the target of this dissertation and the core of the solution, and corresponds to the use case UC_C1 (see section 4.1.2). It represents the interaction of a customer using the retailer's specific software to view a shopping list recommendation.

The figure 4.6 represents the sequence diagram that describes this process. The customer starts the flow using its own device, and a call to the public Recommendations Front-end Service is executed. The Recommendations Front-end Service validates the authorization against the Customer Authorization Service and, if it is valid, executes a request to the Machine Learning Interface. This API loads the machine learning model and uses it to obtain a prediction for a shopping list. The information is fed back to the customer.

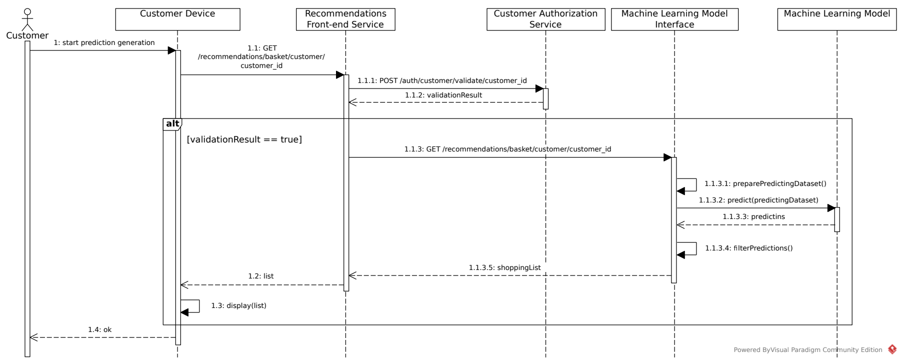

Customer
Customer Device
Recommendations Front-end Service
Customer Authorization Service
Machine Learning Model Interface
Machine Learning Model
1: start prediction generation
1.1: GET /recommendations/basket/customer/customer_id
1.1.1: POST /auth/customer/validate/customer_id
1.1.2: validationResult
alt
[validationResult == true]
1.1.3: GET /recommendations/basket/customer/customer_id
1.1.3.1: preparePredictingDataset()
1.1.3.2: predict(predictingDataset)
1.1.3.3: predictins
1.1.3.4: filterPredictions()
1.1.3.5: shoppingList
1.2: list
1.3: display(list)
1.4: ok
Powered ByVisual Paradigm Community Edition

Figure 4.6: Sequence diagram for use case UC_C1

View an Item Recommendation

Generating a prediction for the next item in the list is not a core functionality of this solution. However, it is something that could be accomplished by training another machine learning model, analogously to what it is done for a full-sized list. It is represented by the use case

UC_C2 (see section 4.1.2). The use case represents the interaction of a customer, using the retailer's specific software, in order to see desired items and obtain item recommendations.

The figure 4.7 represents the sequence diagram that describes this process. The customer starts the flow using its own device, after using the retailer software to express interest in an item (e.g., by viewing it or adding it to a shopping list). At this moment, a call to the public Recommendations Front-end Service is executed. The Recommendations Front-end Service validates the authorization against the Customer Authorization Service and, if it is valid, executes a request to the Machine Learning Interface, informing about the user and the current item. This API loads and uses the machine learning model to obtain a prediction for the next item to recommend. The information is fed back to the customer.

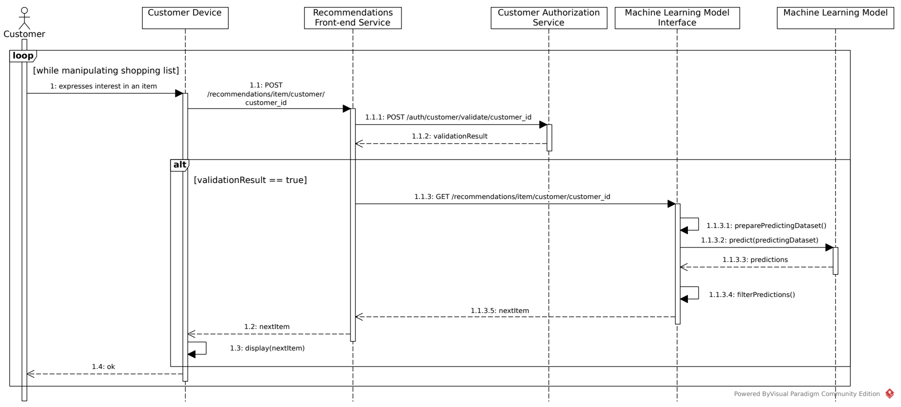

Customer
Customer Device
Recommendations Front-end Service
Customer Authorization Service
Machine Learning Model Interface
Machine Learning Model
loop
[while manipulating shopping list]
1: expresses interest in an item
1.1: POST /recommendations/item/customer/customer_id
1.1.1: POST /auth/customer/validate/customer_id
1.1.2: validationResult
alt
[validationResult == true]
1.1.3: GET /recommendations/item/customer/customer_id
1.1.3.1: preparePredictingDataset()
1.1.3.2: predict(predictingDataset)
1.1.3.3: predictions
1.1.3.4: filterPredictions()
1.1.3.5: nextItem
1.2: nextItem
1.3: display(nextItem)
1.4: ok
Powered ByVisual Paradigm Community Edition

Figure 4.7: Sequence diagram for use case UC_C2

Update the Shopping History

Obtaining updates regarding shopping history is a crucial part of the flow. This functionality is represented by the use case UC_S3 (see section 4.1.2), and it represents the interaction of an automatic task with the Machine Learning Model Interface. This task triggers the update of the internal Shopping History component with the new data.

Figure 4.8 represents the sequence diagram that describes this process. The Machine Learning Model Interface executes a request to Grocery Retailer API in order to obtain new historical data. Since this component is retailer-specific, it does internal logic in order to provide access to the requested information and return it. The Machine Learning Model Interface digests the new data and updates the internal Shopping History.

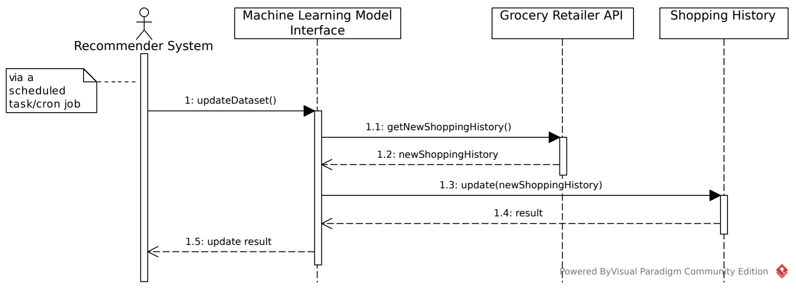

Recommender System
via a scheduled task/cron job
Machine Learning Model Interface
Grocery Retailer API
Shopping History
1: updateDataset()
1.1: getNewShoppingHistory()
1.2: newShoppingHistory
1.3: update(newShoppingHistory)
1.4: result
1.5: update result
Powered ByVisual Paradigm Community Edition

Figure 4.8: Sequence diagram for use case UC_S3

Train and Update the Model

Training a new model and updating the running instance is of paramount importance to the problem since it allows the recommender system to know new data and improve. This functionality is represented by the use case UC_S4 (see section 4.1.2). This use case represents the interaction of an automatic task with the Machine Learning Model Interface, which uses the existing data to train, evaluate, and deploy a new model.

Figure 4.9 represents the sequence diagram that describes this process. The Machine Learning Model Interface uses the data in the internal Shopping History component to extract features and prepare the training dataset. This dataset is used to train a new model and a portion of it to evaluate its quality. If the model meets the quality criteria (e.g., performs better than the previous one), it can be deployed - replacing the previous version.

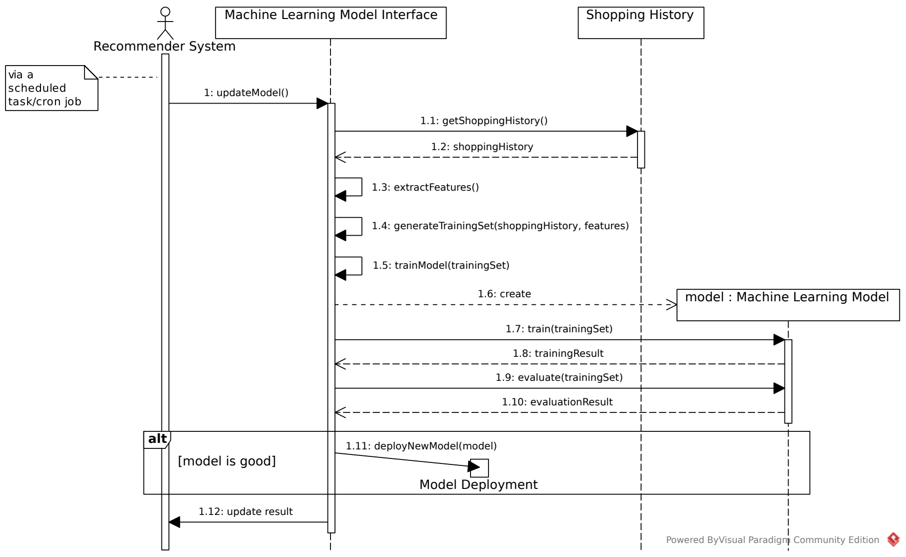

Recommender System
Machine Learning Model Interface
Shopping History
via a scheduled task/cron job
1: updateModel()
1.1: getShoppingHistory()
1.2: shoppingHistory
1.3: extractFeatures()
1.4: generateTrainingSet(shoppingHistory, features)
1.5: trainModel(trainingSet)
1.6: create
model : Machine Learning Model
1.7: train(trainingSet)
1.8: trainingResult
1.9: evaluate(trainingSet)
1.10: evaluationResult
alt
[model is good]
1.11: deployNewModel(model)
Model Deployment
1.12: update result
Powered ByVisual Paradigm Community Edition

Figure 4.9: Sequence diagram for use case UC_S4

4.2.3 Deployment

After adopting an architectural approach (see section 4.2.1), the solution deployment was designed. Figure 4.10 represents a deployment view for the adopted architectural proposal.

Since the customer is expected to use a retailer's specific software, this module is generically represented in its own environment. The Recommendations Front-end Service, since it assumes a major role in the solution (mostly regarding performance and security constraints), is deployed in its own Linux Server. It accepts communication using HTTP requests.

Both the Authorization API and the Retailer API are external components to the solution, being, thus, represented as individually deployed in their specific machines. Both of these components are expected to accept HTTP requests.

The Machine Learning Module, as well as its API and Shopping History, are planned to be deployed inside another Linux Server. It is also expected to accept HTTP requests.

The Linux servers should preferably run a Ubuntu distribution but could equally be a container running on some virtualization infrastructure or cloud service.

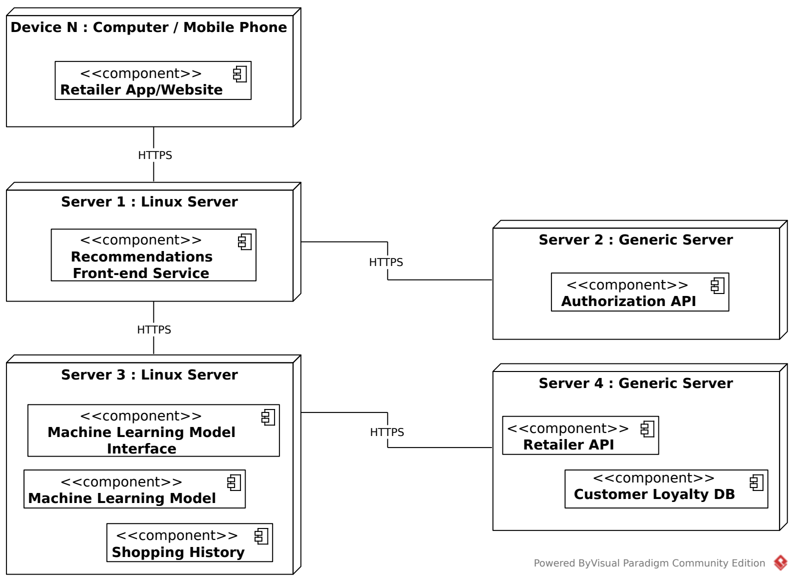


Figure 4.10: Deployment view of the solution

4.2.4 Machine Learning Methodology

To facilitate the different processes involved, from understanding the problem to deploying a final solution, the solution development was based on and adapted from the CRISP-DM (CRoss Industry Standard Process for Data Mining) methodology (Wirth and Hipp 2000).

This methodology was proposed for the development of data mining projects, independent from business and sectors, and presents a reliable and efficient iterative and cyclic flow that can be adapted to different situations (Wirth and Hipp 2000). It can also serve as a base for machine learning projects. The figure 4.11 describes its different phases and the relationships between them.

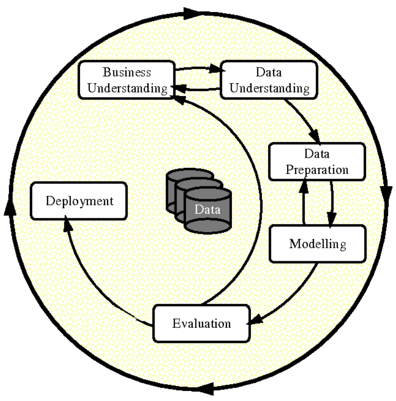


Figure 4.11: Phases of the CRISP-DM model (Wirth and Hipp 2000)

Business Understanding corresponds to understanding the problem, goals, and requirements, and converting them into a data-related problem (Wirth and Hipp 2000) - in this dissertation, it can be seen when understanding the problem and designing a generic solution. Data Understanding corresponds to collecting, exploring, and verifying the quality of data (Wirth and Hipp 2000) - it can be seen when collecting and exploring a retailer's dataset.

Data Preparation corresponds to the data selection, cleaning, construction of new attributes, and transformation (Wirth and Hipp 2000) - it is present when analyzing the data and engineering features for a retailer dataset, during integration. Modeling corresponds to selecting and applying a model and calibrating the parameters (Wirth and Hipp 2000) - it can be seen when adopting a machine learning model and tuning its hyperparameters.

Evaluation corresponds to the evaluation and review of the used models (Wirth and Hipp 2000) - it can be seen when evaluating the quality of the solution on a retailer's dataset. Deployment corresponds to organizing and presenting the knowledge so that the customer can use it (Wirth and Hipp 2000) - in this dissertation, it is seen when deploying the machine learning model as part of the recommender system's architecture.

4.2.5 Adopted Technologies

The study on the state of the art of frameworks for the development of machine learning models has led to the identification of three possible technologies: Tensorflow, Keras, and PyTorch (see section 3.5.6). TensorFlow was the adopted framework as it has excellent recognition in the field and a big community around it, and it supports an ample amount of machine learning algorithms. The proponent company and the advisors also supported this choice.

The study on the state of the art of algorithms for machine learning models (see section 3.4.2) has evidenced three types of algorithms as being the most promising ones to deal with the flexibility and agitation associated with grocery retail (see section 3.2): gradient boosted trees, SVMs and neural networks. The proposed architecture is machine learning model-agnostic, thus the solution is adaptable to different TensorFlow machine learning algorithms. The choice should be based on comparisons between the different approaches, when integrating with a retailer dataset (as performed in the case study of section 6.2).

The chosen programming language for the TensorFlow model and the different services was Python because of its flexibility, simplicity, and recognition in the field. Some auxiliary scripts for the different testing moments were done using both PHP and Python. A demonstrator (simulating the software in a customer device) was developed using JavaScript and HTML. Unit and functional tests of the services were developed using pytest, which is a simple testing framework for Python.

The services expose HTTP APIs using the internal Python HTTP modules. These modules provide a multi-threaded HTTP server, allowing these APIs to assure responsiveness. The different services communicate using JavaScript Object Notation (JSON). Data operations are performed using the frameworks pandas and NumPy, two Python frameworks that combined provide access to mathematical operations over large datasets.

The main technologies can also be found synthesized in the appendix A.

4.3 Summary

A recommender system for grocery retail deals with different entities and interactions throughout the different steps. The most important roles are played by two actors: customers and the recommender system itself.

Customers perform orders with different products. Several customers and their orders on a retailer represent a shopping history. Customers can also prepare shopping lists, which can be translated into orders. They can also interact with the recommender system in order to obtain tailored recommendations. The recommender system, on the other hand, has access to the shopping history so it can predict shopping lists.

Functional requirements are addressed as use cases, and they are split into two different types: customer use cases and recommender system use cases. The use cases for customers are: view a shopping list recommendation (UC_C1); view an item recommendation (UC_C2). The use cases for the recommender system are: update the shopping history (UC_S3); train and update the model (UC_S4). Use case UC_C2 represents an additional functionality designed to value the solution

Non-functional requirements are addresses using the FURPS+ model, which succeeds the FURPS model (Eeles 2004). Despite the solution being more oriented to a concrete problem, different non-functional requirements are identified, belonging to the FURPS+ categories of usability, reliability, performance, supportability, design constraints, implementation constraints, and interface constraints.

The designed solution involves different modules. Some of these modules are related to the recommender system itself, while others are retailer-specific. The core modules for the recommender system are Recommendations Front-end Service, Customer Authorization Service, Machine Learning Model, Machine Learning Model Interface, and internal Shopping History representation.

When designing the architecture of the solution, different approaches were considered, differing in the learning methodology and in the way data is updated. Two online approaches and one offline approach were studied. However, difficulties inherent in having small but frequent data updates are an obstacle to using online approaches. Also, one online approach creates a need for the retailer to develop a specific piece of software to publish data updates instead of just providing access to it, which can also difficult the adoption of such a system. With these conditions present, the adopted methodology was an offline approach.

Deployment is thought with modularity and maintenance in mind. A retailer-specific app, deployed in its environment, communicates via HTTP with the Recommendations Front-end Service, which is also deployed in its own Linux environment. Both the Authorization API and Retailer API are external components used by the recommender system, so they are also deployed in their environments, and the communication with these two components is also performed via HTTP. The Machine Learning Module and its API and history are deployed in another Linux server, providing communication via HTTP.

The different processes involved when developing the machine learning model, from understanding the problem to deploying a solution, were based on the CRISP-DM methodology.

Python was chosen as the primary programming language for the different components developed. Unit and functional tests are developed using pytest, a testing framework for Python. The most promising machine learning algorithms to be considered when integrating with a retailer were identified: gradient boosted trees, SVMs, and neural networks. The adopted machine learning framework was TensorFlow.

Chapter 5

Solution Implementation

The implemented solution consists of a prototype for a recommender system ecosystem, integrating the different actors and processes involved. The solution can provide customers with tailored recommendations on-demand, as well as operations that simulate the periodic updates associated with the dataset and model versions. It simulates a real-world scenario where retailers release data updates, and a newer version of the model can be generated to work with more recent results.

The adopted architecture presents a general approach for the solution, allowing integration with different grocery retailers. Also, by using TensorFlow, the vast majority of the flows regarding data preparation (detailed in this chapter) is common to the different algorithms, allowing the solution to support distinct machine learning algorithms if the business conditions or the dataset lead to such decision.

Integrating with a grocery retailer requires the adoption of their dataset. This means that the features extracted from the data may suffer some modifications. Also, the algorithm hyperparameters should be tuned to better perform in the retailer dataset. The different components and pipelines were designed to support the integration process. A specific case study, where the solution is integrated with a retailer dataset, is studied later, in section 6.2.

This chapter details the development of the different architectural components present in the recommender system, providing a complete view of the solution. It also covers the journeys of processing the dataset and extracting relevant features and the different software tests that supported the implementation. A summary of the different concerns concludes the chapter.

5.1 Architectural Components

The adopted architecture (see section 4.2.1) presents a way of integrating different components and data in order to create a complete solution for a recommender system based on a machine learning model. These components may be provided explicitly by the retailer or be a specific part of the solution.

The Recommendations Front-end Service works as the gate for the solution, providing external access to the recommender system. The Machine Learning Model Interface exposes the Machine Learning Module (which corresponds to the main element of this dissertation) for recommendations and deals with scheduled tasks related to training of new models and shopping history data updates. These constitute the most critical components of the recommender system.

Some components, namely the Customer Authorization Service, Customer Device, Grocery Retail API, and Customer Loyalty DB, represent elements that, despite playing an important role in the solution, are complementary. The Customer Authorization Service is responsible for making sure that recommendations are authorized. The Customer Device is used to start the recommendation flow and display the recommendations and is retailer-specific. The Grocery Retail API and Customer Loyalty DB are also retailer-specific components that represent a way for the solution to obtain and update the internal Shopping History.

Regardless of the contribution of each component to the solution, the different components were developed and tested with the best practices of software development in mind. This section details the different components on an internal level regarding the way they communicate with each other.

5.1.1 Recommendations Front-end Service

The Recommendations Front-end Service works as the entry point for the solution. The retailer app, in the customer device, communicates with this service in order to obtain tailored shopping list recommendations for that specific customer. This service communicates with the Customer Authorization Service in order to make sure the customer is allowed to, and with the Machine Learning Model Interface to obtain the shopping list recommendations.

This service was developed using Python, and it is organized in a simple structure with different packages, keeping the responsibilities segregated. An API layer communicates with a controller layer, which communicates with different utility entities. Business exceptions and configurations are common components shared by the different layers.

Figure 5.1 represents a simplified view of the class diagram for this service, containing the most important classes and connections. The class RecommendationsServer is the entry point of the service, accepting HTTP requests. This class communicates with the RecommendationsAPI, which parses the input information as necessary and uses the different controller classes to validate the authorization and obtain the prediction. Each controller class uses the HTTPClient class to execute HTTP requests. Different BusinessException implementations are used by the different classes to manage exceptional flows.

The current version of the recommender system is not yet able to predict the quantity of a product. This way, since customers communicate directly with this service, and in order to support additional features beforehand, this service sets a default quantity of 1 for each product predicted by the model, via the Machine Learning Model Interface.

This service includes configurations to control the HTTP port where it is served, the two endpoints it uses, and logging. Since these configurations are important to several moments (e.g., different endpoint configurations are needed when validating the authorization and when requesting a prediction), they are provided using dependency injection, allowing a lower coupling associated with configurations and more flexibility when testing.

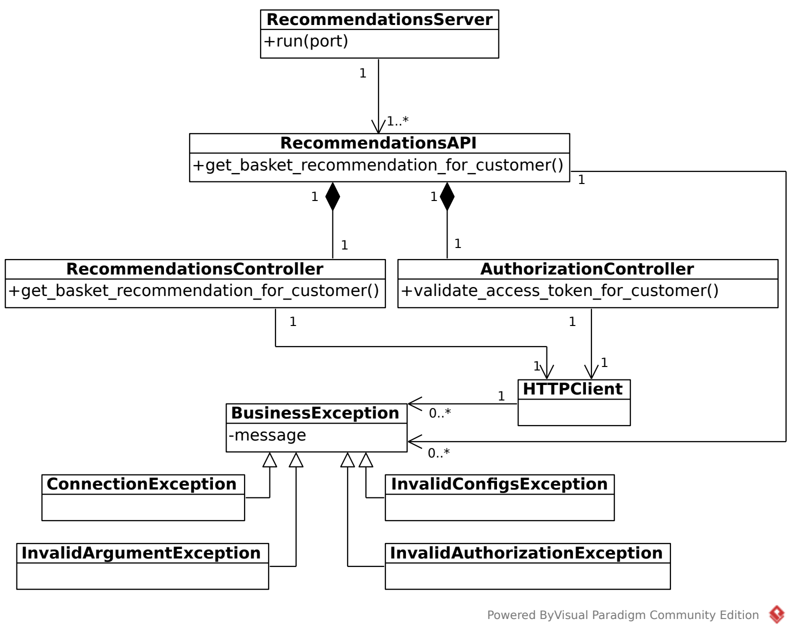

RecommendationsServer
+run(port)
1
1..*
RecommendationsAPI
+get_basket_recommendation_for_customer()
1
1
1
1
1
RecommendationsController
+get_basket_recommendation_for_customer()
AuthorizationController
+validate_access_token_for_customer()
1
1
1
1
1
HTTPClient
BusinessException
-message
0..*
0..*
ConnectionException
InvalidConfigsException
InvalidArgumentException
InvalidAuthorizationException
Powered ByVisual Paradigm Community Edition

Figure 5.1: Class diagram of Recommendations Front-end Service

The Recommendations Front-end Service exposes an endpoint associated with the use case UC_C1, and it is detailed in the following section.

Generate Shopping List Prediction Endpoint

This endpoint is responsible for showing tailored recommendations for a specific customer. It is the entry point of the recommender system, and it is called by the customer device. It returns the recommended list or an HTTP error code if the request could not be executed.

- **Resource URL:** /recommendations/basket/customer/{customer_id}
- **Method:** GET
- **Headers:** Authorization Token
- **Status codes:** 200 (OK: recommendation generated); 400 (Bad Request: there is a problem with the request); 401 (Unauthorized: customer cannot obtain recommendations)
- **Request body:** non-applicable
- **Response body:**

```
{
  "basket_recommendation": [
    {
      "quantity": 1,
      "confidence": 0.976926863193512,
      "product_id": 16797,
      "product_name": "Strawberries"
    },
    {
      "quantity": 1,
      "confidence": 0.9686682820320129,
      "product_id": 24852,
      "product_name": "Banana"
    }
  ]
}
```

Listing 5.1: Response of Generate Shopping List Prediction Endpoint

5.1.2 Customer Authorization Service

The Customer Authorization Service simulates an external service responsible for controlling the access of a customer to a certain resource. It is called by the Recommendations Front-end Service, in order to make sure a certain customer has permissions to request shopping list recommendations. In addition, it provides the ability to generate an authorization token, simulating the retailer service.

The authorization tokens are prepared using JSON Web Token (JWT), which allows the representation of customer information as JSON, digitally signed using a private key. Using the same key, it is possible to validate the content and, so, validate the authorization of a customer.

Since this service is only intended to simulate a retailer service, the validation rule it performs is simple: it decodes the token using the private key and compares the information against the customer identifier sent in the request. This validation makes sure that a request is only valid when using a token encoded for the same customer identifier. This methodology is based on the premise that, in the customer device, the process regarding token generation assures that a customer can only obtain identifiers for herself.

This service was developed using Python, and it is organized in a simple structure with different packages, to have the responsibilities segregated. An API layer communicates with a controller layer, which communicates with a domain layer and different utility entities. Business exceptions are shared by different layers.

The figure 5.2 represents a simplified view of the class diagram for this service, containing the most important classes and connections. The class AuthServer represents the entry point of the service, accepting HTTP requests. This class communicates with the AuthAPI, which deals with the input information and uses the AuthController class to finish the operations with the JWTCustomerTokenService. Different BusinessException implementations are used by the different classes to deal with exceptional flows. This service includes logging and HTTP port configurations.

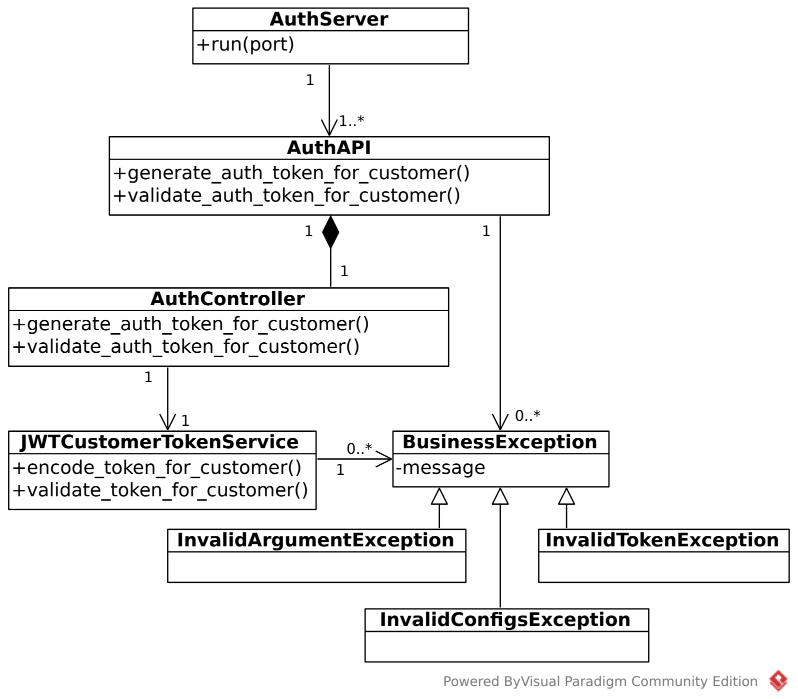

AuthServer
+run(port)
1
1..*
AuthAPI
+generate_auth_token_for_customer()
+validate_auth_token_for_customer()
1
1
1
AuthController
+generate_auth_token_for_customer()
+validate_auth_token_for_customer()
1
1
0..*
JWTCustomerTokenService
+encode_token_for_customer()
+validate_token_for_customer()
0..*
1
BusinessException
-message
InvalidArgumentException
InvalidTokenException
InvalidConfigsException
Powered ByVisual Paradigm Community Edition

Figure 5.2: Class diagram of Customer Authorization Service

Customer Authorization Service exposes two endpoints to manage the authorization, detailed in the following section.

Create Customer Authorization Token Endpoint

This endpoint is responsible for simulating the generation of a customer authorization token. It is called with a customer identifier and a unique installation identifier. This information is encoded in a JWT token and returned to the customer. If the request is invalid, an HTTP error code is returned.

- **Resource URL:** /api/auth/customer/generate
- **Method:** POST
- **Headers:** non-applicable
- **Status codes:** 201 (Created: authorization token generated); 400 (Bad Request: there is a problem with the request)
- **Request body:**

```
{
  "customer_id": 12345,
  "installation_id": "retailer_specific_installation_id"
}
```

Listing 5.2: Request of Create Customer Authorization Token Endpoint

- **Response body:**

```
{
  "access_token": "eyJQiOiJkc2Zqh...",
  "token_type": "bearer"
}
```

Listing 5.3: Response of Create Customer Authorization Token Endpoint

Validate Customer Authorization Token Endpoint

This endpoint is responsible for simulating the validation of a customer authorization token. It is called with the token to validate and the customer identifier. The token is digested and a confirmation is returned. If the request is not valid, an HTTP error code is returned.

- **Resource URL:** /api/auth/customer/validate
- **Method:** POST
- **Headers:** Authorization Token
- **Status codes:** 200 (OK: authorization validated); 400 (Bad Request: there is a problem with the request); 401 (Unauthorized: invalid token)
- **Request body:**

```
{
  "customer_id": 12345
}
```

Listing 5.4: Request of Validate Customer Authorization Token Endpoint

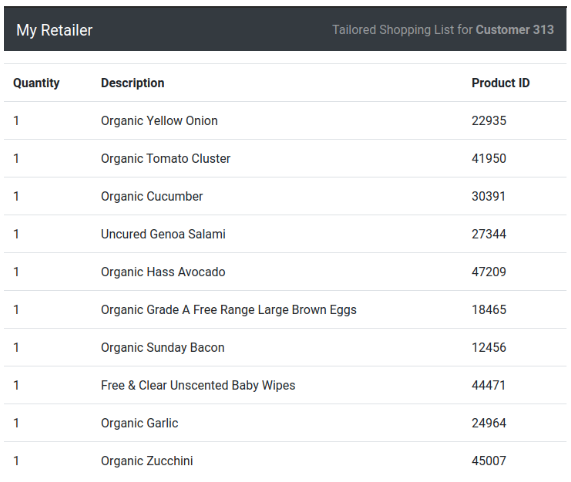
- **Response body:** empty

5.1.3 Customer Device

The Customer Device simulates the software used by a customer of a grocery retailer in order to have access to the tailored recommendations. In a real-world scenario, it could be a mobile app, a website, or even an in-store device, such as a PDA. For this dissertation, a simple website was developed using HTML and JavaScript, to allow the visualization of a shopping list recommendation.

The Customer Device obtains the list by accessing the Recommendations Front-end Service. This request includes an identification token used to represent customer authorization. This token is obtained using the Customer Authorization Service, but in a real-world scenario, it could already be part of the retailer software. The recommender list is then displayed to the customer.

The figure 5.3 presents a screenshot of the demonstrator website, where a tailored shopping list recommendation is being shown for an authorized customer.


My Retailer — Tailored Shopping List for **Customer 313**

Quantity	Description	Product ID
1	Organic Yellow Onion	22935
1	Organic Tomato Cluster	41950
1	Organic Cucumber	30391
1	Uncured Genoa Salami	27344
1	Organic Hass Avocado	47209
1	Organic Grade A Free Range Large Brown Eggs	18465
1	Organic Sunday Bacon	12456
1	Free & Clear Unscented Baby Wipes	44471
1	Organic Garlic	24964
1	Organic Zucchini	45007

Figure 5.3: Customer Device demo with a tailored recommendation

5.1.4 Grocery Retail API

The Grocery Retail API simulates an external service that allows access to a retailer's loyalty dataset. In a real-world scenario, this service could correspond to an API or a data storage (e.g., Amazon S3 object storage [1] or Google Cloud Storage [2]).

To simulate this service, the retailer dataset is compressed and saved into a private Google Cloud Storage. The data is exposed via a unique URL that controls the access. The data updates are simulated by providing access to a new version of the whole dataset in the shared file.

This approach exposes the retailer data similar to what could happen in a real-world scenario, allowing the Machine Learning Model Interface to download a new version periodically and update the internal data representation.

5.1.5 Machine Learning Model

The Machine Learning Model corresponds to the machine learning model developed using TensorFlow. It may be developed using different algorithms, as long as supported by the TensorFlow framework. The model is trained and used via the Machine Learning Model Interface.

The training and predicting processes - designed for generating shopping list recommendations - are constituted by multiple important steps that diverge from the traditional software components and architectures addressed in this section. Details regarding them were wittingly not approached in this section and are detailed in section 5.2.

The model is designed to generate predictions for the items to be present in the next purchase of an existing customer in the dataset used for training. It does not, yet, predict quantities.

[1]https://aws.amazon.com/s3/
[2]https://cloud.google.com/storage

A key feature of the model is that, since it is developed using TensorFlow, it is able to use the GPU during both training and predicting moments because of the CUDA Toolkit. This allows the operations around the model to be executed faster.

5.1.6 Machine Learning Model Interface

The Machine Learning Model Interface is the service responsible for exposing the machine learning model, updating the shopping history, and for training new models. It can, thus, be called by the Recommendations Front-end Service (using a private API-key, since it allows only requests from authorized sources) or by scheduled business tasks.

Predictions are exposed via HTTP, while the two business operations (updating the dataset and training a new model) are only accessible via command-line. This service communicates with the Grocery Retail API to obtain shopping history updates and ingests the data. In addition, it works this dataset in order to extract features and train new models. It provides recommendations using the already deployed models.

The Machine Learning Model Interface was developed using Python and follows a structure that prioritizes responsibility segregation between the different packages and classes involved. An API layer communicates with a controller layer, which uses classes related to the model and data processing and common utility classes.

Since this component is responsible for both the training and predicting processes, it needs to manipulate data to feed the training and predicting phases. The data operations are done using the Python libraries pandas and NumPy. The first library offers access to data structures and operations for manipulating and analyzing numerical tables; the second one provides access to high-level mathematical operations over large, multi-dimensional arrays and matrices.

The figure 5.4 presents a simplified version of the class diagram for this service, showing the most important classes and connections. This service includes two distinct entry points: the Main class, for command-line management access, and the RecommendationsServer, allowing the service to serve HTTP requests.

The command-line access is intended for training new models and updating the dataset but also supports prediction generation. The Main class communicates with the CLIModelAPI, which uses the proper controller class to perform the necessary data and model-related operations. The RecommendationsServer class includes only support for predictions and uses its API class, which communicates with the ModelController to conclude the operation.

Both controller classes (and thus both entry points) use common classes to perform operations on the data, the model, or access configurations and deal with exceptions. The concrete implementations of the classes TrainingDataProcessor and PredictingDataProcessor are responsible for accessing the data needed for the respective process and for obtaining the necessary features. A FeatureProcessor and various Helper classes are used to assist these processes.

This service supports the configuration of the HTTP port where it is served, logging, association rules, machine learning hyperparameters, and location of newly trained models. Similarly to the other services, these configurations are accessible across the solution via dependency injection.

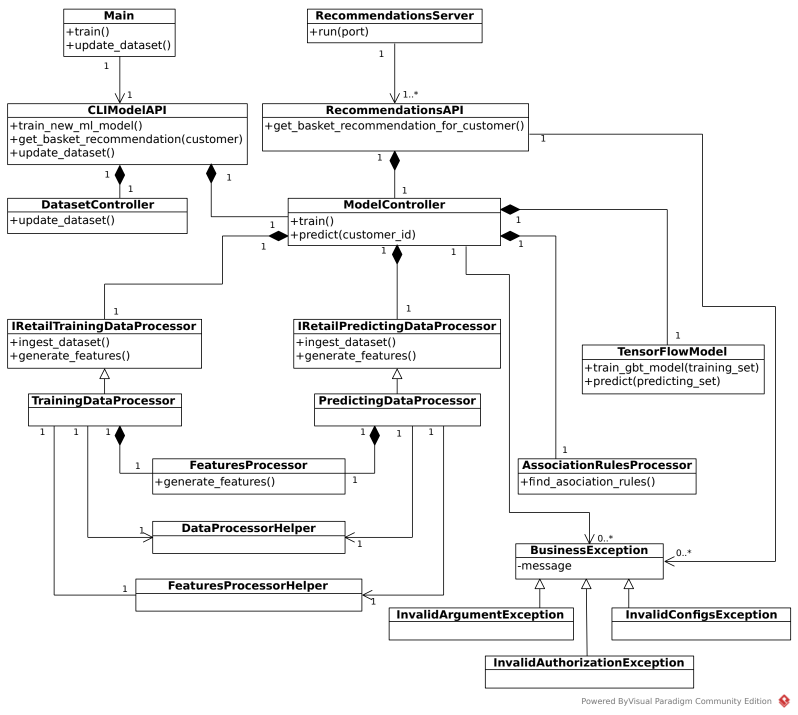


Figure 5.4: Class diagram of Machine Learning Model Interface

Machine Learning Model Interface exposes one HTTP endpoint for recommendations (corresponding to the use case UC_C1) and supports two command-line accesses for management operations (use cases UC_S3 and UC_S4). These three entry-points are detailed below.

Predict Basket Endpoint

This endpoint is responsible for generating the basket recommendations. It is called by the Recommendations Front-end Service, using a private API-key and the customer identifier. If the request is valid, the recommendations are generated and returned. If the request is not valid, an HTTP error code is returned.

- **Resource URL:** /recommendations/basket/customer/{customer_id}
- **Method:** GET
- **Headers:** api-key
- **Status codes:** 200 (OK: basket predicted); 400 (Bad Request: there is a problem with the request); 401 (Unauthorized: invalid private api-key)
- **Request body:** non-applicable
- **Response body:**

```
[
  {
    "product_id": 16797,
    "product_name": "Strawberries",
    "confidence": 0.976926863193512
  },
  {
    "product_id": 24852,
    "product_name": "Banana",
    "confidence": 0.9686682820320129
  }
]
```

Listing 5.5: Response of Predict Basket Endpoint

Update the Shopping History Scheduled Task

This scheduled task allows the shopping history to stay up to date by periodically downloading updates from the retailer and updating the dataset. Since the Grocery Retail API simulates the data updates by providing a new compacted dataset version in a Google Cloud Storage account periodically (see section 5.1.4), this process is done by recurrently accessing the storage.

During each iteration, the task downloads the compressed data file, decompresses it, and creates a newer version of the dataset used to train new models. This scheduled task allows the configuration of the time it is expected to run, as it should be adapted to each real-world scenario.

Train and Update the Model Scheduled Task

This scheduled task allows the training of a newer version of the model based on the existing dataset and adopted machine learning algorithm configurations. During each iteration, this task triggers a training and evaluation process, resulting in both the new model and the newer version of the dataset being configured as the latest.

This scheduled task allows the configuration of the moment it is expected to run, either by being run at a specific time or after the task responsible for updating the shopping history. It should be adapted to each retailer scenario.

5.2 Machine Learning Processes

When creating a machine learning model and when generating recommendations, the Machine Learning Model Interface performs some processes to assure that the dataset is ready to be used. When integrating with a retailer, it is of paramount importance to study their dataset to understand which and how knowledge can be extracted.

This section starts by describing the identification of the target variable and the process of how the features are extracted to create both training and predicting datasets. The section is closed with an analysis of both training and predicting flows.

5.2.1 Identifying the Target Variable in the Dataset

When training a classification machine learning model, the training data needs a label. The model is trained knowing the target variable and learns to analyze new future scenarios and predict them. In this problem, the prediction represents whether or not a specific scenario may lead to an item's purchase by a specific customer.

During the training phase, the dataset is loaded and parsed in order to obtain a matrix representing a scenario where a customer, at a specific moment in time, with a certain historical data, has purchased an item. Whether or not an item was bought, using these conditions, corresponds to the target variable - this is a binary classification, and it is expressed with "0" for "no" and "1" for "yes". It is important to note that, when integrating with a grocery retailer, the dataset should be analyzed in order to identify how the target variable can be obtained.

When predicting, a similar logical matrix is built, representing the specific moment in time, the products, and the customer being recommended. The machine learning model assigns confidence in this prediction. The top-N items that the model considers most likely to be bought by a customer are joined together to create the recommended next shopping list.

5.2.2 Extracting New Features From the Data

The training set used to train a classification machine learning model is constituted by pairs of a target variable and a set of details that lead to each classification. These details are commonly referred to as features, and they can be extracted directly from the dataset or created - either by combining different details from the original dataset, or by applying mathematical operations to them. Some examples of traditional operations used when generating new features include averages, counts, maximum and minimum values. This process benefits, typically, from knowledge on the domain.

In this particular problem, different types of features can be extracted to represent specific behaviors that lead to buying or not an item. Regardless of the retailer or dataset, for this solution, one should aim to extract features about four categories: explicit features (e.g., the hour of the day), customer features (e.g., buying frequency and number of purchases), product features (e.g., number of orders and number of reorders), and customer-product features. Different features can be extracted for each of these categories, according to the dataset. However, excluding any of them may compromise the recommendations' quality.

Features are typically numerical or categorical (more types of features exist, but the technology used limits the usage of these two types (TensorFlow 2020b)). Numerical features correspond to quantitative information, represented as numbers; categorical features, on the other hand, correspond to information represented as groups or categories.

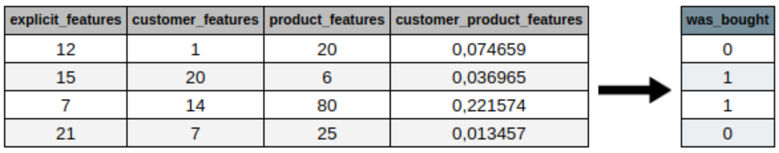
The contribution of each extracted feature for the retailer dataset should be individually understood before being used by the model. Some features may need to be refactored or dropped because they provide little to no benefits or are too computationally expensive for the value they provide.

This feature engineering should be addressed in the Machine Learning Model Interface when integrating with a retailer dataset. These features are then used when training a new model and when generating recommendations. It is a crucial step in both training and predicting pipelines.

5.2.3 Training and Predicting Datasets

With the target variable identified and the different feature categories obtained, one can prepare both training and predicting datasets. These datasets map the features with the target variable, and the main difference is in the fact that the training dataset corresponds to all the existing customers and products, whilst the predicting dataset corresponds to one customer at a time. Also, the target variable is not known in the second one.

The figure 5.5 represents a simplified example of a training dataset, where certain features lead to the purchase of a particular item by a specific customer. The different features were extracted and mapped against the target variable, and are ready to feed the machine learning algorithm.

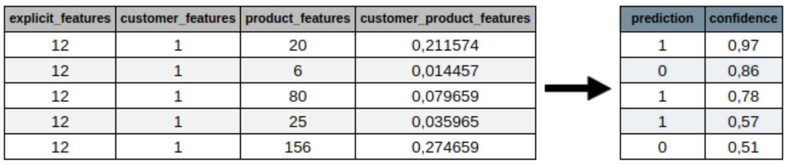


explicit_features	customer_features	product_features	customer_product_features
12	1	20	0,074659
15	20	6	0,036965
7	14	80	0,221574
21	7	25	0,013457

was_bought
0
1
1
0

Figure 5.5: Example of a training dataset with a mapped target variable

The figure 5.6 represents an example of the predicting dataset, where the chance of a specific customer purchasing a set of products is being evaluated for a particular moment in time. The different features were extracted for the customer and the different products, and the target variable for each row is ready to be predicted by the machine learning model (i.e., predict if that product should be recommended).



explicit_features	customer_features	product_features	customer_product_features
12	1	20	0,211574
12	1	6	0,014457
12	1	80	0,079659
12	1	25	0,035965
12	1	156	0,274659

prediction	confidence
1	0,97
0	0,86
1	0,78
1	0,57
0	0,51

Figure 5.6: Example of a predicting dataset and its predictions

5.2.4 Preparing the Training Dataset from the Features

Different features are obtained for every customer, every product, and every combination of customer-product, creating three separate and significant in-memory tables. These tables are joined with the tables representing the training orders, order details, and products, creating a big table with all the features projected against each occurrence of customer-product, indexed by customer and product identifiers.

Each line of this logical table represents a unique scenario, where a specific product was bought by a specific customer, for a specific moment in time, and where both customer and product have certain details (from their features). Each of these lines also includes an entry for the target variable, representing whether or not that scenario originated the purchase of an item already purchased before.

The dataset has to go through three additional steps before the training phase: feature normalization, training-validation split, and setup of the TensorFlow feature columns.

Feature Normalization

The numerical features are sparse, which would make harder for the model to work during both training and predicting phases. Several techniques exist to overtake this difficulty by normalizing (or rescaling) the data. In this dissertation, min-max normalization was the technique chosen to normalize the data. This technique performs a linear transformation on the original data, keeping the relationship between the original data (Patro and Sahu 2015). It can rescale the data between a specific range of values or simply between 0 and 1.

The normalization is performed according to the following equation (Patro and Sahu 2015), where x corresponds to the value being rescaled, x' to the rescaled value and a and b to the range.

$$x' = \frac{x - min(x)}{max(x) - min(x)} * (b - a) + a \tag{5.1}$$

The normalization was performed between 0 and 1, thus it can be simply described as:

$$x' = \frac{x - min(x)}{max(x) - min(x)} \tag{5.2}$$

In order to perform min-max normalization, one needs to calculate the minimum and maximum values of each feature in order to apply the equation to all the entries in every feature. These normalization parameters are saved so that they can be used during the predicting phase.

With the training dataset normalized, the machine learning model can be trained.

Training-validation Split

The dataset needs to be divided into two subsets before training: training and evaluation subsets. The machine learning algorithm uses the first subset to train the model, according to its internal specifications and configurations; the second subset is used in order to evaluate the trained model and provide some statistical information regarding the training process (e.g., accuracy, precision, recall). These two subsets include the features and the label associated with it.

Several works exist on finding the best percentage value for this training-validating split. Some authors consider a good percentage to be around 20-25% (Medar, Rajpurohit, and Rashmi 2018). TensorFlow also specifies 25% as the default value for this setting. For this work, 20% was the adopted split percentage.

Setup of TensorFlow Feature Columns

In order to allow TensorFlow to manage the data in the training dataset properly, it is necessary to configure the columns in the dataset specifically. For numerical features, the process is straight forward: a numeric column (*tf.feature_column.numeric_column*) is added, specifying the feature name and the data type (e.g., float32, int64).

Categorical features, on the other hand, are a different matter, as they have to be encoded in order to allow the machine learning algorithm to work with labeled data (TensorFlow 2020b). The way recommended by the documentation to solve this issue is by creating simple embeddings with the categorical features, using *one-hot encoding*. This technique consists of simply using a vector representing all the distinct categories for that feature, having a "1" value in the position corresponding to the specific value, and "0" in every other position.

Categorical columns are, then, configured by creating a column (*tf.feature_column. categorical_column_with_vocabulary_list*) with the feature name and a list of every distinct value for that feature. This configuration is dynamically used by TensorFlow and persisted with the model metadata, to be used when predicting.

5.2.5 Training Pipeline

The steps addressed in section 5.2.4 allow the preparation and setup of a training dataset ready to create a machine learning model in TensorFlow. To ensure that datasets with different sizes can be used, the training dataset is sliced, and the training is performed in multiple iterations.

Experiments performed on the used hardware shown that training iterations with 5M lines caused a memory usage of around 90%, which was an affordable value. The training dataset (more specifically, the subset of the training dataset used for training) is split into slices of 5M lines, and one iteration is performed per slice. In order to obviate any existing bias, each slice is shuffled before training. TensorFlow provides a set of hyperparameters to be tuned in order to increase the performance of the model (see the tests performed in 6.3).

After the training flow, the evaluation subset from the training dataset is used to generate metrics regarding the current training. TensorFlow uses the data from this subset to predict the target variable and returns some statistics. However, since the main goal of this dissertation is to predict a shopping list, the results returned by the evaluation dataset are not the results one aims to obtain - it is necessary to evaluate the shopping list recommendation. Nevertheless, these results are a good indicator of success.

Once the model is trained and evaluated, a useful feature provided by TensorFlow is to extract the contribution of each of the features during the training phase. This information is persisted with the model for future consultations and is approached later, in the case study. The next and final step is, then, to save the latest model. The path for the latest saved model was made configurable. In addition to the final model, TensorFlow saves several checkpoints to allow the recovery or continuation of the training flow.

The figure 5.7 represents the trained pipeline, which wraps up the multiple processes analyzed in the previous sections. The flow is started with loading and preparing the training data, extracting the features, normalizing the data, and saving the normalization parameters for the predicting phase; after this moment, TensorFlow columns are configured, and the dataset is split into training and validating subsets; the training subset is then split into several slices to allow an iterative training flow; finally, the TensorFlow classifier is created, and the model is trained in multiple iterations with the training dataset slices; the process is concluded with the evaluation and persistence of the latest model.

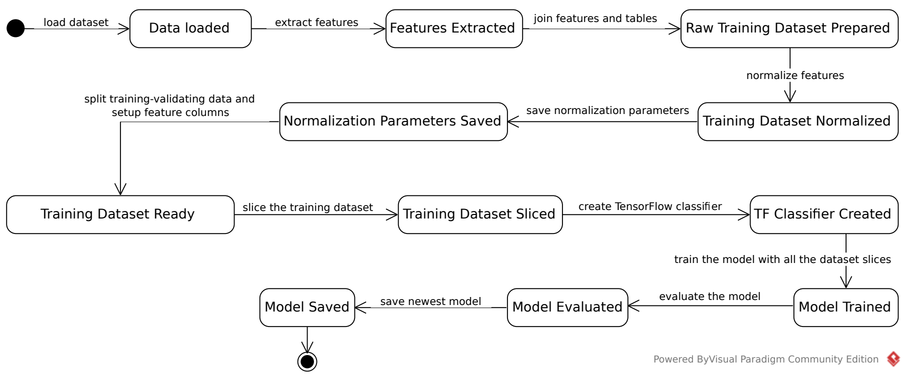

load dataset
Data loaded
extract features
Features Extracted
join features and tables
Raw Training Dataset Prepared
normalize features
Training Dataset Normalized
save normalization parameters
Normalization Parameters Saved
split training-validating data and setup feature columns
Training Dataset Ready
slice the training dataset
Training Dataset Sliced
create TensorFlow classifier
TF Classifier Created
train the model with all the dataset slices
Model Trained
evaluate the model
Model Evaluated
save newest model
Model Saved
Powered ByVisual Paradigm Community Edition

Figure 5.7: Training pipeline of the machine learning model using TensorFlow

5.2.6 Preparing the Predicting Data from Features

Preparing the predicting data is a process very similar to preparing the training data, detailed at 5.2.4. However, whereas training is done for every customer in the dataset, predicting is customer-specific, which reduces the amount of data being dealt with.

Recommendations can be generated in two different ways: predicting the last purchase and predicting a shopping list for that specific moment. The first approach is useful when evaluating the recommendation, as it predicts for the moment of the customer's last purchase (which was naturally not considered during training). The second approach is useful when predicting a shopping list for the current moment, since it uses the day and hour of the recommendation request.

The different features described in 5.2.2 are obtained for every product ever purchased by the customer (since only these products are being recommended), the customer itself, and every combination of customer-products. These three different features are joined with the information regarding products bought by that customer and the order being predicted.

Each line of this predicting dataset represents a scenario where, for a certain moment in time, a customer and different products have certain details (from their features) - the same kind of details used when training the model. The difference is that, now, the label is not known and needs to be predicted.

The predicting dataset has to go through two additional steps: feature normalization and setup of the TensorFlow examples.

Feature Normalization

Similarly to what happens during training (detailed in 5.2.4), the predicting dataset goes through min-max normalization. The normalization is performed in exactly the same way, differing only in the way the normalization parameters (i.e., min and max values for each feature) are obtained. When training, these parameters are retrieved from the dataset (and saved); when predicting, the training parameters are loaded into memory.

The feature normalization done at predicting time uses the same parameters as to when training in order to assure coherence between the data. If this normalization was performed using different parameters than the ones used when training, no guarantees would exist that a certain normalized value meant the same in both steps, causing the results to lose reliability.

The predicting set has, naturally, the same structure as the training set, as presented in section 5.2.3.

Setup of TensorFlow Examples

When predicting using a previously saved machine learning model, each row of the predicting set is transformed into an instance of *tf.train.Example()*. Each TensorFlow Example is constituted of several feature columns with the feature name, where the data type differs according to the type of feature: numerical features set a float value, and categorical features set an integer value.

Whilst during training it is useful to encode categorical features, as studied in 5.2.4, when predicting this step is not done. TensorFlow includes this meta information in the saved model, allowing it to dynamically interpret the value of each categorical feature.

Finished this process, the predicting dataset is ready to be predicted.

5.2.7 Predicting Pipeline

The steps detailed in section 5.2.6 allow the preparation and setup of a predicting dataset, ready to be predicted by a machine learning model using TensorFlow. This model, which was saved at the end of the training pipeline (5.2.5), is loaded into memory and is used to predict each of the TensorFlow Examples in the predicting set. The result is a list of predictions with the same number of rows as the predicting dataset.

Each prediction includes information regarding the existing classes (i.e., the possible values for the target variable (see section 5.2.1)), the predicted class and the confidence in the prediction. To generate a shopping list recommendations, these predictions are filtered, retrieving only the predicted items, and ordered decreasingly by confidence.

The maximum amount of items in the recommended list is a configuration in the project and should be specified for each retailer.

The figure 5.8 represents the pipeline, wrapping up the multiple processes involved in the previous sections. The flow is started by loading the necessary data, extracting the features, and preparing the predicting dataset; the dataset is then normalized, and TensorFlow examples are prepared to allow predicting; next, the machine learning model is loaded and used to predict over the prepared dataset; predictions are then filtered and sorted, and the process is concluded by resizing the recommendations list.

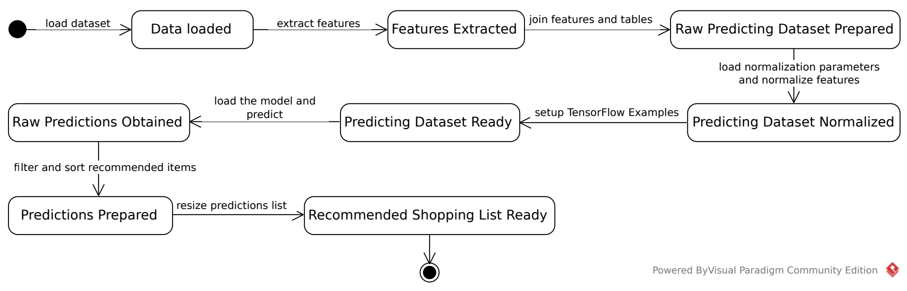

load dataset
Data loaded
extract features
Features Extracted
join features and tables
Raw Predicting Dataset Prepared
load normalization parameters and normalize features
Predicting Dataset Normalized
setup TensorFlow Examples
Predicting Dataset Ready
load the model and predict
Raw Predictions Obtained
filter and sort recommended items
Predictions Prepared
resize predictions list
Recommended Shopping List Ready
Powered ByVisual Paradigm Community Edition

Figure 5.8: Predicting pipeline of the machine learning model using TensorFlow

5.3 Software Tests

The development of the recommender system was kept up with different tests. Depending on the development phase, different types of tests were developed to help making sure that the implementations have a fewer number of flaws possible and that adding new features can be done without risking compromising other parts of the solution.

The adopted test types differ on the granularity they hit: small portions of code are tested by unit tests; the main goals of each component are tested using integration tests; and functional requirements are tested using end-to-end tests. This section presents the concerns behind the developed unit, integration, and end-to-end tests.

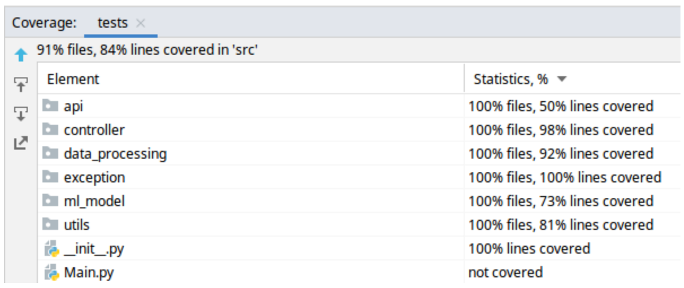
5.3.1 Unit Tests

Unit tests consist of tests for small portions of the code, such as functions. The main goal is to develop tests for the smallest and useful portion of code possible, in order to test complex applications with sets of small and comprehensive tests. These tests target the identification of logical bugs (whether in the existing code or when adding features). However, they are also important to help developers improving code quality and promoting low coupling.

Unit tests were developed using pytest, a framework for the adopted language. Each architectural component has a set of tests, developed throughout the project life cycle, and organized similarly to the source code structure, for better organization. These tests are run every time the code is published to the repository, allowing visualization of the project status.

In order to test heavier computational flows (e.g., training a model or extracting features from a dataset) or communications with external services, the configurations were overridden with test data, namely with a subset of the dataset and mock services. This was easily accomplished since dependency injection was adopted in the different services.

The percentage of code covered by unit tests can be automatically retrieved from the execution of the tests. This procedure helped to identify the parts of the code that were not tested yet, facilitating the design of new tests. As an example, the figure 5.9 shows the coverage report of unit testing in the Machine Learning Model Interface service.



Coverage: tests

91% files, 84% lines covered in 'src'

Element	Statistics, %
api	100% files, 50% lines covered
controller	100% files, 98% lines covered
data_processing	100% files, 92% lines covered
exception	100% files, 100% lines covered
ml_model	100% files, 73% lines covered
utils	100% files, 81% lines covered
init.py	100% lines covered
Main.py	not covered

Figure 5.9: Unit testing coverage for the component Machine Learning Model Interface

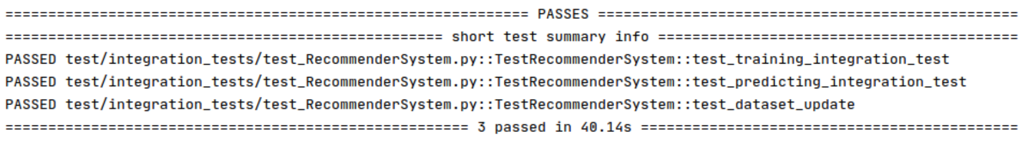
5.3.2 Integration Tests

Integration tests can be seen as an extension of unit tests. The goal of these tests is to assure that the communication between the different parts of a component works as expected, by testing functionalities as a whole. These tests are more complex than unit tests, but they also provide bigger confidence in the component success.

Since each component includes well-defined responsibilities, the number of integration tests is inevitably smaller than unit tests. Nevertheless, they were an important part when developing each software component by testing the different scenarios within the functionalities of a component.

Integration tests were also developed using the test framework used in unit tests (i.e., pytest). Each component includes a set of integration tests, assuring that every functionality is working as expected. Such as unit tests, integration tests benefited from using a smaller dataset and mock endpoints, when needed, to assure that the tests hit the different scenarios within a feasible amount of time.

The figure 5.10 presents an output summary of an integration testing run. These tests are related to the Machine Learning Model Interface, and it is possible to evidence that the different functionalities of this component (i.e., training a model, generating prediction, and updating the dataset) are tested independently.



```
============================================================= PASSES ==============================================
=================================================== short test summary info =========================================
PASSED test/integration_tests/test_RecommenderSystem.py::TestRecommenderSystem::test_training_integration_test
PASSED test/integration_tests/test_RecommenderSystem.py::TestRecommenderSystem::test_predicting_integration_test
PASSED test/integration_tests/test_RecommenderSystem.py::TestRecommenderSystem::test_dataset_update
======================================================= 3 passed in 40.14s ============================================
```

Figure 5.10: Integration tests of the component Machine Learning Model Interface

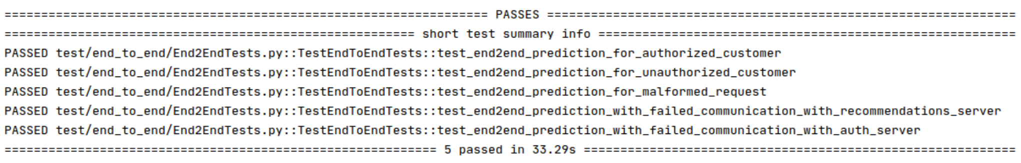
5.3.3 End-to-end Tests

End-to-end tests aim at testing the success of a functional requirement. These tests have a higher granularity than unit and integration tests and also provide higher confidence regarding the success of the solution. The success of end-to-end tests represents the success of a use case, since both the different components involved and communication protocols are tested.

These tests were developed using pytest as well, but in the latest moment of the development cycle, after the success of both unit and integration tests. Since they hit a complete functional requirement, it was of paramount importance to test the different error flows beside the happy-path. The end-to-end tests hit success, failure, and error scenarios.

The use cases that compose the main functionality of the recommender system were extensively tested. The support use cases, namely training and deploying a model and updating the dataset, do not involve different components and thus were not translated into end-to-end tests - they were tested via integration tests.

The figure 5.11 presents the output summary of an end-to-end testing run. One can observe that the main recommendation scenario is covered, as well as exceptional scenarios (with invalid and unauthorized requests) and error scenarios in internal communications. The real deployed components were used in these tests, replicating real-world requests.



```
================================================================== PASSES ================================================================
======================================================== short test summary info =========================================================
PASSED test/end_to_end/End2EndTests.py::TestEndToEndTests::test_end2end_prediction_for_authorized_customer
PASSED test/end_to_end/End2EndTests.py::TestEndToEndTests::test_end2end_prediction_for_unauthorized_customer
PASSED test/end_to_end/End2EndTests.py::TestEndToEndTests::test_end2end_prediction_for_malformed_request
PASSED test/end_to_end/End2EndTests.py::TestEndToEndTests::test_end2end_prediction_with_failed_communication_with_recommendations_server
PASSED test/end_to_end/End2EndTests.py::TestEndToEndTests::test_end2end_prediction_with_failed_communication_with_auth_server
=========================================================== 5 passed in 33.29s ===========================================================
```

Figure 5.11: End-to-end tests of the recommender system

5.4 Summary

The recommender system developed includes several architectural components that communicate with each other to provide a general solution for personalized recommendations. The adopted architecture is composed of three main components, three support components, and two data sources.

The three main components are the Recommendations Front-end Service; the Machine Learning Model Interface, and the Machine Learning Model. The three support components correspond to the Customer Authorization Service, the Grocery Retailer API, and the Customer Device. The two data sources are the dataset exposed by the retailer and the dataset used by the model. Since the support services are retailer-specific, they are simulated in this dissertation. The communication between the services is done via HTTP, and each component is built using a structure with multiple packages and layers, promoting the responsibilities segregation. Good software development practices were in the foundation of the different modules.

In this problem, the target variable corresponds to the purchase of an item associated with a set of conditions that lead to this decision. The way it is obtained depends on the adopted dataset. Features correspond to information generated from the dataset, representing certain characteristics, and TensorFlow supports either numerical or categorical features.

Four types of features were identified: customer-related, product-related, customer-product-related, and explicit features. Customer-related features represent details about customer's shopping habits, product-related features describe the product, customer-product-related features represent the relationship between the two entities, and explicit features are present directly in the dataset. These features are used when training and predicting.

Before training, the training dataset is normalized using the min-max technique. The training pipeline is finished with the split of the dataset into training and validation, and the configuration of the feature columns for TensorFlow.

When predicting a shopping list, only the orders of the customer executing the request are considered. The predicting dataset is also normalized using the normalization parameters obtained during training, and the model is loaded into memory in order to predict the data. The predictions are then filtered by the classification result and sorted by confidence. The predicting pipeline is concluded by returning the top recommendations.

The development was followed by different methodologies of software tests. Unit testing was done to ensure that the code was bug-free and that new features could be added with confidence. Integration testing was implemented to make sure that the different components were working as expected as a whole. End-to-end testing was performed in the last moment of the development life cycle, in order to assure that the different use cases work as expected.

Chapter 6

Evaluation and Results

This chapter presents the solution evaluation and the discussion of its results. It starts by analyzing the experimentation and evaluation processes, where the test hypothesis is analyzed, the metrics used are presented, and the evaluation methodology is described. It is followed by a case study, where the implemented solution is applied to a public dataset. After, experiments around hyperparameter tuning are performed, and the solution is compared against other recommender systems over the same conditions. These comparisons are followed by a validation of the recommendations with association rules and performance tests over different workloads. A summary of the different concerns concludes the chapter.

6.1 Experimentation and Evaluation

Experimentation and evaluation are important tasks for research and critical thought (Gomes 2016). They are an important part of achieving conclusions. In this section, test hypotheses are presented, the metrics used to evaluate them are enumerated, the evaluation methodology is detailed, and the test environment where the experiences are executed is presented.

6.1.1 Test Hypothesis

Since the primary goal of this dissertation is to develop a machine learning recommender system capable of predicting useful shopping baskets, it is of paramount importance to evaluate its quality, using appropriate metrics, and its performance. It is essential to choose a machine learning model and tune its hyperparameters to prepare a good comparison point. As a comparison, two non-machine learning recommender systems existing in the proponent organization were applied to the same dataset.

Analyzing the results obtained by the three solutions, one is able to detect quality differences. In addition to plots and tables, statistical tests help to achieve conclusions (Gomes 2016). This way, the comparisons aim to reject the null hypothesis, where all the different classifiers would behave similarly on the same data, under similar conditions.

Also, the system is tested to validate its behavior under the presence of different workloads and respond in an adequate amount of time so that customers can seamlessly wait for predictions to be generated without sacrificing the user experience. The main goal in this validation is to make sure the system can respond in real-world scenarios and understand how the adopted architecture is able to deal with different stress moments.

6.1.2 Evaluation Metrics

The developed solution and the two solutions used for comparison are tested over the same data to generate recommendations. As detailed in 3.2.5, the quality of the generated recommendations can be evaluated using the metrics accuracy, precision, recall, and f1-score (Jariha and S. K. Jain 2018). In this specific domain, the evaluation metrics can be understood as:

- **accuracy** - Number of correct predicted products divided by the total number of predicted and not predicted products;
- **precision** - Number of correct predicted products divided by the total number of predicted products;
- **recall** - Number of correct predicted products divided by the total number of purchased products;
- **f1-score** - Harmonic average of the precision and recall.

In addition to the quality metrics, performance is also evaluated. Parallel and sequential requests are executed programmatically, simulating the different stress moments of real-world use. The average duration times are acquired and analyzed.

6.1.3 Evaluation Methodology

The implemented solution is applied to a dataset, against which is evaluated according to two complementary ways: it is measured the quality of the recommendations and the ability of the solution to generate predictions in useful time. The first measure is obtained by running the solution over the dataset and the last one by simulating user requests.

The quality evaluation is done by collecting and comparing the average results of each evaluation metric for all the test customers, in the different tests, by the three solutions, using the same data. This evaluation provides information to plot the comparisons and evaluate the results.

In the end, one is able to apply the non-parametric test of Friedman, to conclude with higher confidence the possibility to refute the null hypothesis (Gomes 2016) - in other words, verify the existence of differences between the behavior of multiple recommender systems. One is able to use the Friedman test since there are more than two classifiers and because the data is not expected to follow a normal distribution. If the null hypothesis is proven rejected, one can also use the Nemenyi test (*post-hoc*) in order to obtain details on the origin of the difference.

The performance of the solution is tested after an automatic simulation of requests to generate recommendations. These requests are made sequentially and in parallel, simulating a real scenario. The duration times are registered and plotted.

This evaluation methodology is supported by studies such as (Fernández-Delgado et al. 2014; Gomes 2016; Olson et al. 2017), where non-parametric tests and post-hoc analysis are applied to the evaluation metrics from machine learning classifiers.

6.1.4 Test Environment

The different tests performed in this section were executed in a machine, courtesy of the proponent organization, with the following specifications: Ubuntu 18.04 OS; 64 GB Crucial DDR4 4x16GB RAM; Intel i7-8700K CPU; 480 GB Micron 5200 SSD storage; Asus ROG STRIX Z370-G GAMING motherboard; Nvidia GeForce GTX 1070 8 GB GDDR5 GPU.

6.2 Instacart Dataset - A Case Study

The architecture detailed in chapter 4 and implemented in chapter 5 allows the development of a generic recommender system for grocery retail, based on a TensorFlow-based machine learning model. The real value of the solution can only be obtained when integrating it with a grocery retailer.

In order to perform the case study, different public datasets were analyzed, and one was adopted and detailed. Also, as studied in section 5.2.2, valuable features were extracted from the dataset and analyzed, to improve the results of the recommender system. This process was followed by an analysis of the different supported machine learning algorithms to identify which provides better results over the adopted dataset. The adopted dataset and algorithm are further explored and compared in this chapter.

From the public datasets analyzed, some were not strictly related to grocery retail (e.g., the *Online Retail Data Set* [1]). The dataset that better reflected a grocery retail scenario, with a fair amount of data, was *Instacart Online Grocery Shopping Dataset* (Instacart 2017) - from now on referred to as the *Instacart dataset*.

The Instacart dataset contains data regarding 3 million grocery orders, divided by more than 200 thousand users. It was also used in a former *Kaggle* competition [2]. The proponent company and the advisors also supported the choice of the dataset.

This section includes an analysis of the dataset, an exploratory data analysis that allows a deeper understanding of the data, the tests performed to adopt a machine learning algorithm, and the performed feature engineering.

6.2.1 Dataset Overview

The Instacart dataset was provided for non-commercial use by the American company Instacart [3], which partners with some of the biggest retailers in the US, in order to provide a pick-up and delivery service.

The dataset is provided as a set of *.csv* files, translating data tables for departments, aisles, orders, products, and product_orders. The last table is divided into two separate files, containing different portions of the data: prior and train. The prior file contains all the items in all the transactions for each customer, except for the last transaction. The last transaction is present in the training file or is absent. The figure 6.1 represents the relationships between the data tables.

[1]https://archive.ics.uci.edu/ml/datasets/Online+Retail
[2]https://www.kaggle.com/c/instacart-market-basket-analysis/data
[3]https://www.instacart.com/

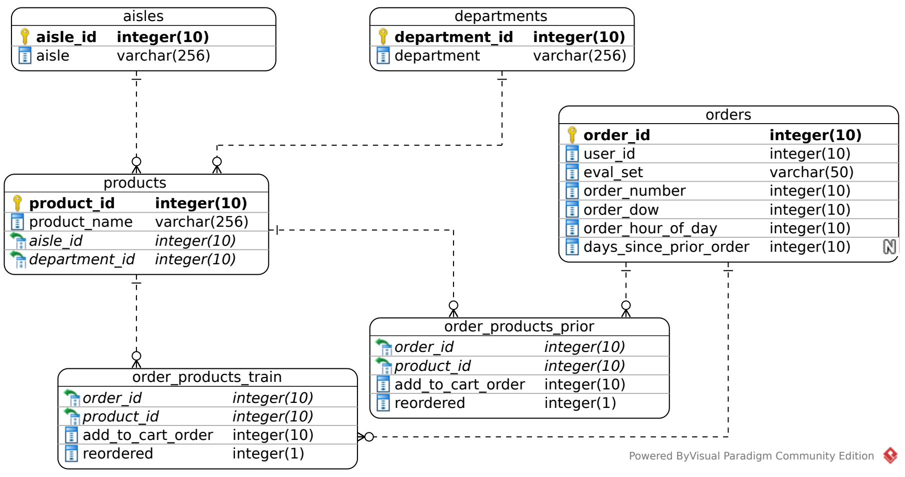


Figure 6.1: Entity Relationship Diagram of Instacart the Instacart Dataset

Each product in the dataset has an identifier and a direct reference to a specific aisle and department, categorizing its location in the store and an exact name. Each aisle and each department include an exact name, as well as its identifier. Orders include an identifier, the identifier of the customer (there are no more details regarding a customer), a tag regarding the set it belongs (i.e., train, prior, or test), an order number representing the position in a specific customer shopping list, the day of the week, the hour of the day and, when available, the number of days since a previous order.

The dataset includes N prior orders for each customer, and one train or test order (designed for the competition). This is useful to separate the last transaction and, thus, evaluate its recommendation. This way, for the purpose of this dissertation, only prior orders are considered during training, saving the last transaction from the training flow.

The association between these orders and their products are present in two additional tables: one with all the N-1 orders for each customer, and one for the train orders (for customers with one train order). Other customers have no train orders but have one test order, and they are used only for training and not for evaluating recommendations, as no information regarding their last transaction exists. These association tables also include the position in which a product was added to the cart, and a flag representing if the product was reordered (i.e., if it is not new for a specific customer).

There is not a specific table with customer data. The only detail regarding customers is their unique identifier. Even if there existed customer details, they would not be considered by the recommender system to assure privacy. In addition, recent rules regarding data protection (e.g., GDPR [4]) encourage this concern.

6.2.2 Exploratory Data Analysis

The Instacart dataset includes 206209 customers and 3421083 orders. From these orders, 3214874 are classified as prior, 131209 as train, and 75000 as test. Since only the prior orders are useful to train the dataset (as detailed in 6.2.1), the model can use around 94% of all the transactions available during training.

[4]https://gdpr.eu/what-is-gdpr/

The biggest amount of orders for a specific customer is 100, and the minimum amount of orders is 4. In addition, the average amount of products in the orders is 10. From all the orders used for training (prior), 388513 include no reordered products (i.e., around 12% of these orders include only new items or are the first order), meaning that more than 88% of the orders contain the items that that customer uses to buy.

This section presents an exploratory data analysis of the dataset, aiming to better understanding the dataset to create the best model possible. It includes several diagrams exploring the data and an explanation of its meaning and generation. These diagrams were generated using pandas and seaborn libraries, for Python, over the Instacart dataset. This analysis was inspired by the work of two authors (Lekha 2019; SRK 2017), but adapted and evolved to target the specific problem of this dissertation.

The figure 6.2 represents the distribution of products per order, and it was generated by simply counting the number of items in each order. This provides useful additional information regarding the basket size (as seen in 6.2.2, the average basket size is 10). The number of products per order increases until the number of items is 6; after this moment, it decreases gradually, approaching 0 when the amount of products per cart goes around 50.

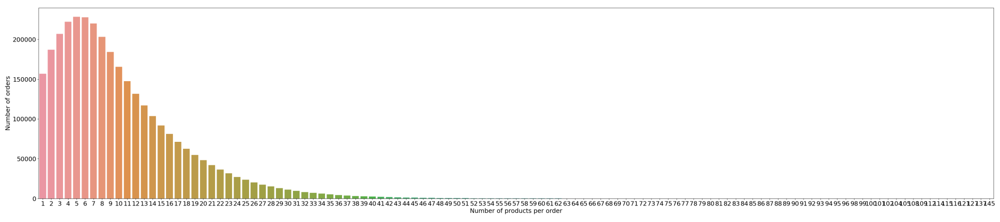


Figure 6.2: Distribution of products per order

The figure 6.3 details the distribution of orders per customer. This chart was generated by grouping the orders by customer and order identifiers and counting the orders. It is possible to observe that the majority of customers have less than 10 orders. The amount of orders per customer falls progressively, to the point where the number of customers with more than 30 orders is meager. Customers with more than 70 orders represent a residual part of the dataset.

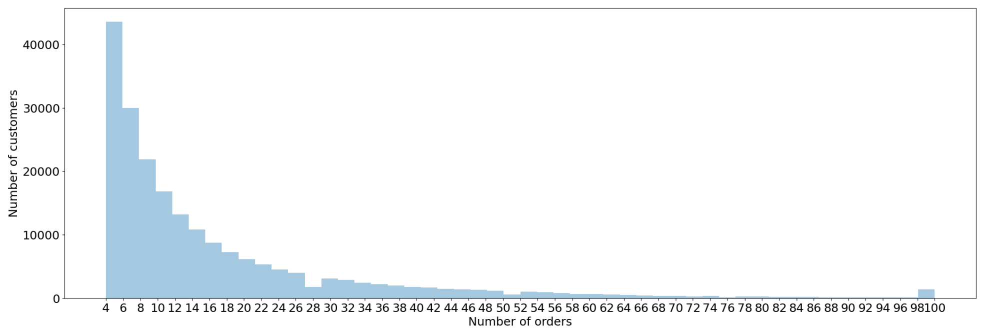


Figure 6.3: Distribution of orders per customer

The figure 6.4 shows an interesting customer behaviour regarding shopping habits. Since each order register in the dataset includes the number of days since prior order, it was possible to generate a diagram mapping the number of days since prior order by the number of orders with a specific value. This value seems to be chopped at 30, showing that the dataset might be missing customers whose shopping frequency is more than once a month. Interestingly, the peaks at 7, 14, 21, and 30 days, evidence a clear preference for weekend shopping, either weekly, bi-weekly, tri-weekly, or monthly. Shopping on a weekly and monthly basis is, however, the most common habit, which is easily understood when dealing with grocery shopping.

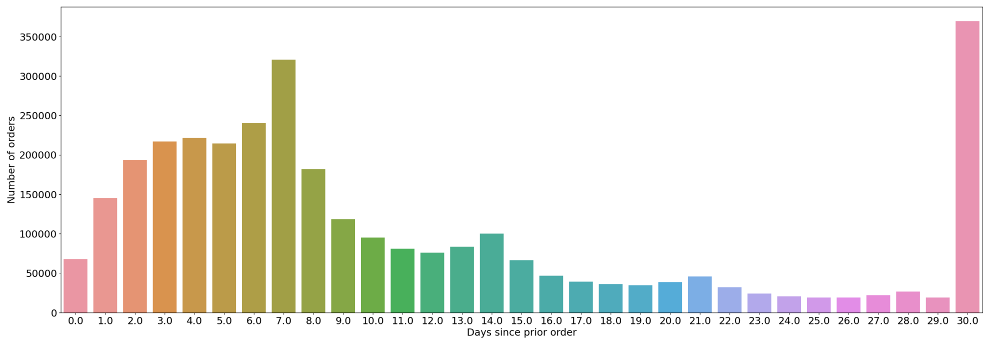


Figure 6.4: Distribution of orders by days since prior order

Figure 6.5 shows the distribution of orders by day of the week, and it was generated by using the information for the day of the week in each order. There is a clear preference for shopping on days 0 and 1. Since days 2-6 have similar behavior, it is likely that 0 corresponds to Saturday and 1 to Sunday. This idea is also supported by a study that presents Saturday as being the most popular day to go shopping (Mitova 2020).

The Instacart dataset provides the day of the week as an integer from 0 to 6, but there is no official information regarding which day belongs to each value. At the time of writing, there are several topics asking for this clarification, but without an answer (e.g., the official data dictionary, by Instacart [5]).

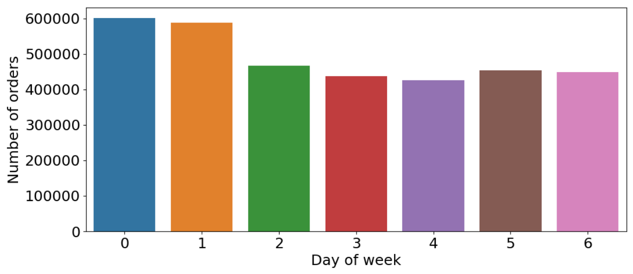


Figure 6.5: Distribution of orders by day of week

Figure 6.6 describes the distribution of orders by the hour of the day. It was generated by using the information regarding the hour of the day in each order. A peak for shopping is shown between 8 am and 6 pm, with slight increases at the beginning and end of the interval. Besides, a higher concentration of orders is present in both the hours immediately before

[5]https://gist.github.com/jeremystan/c3b39d947d9b88b3ccff3147dbcf6c6b

and after lunch. This behavior seems to reflect the overall day of people, where shopping is done after and before work activities, and around lunchtime, either by using the lunch period for grocery shopping or to buy food.

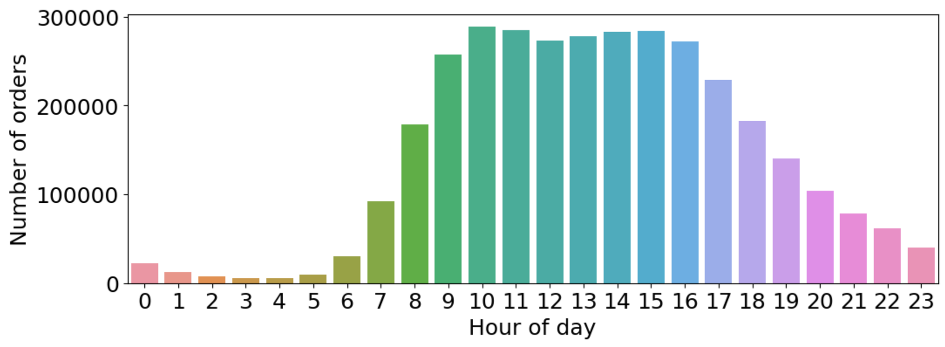


Figure 6.6: Distribution of orders by hour of day

Figure 6.7 projects the information in the previous diagram to the day of the week it happened. The same behavior is evidenced, keeping the same preference periods. However, two interesting peaks are shown in the afternoon of day 0 and the morning of day 1, which might value the theory of 0 corresponding to 0 and 1 to Sunday.

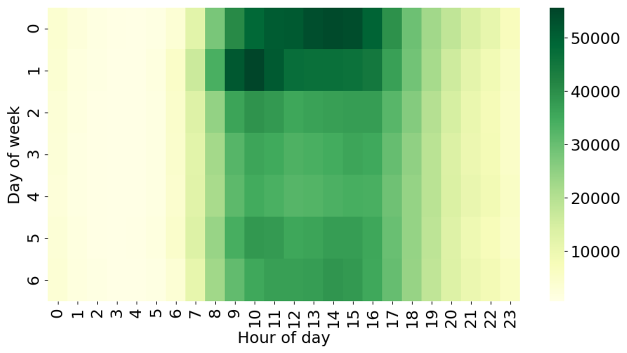


Figure 6.7: Distribution of orders by hour of day and day of week

The position a product is added into a specific order is also a parameter present in the Instacart dataset. Figure 6.8 shows the distribution of the product position in all the orders. This bar plot was build by grouping the positions for each product in all the training orders, and counting and summing each occurrence. It is possible to observe that, naturally, there are more products added in the first positions because of the order size - the most significant amount of products correspond to the first 10 positions, which correlates with the average order size being 10 (as seen in 6.2.2). This value drops progressively, approaching 0 for positions over 80.

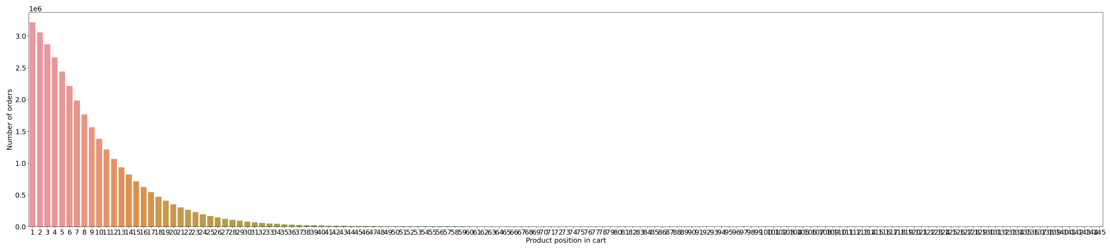


Figure 6.8: Distribution of the product position in the orders

In addition to the number of products per position, it is relevant to check the behavior of the reordering flow of products. The figure 6.9 represents the distribution of reordered products per order. This plot was generated by grouping the products in each order and summing the amount of reordered items present. Besides a large number of orders with 0 reordered items (which are easily justified by the more than 200k first orders existent), it is possible to observe that most orders have up to 10 products that are recurrent for that customer. This value drops gradually, approaching 0 when the number of reordered products per order is 30.

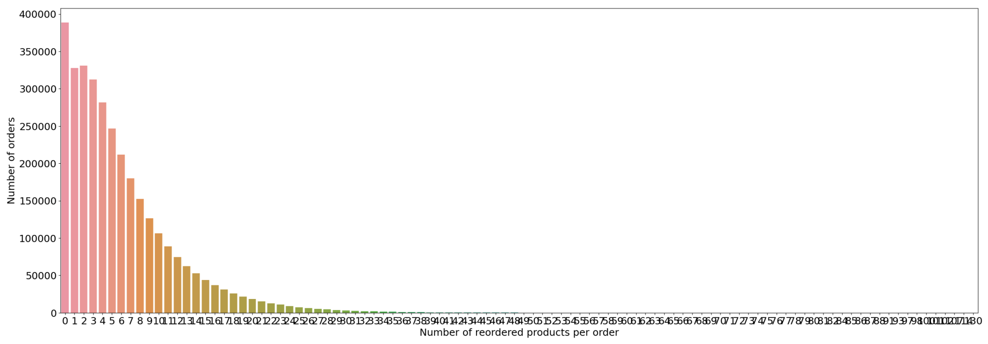


Figure 6.9: Distribution of reordered products per order

In addition to the previous plot on reordered, it is possible to understand when reordering occurs with a higher rate during the shopping flow. The figure 6.10 represents the distribution of reordered items according to the position in the cart. This diagram was built by grouping the products in all the orders by the position they were added in the cart and counting and summing, which were reordered. The percentage was calculated using the total purchases and the total recurrent items for each position.

This diagram provides fascinating information: the regular purchases are added into the cart in the first place. There is an accentuated decrease in the reorder rate as the position in the cart goes forward. This diagram shows an uneven behavior when the position moves after 50 (even a bit earlier) - which has a direct correlation with the amount of reordered items in the cart, as seen in the figure 6.9, when almost no data exists past 30 reordered items in the cart.

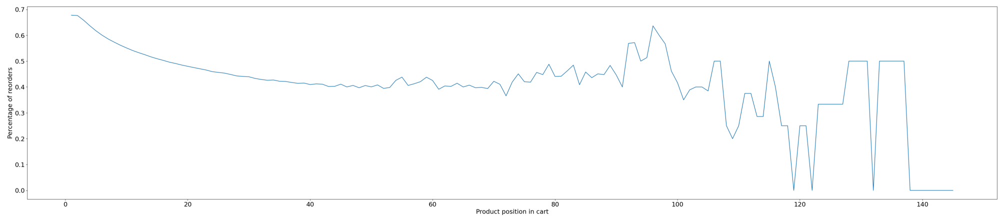


Figure 6.10: Distribution of reordered products per position in the order

By counting the number of times each product is reordered and projecting it into a diagram, as seen in figure 6.11, it is possible to understand the way reordering correlates with the kind of product. There is a visible peak at 0%, meaning that several products tend not to be reordered. This is easily justifiable in a grocery retail scenario, as people tend not to need some kind of products more than once in a lifetime (e.g., the support for the gas canister or floor mats for their cars). On the other hand, most products have a repeatability rate of around 50%. This percentage follows a curve where most products have a reorder rate between 20% and 60%.

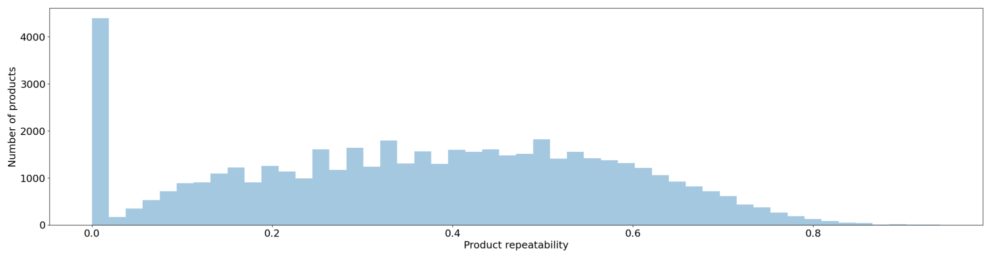


Figure 6.11: Repeatability of products

6.2.3 Obtaining The Target Variable

As studied in section 5.2.1, the way of retrieving the target variable may be dataset-specific, so the adopted one is analysed to understand how to retrieve it. The target variable could be represented via several ways, but two alternatives appeared to be specifically powerful: using the presence or absence of every product in a list as a label; using the reorder flag as the label. The main difference between these approaches is that the first one represents the purchase of every item in the list, and the second one the purchase of every reordered item in the list.

By targeting every purchased item, one would face a higher density of data, causing a higher temporal and computational load when training and predicting. The model would, theoretically, be able to suggest items that a customer has never purchased before, but this would also bring a bigger chance of failing since more products would be considered.

Targeting only the products previously purchased by a customer would be lighter when it comes to the amount of data and time spent training, but would leave behind the suggestion of new products. However, by recommending only products that a customer has already bought in previous transactions, a smaller chance of failing would, theoretically, exist.

There was consensus with the advisors to follow the second approach. This way, the target variable represents the reorder of an item (where 1 means reordered and 0 means not reordered), and recommendations are purely consisting of previously reordered items.

6.2.4 Feature Engineering

As studied in section 6.2.1, prior orders provide a clear way of having all the transactions but the last one, for every customer, and orders and products are present in their own tables. In addition to the tables representing orders and products, a logical table joining prior orders with orders and products is created as a way of providing detailed information regarding each order and the items that constitute them.

This combination provides a way of representing a very detailed view of every prior order existent in the dataset. Since departments and aisles do not have many details, and their name would provide a very sparse representation, only their unique identifiers are used when joining the data regarding products with orders.

All these tables represent several occurrences of customers buying products, and by combining them, certain shopping habits are enhanced, and they are intended to be perceived by the model. This provides us with the ability to cross information and calculate different features.

As studied in section 5.2.2, the solution expects the presence of four types of features: customer-related, product-related, and customer-product-related features, as well as features extracted explicitly from the data. Also, the TensorFlow framework supports numerical and categorical features (TensorFlow 2020b) - where numerical features correspond to quantitative information, represented as numbers, and categorical features correspond to information represented as groups or categories.

This section describes how the dataset was used to extract the different features used to feed the model in both training and predicting pipelines of the implemented solution (see section 5.2). An analysis on the the extraction of different types of features is performed at the end of the section.

Customer-related Features

The Instacart dataset does not provide many details regarding customers. There are no demographic or biological details present. Despite this fact, plenty of useful details can be obtained when joining the information regarding their previous orders.

There were generated 12 different features regarding each customer, and they are described below.

1. **c_num_purchased_products** - This numerical feature represents the total number of products purchased by the specific customer and is calculated by grouping all the previous orders by customer identifier and counting. It is helpful to understand the type of customer but only combined with the other features.

2. **c_unique_purchased_products** - This numerical feature represents the number of different products purchased by a specific customer. It is calculated by grouping all

the previous orders by customer identifier and counting how many unique products are present. It was chosen as an attempt to understand how experimental is the customer.

3. **c_unique_prior_orders** - This numerical feature represents the total number of unique orders for a specific customer and is calculated by grouping all the previous orders by customer identifier and counting the unique number of orders.
4. **c_num_reordered_products** - This numerical feature represents the number of products that a specific customer has reordered. It is calculated similarly to the features before, but counting how many of the products a user has bought correspond to reorders and is intended to help the model identifying customers with more stale shopping habits.
5. **c_avg_days_since_prior_orders** - This numerical feature measures the average amount of days since an order. It is calculated by grouping the orders by the customer identifier and measuring the average value for the number of days since the previous order. It was chosen as an attempt to help representing the customer's shopping frequency.
6. **c_num_orders** - This numerical feature represents the number of orders for a specific customer and is calculated by grouping all the previous orders by customer identifier and counting the number of orders. It is similar to the number of unique orders and serves only the purpose of making sure that the model can handle eventual splits between orders in the dataset. This feature was chosen as an attempt to represent how frequent or usual the customer is.
7. **c_avg_basket_size** - This numerical feature measures the average basket size for a specific customer. It is obtained by dividing the number of purchased products by the number of unique orders (features already obtained and detailed) and is useful to understand what kind of customer the model is dealing with (i.e., customers with a preference for small or big purchases).
8. **c_ratio_reorders** - This numerical feature measures the ratio of reorders for a specific customer (i.e., how many of the products bought were reordered). It is also obtained by the result of the division of two already existing features, the number of reordered products, and the total number of purchased products. This feature, as well as the number of reordered items, is chosen as an attempt to identify customers who take fewer risks and are more loyal to certain items.
9. **c_median_order_dow** - This numerical feature corresponds to the median value of the day of the week of all the orders of a specific customer. It is calculated by simply grouping all the orders by the customer identifier and calculating the median value of this parameter.
10. **c_last_basket_size** - This numerical feature corresponds to the number of items present in the last basket of a specific customer. It is calculated by simply counting the number of items in the last order. It was chosen as an attempt to value extraordinary last purchases and identify size patterns when predicting a new one.
11. **c_orders_with_new_products** - This numerical feature represents the number of orders with new products (i.e., products never bought before by that customer), and is calculated by grouping the items in each order by the customer identifier and counting

the products that were not reordered. It was calculated in order to attempt to represent the kind of shopper being dealt with.

12. **c_std_days_since_prior_orders** - This numerical feature represents the standard deviation on the days since a previous order. It is calculated by simply grouping all the orders by the customer identifier and calculating the standard deviation of this parameter. This feature was chosen as an attempt to represent how agitated is the customer's shopping behavior.

Product-related Features

The dataset includes simple details regarding products (i.e., identifier, name, and which aisle and department they belong to). However, one can extend the amount of information known regarding each item by calculating some metrics over the data.

There were generated 6 different features regarding each product, and they are described below.

1. **p_num_orders** - This numerical feature represents the number of times a specific product has been purchased. It is calculated by grouping the order details by the product identifier and counting the occurrences. This feature was chosen as an attempt to detect the items that are ordered the most.

2. **p_num_reorders** - This numerical feature represents the number of times a specific product has been reordered. It is calculated by grouping the order details by the product identifier and counting this parameter. It was chosen as an attempt to detect the items that are ordered in a more recurrent way.

3. **p_ratio_reorders** - This numerical feature describes the ratio of reorders for a specific product (i.e., how many times the product was bought as a reorder out of all the times it was bought). It is calculated by dividing the two already known features, number of reorders, and number of orders. This feature was used in order to help to identify the more popular items, with the help of the other features.

4. **p_num_customers_purchased** - This numerical feature measures the number of times a customer has bought that specific product. It is calculated by simply counting the number of times it appears when grouping the order details by customer and product identifiers.

5. **p_num_customers_one_shot_purchased** - This numerical feature measures the number of times that specific product was bought as a one-shot in all its buying record (i.e., how many times someone has bought the product only once in their whole shopping history). This feature is calculated by grouping the order details by customer and product identifiers, calculating the number of occurrences, and counting how many of those correspond to an isolated purchase. The interest in this feature was originated in trying to help the model identify items that, typically, people do not need to buy more than once (e.g., the support for the gas canister).

6. **p_ratio_one_shot_customers** - This numerical feature measures the ratio of one-shot purchases for a specific product. It is simply calculated by dividing the number of times it is bought as a one-shot item by the total number of purchases, and is intended to serve the same purpose as the number of one-shot occurrences.

Customer-Product-related Features

In addition to the features regarding products and features, it is possible to cross both entities and extract features regarding their relationship. These features aim to clarify the shopping behavior between customers and all the products they have already purchased.

There were generated 5 features regarding this relationship, and they are described below.

1. **cp_num_orders** - This numerical feature describes the number of times a specific product was bought by a customer. It is obtained by grouping the order details by customer and product identifiers and counting the occurrences. This feature was designed to help to identify the favorite items of each customer.
2. **cp_num_reorders** - This numerical feature represents the number of times a product was reordered by a customer. It is obtained in a similar way to the number of orders, but counting the number of times it was reordered.
3. **cp_ratio_reorders** - This numerical feature measures the ratio of reordering by a specific customer for a specific product, and is simply calculated by dividing the number of times a product was reordered by that customer by the number of times it was ordered. It is also intended to help to identify the most bought items of a customer.
4. **cp_avg_order_in_cart** - This numerical feature describes the average position a product is bought by a customer in all the purchases. It is calculated by grouping order details by customer and product identifiers and calculating the average of the position in the cart. This feature is intended to clarify in which part of the shopping flow a product is typically bought by the customer.
5. **cp_order_strike** - This numerical feature describes how recently and/or frequently a product is bought by a customer. It is calculated by applying a function that tends to 0 when the variable tends to infinite, to the reverse value of the order number (i.e., if the user has 10 orders, the first one will have the value 10). The function used was simply $(1/2)^n$, since it returns a value that, the bigger the value of n, the smaller it is. This way, older orders (which will have the highest values for order number) will have the lowest importance. This allows us to sum the result of this function in all the times the product was bought by the user, and obtain a strike value that, the bigger it is, the more recently a user has bought the product. This feature is intended to allow the model to identify products bought recently or frequently by the customer, and those that it has probably stopped buying.

Explicit Features

The dataset provided important information that was explicitly extracted in order to generate some features. These features represent information directly related to the orders and products.

There were generated 6 explicit features, and they are described below.

1. **order_hour_of_day** - This numerical feature represents the hour of the day where a specific order was made, and it was directly provided in the dataset associated with each order. This feature aims at helping to associate the purchase of certain items with specific times of the day.

2. **department_id** - This categorical feature represents the department where a certain product belongs to, and it is directly associated with each product. This feature aims at helping the model dealing with the proximity of items, both physically and categorically.
3. **days_since_prior_order** - This numerical feature represents the number of days since the previous order. This feature is also provided directly in the dataset and aims at helping the model inferring time-related constraints, such as seasonality.
4. **add_to_cart_order** - This numerical feature represents the position where a certain product was added in a certain purchase, and it is provided in the order details. The goal of this feature is to help understanding reorder patterns, since the study performed in 6.2.2 shows a higher reorder rate associated with lower positions in the cart.
5. **order_dow** - This categorical feature represents the day of the week where a certain order was done and is also provided directly associated with each order. This feature aims at helping the model to deal with time-related constraints, such as the preference for certain items in certain days (e.g., weekend-purchases).
6. **aisle_id** - This categorical feature represents the aisle where a certain product belongs, and it is directly associated with each product. It also aims at helping the model dealing with the proximity of items, both physically and categorically.

Selecting the Extracted Features

Choosing which features to develop, how to calculate them, or which to keep was an iterative and experimental process. The analysis of the documentation on the usage of machine learning algorithms was important to understand which kind of general features are typically used for similar scenarios. Reading and experimenting with several notes, discussions, and solution proposals available for the Instacart Kaggle competition, as well as works regarding other recommender systems around the retail area, made it easier to select which features to pursue.

Besides these features, several others were considered and experiment, but for multiple reasons had to be left behind. For instance, considering the number of orders in a row that each customer bought each product caused a massive increase in the computational time evolved in the process and brought little to no benefits.

Some features brought, however, clear benefits and thus had to be considered. For instance, the feature *cp_order_strike*, which was analysed in 6.2.4 is also very computationally and time expensive, but was able to contribute in a significant way to the model results.

Multiple features are computationally inexpensive and seem to have little relevance to the model, but the results have proved us wrong. All the experiments made around features made it visible that, just because one thinks a feature is useful or can benefit the results in a significant way, it does not necessarily work that way. However, this does not mean that business knowledge is useless. It is of paramount importance to know the domain in order to judge on feature selection and have intuition on where to experiment. Learning better data representations improves the results of machine learning algorithms (Olson et al. 2017).

TensorFlow allows the generation of a diagram presenting the top-N features that have contributed the most to the model, helping this selection. As an example, the figure 6.12

represents this diagram for the top 10 features for the final model configuration (see the journey to find the final configurations in section 6.3).

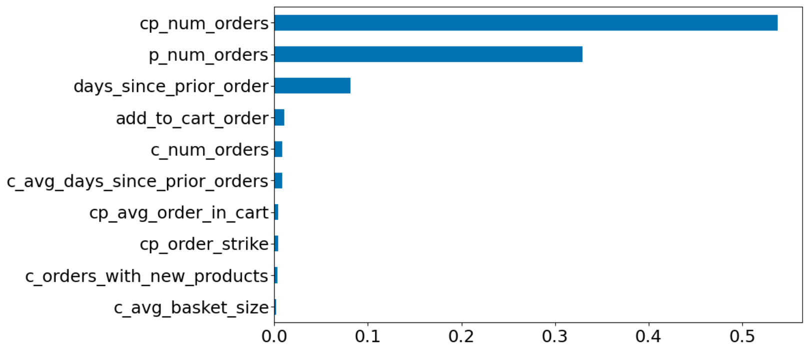


Figure 6.12: Feature contribution diagram for the chosen model configuration

One can evidence that, for this specific model, the feature that contributes the most is *cp_num_orders*, which represents the number of times that a product was purchased by that specific customer. This feature is followed by *p_num_orders*, which represents the number of purchases of a specific item. The next most-contributing features are the number of days since the last order of the specific customer (days_since_prior_orders) and the position occupied by an item in the cart for a specific customer and a specific order (add_to_cart_order). The heavy feature *cp_order_strike*, referred to above, is the 8th most powerful feature for the model.

Computational Challenges

The amount of data present in the dataset, increased by the processes described in section 5.2 brings some challenges when performing heavier operations in the data.

The customer-product feature *cp_order_strike* (see section 6.2.4) requires the parameter representing the position where each product was added into each order, to be reversed. During training, this operation has to go through 3 million orders; when predicting, this number will depend on the number of previous orders that a customer has, but it will be significantly lower. Nonetheless, this is a very heavy computation in both scenarios. During training, this step was, initially, taking more than 45 minutes alone to execute.

When preparing the TensorFlow example columns (see section 5.2.6), the whole predicting dataset needs to be converted, thus every row and every column need to be accessed. Depending on the number of orders and products, this process can also take some time to execute. On average, more than 2 seconds were being taken to execute, which is considerable when thinking that a customer might be waiting for it to finish.

In order to reduce the time spent performing these operations, a multi-threaded approach was developed. A pool of processes (scaled by Python, depending on the hardware it is running on) processes chunks of the whole dataset in parallel. The amount of data per process is also managed by the language to keep it optimized.

This parallel mapping allows an average CPU usage of more than 95% during these operations, and a reduction of around 7 times the time spent (probably depending on the hardware). With this modification, the first process was reduced from 45 minutes to less than 7 minutes, and the second one from more than 2 seconds to around 0.3 seconds, on the test environment (see section 6.1.4).

This computational challenge brings another issue to the table: by using so much CPU for each predicting process, the scaling of this solution will, inevitably, suffer. The predicting time will increase under higher stress loads, as the CPU will be occupied by many processes (plus the additional time spent by the process scheduler). This solution is capable of providing better results if the Machine Learning Model Interface is scaled horizontally.

6.2.5 Adopting a Machine Learning Algorithm

The study on state of the art for machine learning algorithms for the current problem has evidenced three algorithms as being more promising (see section 4.2.5): gradient boosted trees, SVMs and neural networks. However, the support for SVMs using TensorFlow estimators has been deprecated (TensorFlow 2018), which makes this algorithm unable to be further studied using the developed architecture as is. So, gradient boosted trees, and neural networks were compared against other algorithms, to understand which should be adopted in the following tests.

A simple linear classifier was also used, to serve as a baseline, as well as a combined technique mixing a deep neural network and a linear classifier (commonly referred to as *wide-n-deep* approach). From the original three most promising approaches, a gradient boosted trees implementation and deep neural network implementation were used.

Since the main goal of this experimentation is to understand which algorithm to explore further, the tests have been performed using no hyperparameter tuning. Each algorithm was executed using the default parameter configuration, provided by TensorFlow. The comparison was performed using the training pipeline of the complete solution (which can be found detailed in 5.2.5). The predicted target variable was also the one used in the complete solution (detailed in 6.2.3).

Since all the algorithms were executed using the default hyperparameter configuration, the starting point was equivalent. The gradient boosted trees algorithm was configured with 100 trees, a maximum depth of 6, one single batch per layer and a learning rate of 0.1. Both the deep neural network algorithm and the wide-n-deep algorithm were built using two hidden layers of 50 and 25 nodes, respectively, and used the available *Adam* optimizer. The linear classifier was built using the available *Ftrl* optimizer.

The tests were performed over the adopted Instacart dataset to assure that they provide accurate conclusions. Two different test sets were conducted: test set 1, using a portion of the dataset (corresponding to 991714 orders over a total of 187435); and test set 2, using the complete dataset (see section 6.2.1). These two test sets were executed in order to understand the results in different amounts of data and avoid overfitting.

Before training, 20% of the training dataset is split for evaluation. During the evaluation, the TensorFlow framework predicts the target variable on this subset and provides access to evaluation metrics. The table 6.1 presents the results obtained by the compared algorithms, over the different test sets, as far as accuracy, precision, recall, f1-score, and duration are concerned.

Table 6.1: Comparison tests using different machine learning algorithms

test set	**algorithm**	**accuracy**	**precision**	**recall**	**f1-score**	**duration (min)**
1	Gradient Boosted Trees	0,916699	0,909790	0,953114	0,930948	24
1	Linear Classifier	0,798954	0,745651	0,999925	0,854268	12
1	Deep Neural Network	0,880106	0,855256	0,958862	0,904100	13
1	Wide-n-Deep	0,633094	0,634689	0,890384	0,741102	13
2	Gradient Boosted Trees	0,867172	0,838963	0,958773	0,894876	36
2	Linear Classifier	0,822802	0,786945	0,959452	0,864678	23
2	Deep Neural Network	0,843371	0,836527	0,912820	0,873010	25
2	Wide-n-Deep	0,645525	0,641447	0,904153	0,750474	25

The results detailed in the table 6.1 evidence overall best results obtained by the gradient boosted trees algorithm, throughout the different test sets, despite taking the highest to train. Looking at the f1-score (which is the combined effect of precision and recall), one can observe that this algorithm has obtained 0,930948 on the test set 1, and 0,894876 on the test set 2. The second place was obtained by the deep neural network algorithm, which has achieved f1-scores of 0,904100 and 0,873010, respectively. The last position was achieved by the wide-n-deep algorithm, with 0,741102 on the test set 1, and 0,750474 on the test set 2.

These results led to the choice of the gradient boosted trees algorithm as the machine learning algorithm to explore in this dissertation. The results evidenced this algorithm as very successful during the evaluation phase, making it an excellent candidate to explore the hyperparameter tuning and to generate shopping list recommendations. Also, both advisors and the proponent company support its adoption. One key-benefit of the designed architecture, as well as TensorFlow, is that the algorithm can be changed in future if the business reality indicates so.

6.3 Tuning the Hyperparameters

A good model, as far as the goals of this dissertation are concerned, is a model that is able to achieve good recommendation results regarding precision, accuracy, and recall on a set of test predictions, but without compromising too much on the training and predicting duration. A model that achieves results that overtake another model's by a small percentage, needing several hours more to train, might not be a better model, as it can result in bigger challenges when being used in real-world scenarios. This section uses the solution prepared in the Instacart case study (see section 6.2).

TensorFlow provides multiple hyperparameters that support tuning. At the moment of writing this dissertation, all the tunable hyperparameters can be seen in the constructor of *BoostedTreesClassifier*, by TensorFlow, in the code example 6.1 (TensorFlow 2020a).

```
tf.estimator.BoostedTreesClassifier(
     feature_columns, n_batches_per_layer, model_dir=None,
    n_classes=2,
     weight_column=None, label_vocabulary=None, n_trees=100,
    max_depth=6,
     learning_rate=0.1, l1_regularization=0.0, l2_regularization
    =0.0,
     tree_complexity=0.0, min_node_weight=0.0, config=None,
    center_bias=False,
     pruning_mode='none', quantile_sketch_epsilon=0.01,
    train_in_memory=False
)
```

Listing 6.1: TensorFlow BoostedTreesClassifier Constructor

The literature presents learning rate, number of trees, trees' depth, and regularization parameters as the hyperparameters that provide the best results when tuning gradient boosted trees (A. Jain 2016a,b; TensorFlow 2020c). The learning rate corresponds to a shrinkage value added when adding a new tree to the model (TensorFlow 2020a). The number of trees represents the total number of trees created in the model. Max depth restricts the maximum depth of each tree to grow.

Regularization parameters, on the other hand, are common parameters to many machine learning algorithms, working as a cost related to large weights. In TensorFlow's implementation of this algorithm, they consist of multipliers that are applied to tree leaves in order to penalize and control growth (TensorFlow 2020a). Two different regularization parameters exist: l1 and l2 regularization. The first one is applied to the absolute weights of the leaves, whilst the second one is applied to the square weights of the leaf. L2 is typically more used as it does not encourage sparse models, since the penalty tends to zero for smaller weights (TensorFlow 2020c).

Because of the large number of tunable hyperparameters, this process has to follow a certain guideline. Despite the existence of some works published in this area, most of them target the process of tuning other machine learning algorithms. Furthermore, those who target gradient boosted trees tend to focus on more classical approaches such as XGBoost - which is not a problem, since most parameters are common to the different implementations.

The steps followed in order to tune the hyperparameters were based on an official paper by TensorFlow regarding the implementation of gradient boosted trees (Ponomareva et al. 2017) and on two articles describing some guidelines for tuning these models (A. Jain 2016a,b). Based on these sources, the following four-step algorithm was adopted to tune the hyperparameters of the TensorFlow gradient boosted trees model.

1. Start with a learning rate value that is high enough to allow fast tests but small enough to provide good results (typically between 0.05 to 0.2). The default value in TensorFlow is 0.1;
2. Find an optimum number of trees for the chosen learning rate (avoiding too big training times, since several tests are executed);
3. Tune tree-specific and regularization parameters (e.g., depth, batches per layer, l1 regularization and l2 regularization);
4. Lower the learning rate and increase the number of trees.

A limitation of the TensorFlow implementation of gradient boosted trees is, as far as the research made at the time of this dissertation, the lack of an automatic way to optimize specific parameters. For instance, XGBoost allows the usage of cross-validation during each boosting iteration of the training phase to calculate the optimum number of trees (A. Jain 2016a). Nevertheless, this automation would perform similar tasks to the ones described, thus the achieved conclusions should not be much different, but would probably allow the saving of research and experimentation time.

6.3.1 Test Conditions

As analyzed in the previous chapter (see section 6.2.1), by not considering the last order of each customer when training the model, one is allowed to generate shopping list predictions and to compare them against the actual purchase done by the customer. Similarly to the detailed evaluations and comparisons are done in chapter 6, a subset of customers was used in this step to evaluate the recommendations. To keep the results reliable and within a reasonable time span, 100 customers were used for these tests.

Several tests over the steps from the previous tuning algorithm were performed. Each trained model was used to predict recommendations for each one of the 100 test customers and evaluated. The average results of accuracy, precision, recall, and f1-score were registered, as well as the duration and the number of customers where the model was not able to predict a single item from the actual purchase (this is not an official metric, but it provides a good high-granularity point-of-view on how the model is behaving), referred in this dissertation as *zeros*. Details on the comparison metrics are present in section 3.2.5.

The tables 6.2 and 6.3 present the most important tuning tests performed. The first table describes the goal of the test (according to the tuning algorithm adopted) and the different steps involved. The second one presents the different hyperparameter configuration of each test. Both tables are indexed by the test number, so the information can be crossed.

Table 6.2: Descriptions of the hyperparameter tests

test #	**test description**	**step description**
1	finding a good learning rate	starting with a high learning rate
2	finding a good learning rate	trying a smaller learning rate
3	finding a good learning rate	even smaller learning rate
4	higher number of trees	higher number of trees
5	tuning tree-specific and regul. params.	more batches per layer
6	tuning tree-specific and regul. params.	adding l2 regularization
7	tuning tree-specific and regul. params.	higher l2 regularization
8	tuning tree-specific and regul. params.	adding l1 regularization
9	tuning tree-specific and regul. params.	best l2 value and more trees
10	tuning tree-specific and regul. params.	more batches per layer
11	tuning tree-specific and regul. params.	no l2 to assure it is useful
12	tuning tree-specific and regul. params.	keeping l2 but higher max depth
13	higher number of trees	higher number of trees
14	higher number of trees	even higher number of trees
15	tuning tree-specific parameters	more batches per layer
16	higher num. trees and tuning tree params.	more trees and batches per layer
17	higher num. trees and less learn. rate	more trees and less learning rate

Table 6.3: Values used per hyperparameter test

test #	**n trees**	**max depth**	**batches layer**	**learning rate**	**l2 regul**	**l1 regul**
1	100	7	1	0.1	0.0	0.0
2	100	7	1	0.01	0.0	0.0
3	100	7	1	0.05	0.0	0.0
4	150	7	1	0.01	0.0	0.0
5	150	7	2	0.01	0.0	0.0
6	150	7	2	0.01	0.001	0.0
7	150	7	2	0.01	0.05	0.0
8	150	7	2	0.01	0.05	0.01
9	200	7	2	0.01	0.001	0.0
10	200	7	3	0.01	0.001	0.0
11	200	7	3	0.01	0.0	0.0
12	200	8	3	0.01	0.001	0.0
13	230	7	3	0.01	0.001	0.0
14	260	7	3	0.01	0.001	0.0
15	230	7	4	0.01	0.001	0.0
16	260	7	4	0.01	0.001	0.0
17	260	7	4	0.007	0.001	0.0

Tests were started with 100 trees, a learning rate of 0.1 and no l1 or l2 regularization, which correspond to the default values in TensorFlow. The starting max depth was 7 (started with one more than the default one, because of the big amount of data) and 1 batch per layer was used, corresponding to the lowest value allowed.

The main goal of these tests is to understand how the results evolve with different hyperparameter configurations and determine one that allows the model to achieve the best results in the best training duration achievable.

6.3.2 Test Results

The table 6.4 presents the average results for all the 100 test users of the different evaluation metrics, obtained in the different tests described in the previous section. The table is also indexed by the test number, to allow information to be crossed. For a matter of simplifying the analysis, these tests are described bellow synthetically, presenting only the average f1-score (i.e., the combined effect of precision and recall) and zeros. All the remaining details can be obtained in the tables.

Table 6.4: Results obtained in the hyperparameter tests

test #	avg accuracy	avg precision	avg recall	avg f1-score	num zeros	duration (min)
1	0.164181	0.275238	0.298533	0.260819	22	51
2	0.186305	0.309943	0.333533	0.291619	14	51
3	0.183724	0.304762	0.329686	0.287733	14	51
4	0.183857	0.305181	0.331590	0.288743	14	61
5	0.185343	0.308038	0.323476	0.289181	16	87
6	0.184086	0.307086	0.322419	0.288229	15	90
7	0.178876	0.299467	0.314610	0.280790	16	91
8	0.166743	0.278514	0.298333	0.263381	19	87
9	0.185267	0.308990	0.323829	0.289857	15	110
10	0.185648	0.309943	0.324581	0.290657	15	149
11	0.183133	0.306133	0.322543	0.287781	15	144
12	0.183552	0.307086	0.320895	0.287505	15	170
13	0.187705	0.311848	0.326276	0.292429	15	171
14	0.186162	0.309943	0.324495	0.290543	15	190
15	0.185381	0.308990	0.323752	0.289724	15	212
16	0.186581	0.310895	0.324895	0.291143	15	244
17	0.184495	0.308038	0.322952	0,288905	15	246

The average results for 100 users under the initial conditions (test 1) were a f1-score of 0.260819 and 22 zeros, taking 51 minutes to train. In an attempt to find a smaller learning rate, 0.01 was experimented (test 2), resulting in an increase of f1-score to 0.291619, a drop in the number of zeros to 14 and the same duration. Another test on the learning rate was performed, increasing the value to 0.05 (test 3), but it provided a slightly worse f1-score of 0.287733. A learning rate of 0.01 was, then, chosen to pursue the tests.

The number of trees was increased in an attempt to find the optimum number for the chosen learning rate, as specified in the tuning algorithm. Using 150 trees and the same previous conditions (test 4), the results dropped slightly (f1-score of 0.288743) and the duration increased to 61 minutes. However, it is premature to conclude that the number of trees was too big for the dataset without experimenting with a higher number of batches per layer or depth. The test 5 includes one additional batch per layer and the results got slightly better (the average f1-score increased to 0.289181, despite the increase in the number of zeros to 16 and the duration to 87), which is a good indicator that the number of trees is getting closer to the optimum value.

By adding a l2 regularization of 0.001 over the previous hyperparameter values (test 6), the f1-score suffered another slight drop to 0.288229, at the same time that the amount of zeros was reduced by one and the increase in the duration was insignificant. This parameter is described as being very helpful by the documentation when it comes to reducing overfitting (TensorFlow 2020c) so, despite the little drop in the efficiency, additional tests were done before deciding on its contribution for this model. The value of l2 regularization was increased to 0.05 in test 7, leading to an even worse result, as the f1-score dropped down to 0.280790 and the amount of zeros increased to 16 again.

In an attempt to increase the results of the model, a l1 regularization of 0.01 was introduced (test 8), but the results were reduced once again, as the average f1-score dropped to 0.263381 and the number of zeros increased to 19. Since this hyperparameter is known to cause models to become too sparse (TensorFlow 2020c), it was abandoned. Using the best l2 regularization value (0.001), the number of trees was increased to 200 (test 9), leading to an increase of the average f1-score to 0.289857, a reduction of the number of zeros to 15 and an increase in the duration to 110 minutes.

The number of batches per layer was increased in test 10, improving the f1-score value to 0.290657 and jumping the training duration to 149 minutes. Since good results were being obtained while increasing the trees' depth, number and batches per layer, a test without l2 regularization was performed (test 11), to assure that it was benefiting the results - without l2 regularization, the average f1-score dropped significantly to 0.287781, evidencing its importance for the results.

In an attempt of tuning even better the model, the number of max depth was increased to 8 (test 12). This test caused the model to perform slightly worse than before increasing this parameter, and made it take almost twice as much to train. This way, this parameter was left as was. The number of trees was, then, increased to 230 (test 13) and the results were positive: the average f1-score increased to 0.292429 and the number of zeros was maintained at 15, with a training time of 171 minutes.

Despite the positive results, the tests were pursued, and in test 14 the number of trees was increased to 260, causing the model to perform slightly worse and take 20 more minutes to train. The number of trees was kept as 230 and the number of batches per layer was increased to 4 (test 15), but the performance kept decreasing and the training duration following the opposite route. A mix of both previous tests was done in test 16, causing the results to improve, but being still worse than in test 13, but taking 73 more minutes to train.

Following the training algorithm, the number of trees was increased and the learning rate decreased in test 17, using 260 trees and a learning rate of 0.007, and the results were surprisingly lower: it obtained an average f1-score of 0.288905 and kept the same number of zeros as the previous test and approximately the same duration. This results were better than the previous tests, but still lower than the values achieved in test 13.

The results obtained in the last test lead us to believe that, in order to achieve better results, the number of trees, maximum depth and number of batches per layer have to be increased even more, causing the model to take significantly longer to train, as evidenced by the tests and defended by the literature (A. Jain 2016b). This means that the best ratio results/duration might have been reached in test 13, and that in order to improve it, the training duration might not be worth it. Since grocery retail is an agitated business area, with big amounts of data produced every day, having models that take too much to train may reduce the ability to update the recommender system frequently.

The figure 6.13 represents the projection of the average values of f1-score from the previous tests, against the training duration. This allows us to better perceive how the results got, typically, better with higher training times, and how it is predictable that the results outperform the ones obtained in test 13, with 171 minutes of training, but for significantly higher training times.

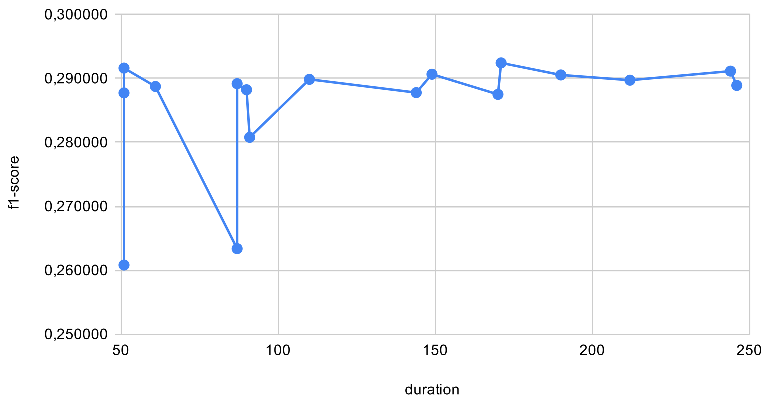


Figure 6.13: Training duration projected against the average f1-score

Predicting time, on the other hand, does not suffer big fluctuations with the different models trained. In all the different hyperparameter configurations, the overall prediction time (with all the architectural services and HTTP connections involved) is around 11 seconds. Most of this time is spent during data-related tasks, before feeding the model. All these gradient boosted trees models take around one second to predict on the whole predicting set.

With these considerations and results, hyperparameter tuning was concluded at this moment, and the satisfactory results and decent training duration of test 13 have led to the adoption of its hyperparameter configurations. The final model was trained with 230 trees, a max depth value of 7, 3 batches per layer, a learning rate of 0.01, and an l2 regularization value of 0.001.

The table 6.5 synthesizes the evolution obtained by the model before tuning the hyperparameters (i.e., test 1) and after training with a promising hyperparameter configuration (i.e., test 13).

Table 6.5: Results obtained before and after hyperparameter tests

description	**test #**	**avg accuracy**	**avg precision**	**avg recall**	**avg f1-score**	**num zeros**	**duration (min)**
before tuning	1	0,164181	0,275238	0,298533	0,260819	22	51
after tuning	13	0,187705	0,311848	0,326276	0,292429	15	171

6.4 Comparison Against Other Recommender Systems

After finding a hyperparameter configuration that allows the developed recommender system based on machine learning to achieve good results and have a decent training duration, one can compare it to different solutions, under different circumstances.

This section presents the different recommender systems compared, the existing test conditions, and the results obtained. It is concluded by a statistical test of the null hypothesis.

6.4.1 Recommender Systems Used for Testing

Two non-machine learning recommender systems were used as a comparison against the developed gradient boosted trees model. These two models represent two distinct approaches: a pattern-mining solution and a heuristic solution. The three models compared are detailed below.

Gradient Boosted Trees Model

The gradient boosted trees model represents the model prepared using the algorithm, dataset and features from the Instacart case study (see section 6.2), and optimized in section 6.3. It is a TensorFlow implementation of this algorithm and is ready to predict personalized shopping lists, with no information regarding quantity.

Details regarding its architecture can be found in the chapter 4; its development is widely detailed in the chapter 5; the optimization is done via the hyperparameter tuning in order to achieve the final configuration is detailed in section 6.3.

Pattern Mining Model

A periodic pattern mining model (see section 3.3.2) from the organization was used for the comparison. This type of non-machine learning algorithm aims at discovering temporal patterns within the dataset in order to generate predictions (e.g., preference for certain items during the weekends). Similarly to the gradient boosted trees implementation, this model was not trained using the last order of each customer. This model is a property of the proponent company, thus it is only briefly presented here.

This algorithm filters customers with less than 30 orders, restricting the number of users for which it is able to generate recommendations. It uses 80% of the data for training and the remaining as evaluation data. The PFPM algorithm [6] is used to find shopping patterns, a frequency and a score. This score is intended to penalize the most recent and old products.

Patterns with the lowest frequency or score are pruned. The periodic patterns are then sorted, and the absolute frequency of products and patterns is calculated, valuing the favorite ones. The last step is normalizing the scores and re-sorting the data. The 10 products with a higher score are recommended. The predictions generated by this model do not include quantity recommendations.

[6]https://www.philippe-fournier-viger.com/spmf/PFPM.php

SQL Heuristic Model

An heuristic-based model (see section 3.3.1) from the proponent organization was also used for the comparison. This non-machine learning approach is based on a Structured Query Language (SQL) heuristic that aims at finding products that should be repurchased based on the average buying frequency of each customer. This implementation uses the predicting moment as a time limit when accessing the data. This solution is also a property of the company, thus it is only briefly presented here.

When recommending a basket for a customer, this algorithm starts by accessing the shopping history and calculating the average shopping frequency and obtaining the period since the last purchase. Then, it searches for products that the number of days since their last purchase is higher than the average shopping frequency.

The most popular products for that customer that were not bought between the last two weeks from the recommendations and the average frequency are also obtained. These two sets of products are used to create a list with a configurable amount of products, with an equal amount of products from each list (if the number is odd, the first set includes an extra element).

Unlike the gradient boosted trees and pattern-mining models, this heuristic is ready to predict quantities. However, since the adopted Instacart dataset does not provide such information, this feature is not used.

6.4.2 Test Conditions

The comparison between this 3 classifiers was performed by compiling the results of each model when predicting the last purchase of a set of customers. Since the last purchase was not considered when training the gradient boosted trees model or by the pattern mining algorithm, and the SQL heuristic ignores it by design by using only the information before the predicting time, it is safe to compare the recommended shopping list against the last purchase to evaluate the results (see section 6.2.1).

The evaluation is done using the metrics accuracy, precision, recall, f1-score, and zeros (the number of customers where the model could not predict a single item from the actual purchase).

A subset of 360 customers was arbitrarily chosen, and 3 distinct sets of tests were performed, varying the number of products being recommended. Since the average cart size in the adopted dataset is 10 (see section 6.2.2), it was the first value chosen for the maximum recommendation size. The two additional sets of tests were performed, recommending a maximum of 7 and 5 products, respectively. The reason behind adopting two values smaller than the average was due to trying to reduce errors inserted by a lack of certainty by the recommender systems associated with a higher number of predictions.

The main goal of these tests is to understand how the developed solution behaves when comparing to solutions with a similar purpose. The variation on the amount of data is to remove any possible bias associated with the number of items predicted, not to find the best ones. That decision may be more business-related than technical, so it is not approached in this dissertation.

6.4.3 Test Results

The average results of each model regarding the accuracy, precision, recall, f1-score, and zeros are compiled in the table 6.6. For each test set in the table, there are three rows, representing the results attained by each model. The model description, in the column with the same name, is abbreviated in the table: GBT stands for the gradient boosted trees model, PM to the pattern mining model, and SQL to the SQL heuristic.

Table 6.6: Results obtained by the thee compared recommender systems

test set	num customers	num items	model	avg accuracy	avg precision	avg recall	avg f1-score	num zeros
1	360	10	GBT	0.169613	0.300591	0.295923	0.269607	50
1	360	10	PM	0.158481	0.291137	0.267530	0.253327	64
1	360	10	SQL	0.053492	0.110294	0.105755	0.096022	142
2	360	7	GBT	0.158852	0.337198	0.239832	0.253349	65
2	360	7	PM	0.143753	0.316618	0.219819	0.232124	76
2	360	7	SQL	0.042126	0.104953	0.069192	0.075580	189
3	360	5	GBT	0.149146	0.381868	0.199643	0.237115	74
3	360	5	PM	0.121412	0.328066	0.168475	0.198989	97
3	360	5	SQL	0.038679	0.116437	0.059544	0.069102	210

Unlike the analysis performed in the different hyperparameter tuning tests (see section 6.3), these tests do not have the same amount of products being predicted, meaning that one cannot compare the f1-score alone, between the different test sets. The divider when calculating precision is the number of predicted products, thus it is expected to grow inevitably. Besides, since recall is built dividing the number of correctly predicted items by the number of bought items, its value is expected to decrease with the decrease in the number of predicted items. The f1-score is obtained by these two values, so it is also influenced. The accuracy metric is obtained considering the predicted and not predicted data, making it a bit more adequate to the analysis, however it cannot be compared alone. Details on the metrics for this domain can be found in the section 6.1.2.

With this in mind, comparisons between test sets can be simplified using the average f1-score, since the different models were executed under the same circumstances. Between test sets, the accuracy allows us to perceive with a higher granularity how the different models evolved with the changes in the condition. The number of zeros is a good indicator of how broad the model is, thus it is useful in the different comparisons.

When recommending a maximum of 10 items (test set 1), the gradient boosted trees model achieved an average f1-score of 0.26960 and 50 users where it failed the prediction completely. For the same test, the pattern mining model obtained significantly worse results, having an average f1-score of 0.25332 and 64 zeros. The SQL heuristic obtained the worse results, having an average f1-score of 0.09602 and 142 where it could not predict a single item from the list.

When generating baskets of a maximum of 7 items (test set 2), the gradient boosted trees model obtained an average f1-score of 0.25334 and 65 zeros. The pattern mining model achieved a slightly worse average f1-score of 0.23212 and an increased number of zeros of

76. The SQL heuristic obtained the worse results again, having an f1-score of 0.07558 and a number of zeros of 189.

When recommending a maximum of 5 items (test set 3), the gradient boosted trees model obtained the best results once again, having an average f1-score of 0.237115 and increasing to 74 zeros. For the same test set, the pattern mining obtained an average f1-score of 0.198989 and 97 zeros. The SQL heuristic occupied the third place once again, having an average f1-score of 0.069102 and 210 customers where it could not predict a single item correctly.

It becomes evident that the different positions are maintained by the different models across the test sets. The gradient boosted trees model occupies the first position with a slightly better average f1-score in every test set, as evidenced in the figure 6.14, and a much lower number of customers with no correctly predicted products. The SQL heuristic obtains the worst results in all the test sets in the different metrics as well.

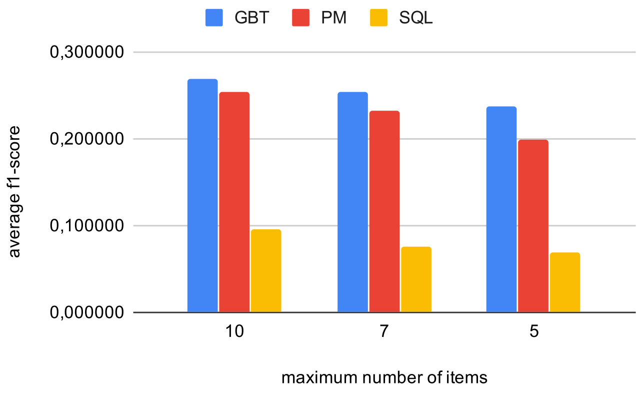


Figure 6.14: Variation of the average f1-score between tests for the 3 compared models

Despite the number of customers where the prediction failed completely (i.e., zeros) having increased when the maximum number of recommended items decreased, one can observe that the gradient boosted trees model presents the smallest increase: it failed the predictions to 24 more customers when comparing test sets 1 and 3, while the pattern mining and SQL models failed to 33 and 68 more customers, respectively. The figure 6.15 presents the relationship between zeros and maximum predicted items.

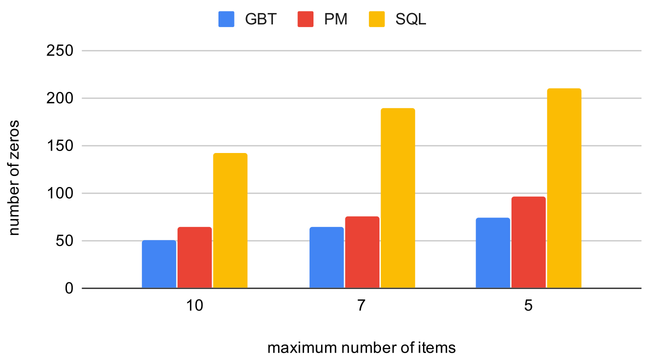


Figure 6.15: Variation of the number of zeros between tests for the 3 compared models

In addition to the absolute number of zeros, the percentage of this value in the whole test customers set is visible in the figure 6.16. The increase obtained by the gradient boosted trees model is the lowest. Both pattern mining and SQL implementations had a higher accentuation, but in different moments: the higher increase for the pattern mining algorithm was when reducing the maximum predicted items to 5, whilst the SQL heuristic was when reducing this value from 10 to 7.

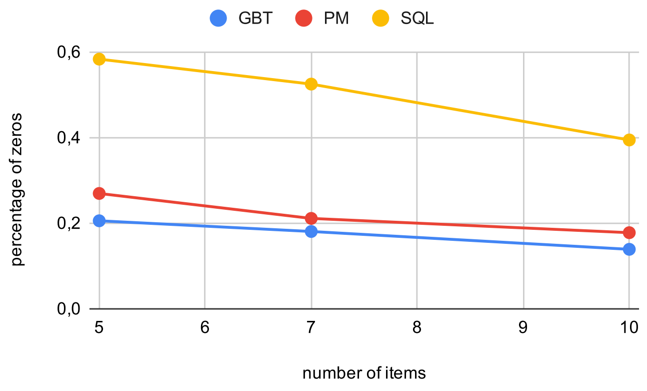


Figure 6.16: Percentage of zeros between tests for the 3 compared models

The variations regarding the accuracy variation across the different test sets were slightly different. Both the gradient boosted trees and the SQL heuristic implementations suffered a slight drop when decreasing the maximum predicted products, while the pattern mining algorithm suffered a more accentuate drop. The figure 6.17 shows the accuracy variation across the different test sets. Once again, the gradient boosted trees implementation shows itself as a slightly more robust solution.

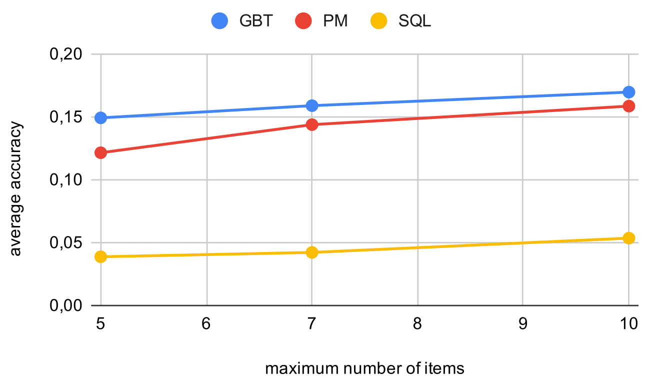


Figure 6.17: Variation of accuracy between tests for the 3 compared models

Because the different metrics are presented as the average value, the increase in the zeros may lead to devaluing the better results of the gradient boosted trees implementation to the detriment of the pattern mining model. However, it also means that the first model is significantly more robust and generalized. It is able to provide useful recommendations to a higher number of customers in the dataset, meaning that it was able to learn in a way that made it know a little bit more about the different individual shopping habits.

The different comparisons and results make it possible to conclude that the developed solution (i.e., the gradient boosted trees approach) when compared to the two comparison recommender systems used, is able to provide overall better results. It stands out the most when it comes to consistency: it shows the highest capacity of providing useful recommendations to a bigger portion of the test customers and the fewer variations associated with the maximum amount of predicted items.

A dissection of some of the recommendations generated by the developed recommender system can be found in the appendix B.

6.4.4 Testing the Null Hypothesis

As presented in 6.1.1, the null hypothesis of this work (in particular, with the previous comparisons) can be defined as all the different recommender systems behaving in the same way, under the same data and conditions. By having more than two classifiers and because no normal distribution is expected to be followed, the non-parametric Friedman statistical test can be used (Gomes 2016) to test the null hypothesis.

Friedman test is used to validate differences between groups and can be executed over a random population from the dataset, measured under 3 or more different scenarios (Gomes 2016). To ensure these conditions, the f1-score results obtained by the 3 models compared in this section, for recommendations with a maximum of 10 items, were selected for 10 arbitrary customers. The reason behind choosing the f1-score is its coverage, since it combines both precision and recall, reducing any bias. This input data for the Friedman test is present in table 6.7.

Table 6.7: Input data for the Friedman hypothesis test over the 3 models

GBT	PM	SQL
0.242	0.242	0.121
0.533	0.533	0.286
0.235	0.125	0.118
0.32	0.32	0.083
0.421	0.471	0.105
0.125	0.125	0.25
0.4	0.267	0.133
0.25	0.167	0.083
0.261	0.174	0.091
0.296	0.519	0.222

The alpha value for p-value validation is 0.05 (Gomes 2016). The p-value obtained by the test (i.e., the result of the test, compared against the p-value) is 0.004320. The null hypothesis (h0) can be rejected since the p-value < alpha (Gomes 2016), meaning that not all the different recommender systems perform the same way under this circumstances.

The difference between p-value and alpha is significant, which reinforces the results obtained in the previous comparison between the three systems (see section 6.4.3).

The Friedman test was executed using the SciPy Python library, based on a public work on this matter (Brownlee 2018). The code used for the test can be found in the appendix C.

6.5 Association Rules in the Recommendations

Association rules provide knowledge regarding the data, and can be used in order to add value to a recommender system (see section 3.3.3). These rules represent the most common behaviors in the dataset (i.e., the more commonly bought together sets of products), but are completely customer-agnostic. This section uses the solution prepared in the Instacart case study (see section 6.2) and optimized in section 6.3.

At first glance, it becomes tempting to extract the most frequent association rules and use them as binary features in the training set for the developed gradient boosted trees model, but this idea was refuted for two reasons: first, the number of hyperparameters would increase significantly, causing much higher training and predicting times; second, by projecting these rules in the current training dataset, one could be causing a potentially erroneous interpretation of the model, as it could learn the rule as associating feature behaviors instead of just products.

However, by training the model, most of the rules are expected to be already present, since the training data corresponds to a set of purchases of a set of customers during a period, and some of the developed features are intentionally designed to help the identification of these patterns. This way, one might expect to add little value by finding a way of mixing the output of the machine learning model with the identified association rules.

This section describes the tests designed to understand the value that adding association rules to the solution could bring, and presents the results obtained.

6.5.1 Test Conditions

In order to understand how association rules were being perceived by the trained model, the recommendations used in the tests 6.4, where a maximum of 10 items were predicted for 360 random users, were analysed.

The Apriori algorithm (see section 3.3.3) was chosen in order to find association rules in the Instacart dataset. A set of 12 different tests, with different configurations of minimum support, confidence and lift, was executed and the resulting association rules saved. The table 6.8 presents the configurations used in the different iterations.

Table 6.8: Tests for identifying association rules in the dataset.

test #	**min support**	**min confidence**	**min lift**
1	0.0009	0.6	3
2	0.0008	0.6	3
3	0.0007	0.6	3
4	0.0006	0.6	3
5	0.0005	0.6	3
6	0.0005	0.6	2
7	0.0005	0.5	3
8	0.0005	0.5	2
9	0.0005	0.4	3
10	0.0005	0.3	3
11	0.0004	0.6	3
12	0.0004	0.5	3

As studied in 3.3.3, support measures the times a rule occurs in the data; confidence represents the percentage of transactions with the first part of a rule that also contain the second part; lift measures the number of times the second part is more likely to be bought when buying the first part.

The tests start with searching for the strongest rules in the dataset and then lightening the criteria. By starting with a minimum confidence of 60%, the algorithm searches for rules where 60% or more of the transaction contain the complete rule, within the transactions with the first part of a rule. This value was chosen to assure the detection of rules stronger than 50%.

With a minimum lift of 3, the algorithm searches for rules where the second part of a rule is, at least, 3 times more likely to be bought when the first part is bought. Once again, this value was adopted to make sure only the strongest rules could be identified.

The initial minimum support of 0.0009 means that the rule is present at least 3000 times in all the purchases (3000/3214874 (details regarding the amount of transactions in 6.2.2)). This value was adopted by performing some tests in an attempt to find a value so tight that no rules could be generated, in order to progressively experiment with slower values in the other tests (i.e., less frequent rules).

For each of these tests, the recommendations for the test customers were analysed in order to evaluate how these rules were being reflected. The results for each test contain the amount of rules found complete in all the test recommendations, as well as the incomplete rules (i.e, rules where only the first part of a rule was present).

The goal of these tests is to validate how strong the recommender system is as far as popular shopping habits are concerned, and understand if by using association rules, one could improve its results and make it stronger.

6.5.2 Test Results

The results of each test can be found in table 6.9. This table is indexed by the test number, allowing the information to be crossed with the table describing the tests (table 6.8). The results are analyzed bellow.

Table 6.9: Results for the association rules tests for 360 customers

test #	**num rules**	**num complete rules**	**num incomplete rules**
1	0	-	-
2	2	3	0
3	4	5	0
4	6	7	1
5	7	7	1
6	7	7	1
7	24	20	7
8	24	20	7
9	153	104	69
10	403	255	229
11	22	19	2
12	61	42	15

The first test, as designed, had a minimum support so small that no rules could be found with the desired confidence and lift values. The second test had a minimum support value of 0.0008 and the same other configurations, allowing 2 rules to be found in the dataset - these rules are the strongest in the dataset and were found 3 times in the recommendations, and were not found incomplete. As a curiosity, both these rules are regarding the association of different sparkling water flavors.

The minimum support was decreased again, to 0.0007 in test 3, allowing the identification of 4 distinct rules - these rules were found fulfilled 5 times in the 360 recommended shopping lists and were never found incomplete. The minimum support was then dropped down to 0.0006 (test 4), increasing the number of rules identified to 6 - they appear complete in 7 shopping lists and incomplete in 1 recommendation (i.e., only the first part of the rule appeared).

The minimum confidence and lift values were kept, and the minimum support was decreased once again, to 0.0005 (test 5), allowing the identification of 7 association rules - again, these rules appeared complete 7 times in the dataset and 1 time incomplete. This time, the minimum lift was decreased to 2 (test 6), in order to test its impact in the rules (since less strict rules could, theoretically, be detected) - this test caused no changes in the results.

The minimum confidence was dropped down to 0.5 in test 7, allowing weaker rules to be found, and a total of 24 rules were identified throughout the recommendations - these rules were found complete 20 times and incomplete 7 times. The lift was decreased again to 2, keeping the previous confidence and support values (test 8), causing no changes in the results again.

The minimum confidence value was relaxed once again, to 0.4 (test 9), keeping a minimum lift of 3 and a minimum support of 0.0005, causing the highest increase in the number of the identified rules - 153 rules were identified, and they were found complete in 104 recommendations, and incomplete 69 times. It is important to remember that these rules are significantly weaker than the ones found using higher confidence values.

In an attempt to verify the presence of even weaker rules, the minimum confidence value was decreased to 0.3 in test 10, allowing the identification of 403 association rules - they were found complete 255 times in the different recommendations and 229 times incomplete. The minimum confidence value was increased to 0.6, and the minimum support was decreased to 0.0004 (test 11), and 22 strict rules were identified - they were found entirely in 19 recommendations and 2 times incomplete.

The minimum confidence was decreased one last time to 0.5 for test 12, and 61 rules were found - they were found 42 times complete in the recommendations and 15 times incomplete.

Lowering the minimum confidence, the number of rules identified by the Apriori algorithm increased, naturally. The weaker association rules have a higher probability of not being present in the recommendations, which was evidenced throughout the different tests. The strictest rules, on the other hand, were present many times in the recommendations.

These results should not be interpreted quantitatively - it is not because the recommendations generated by the gradient boosted trees model include many association rules that it becomes stronger. The association rules are obtained using the transaction of all the customers, while the machine learning model generates recommendations using details regarding the particular behavior of each customer.

However, by comparing the evolution of the number of rules found complete and incomplete with their strength, it becomes evident that the strictest rules are recognized by the model and exist entirely in the recommendations. The weaker the rules, the more they were found incomplete in the recommendations. The fact that the number of rules found incomplete is significantly smaller than the ones found complete and that it increases with the decrease in the strength of the rules is a good indicator that the rules may actually not reflect the reality of that customer for those recommendations.

The gradient boosted trees model seems capable of identifying the strictest rules in the dataset and reflect that reality in the recommendations. However, it seems that it could benefit from including a heuristic layer of association rules in the recommendations pipeline. Maybe increasing the confidence of a suggestion from the raw recommendations list (output of the model, prior to resizing the recommendations) when a rule was fully present, using its confidence or lift, could benefit the final results. This could, nonetheless, decrease the reliability of the recommender system, since the results would be manipulated with the rule. As far as the research made at the time of this dissertation, it does not seem to exist any work combining these two approaches in the described way, so no conclusions regarding the potential benefit can be obtained beforehand, but it would be interesting to explore.

6.6 Performance Tests

This section uses the solution prepared in the Instacart case study (see section 6.2) and optimized in section 6.3. Performance tests aim to validate how stable and responsive a software component or architecture stays under different workloads. Since scalability was not a concern of this dissertation, the high computational costs of the predictions are expected to sacrifice the duration when multiple requests run in parallel. However, the architecture by itself is expected to remain responsive, allowing the solution to be scaled in the future and solve eventual performance issues.

These tests were developed after all the previous test sets, and hit the main use case: obtain a tailored shopping list recommendation (UC_C1). A set of tests were developed, testing sequential and parallel HTTP requests, simulating a real-world scenario. In order to simulate these workloads, the Siege tool was used (Fulmer 2012). Siege is an open-source tool for stress testing that allows the simulation of sequential and parallel requests.

The table 6.10 presents the results obtained in the different tests, where different numbers of requests were executed, both in parallel and sequentially. The total duration of each test and the availability (i.e., the ability to respond to every request) were also registered.

Table 6.10: Results for the performance tests on the developed solution

num requests	avg duration (parallel)	total duration (parallel)	avg duration (sequential)	total duration (sequential)	availability
1	-	-	10.46	10.46	100
2	13.94	14.25	10.39	20.78	100
5	29.60	29.77	10.48	52.43	100
10	56.77	60.63	10.37	103.68	100
20	131.58	136.33	10.44	208.78	100
30	193.39	197.45	10.37	311.34	100

The average durations were projected against the number of requests for both types of tests and are visible in the figure 6.18.

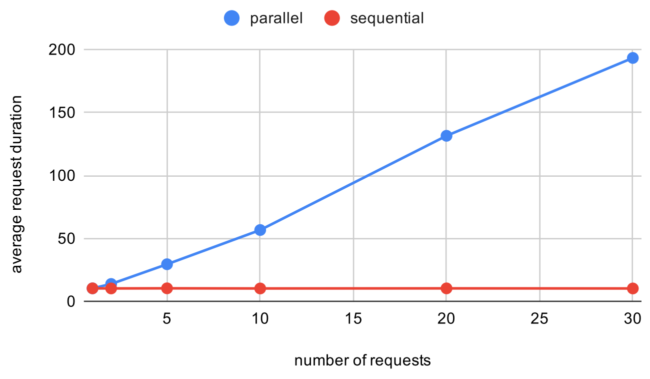


Figure 6.18: Average duration of parallel and sequential tests

Analyzing the previous table and figure, one can conclude that a recommendation takes around 11 seconds to be generated. This value follows a linear trend when only one request is made at a time. However, when the number of concurrent requests increases, the average response time follows the same trend. The average response time of 5 concurrent requests is 29.60 seconds, which is almost 3 times the duration of a single request.

When 10 parallel requests are performed, the average request duration increases to 56.77 seconds - more than 5 times the typical duration of a request. This value keeps following this linear trend, and, for 30 parallel requests, the average duration is 193.39 seconds - more than 18 times the duration of a request.

The increase in the average duration can be easily explained by the high CPU needs of the algorithm. The extraction of some features required some tasks to be parallelized in order to execute in (significantly) shorter times. A higher number of processes being scheduled in a scenario where each one uses a pool of processes that maximizes the CPU usage is inevitably translated into a scenario where the overall time is increased by the CPU time and time between process scheduling. This analysis shows that the process pools used by the Machine Learning Model Interface should be configured not to use the maximum CPU possible or that this component should be scaled horizontally (scenario where multiple instances of this component would be used in parallel).

6.7 Summary

A case study was developed in order to validate the implemented architecture. The adopted dataset for the case study was the Instacart Online Grocery Shopping Dataset since it represents a real-world grocery retail scenario. This public dataset contains data regarding 3 million orders and 200 thousand users and was used in a public Kaggle competition.

The dataset includes information regarding orders, products, aisles, departments, and order-products. Each customer includes N prior orders, and one order classified as either testing or training, randomly split (testing orders include no information about the products that constitute them). This last order is designed for the competition, but it is useful for this dissertation as well: by removing the last order of each user, one can generate recommendations for that moment and compare it with the real one.

Analyzing the dataset, one can conclude that 94% of the orders can be used when training the model (i.e., they are not the last order of each customer). The amount of orders of each customer floats between 4 and 100. The average number of products per order is 10. Around 88% of the orders include products already bought by the customer. It is possible to detect a preference for shopping on a weekend-basis, either being weekly, biweekly, triweekly, or monthly, with higher peaks weekly and monthly. There is a peak for shopping between 8 am and 6 pm, with slight increases at the beginning and end of this interval.

The target variable that reflects whether or not a certain condition led to the purchase of an item was chosen from two main approaches: using the presence or absence of an item as a label or using the information regarding an item in an order being reordered as a label. Despite the second approach meaning that only items that were already previously purchased by a customer can be recommended, it was selected as it allows lighter training and predicting datasets, as well as a smaller chance of failing. From the analysis of the dataset, one could extract the proposed four types of features: 12 customer-related, 6 product-related, 5 customer-product-related, and 6 explicit features.

Extracting useful features is a complicated process. Some features seem to provide useful business information and end up not improving the model. Others can be too computationally expensive to be worth it. However, good knowledge of the business helps the process.

Comparison tests were performed using the three most promising machine learning algorithms identified (gradient boosted trees, neural networks, and SVMs), to choose the algorithm to adopt in the case study. Since the support for SVMs using TensorFlow estimators has been deprecated, two other approaches were tested: linear and wide-n-deep algorithms. The tests were performed in the adopted dataset, using the default hyperparameter configuration. Gradient boosted trees outperformed the compared algorithms, being the adopted one. This case study was used in the different comparisons of this chapter.

When training a gradient boosted trees model using TensorFlow, the literature argues the learning rate, number of trees, trees' depth, and regularization parameters as being the most effective ones. In order to obtain a good result, several experiments were performed, where the output model was used to predict the last purchase of 100 customers, and evaluation metrics (precision, accuracy, recall, f1-score, and zeros) were analyzed. These tests lead to a final model configuration of 230 trees, a max depth value of 7, 3 batches per layer, a learning rate of 0.01, and an l2 regularization value of 0.001. The model takes around 3 hours to train, and predictions take around 11 seconds to be obtained.

The case study solution was compared against two non-machine learning approaches from the proponent organization: a pattern mining model and a SQL heuristic. The models were tested under 3 sets of tests, where each one predicted the last purchase for 360 test customers, with a maximum of 10, 7, and 5 items, respectively. The developed solution obtained the overall best results and was the most flexible and generalized one.

A Friedman hypothesis test was executed to test the null hypothesis. A random sample of 10 customers was selected, and the average f1-score was registered. The p-value obtained by the test was smaller than the alpha (0.05), proving the rejection of the null hypothesis (all the different classifiers behave similarly on the same data under the same conditions) - supporting the previous comparison tests.

The detection of association rules by the current model was analyzed. The Apriori algorithm was applied to the dataset multiple times, with different configurations, to identify sets of association rules with different strengths and search them in the recommendations generated for 360 test customers. The results have shown that the strictest rules were identified in a complete way in the recommendations and that the weakest recommendations appeared many times incomplete. Despite the developed model being able to learn the strictest rules, it may be able to benefit from a heuristic layer of association rules.

Performance tests were developed to validate how the recommender system behaves under different workloads. These tests evidenced an increase in the recommendations time associated with a higher number of parallel requests, enhancing the need to scale the solution in a horizontal way.

All the different experiments were performed in a Ubuntu 18.04 machine, with 64 GB of RAM, 480 GB of SSD storage, and an Nvidia GeForce GTX 1070 GPU.

Chapter 7

Conclusions

Grocery retail differs from other fields of retail because of some of its uniquenesses, such as the wide variety of different products it deals with, their seasonality, its promotional character, its evolution throughout time, and the discrepancy between the number of customers and items they buy (Sano et al. 2015). Also, grocery retail shopping habits are not easily described. Certain items are bought only once in a customer's lifetime, while others are purchased very frequently; some products are not replaceable for a customer, while others can be replaced by similar or very distinct items; the need for some products evolves in time (e.g., sizes or flavours), while for others stays the same forever.

Grocery retailers understand the progress in technology and the benefits associated with investing in personalized customer experiences, either in-store or in online stores (Deloitte 2018). A way of offering such experience is by providing access to personalized recommendations, facilitating the process of creating and managing the shopping list for next purchases (Guidotti et al. 2017).

The academical interest in recommender systems has increased since the mid-90s (Adomavicius and Tuzhilin 2005), and several methodologies have emerged since then. The more basic recommender systems estimate ratings for items that users have not yet demonstrated interest on, based on their interest in other items or other user's interests, and then recommend the most highly-rated ones. Shopping list recommendations for a specific moment in time are, however, a more complex use case, as there is a dependency on the history and shopping habits of each customer.

Recommender systems can be developed using traditional software approaches or using machine learning-based approaches. This dissertation focused on the second methodology, and the research performed has shown that several machine learning approaches exist and provide different results, according to the type of recommendation and to the field or dataset. The lack of academic and scientific work regarding recommender systems for grocery retail brought a need to study broader applications when selecting an approach.

By studying academical results (Fernández-Delgado et al. 2014; Olson et al. 2017; Vanschoren et al. 2012) on areas with similar domains (i.e., where a user has historical data associated with it, such as movies, e-commerce, or health procedures), gradient boosted-tree algorithms, neural networks, and SVMs appeared as the most promising approaches to develop a machine learning model for grocery retail, achieving the best results and being more consistent throughout the different domains (see section 3.4.2). Several technologies exist when developing machine learning algorithms. TensorFlow is a very well-known technology and has a big community around it, making it the ideal framework to be used in this work. Besides, it includes support for both CPU and GPU, allowing faster results when working with models.

A case study was developed in order to validate the developed solution. This way, a dataset capable of reflecting a real-world scenario was needed, and from all the studied datasets, the *Instacart Online Grocery Shopping Dataset* was the most promising one (see section 6.2.1). It includes more than 3 million grocery orders divided into more than 200 thousand customers, representing a good base to work on in order to extract information to train a machine learning model. A limitation of the dataset is the lack of quantities, causing the recommender system not to support quantity recommendation yet. Once the dataset was adopted, the machine learning algorithm for the case study was selected, based on a comparison between the most promising ones. Gradient boosted trees achieved the best results on the different test sets performed.

The particularities of grocery retail make the development of a recommender system particularly challenging. It was of paramount importance to extract the biggest amount of knowledge as possible from the dataset used in the case study, in an attempt to help the gradient boosted trees model learn distinct patterns for different customers, different products, and different relationships between both. This knowledge was represented by features. Four types of features were extracted, according to the implemented solution: customer-related, product-related, customer-product-related, and explicit features.

Designing and selecting the best features has proved itself to be a difficult task. Some features seemed to provide very specific information, but ended up by not improving the results or by increasing the training time for too much (e.g., identifying the number of orders in a row that a customer bought a product). Some simple features were able to provide the best results, and some other features were computationally expensive; however, the benefits were too big to be ignored (e.g., identifying how recently or frequently a product was bought by a customer). This process was very time consuming, but it was a crucial part of the solution.

The available data allowed the development of a recommender system capable of predicting items already purchased by a customer or from all the available items. A business decision was made in order to target items that the customer has already bought before, in order to have a theoretical smaller margin of error in the recommendations, and a smaller amount of data when training and predicting, and consequently smaller amounts of time spent. This means, however, that the developed recommender system is not ready to suggest new items for a customer, by itself.

After having decent work feature-wise, the results were improved by tuning the hyperparameters when training the model in the case study. Each machine learning technique provides different sets of parameters to tune, and the tuning process depends on that. There is no general rule for training, but some methodologies (e.g., XGBoost) provide ways of optimizing the choice of the best setup. TensorFlow does not provide such implementation for gradient boosted trees, so a tuning algorithm based on the official hyperparameter documentation and general boosting trees algorithms was followed. Several test iterations were executed in order to identify a good hyperparameter configuration, and the final choice was based on a ratio between training time and results.

The case study model takes around 3 hours to train and is able to generate recommendations in around 11 seconds. Tests performed comparing this solution with two non-machine learning-based approaches have shown it to outperform these solutions as far as results, flexibility, and consistency are concerned, for all the tests. The three recommender systems were compared when predicting a maximum of 10 items per customer (average basket size in the Instacart dataset), a maximum of 7 items, and a maximum of 5 items. The test sample consisted of 360 random customers.

By recommending a maximum of 10 items, the model from the case study was able to predict at least one correct item for 310 customers and obtain an average precision of 30% (i.e., on average, 3 correct items per recommended list were correct). This value increased, naturally, with smaller amounts of maximum recommended items, achieving 38% when recommending a maximum of 5 items. Even though the best results were achieved with this value, the maximum basket size depends essentially on business and market decisions.

An association rules algorithm was applied to the dataset in order to retrieve association rules with different strengths. These rules were searched for in the recommendations for the previous test customers, and the results were rewarding. The strictest rules were found within the recommendations several times. Weaker rules were found complete many times as well but were also found incomplete. This evidences that the model was able to learn the strictest rules and that the results may, still, be improved. The lack of literature combining these two techniques allows the contribution with empirical knowledge regarding a heuristic association rules layer that could be used to reorganize the recommendations returned by the gradient boosted trees model.

The adopted architecture follows an offline learning approach (see section 4.2.1), so the machine learning model needs to be trained and then deployed - it is not capable of learning as it is being used. This solution includes an automatic training and deployment flow. An automatic process for downloading retailer data was developed, followed by an automatic process for training new models periodically, as configured.

Since the machine learning model learns from the historical behavior and preferences of all the customers and recommends items based on previous purchases, more items will be available at prediction time, increasing the chance of predicting both correct and incorrect items. However, if the number of the previous transaction is small, the recommendation's quality will inevitably suffer (i.e., cold-start problem (see section 3.1.2)). Using a quality filter to restrict the recommendations for customers with at least N previous transactions might be valuable in a real-world scenario.

This chapter presents a review of the achieved requirements, limitations, and future work, and it is concluded by a final appreciation of the dissertation.

7.1 Achieved Requirements

The main goal of this dissertation was to develop a recommender system for grocery retail, capable of predicting personalized shopping lists. Throughout this project, several discussions led to the identification of the functional requirements (in the form of use cases) that suit the goals. Non-functional requirements were also identified, describing how the solution is expected to behave.

As far as functional requirements are concerned (see section 4.1.2), the following requirements were fulfilled:

- UC_C1 - View a shopping list recommendation

- UC_S3 - Update the shopping history
- UC_S4 - Train and update the model

The previous use cases were designed to match the solution goals. However, an additional requirement (UC_C2) was proposed to value the work by adding the ability to recommend the next item, but ended up not being achieved due to the fact that the structure developed to generate shopping list recommendations is not ready to predict the next item of a shopping flow. In order to do so in a proper way, a different machine learning algorithm or training flow would need to be developed.

As far as non-functional requirements go (see section 4.1.3), the following categories were fulfilled: *usability*; *reliability*; *supportability*; *design constraints*; *implementation constraints*; and *interface constraints*.

A goal of generating shopping lists in a small amount of time was proposed as another way of valuing the recommender system. In the developed case study, the average recommendation time is 11 seconds, depending on the customer (smaller shopping histories can be processed faster) and on the hardware. The perception of this duration can easily be minimized by performing the recommendations in background and pushing them to the customers once ready, reducing its impact.

7.2 Limitations

Despite being able to meet the requirements that motivated this dissertation, some aspects are limited by different reasons.

The machine learning model is trained over a specific dataset, containing multiple customers and their purchases during a time period. Since recommendations are tailored to each customer, it also means that new customers that have no history in the training set cannot benefit from them. Also, the solution includes no support for forgetting old data.

The choice behind the target variable in the case study (see section 6.2.3) has brought some benefits to the solution, as analyzed before, but has also brought a limitation: no new items can be suggested by this recommender system. Even though customers tend to buy a reduced portion of new items (see section 6.2.2), the solution is limited by predicting only items that were already bought by the customer.

Also, the adopted dataset in the case study has no details on the quantity of each item bought. For this reason, this information was not considered during training nor during the design of the solution. This way, the developed recommender system includes no support to quantity recommendations.

A limitation of the implemented solution has to do with scaling: the use of several parallel computations made it able to provide recommendations in decent amounts of time, but with high CPU usage. This, together with the several steps involved in the predicting pipeline (see section 5.2.7), caused the predicting phase to be very expensive, computation-wise.

When several requests occur in parallel (see tests made in 6.6), the average response time of each request increases significantly. In order to solve this limitation, horizontal scaling should be adopted. The service responsible for exposing the machine learning model (i.e., Machine Learning Model Interface) should have multiple instances deployed, with a gateway upfront, routing the requests. This would mean that there could be, at a time, as many instances as needed, assuring consistent response times. The architecture is prepared to support this situation.

7.3 Future Work

The work done for this dissertation has much potential to be expanded, either by improving specific parts or by adding new features to boost the results. Also, the solution can be further tested using a Nemenyi statistical test to obtain details on the origin of the difference when compared with other solutions.

A matter to pursue would be using a real-world dataset. Although a grocery retail dataset was adopted in the case study, this work would benefit from working with real customers and receiving feedback from its usage. This could lead to eventual fixes or suggestions.

In addition, by working with real customers, the chance to acquire real-time feedback would exist (either by asking the customer their opinion on the recommendation or by interpreting how it was then translated into a purchase). This would be a significant step towards experimenting with models based on online learning and could be completed with periodical data updates from the retailer.

Predicting quantities associated with an item would also value the solution. Since this information did not exist in the case study dataset, it was not pursued. In order to accomplish this, modifications to the training pipeline would be necessary, or even the usage of an alternative or complementary technique.

Another feature that would benefit the current recommender system is suggesting the next-item when a customer is shopping. This feature was explored in this dissertation (see section 4.2.2), but ended up not being implemented as the current solution includes no support to this methodology.

Reducing the recommendation time would also allow a better user experience. This value can still be reduced with additional parallel computations (e.g., when extracting the different features from the data) or by optimizing other parts of the code. However, by increasing the CPU usage, the average duration of parallel recommendations is expected to increase (see the tests in section 6.6). This enhances the need to scale the service responsible for exposing the machine learning model horizontally. By having multiple instances of this service, the previously referred optimizations would bring extra value to the user experience.

The experiments with association rules (see section 6.5) have shown that the model was able to learn most of the strictest rules. Weaker rules, on the other hand, were found incomplete in the recommendations some times (i.e., only part of the rule was suggested). These results, together with the lack of studies on the usage of gradient boosted trees with association rules (at the time of this dissertation), bring the idea that a heuristic layer could be added to the results of the machine learning model in order to re-sort the recommendations based on the confidence of a rule. This is empirical knowledge that would be interesting to pursue.

An exciting matter to explore is how the shopping list is evaluated. Traditional evaluation metrics cover the strict match between predicted and bought items without considering alternatives. By analyzing some recommendations (see appendix B), one can find many similarities between recommended and bought items, and they may be good enough to be accepted by the customer. Exploring product similarity (e.g., using embeddings) during evaluation may lead to better and more accurate results.

7.4 Final Appreciation

The challenges associated with machine learning and with the different types of recommender systems have made this project particularly interesting to the student. Despite the already existent interest in grocery retail, this dissertation has widened his knowledge in the field and made him learn more about the uniqueness of the area.

This dissertation has made the student step out of his comfort zone, as far as software development is concerned. It has made him research and experiment in order to understand deeper concerns of the area. It has also allowed the student to meet people from the machine learning area and have enriching conversations.

Analyzing the different approaches to machine learning-based recommender systems was considered to be a valuable experience. Understanding some of the issues faced by some authors in different retail subsets allowed the student to acquire a better understating of the grocery retail uniquenesses and difficulties associated with it.

Feature engineering and hyperparameter tuning were considered to be particularly challenging tasks. The first one, because of the lack of experience in the field and because of the issues related to dealing with such amounts of data. The second one, because of the more profound concepts behind every configuration and every result. However, the student considers these as the most rewarding tasks.

The student considers also that evaluating recommendations for grocery retail using traditional evaluation metrics, where only the product identifier is compared, might reduce its value. The real value for these retailers is related to increasing the basket or its value, which might not be reflected in the traditional measurements.

The periodical meetings with both advisors have helped to keep this project heading in the right direction and have brought some interesting discussions regarding retail and machine learning that were later reflected in the dissertation.

Recommender systems for grocery retail are an exciting but embryonic area. All the studies and analyses performed have evidenced a lack of research and academic work in this field. There is still a lot to explore in the area, and it will undoubtedly be done during the next years. The student feels enriched to have been part of it.

Bibliography

Adomavicius, Gediminas and Alexander Tuzhilin (2005). “Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions”. In: *IEEE Transactions on Knowledge and Data Engineering* 17.6, pp. 734–749. issn: 10414347. doi: 10.1109/TKDE.2005.99.

Agarwal, Pragya, Madan Lal Yadav, and Nupur Anand (2013). *Study on Apriori Algorithm and its Application in Grocery Store*. Tech. rep. 14, pp. 975–8887.

Allee, Verna (2006). “What is ValueNet Works^{TM} Analysis?” In:

Anuradha and Gaurav Gupta (2014). “A self explanatory review of decision tree classifiers”. In: *International Conference on Recent Advances and Innovations in Engineering, ICRAIE 2014*. Institute of Electrical and Electronics Engineers Inc. isbn: 9781479940400. doi: 10.1109/ICRAIE.2014.6909245.

Auria, Laura and R. A. Moro (2011). “Support Vector Machines (SVM) as a Technique for Solvency Analysis”. In: *SSRN Electronic Journal*. doi: 10.2139/ssrn.1424949.

Bengio, Yoshua, Aaron Courville, and Pascal Vincent (2014). “Representation Learning: A Review and New Perspectives”. In: arXiv: 1206.5538v3. url: http://www.image-net.org/challenges/LSVRC/2012/results.html.

Bittner, Kurt (2016). *Driving Iterative Development With Use Cases*. url: https://www.ibm.com/developerworks/rational/library/4029.html (visited on 09/01/2020).

Bloice, Marcus D. and Andreas Holzinger (2016). “A tutorial on machine learning and data science tools with python”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9605 LNCS. Springer Verlag, pp. 435–480. doi: 10.1007/978-3-319-50478-0_22.

Breiman, Leo (2001). “Random forests”. In: *Machine Learning* 45.1, pp. 5–32. issn: 8856125. doi: 10.1023/A:1010933404324.

Brownlee, Jason (2018). *How to Calculate Nonparametric Statistical Hypothesis Tests in Python*. url: https://machinelearningmastery.com/nonparametric-statistical-significance-tests-in-python/ (visited on 09/06/2020).

Burke, Robin (2002). “Hybrid recommender systems: Survey and experiments”. In: *User Modelling and User-Adapted Interaction* 12.4, pp. 331–370. issn: 9241868. doi: 10.1023/A:1021240730564.

Cheng, Heng-Tze et al. (2016). *Wide & Deep Learning for Recommender Systems*. Tech. rep. arXiv: 1606.07792v1. url: http://tensorflow.org..

Deloitte (2018). *Global Powers of Retailing 2018 - Transformative change, reinvigorated commerce*. Tech. rep.

– (2019). *Global Powers of Retailing 2019*. Tech. rep.

Dhumale, R. B., N. D. Thombare, and P. M. Bangare (2019). “Machine Learning: A Way of Dealing with Artificial Intelligence”. In: *Proceedings of 1st International Conference on Innovations in Information and Communication Technology, ICIICT 2019*. Institute of Electrical and Electronics Engineers Inc. isbn: 9781728116044. doi: 10.1109/ICIICT1.2019.8741360.

Eeles, Peter (2004). *What, no supplementary specification?* url: https://www.ibm.com/developerworks/rational/library/3975.html.

Fernández-Delgado, Manuel et al. (2014). *Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?* Tech. rep., pp. 3133–3181. url: http://www.mathworks.es/products/neural-network..

Ferreira, Carlos Abreu, João Gama, and Vítor Santos Costa (2011). "Predictive Sequence Miner in ILP Learning". In:

Fournier-Viger, Philippe et al. (2016). "PHM: Mining periodic high-utility itemsets". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9728. Springer Verlag, pp. 64–79. isbn: 9783319415604. doi: 10.1007/978-3-319-41561-1_6.

Freund, Yoav and Robert E Schapire (1996). *Experiments with a New Boosting Algorithm*. Tech. rep. url: http://www.research.att.com/.

Fulmer, Jeffrey (2012). *Siege Home*. url: https://www.joedog.org/siege-home/ (visited on 09/09/2020).

Gama, João et al. (2015). *Extração de Conhecimento de Dados*. isbn: 9789726188117.

Gomes, Elsa Ferreira (2016). *Experimentação e Avaliação*.

Grbovic, Mihajlo and Haibin Cheng (2018). "Real-time Personalization using Embeddings for Search Ranking at Airbnb". In: doi: 10.1145/3219819.3219885. url: https://doi.org/10.1145/3219819.3219885.

Gron, Aurlien (2017). *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 1st. O'Reilly Media, Inc. isbn: 1491962291.

Group, FMI Hartman (2018). *How Technology is Changing Grocery Shopping, From the Consumer Perspective*. url: https://www.fmi.org/docs/default-source/webinars/trends-2018-webinar-3-final94d6250324aa67249237ff0000c12749.pdf?sfvrsn=547c426e%7B%5C_%7D0 (visited on 01/09/2020).

Guidotti, Riccardo et al. (2017). "Next Basket Prediction using Recurring Sequential Patterns". In: *Proceedings - IEEE International Conference on Data Mining, ICDM* 2017-November, pp. 895–900. doi: 10.1109/ICDM.2017.111. arXiv: 1702.07158. url: http://arxiv.org/abs/1702.07158%20http://dx.doi.org/10.1109/ICDM.2017.111.

Gupta, Swati and Sushama Nagpal (2015). "An empirical analysis of implicit trust metrics in recommender systems". In: *2015 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2015*. Institute of Electrical and Electronics Engineers Inc., pp. 636–639. isbn: 9781479987917. doi: 10.1109/ICACCI.2015.7275681.

Hanke, Jannis et al. (2018). *REDEFINING THE OFFLINE RETAIL EXPERIENCE: DESIGNING PRODUCT RECOMMENDATION SYSTEMS FOR FASHION STORES*. Tech. rep.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2009). *The Elements of Statistical Learning*. Vol. 27. 2. isbn: 978-0-387-84857-0. doi: 111. arXiv: arXiv:1011.1669v3. url: http://www.springerlink.com/index/10.1007/b94608.

Hauser, John R. and Don Clausing (1988). *The House of Quality*. url: https://hbr.org/1988/05/the-house-of-quality (visited on 09/22/2020).

Haykin, Simon et al. (2009). *Neural Networks and Learning Machines Third Edition*. isbn: 9780131471399.

He, Qiang and Jun Fen Chen (2005). "The inverse problem of support vector machines and its solution". In: *2005 International Conference on Machine Learning and Cybernetics, ICMLC 2005*, pp. 4322–4326. isbn: 078039092X. doi: 10.1109/icmlc.2005.1527698.

Instacart (2017). *The Instacart Online Grocery Shopping Dataset 2017*. url: `https://www.instacart.com/datasets/grocery-shopping-2017` (visited on 08/21/2020).

Jain, Aarshay (2016a). *Complete Guide to Parameter Tuning in XGBoost with codes in Python*. url: `https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/` (visited on 08/30/2020).

– (2016b). *Complete Machine Learning Guide to Parameter Tuning in Gradient Boosting (GBM) in Python*. url: `https://www.analyticsvidhya.com/blog/2016/02/complete-guide-parameter-tuning-gradient-boosting-gbm-python/` (visited on 08/30/2020).

Jariha, Priyanka and Sanjay Kumar Jain (2018). "A state-of-the-art Recommender Systems: An overview on Concepts, Methodology and Challenges". In: *Proceedings of the International Conference on Inventive Communication and Computational Technologies, ICICCT 2018*. Institute of Electrical and Electronics Engineers Inc., pp. 1769–1774. isbn: 9781538619742. doi: `10.1109/ICICCT.2018.8473275`.

Jooa, Jinhyun, Sangwon Bangb, and Geunduk Parka (2016). "Implementation of a Recommendation System Using Association Rules and Collaborative Filtering". In: *Procedia Computer Science*. Vol. 91. Elsevier B.V., pp. 944–952. doi: `10.1016/j.procs.2016.07.115`.

Koen, Peter A et al. (2002). *FuzzyFrontEnd: Effective Methods, Tools, and Techniques*. Tech. rep.

Koen, Peter et al. (2001). *PROVIDING CLARITY AND A COMMON LANGUAGE TO THE "FUZZY FRONT END"*. Tech. rep.

Komer, Brent, James Bergstra, and Chris Eliasmith (2014). *Hyperopt-Sklearn: Automatic Hyperparameter Configuration for Scikit-Learn*. Tech. rep.

Kovalev, Vassili, Alexander Kalinovsky, and Sergey Kovalev (2016). "Deep Learning with Theano, Torch, Caffe, Tensorflow, and Deeplearning4J: Which One is the Best in Speed and Accuracy?" In:

Le, Quoc V et al. (2012). *Building High-level Features Using Large Scale Unsupervised Learning*. Tech. rep. url: `http://opencv.willowgarage.com/wiki/`.

Learned-Miller, Erik G (2014). "Introduction to Supervised Learning". In:

Lekha, Madhu (2019). *Instacart MBA - Tensorflow GBM - PART1*. Tech. rep. url: `https://www.kaggle.com/madhulekha/instacart-mba-tensorflow-gbm-part1-eda`.

Lerato, Masupha et al. (2016). "A survey of recommender system feedback techniques, comparison and evaluation metrics". In: *2015 International Conference on Computing, Communication and Security, ICCCS 2015*. Institute of Electrical and Electronics Engineers Inc. isbn: 9781467393546. doi: `10.1109/CCCS.2015.7374146`.

Liu, David Zhan and Gurbir Singh (2016). *A Recurrent Neural Network Based Recommendation System*. Tech. rep. url: `https://www.yelp.com/dataset%7B%5C_%7Dchallenge`.

Lops, Pasquale, Marco de Gemmis, and Giovanni Semeraro (2011). "Content-based Recommender Systems: State of the Art and Trends". In: *Recommender Systems Handbook*. Springer US, pp. 73–105. doi: `10.1007/978-0-387-85820-3_3`.

Malik, Usman (2020). *Association Rule Mining via Apriori Algorithm in Python*. url: `https://stackabuse.com/association-rule-mining-via-apriori-algorithm-in-python/` (visited on 09/06/2020).

Manufacturing Group, Warwick (2007). "Quality Function Deployment". In: *Product Excellence using Six Sigma*.

Medar, Ramesh, Vijay S. Rajpurohit, and B. Rashmi (2018). "Impact of Training and Testing Data Splits on Accuracy of Time Series Forecasting in Machine Learning". In: *2017*

International Conference on Computing, Communication, Control and Automation, ICCUBEA 2017. Institute of Electrical and Electronics Engineers Inc. isbn: 9781538640081. doi: 10.1109/ICCUBEA.2017.8463779.

Mitova, Teodora (2020). *21+ Grocery Shopping Statistics for Every CUSTOMER in 2020*. url: https://spendmenot.com/blog/grocery-shopping-statistics/ (visited on 09/21/2020).

Mohamed, Marwa Hussien, Mohamed Helmy Khafagy, and Mohamed Hasan Ibrahim (2019). "Recommender Systems Challenges and Solutions Survey". In: *Proceedings of 2019 International Conference on Innovative Trends in Computer Engineering, ITCE 2019*. Institute of Electrical and Electronics Engineers Inc., pp. 149–155. isbn: 9781538652602. doi: 10.1109/ITCE.2019.8646645.

Nguyen, Giang et al. (2019). "Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey". In: *Artificial Intelligence Review* 52.1, pp. 77–124. issn: 15737462. doi: 10.1007/s10462-018-09679-z.

Olson, Randal S. et al. (2017). "Data-driven Advice for Applying Machine Learning to Bioinformatics Problems". In: arXiv: 1708.05070. url: http://arxiv.org/abs/1708.05070.

Patro, S.Gopal Krishna and Kishore Kumar Sahu (2015). "Normalization: A Preprocessing Stage". In: *IARJSET*, pp. 20–22. issn: 2393-8021. doi: 10.17148/iarjset.2015.2305. arXiv: 1503.06462.

Petrovic, Otto and Christian Kittl (2003). "Capturing the value proposition of a product or service". In:

Ponomareva, Natalia et al. (2017). "Compact Multi-Class Boosted Trees". In: arXiv: 1710.11547. url: http://arxiv.org/abs/1710.11547.

Potdar, Kedar, Taher S., and Chinmay D. (2017). "A Comparative Study of Categorical Variable Encoding Techniques for Neural Network Classifiers". In: *International Journal of Computer Applications* 175.4, pp. 7–9. doi: 10.5120/ijca2017915495.

Qastharin, Annisa R (2016). "Business Model Canvas for Social Enterprise". In: *Journal of Business and Economics* 7.4, pp. 627–637. doi: 10.15341/jbe(2155-7950)/04.07.2016/008. url: http://www.academicstar.us.

Raju, C. V.L., Y. Narahari, and K. Ravikumar (2003). "Reinforcement learning applications in dynamic pricing of retail markets". In: *Proceedings - IEEE International Conference on E-Commerce, CEC 2003*. Institute of Electrical and Electronics Engineers Inc., pp. 339–346. doi: 10.1109/COEC.2003.1210269.

Raut, Laukik, Rajat Wakode, and Pravin Talmale (2015). *Overview on Kanban Methodology and its Implementation*. url: https://www.researchgate.net/publication/280865949%7B%5C_%7DOverview%7B%5C_%7Don%7B%5C_%7DKanban%7B%5C_%7D%20Methodology%7B%5C_%7Dand%7B%5C_%7Dits%7B%5C_%7DImplementation (visited on 09/17/2020).

Rojas, Raúl (2009). *AdaBoost and the Super Bowl of Classifiers A Tutorial Introduction to Adaptive Boosting*. Tech. rep.

Sano, Natsuki et al. (2015). "Recommendation system for grocery store considering data sparsity". In: *Procedia Computer Science*. Vol. 60. 1. Elsevier B.V., pp. 1406–1413. doi: 10.1016/j.procs.2015.08.216.

Shalev-Shwartz, Shai and Shai Ben-David (2014). *Understanding Machine Learning: From Theory to Algorithms*.

Sheil, Humphrey, Omer Rana, and Ronan Reilly (2018). "Predicting purchasing intent: Automatic Feature Learning using Recurrent Neural Networks". In: doi: 10.1145/nnnnnnn.nnnnnnn. arXiv: 1807.08207v1. url: https://doi.org/10.1145/nnnnnnn.nnnnnnn.

Si, Si et al. (2017). *Gradient Boosted Decision Trees for High Dimensional Sparse Output*. Tech. rep. url: https://github.com/Microsoft/LightGBM.

Slivkins, Aleksandrs (2019). "Introduction to Multi-Armed Bandits". In: arXiv: 1904.07272. url: http://arxiv.org/abs/1904.07272.

SRK (2017). *Simple Exploration Notebook - Instacart*. Tech. rep. url: https://www.kaggle.com/sudalairajkumar/simple-exploration-notebook-instacart.

Staudemeyer, Ralf C. and Eric Rothstein Morris (2019). "Understanding LSTM – a tutorial into Long Short-Term Memory Recurrent Neural Networks". In: arXiv: 1909.09586. url: http://arxiv.org/abs/1909.09586.

Sun, Chen, Rong Gao, and Hongsheng Xi (2014). "Big data based retail recommender system of non E-commerce". In: *5th International Conference on Computing Communication and Networking Technologies, ICCCNT 2014*. Institute of Electrical and Electronics Engineers Inc. isbn: 9781479926961. doi: 10.1109/ICCCNT.2014.6963129.

TensorFlow (2020a). *A Classifier for Tensorflow Boosted Trees models*. url: https://www.tensorflow.org/api%7B%5C_%7Ddocs/python/tf/%20estimator%20/Boosted%20Trees%20Classifier (visited on 08/31/2020).

– (2020b). *Boosted trees using Estimators | TensorFlow Core*. url: https://www.tensorflow.org/tutorials/estimator/boosted%7B%5C_%7Dtrees (visited on 08/28/2020).

– (2020c). *Overfit and underfit | TensorFlow Core*. url: https://www.tensorflow.org/tutorials/keras/overfit%7B%5C_%7Dand%7B%5C_%7Dunderfit%7B%5C#%7Dadd%7B%5C_%7Dweight%7B%5C_%7Dregularization (visited on 08/31/2020).

– (2018). *TensorFlow Contrib Estimators are Deptrecated*. url: https://github.com/tensorflow/tensorflow/blob/r1.8/tensorflow/contrib/learn/README.md (visited on 10/09/2020).

Thornton, Chris et al. (2013). *Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms*. Tech. rep. arXiv: 1208.3719v2.

Vanschoren, Joaquin et al. (2012). "Experiment databases: A new way to share, organize and learn from experiments". In: *Machine Learning* 87.2, pp. 127–158. issn: 08856125. doi: 10.1007/s10994-011-5277-0.

Vermorel, Joannès and Mehryar Mohri (2005). *Multi-armed Bandit Algorithms and Empirical Evaluation*. Tech. rep.

Vinagre, João, Alípio Jorge, and João Gama (2014). "Evaluation of recommender systems in streaming environments". In: doi: 10.13140/2.1.4381.5367.

Wang, Qiang and Zhongli Zhan (2011). "Reinforcement learning model, algorithms and its application". In: *Proceedings 2011 International Conference on Mechatronic Science, Electric Engineering and Computer, MEC 2011*, pp. 1143–1146. isbn: 9781612847221. doi: 10.1109/MEC.2011.6025669.

Wirth, R. and J. Hipp (2000). "Crisp-dm: towards a standard process modell for data mining". In:

Wolfgang, Ulaga and Eggert Andreas (2006). "Relationship value and relationship quality". In: *European Journal of Marketing* 40, pp. 836–867. doi: 10.1108/03090560610648075. url: http://dx.doi.org/10.1108/03090569610106626http://dx.doi.org/10.1108/03090560710752429.

Xu, Qingyang et al. (2017). "The difference learning of hidden layer between autoencoder and variational autoencoder". In: *Proceedings of the 29th Chinese Control and Decision Conference, CCDC 2017*. Institute of Electrical and Electronics Engineers Inc., pp. 4801–4804. isbn: 9781509046560. doi: 10.1109/CCDC.2017.7979344.

Zhang, Shuai et al. (2017). “Deep Learning based Recommender System: A Survey and New Perspectives”. In: doi: 10.1145/3285029. arXiv: 1707.07435. url: http://arxiv.org/abs/1707.07435%20http://dx.doi.org/10.1145/3285029.

Appendix A

Technologies used in the solution

The main technologies used to develop this solution can be found synthesised in the table A.1.

Table A.1: Technologies used to develop the solution

Technology	Description	Usage
CUDA	Toolkit used provide GPU-acceleration to TensorFlow models	Allowing GPU acceleration of the machine learning model
Gradient Boosted Trees	Supervised learning algorithm for machine learning models, with high accuracy and training speed and fast prediction time	The algorithm used for the machine learning model of the case study
HTML	Standard markup language used by web browsers to display documents graphically; it is capable of embedding scripting languages	Client device demonstrator
JavaScript	High-level programming language, scripted or compiled; it is mainly used to animate web pages and in smaller server-side applications	Logic and data manipulation in the client device demonstrator
JSON	Human readable key-value representation of programming objects	Data representation between the different services
NumPy	Python library that provides access to high-lever mathematical operations over large, multi-dimensional arrays and matrices	Data operations over the dataset
pandas	Python library that offers access to data structures and operations for manipulating and analysing numerical tables	Data representation and manipulation, and charts
PHP	General purpose scripting language, mainly used for web development and simple script operations	Auxiliary scripts for comparison testing
Python	High-level interpreted programming language, with dynamic typing and garbage-collector	Main programming language; used in the different components

Table A.1: Technologies used to develop the solution (continued)

Technology	**Description**	**Usage**
pytest	Testing framework for Python programming language	Unit, functional and end-to-end testing
seaborn	Statistical data visualization library that allows the generation and customization of charts	Plot generation
SciPy	Set of frameworks for mathematics, science, and engineering, containing frameworks like pandas and NumPy.	Data operations and Friedman hypothesis tests
Siege	Open source tool for stress testing, that allows the simulation of sequential and parallel requests	Performance testing
TensorFlow	Open-source library for numerical computation using data flow graphs	Training the machine learning model and predicting

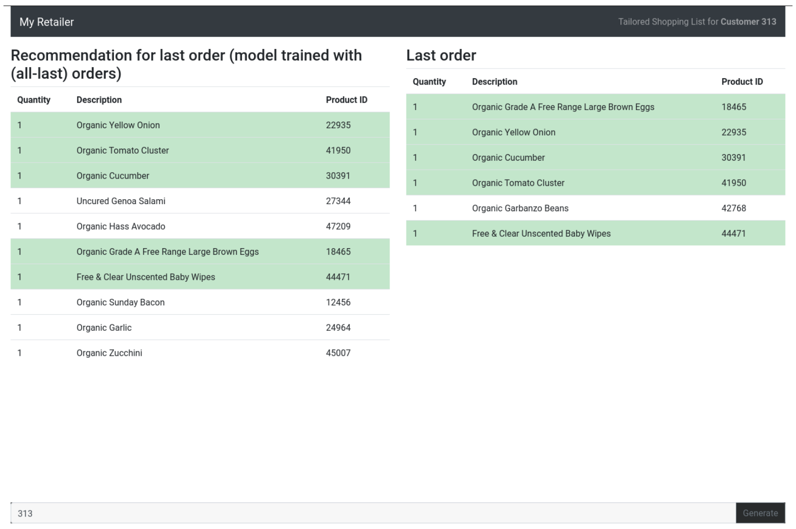
Appendix B

Dissection of recommendations for last order

A graphical demonstrator was developed to visualize the quality of the last order recommendations, used throughout the tests. On the left, the recommended shopping list is presented; on the right, the actual last order is described. Common items are marked in green. Three examples from the recommendations generated with a maximum of 10 items for 360 test customers (see section 6.4.2) are dissected in this appendix.

This analysis leads to thinking that traditional evaluation metrics may provide limited results when evaluating these recommendations, because of the difficulty to identify a wrong prediction.

The figure B.1 presents a recommendation of 10 items for customer with the identifier 313. The actual order contained 6 items, of each 5 were correctly predicted by the model.



My Retailer

Tailored Shopping List for **Customer 313**

Recommendation for last order (model trained with (all-last) orders)

Quantity	Description	Product ID
1	Organic Yellow Onion	22935
1	Organic Tomato Cluster	41950
1	Organic Cucumber	30391
1	Uncured Genoa Salami	27344
1	Organic Hass Avocado	47209
1	Organic Grade A Free Range Large Brown Eggs	18465
1	Free & Clear Unscented Baby Wipes	44471
1	Organic Sunday Bacon	12456
1	Organic Garlic	24964
1	Organic Zucchini	45007

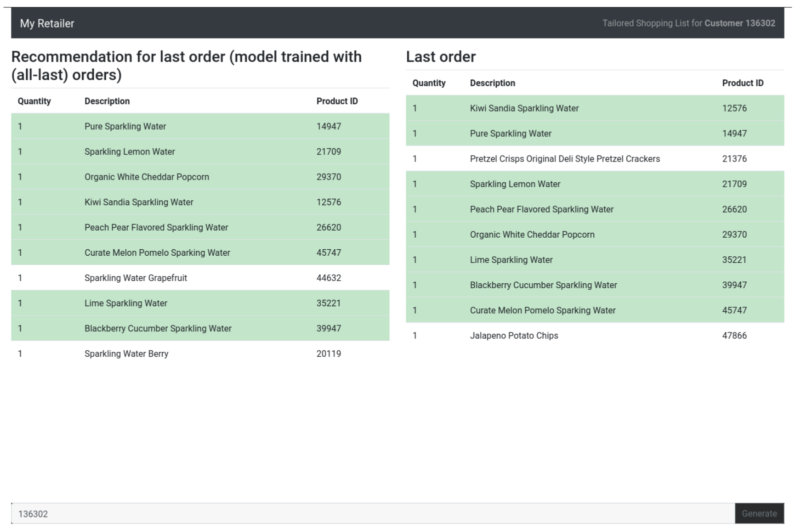
Last order

Quantity	Description	Product ID
1	Organic Grade A Free Range Large Brown Eggs	18465
1	Organic Yellow Onion	22935
1	Organic Cucumber	30391
1	Organic Tomato Cluster	41950
1	Organic Garbanzo Beans	42768
1	Free & Clear Unscented Baby Wipes	44471

313

Figure B.1: Evaluation of the last order recommendation for customer 313

The figure B.2 presents a recommendation of 10 items for customer with the identifier 136302. The actual order contained 10 items, of each 8 were correctly predicted by the model.



Recommendation for last order (model trained with (all-last) orders)

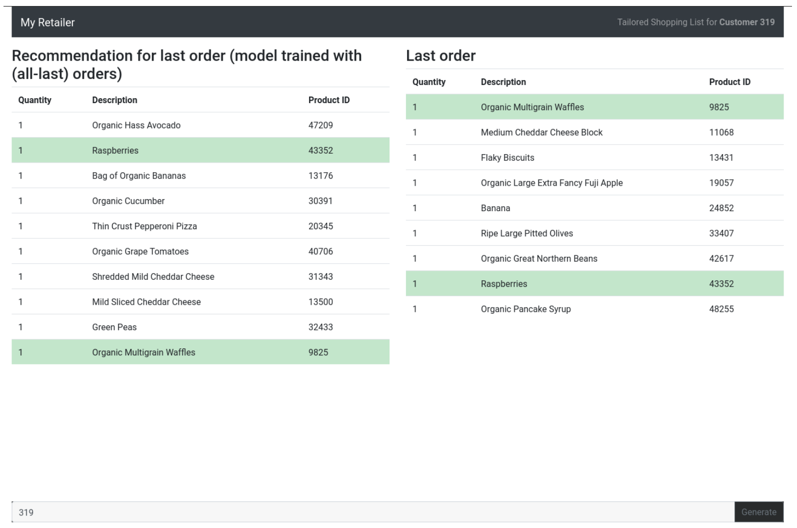
Quantity	Description	Product ID
1	Pure Sparkling Water	14947
1	Sparkling Lemon Water	21709
1	Organic White Cheddar Popcorn	29370
1	Kiwi Sandia Sparkling Water	12576
1	Peach Pear Flavored Sparkling Water	26620
1	Curate Melon Pomelo Sparking Water	45747
1	Sparkling Water Grapefruit	44632
1	Lime Sparkling Water	35221
1	Blackberry Cucumber Sparkling Water	39947
1	Sparkling Water Berry	20119

Last order

Quantity	Description	Product ID
1	Kiwi Sandia Sparkling Water	12576
1	Pure Sparkling Water	14947
1	Pretzel Crisps Original Deli Style Pretzel Crackers	21376
1	Sparkling Lemon Water	21709
1	Peach Pear Flavored Sparkling Water	26620
1	Organic White Cheddar Popcorn	29370
1	Lime Sparkling Water	35221
1	Blackberry Cucumber Sparkling Water	39947
1	Curate Melon Pomelo Sparking Water	45747
1	Jalapeno Potato Chips	47866

Figure B.2: Evaluation of the last order recommendation for customer 136302

The figure B.3 presents a recommendation of 10 items for customer with the identifier 319. The actual order contained 9 items, of each 2 were correctly predicted by the model. This example is particularly interesting because, in addition to the correctly predicted items, the recommender system predicted items very similar to the ones that ended up being bought.



Recommendation for last order (model trained with (all-last) orders)

Quantity	Description	Product ID
1	Organic Hass Avocado	47209
1	Raspberries	43352
1	Bag of Organic Bananas	13176
1	Organic Cucumber	30391
1	Thin Crust Pepperoni Pizza	20345
1	Organic Grape Tomatoes	40706
1	Shredded Mild Cheddar Cheese	31343
1	Mild Sliced Cheddar Cheese	13500
1	Green Peas	32433
1	Organic Multigrain Waffles	9825

Last order

Quantity	Description	Product ID
1	Organic Multigrain Waffles	9825
1	Medium Cheddar Cheese Block	11068
1	Flaky Biscuits	13431
1	Organic Large Extra Fancy Fuji Apple	19057
1	Banana	24852
1	Ripe Large Pitted Olives	33407
1	Organic Great Northern Beans	42617
1	Raspberries	43352
1	Organic Pancake Syrup	48255

Figure B.3: Evaluation of the last order recommendation for customer 319

The recommendation contained Bag of Organic Bananas, and the customer purchased standard bananas. In addition, two varieties of Cheddar Cheese were recommended, and the customer ended up buying a Medium Cheddar Cheese Block. This scenario is most likely to be present in many other situations, and consists of situations where the recommendations could have been accepted by the customer, increasing the perceived value of the recommender system.

Appendix C

Friedman hypothesis test source code

Source code of the Friedman hypothesis test, used to validate the null hypothesis in section 6.4.4.

```
from scipy.stats import friedmanchisquare

# independent samples
gbt_measurement = [0.242, 0.533, 0.235, 0.32, 0.421, 0.125, 0.4,
    0.25, 0.261, 0.296]
pm_measurement = [0.242, 0.533, 0.125, 0.32, 0.471, 0.125, 0.267,
     0.167, 0.174, 0.519]
sql_measurement = [0.121, 0.286, 0.118, 0.083, 0.105, 0.25,
    0.133, 0.083, 0.091, 0.222]

# algorithm
stat, pvalue = friedmanchisquare(gbt_measurement, pm_measurement,
     sql_measurement)
print('H-stat=%.3f, p=%.6f' % (stat, pvalue))

# interpretation
alpha = 0.05
if pvalue > alpha:
    print('p-value > alpha (fail to reject H0)')
else:
    print('p-value < alpha (reject H0)')
```

Listing C.1: Friedman hypothesis test over the 3 recommender systems