A cloud driven dynamic pricing system for retail companies

José Juan Galán Montero

Master Thesis in
Master in Artificial Intelligence

Company's Supervisor: Jesús Vicente García Hernández
University's Supervisor: Dr. Ulises Cortés García

January 2022

Abstract

The aim of this project is developing a dynamic pricing framework over a cloud based architecture, being scalable and highly configurable, considering the great cardinality of the solution in terms of the analytic models to build and apply. The dynamic optimization of the prices is achieved by combining the training of a sales prediction model and the execution of a discount combination optimizer. We performed simulations with data from a real client from the fashion retail sector, and the results obtained were promising, suggesting an improvement in the company's revenue.

Keywords

Dynamic pricing, cloud-based, scalability, configurable, gradient-boosting tree

Contents

Chapter 1

Introduction

Dynamic pricing is the study of determining optimal selling prices of products or services, in a setting where prices can easily and frequently be adjusted. This applies to vendors selling their products via Internet, or to brick-and-mortar stores that make use of digital price tags. In both cases, digital technology has made it possible to continuously adjust prices to changing circumstances, without any costs or efforts. Dynamic pricing techniques are nowadays widely used in various businesses, and in some cases considered to be an indispensable part of pricing policies.

1.1 Project Focus

Sales period have a crucial importance in the retail sector, a significant amount of the total yearly profit is generated within those weeks. Given this fact, maximizing profit during this time windows will for sure increase the company's profit for the whole year. The way in which discounts are applied to the different articles can make a difference in the volume of sold items, and also in the clients' satisfaction.

The purpose of this project is the implementation of a new dynamic pricing system that can be proven to optimize the discounts applied in order to maximize our client's profit. For that, the system will provide both the magnitude and the order in which discounts should be applied so the total profit at the end of the sales period is maximized.

1.2 Objectives

The main purpose of this project is to build a generic and modular dynamic pricing system which can be adapted easily to a given client. To manage that, it will be mainly cloud-based and highly configurable through several parameters

Generality A dynamic pricing system should be able to optimize prices within a range of weeks and steps that can be chosen by the business. Other parameters such as the initial price or discount, the moment in which prices should start being adjusted or the final desired stock should also be adjustable.

Modularity The ideal solution must be modular, being each of the parts composing the architecture as independent as possible, so any of them can be replaced at any moment if desired. The optimization process must be decoupled from the training one, as well as with the data preprocessing and ingestion.

Easy deployment The cloud nature of the system should make it easier to deploy it independently of the client's resources and location, reducing also the complexity of development given that the systems maintenance and physical architecture is delegated to already well-proven providers.

Dynamism A dynamic pricing system must be executed periodically once after every real iteration introduces more data into the system. By comparing recent real data with the corresponding predictions, the performance of the optimizer can be tested and improved. It is important to check if the predictions made follow a similar path to the real data observed, and to discard them if this is not the case.

1.3 Optimal pricing vs dynamic pricing

Price fixation models can face different business challenges. However, there is no unique model that works for every one. The particularities of each business, the availability of the data and the technological maturity of the company can lead to the use of different specific components, belonging to different pricing models. Each business challenge must be analysed individually, taking into account their singularities.

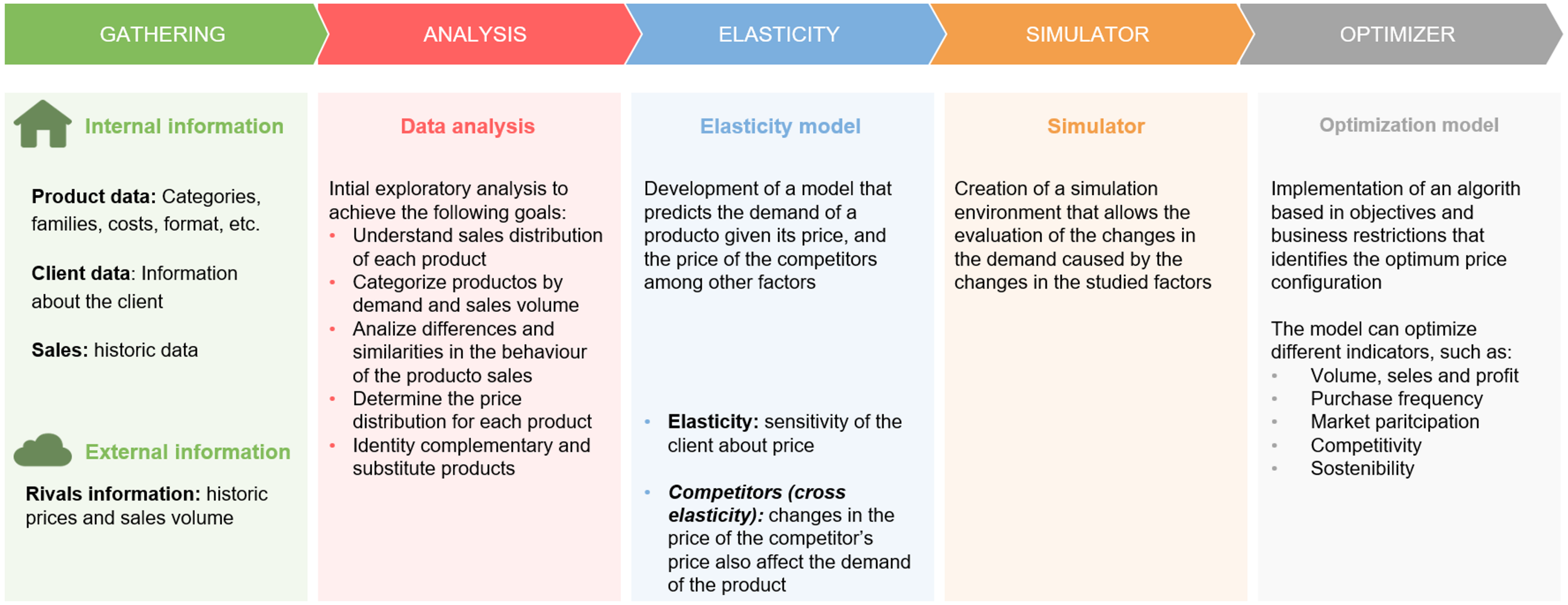
Optimal pricing Recommends prices that comply with the business objectives taking into account their restrictions. This kind of models consider relevant conditions from the context as the demand, the inventory and the prices of their rivals. The computing of the optimal prices it's done under demand and not continuously.

Dynamic pricing Adds the time concept to the optimal pricing approach, what means that prices can adapt in a continuous way. Price can even be modified several times a day depending on the changes in the internal and external factors.

Given that most of today's markets are constantly changing and evolving at a fast pace, dynamic pricing systems have been proven to be more efficient

Pricing methodologies

Optimal pricing


GATHERING → ANALYSIS → ELASTICITY → SIMULATOR → OPTIMIZER

Internal information

Product data: Categories, families, costs, format, etc.

Client data: Information about the client

Sales: historic data

External information

Rivals information: historic prices and sales volume

Data analysis

Intial exploratory analysis to achieve the following goals:
- Understand sales distribution of each product
- Categorize productos by demand and sales volume
- Analize differences and similarities in the behaviour of the producto sales
- Determine the price distribution for each product
- Identity complementary and substitute products

Elasticity model

Development of a model that predicts the demand of a producto given its price, and the price of the competitors among other factors

- **Elasticity:** sensitivity of the client about price
- ***Competitors (cross elasticity):*** changes in the price of the competitor's price also affect the demand of the product

Simulator

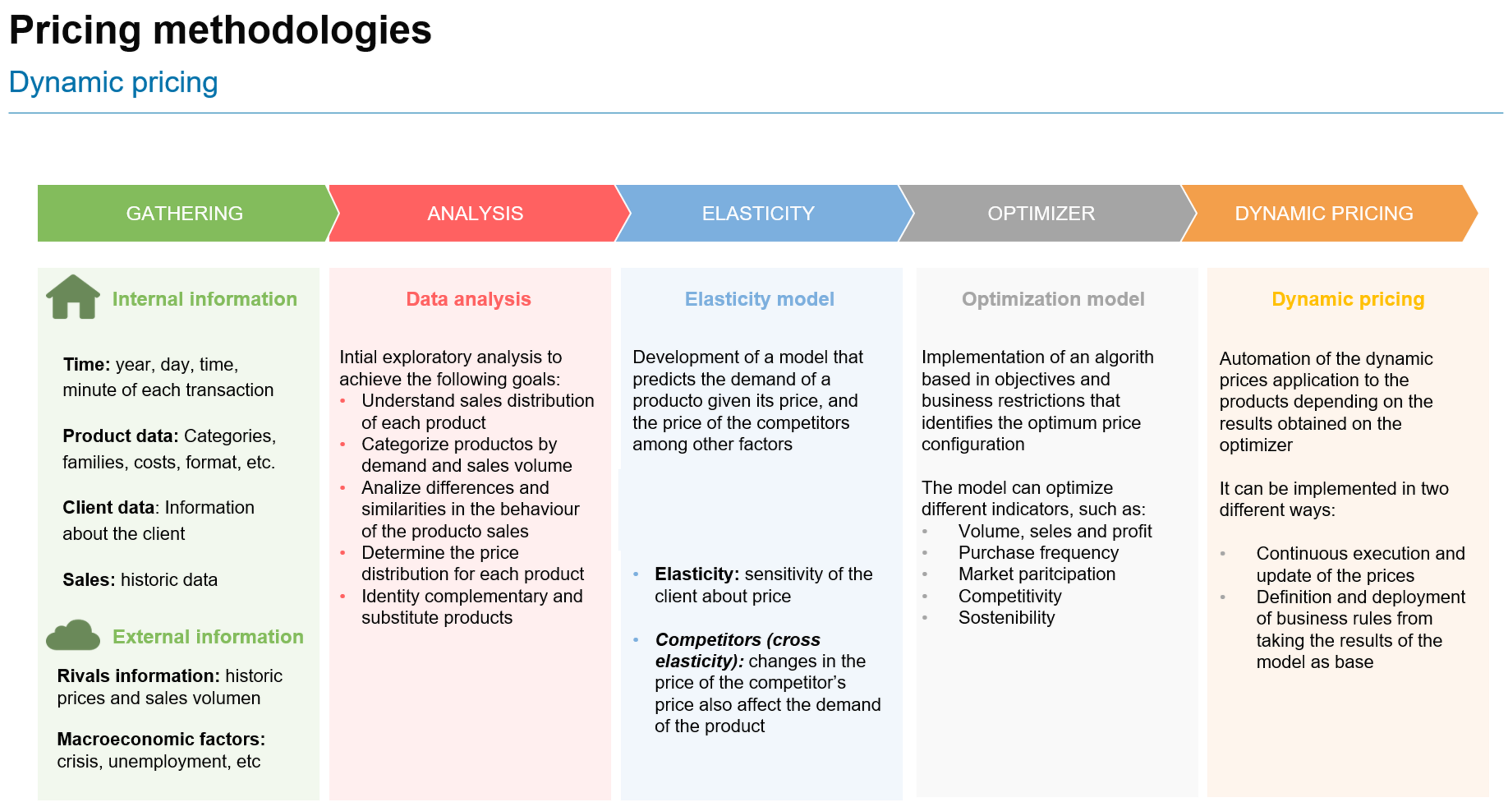
Creation of a simulation environment that allows the evaluation of the changes in the demand caused by the changes in the studied factors

Optimization model

Implementation of an algorith based in objectives and business restrictions that identifies the optimum price configuration

The model can optimize different indicators, such as:
- Volume, seles and profit
- Purchase frequency
- Market paritcipation
- Competitivity
- Sostenibility

Figure 1.1: Optimal pricing workflow



Pricing methodologies

Dynamic pricing

GATHERING → ANALYSIS → ELASTICITY → OPTIMIZER → DYNAMIC PRICING

Internal information

Time: year, day, time, minute of each transaction

Product data: Categories, families, costs, format, etc.

Client data: Information about the client

Sales: historic data

External information

Rivals information: historic prices and sales volumen

Macroeconomic factors: crisis, unemployment, etc

Data analysis

Intial exploratory analysis to achieve the following goals:

- Understand sales distribution of each product
- Categorize productos by demand and sales volume
- Analize differences and similarities in the behaviour of the producto sales
- Determine the price distribution for each product
- Identity complementary and substitute products

Elasticity model

Development of a model that predicts the demand of a producto given its price, and the price of the competitors among other factors

- **Elasticity:** sensitivity of the client about price
- ***Competitors (cross elasticity):*** changes in the price of the competitor's price also affect the demand of the product

Optimization model

Implementation of an algorith based in objectives and business restrictions that identifies the optimum price configuration

The model can optimize different indicators, such as:

- Volume, seles and profit
- Purchase frequency
- Market paritcipation
- Competitivity
- Sostenibility

Dynamic pricing

Automation of the dynamic prices application to the products depending on the results obtained on the optimizer

It can be implemented in two different ways:

- Continuous execution and update of the prices
- Definition and deployment of business rules from taking the results of the model as base

Figure 1.2: Dynamic pricing workflow

Chapter 2

Scope

2.1 User stories

2.1.1 System design

- As tech lead I want an implementation of an access to the database, independent from the database management system
- As tech lead I want an implementation of an interface to access the database
- As tech lead I want an implementation of a file reading/writing layer that allows the storage in different locations
- As tech lead I want an implementation of a data load process independent form the data infrastructure

2.1.2 Allowance of the architecture of the system

- Implementation of the infrastructure as code (IaC) using the Terraform framework
- Implementation of an automated deployment pipeline using bitbucket-pipelines

2.1.3 Definition of the algorithmic solution

- As tech lead I want implementation of an algorithm that predicts units sold from a level of discount and some more inputs
- As tech lead I want implementation of a dynamic price optimizer that chooses the best sequences of discounts among all the possible combinations
- As tech lead I want scheme of the features required by the model as inputs
- As tech lead I want a definition of the queries required for the training and the prediction processes
- As tech lead I want a definition of the transformation process of the data

2.1.4 Enabling a persistence layer

- As tech lead I want an implementation of an access to the database, independent from the database management system
- As tech lead I want implementation of an interface to access the database
- As tech lead I want implementation of a file reading/writing layer that allows the storage in different locations
- As tech lead I want implementation of a data load process independent form the data infrastructure

2.1.5 Enabling a model governance layer

- As tech lead I want implementation of a MLOPS module, to have a registry of the different executions
- As tech lead I want implementation of a metrics registry
- As tech lead I want implementation of a model registry
- As tech lead I want implementation of parameters/hyper-parameters registry

2.1.6 Implementation of a data ingestion module

- As a tech lead I want to implement a data ingestion command, independent from the training one.
- As a tech lead I want to implement a load and execution method for SQL queries
- As a tech lead I want to implement an interface that supports ad-hoc preprocessing locally

2.1.7 Implementation of a data quality module

- As a tech lead I want to implement a data quality module that generates reports with several metrics about the ingested data

2.1.8 Implementation of a data preprocessing module

- As a tech lead I want to implement a data preprocessing command, which applies the desired preprocessing to the desired variables

2.1.9 Implementation of a model training module

- As a tech lead I want a model training module independent from the ingestion one
- As a tech lead I want the training module to register its activity through the model governance module
- As a product owner I need a system of cross-validation compatible with the SKLearn API, allowing the integration of different strategies

2.1.10 Implementation of an optimizer module

- As a tech lead I want the possibility of excluding variables from the optimization process
- As a tech lead I want the possibility of encoding new variables to feed the models from a configuration file
- As a tech lead I want to explore all the different combinations the algorithm could possibly generate
- As a tech lead I want the possibility of setting some cold-start weeks in which the discounts are not optimized but maintained constant
- As a tech lead I want the possibility of fixing an amount of minimum stock at the end of the sales period
- As a tech lead I want to allow the traceability of the optimization process week by week
- As a tech lead I want the optimizer to be configurable by number of discount weeks, range of discounts or number of cold-start weeks
- As a tech lead I want the optimizer to have a configurable hierarchy when performing the training or prediction
- As a tech lead I want the optimizer to use an heuristic to guarantee the provided best combination is also stable
- As a tech lead, I want both training and prediction to be scalable and optimized in time

2.2 Functional requirements

2.2.1 System Design

1. Design of the infrastructure of the system to estimate the required resources

2. Design of the modules of the system and why do we need each of them

2.2.2 Allowance of the architecture of the system

1. Implementation of the infrastructure as code (IaC) using the Terraform framework
2. Implementation of an automated deployment pipeline using bitbucket-pipelines

2.2.3 Definition of the algorithmic solution

1. Implementation of an algorithm that predicts units sold from a level of discount and some more inputs
2. Implementation of a dynamic price optimizer that chooses the best sequences of discounts among all the possible combinations
3. Scheme of the features required by the model as inputs
4. Definition of the queries required for the training and the prediction processes
5. Definition of the transformation process of the data

2.2.4 Enabling a persistence layer

1. Implementation of an access to the database, independent from the database management system
2. Implementation of an interface to access the database
3. Implementation of a file reading/writing layer that allows the storage in different locations
4. Implementation of a data load process independent form the data infrastructure

2.2.5 Enabling a model governance layer

1. Implementation of a MLOPS module, to have a registry of the different executions
2. Implementation of a metrics registry
3. Implementation of a model registry
4. Implementation of parameters/hyper-parameters registry

2.2.6 Implementation of a data ingestion module

1. Implement a data ingestion command, independent from the training one.
2. Implement a load and execution method for SQL queries

2.2.7 Implementation of a data quality module

- Implementation of a data quality module that generates reports with several metrics about the ingested data

2.2.8 Implementation of a data preprocessing module

- Implementation of a data preprocessing command, which applies the desired preprocessing to the desired variables

2.2.9 Implementation of a model training module

1. The model training module must be independent from the ingestion one
2. The training module must register its activity through the model governance module
3. There must be a system of cross-validation compatible with the SKLearn API, allowing the integration of different strategies

2.2.10 Implementation of an optimizer module

1. The optimizer must have a configurable hierarchy when performing the training or prediction, and also allow to configure a set of parameters
2. The optimizer must use an heuristic to guarantee the provided best combination is also stable

2.3 Non-functional requirements

1. A class diagram, for documentation purposes
2. The system must be as less coupled as possible, to allow testing and changing parts independently
3. The system must be secure, each part having it's own credential
4. The process of deployment must be as automatic as possible
5. The system must be configurable easily through configuration files or another method
6. The solution must be traceable, and offer information about the intermediate steps, not only the final answer
7. The activity of the system must be registered as much as possible using registries
8. Both training and prediction must be scalable and optimized in time

9. It must be possible to exclude variables from the optimization process
10. It must be possible to encode new variables to feed the models from a configuration file
11. It must be possible to explore all the different combinations the algorithm could possibly generate
12. It must be possible to set some cold-start weeks in which the discounts are not optimized but maintained constant
13. It must be possible to fix an amount of minimum stock at the end of the sales period
14. The optimizer must be configurable by number of discount weeks, range of discounts or number of cold-start weeks

2.4 Scope

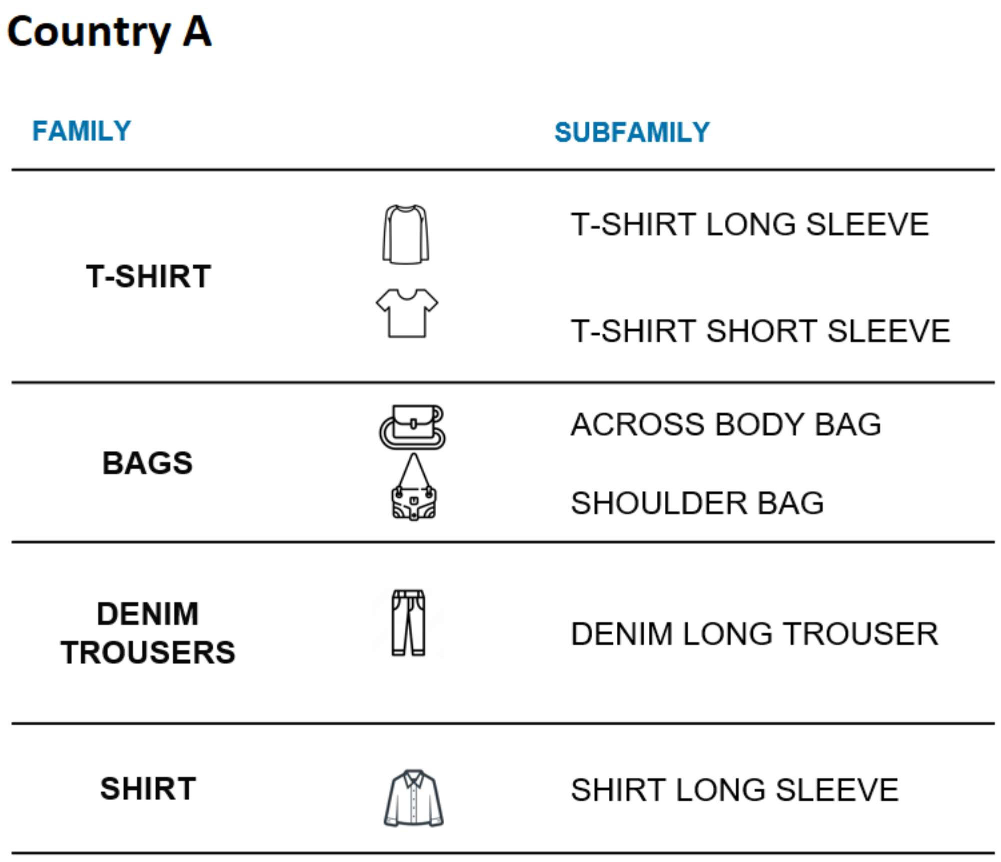
2.4.1 Milestones

Milestone	Description
1	Audit closing
2	Mirror references definition and PoC scope
3	Preliminary design of the solution
4	Mirror references development
5	Project status
6	Data preparation
7	Initial sales prediction model
8	Income optimizer
9	Simulation environment
10	Testing plan
11	Conclusions and documentation

2.4.2 Development and test of a pilot

The main goal of this pilot project is producing a tool able to optimize the sequence of discounts applied to different products in order to maximize the income at the end of the discount season. Thereby the optimizer must be executed week after week to constantly check if the sales prediction made by our model matches the real behaviour of the market.

To prove this better efficiency an experiment concerning two countries and different product subfamilies will be conducted, one f them using the automated markdown process and the other one following the classic method.

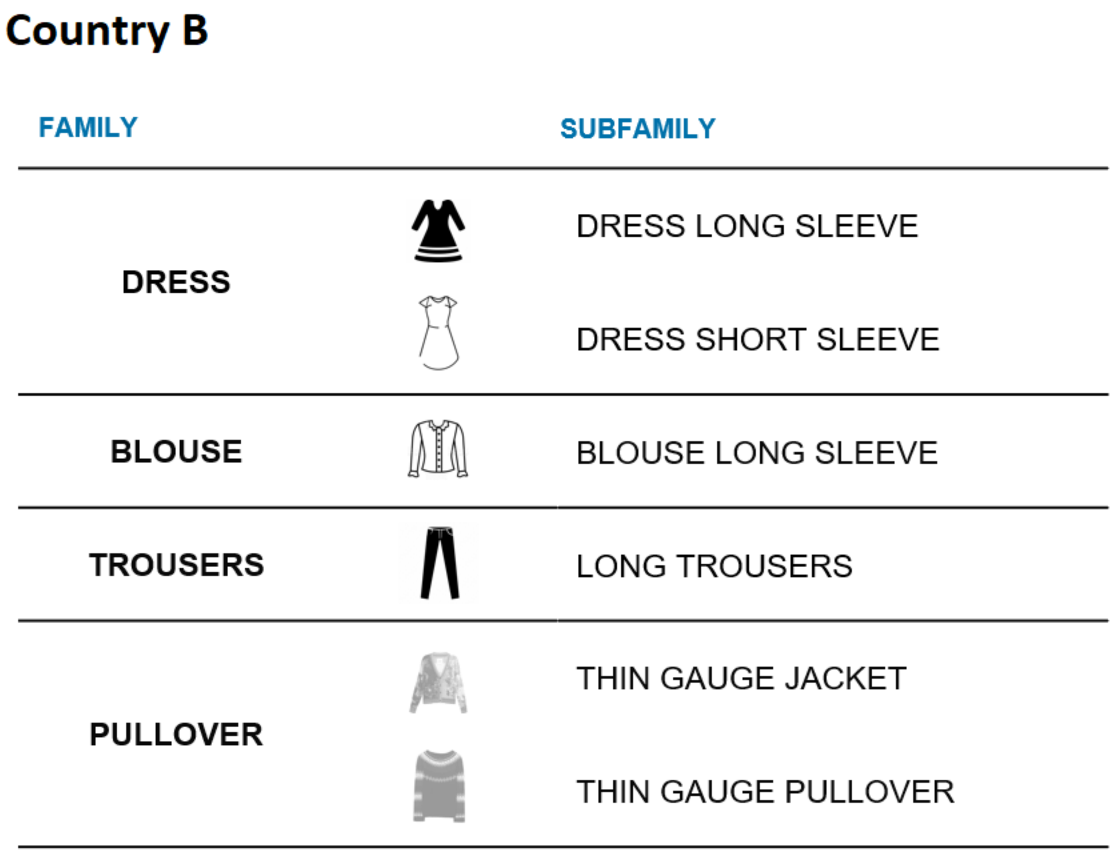


Country A

FAMILY	SUBFAMILY
T-SHIRT	T-SHIRT LONG SLEEVE
	T-SHIRT SHORT SLEEVE
BAGS	ACROSS BODY BAG
	SHOULDER BAG
DENIM TROUSERS	DENIM LONG TROUSER
SHIRT	SHIRT LONG SLEEVE

Figure 2.1: Option A (automated markdown)

Once this pilot demonstrates the better efficiency with respect to the company's income, the tool could be upgraded in order to optimize another business variables, although income will certainly keep being the most important one.



Country B

FAMILY		SUBFAMILY
DRESS		DRESS LONG SLEEVE
		DRESS SHORT SLEEVE
BLOUSE		BLOUSE LONG SLEEVE
TROUSERS		LONG TROUSERS
PULLOVER		THIN GAUGE JACKET
		THIN GAUGE PULLOVER

Figure 2.2: Option B (classic)

Chapter 3

Analysis

3.1 Proposed solution - pipeline

Understanding the business and diagnosing the information

1. Understanding the general context of the project
2. Defining the problem to resolve and objectives to reach
3. Definition of goals: possible casuistic and success criteria
4. Current situation analysis: assumptions and risks
5. Functional definition of the dataset

Design of the solution and preparation of the data

1. Initial data auditory
2. Creation of historic data using mirror references
3. Functional design of the solution: sales prediction and optimizer
4. Design of synthetic variables
5. Creation of the model input data from mirror products history
6. Evaluation and interpretation: analysis of variables and dependencies

Development of analytic and modelation environments

1. Development of the sales prediction model
2. Development of the margin optimizer
3. Development of the simulation environment
4. Integration of developments
5. Creation of different scenarios and comparatives

Chapter 4

State of the art

This thesis derives from the assumption that exists a relationship between price and demand for retail products. This lead to the idea that demand can be predicted from price and another factors, which can then allow us to modify our price variable to achieve maximized income.

In this chapter several methods applied to the dynamic pricing problem will be described, some of them constituting the current state-of-the-art.

The literature on dynamic pricing can roughly be classified as follows:

- Models where the demand function is dynamically changing over time
- Models where the demand function is static, but where pricing dynamics are caused by the inventory level.

In the first class of models, the demand function changes according to changing circumstances: for example, the demand function may depend on the time-derivative of price, on the current inventory level, on the amount of cumulative sales, on the firm's pricing history, etcetera. In the second class of models, it is not the demand function itself that causes the price dynamics. Instead,the price dynamics are caused by inventory effects.

For this thesis, we are more interested in the first group, in which the demand by the consumers depends on several variables, and price is usually the one to be controlled.

4.1 Previous work

4.1.1 Demand depends on price-derivatives

Assumes that the (deterministic) demand is not only a function of price, but also of the time-derivative of price. This models the fact that buyers do not only consider the current selling price in their decision to buy a product, but also anticipated price changes

4.1.2 Demand depends on price history

These models consider the effect of reference price over the demand. Reference prices are perceptions of customers about the price that the firm has been charging in the past. A difference between the reference price and the actual selling price influences the demand, and, as a result, each posted selling price does not only affect the current demand but also the future demand.

4.1.3 Demand depends on amount of sales

In these models, the demand for products does not only depend on the selling price, but also on the amount of cumulative sales. This allows modeling several phenomena related to market saturation, advertisement, word-of-mouth effects, and product diffusion. These problems are solved by principles from optimal control theory, and the optimal pricing strategy is often given by the solution of a differential equation.

4.2 Parametric and Non-Parametric Models

Both types of models have the objective of finding the the function which allows to predict the our dependent variable, in our case, the product sales, given our input data or observations. These observations can be called training data, which will be used to teach a model how to estimate f. Let x_{ij} represent the value of the j th predictor, or input, for observation i, where $i = 1, 2, \ldots, \mathrm{n}$ and $j = 1, 2, \ldots, \mathrm{p}$. Correspondingly, y_i represents the response variable for the i th observation. Then our training data consist of $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ where $x_i = (x_{i1}, x_{i2}, \ldots, x_{ip})^T$. In this sense, the approach is to find, through machine learning, the function f where $Y \approx f(X)$ for any observation (X, Y). The machine learning algorithms which accomplish this task are characterized as parametric or non-parametric.

4.2.1 Parametric

The parametric model, as it name states, defines an objective function that depends on some parameters, for example, a linear model.

$$f(X) = \alpha_0 + \alpha_1 * X_1 + +\alpha_2 * X_2 + +\alpha_n * X_n$$

The potential disadvantage of this method is that the true f is likely to not fit in this kind of parametric function. Parametric models include Linear Regression, Logistic Regression, Penalized Linear Models, Naive Bayes and Lasso.

Under this setting, a parametric model of demand (often linear in price) is assumed and the unknown parameters of the demand model are learned in a sequential manner. den Boer and Zwart (2013) [DBZ14] assume a linear price demand relationship and propose a controlled variance pricing policy that accomplishes sufficient learning by introducing variance in the dynamically chosen prices. Keskin and Zeevi (2014) [KZ14] find sufficient conditions under which a pricing policy is optimal in terms of its rate of regret in the linear demand setting. Handel and Misra (2015) [HM15] use techniques from robust optimization to solve the dynamic pricing with unknown demand parameters. More recently, Ban and Keskin (2021) [BK21] and others have used parametric models of demand that not only include price but other product related covariates for optimal pricing decisions.

4.2.2 Non-Parametric

On the other hand, a non parametric model can model more complex functions from the training data. It takes into account more information from the current set of data that is attached to the model at the current state. It seeks an estimate of f that gets as close to the data points as possible without being too rough or wiggly. By avoiding the assumption of a particular functional form for f, they have the potential to accurately fit a wider range of possible shapes for f

On the other hand, Non-Parametric Models are conformed by Non-Linear models, with algorithms including Neural Networks, K-Nearest Neighbors and Support Vector Machines (SVMs).

Under this setting, demand is assumed to belong to a family of demand curves characterized by some structural properties of the revenue function such as its concavity and unimodality. Besbes and Zeevi (2015) [BZ15] construct a misspecified algorithm that uses linear models of demand to optimize subsequent prices. Somewhat surprisingly, they find that even though misspecified, their constructed policy is near optimal. Others such as and Chen and Gallego (2018) [CG18] also construct non-parametric pricing policies but do not account for price changes.

4.3 Machine learning models

Machine learning techniques allow us to model many phenomena that influence the demand, such as competition, fluctuating demand, and strategic buyer behavior. A drawback is that these models are often too complex to analyze analytically, and insights on the behavior of various pricing strategies can only be obtained by performing numerical experiments

Machine-learning techniques that have been applied to dynamic pricing problems include evolutionary algorithms (Shakya et al., 2009) [SOO09], particle swarm optimization (Mullen et al., 2006) [MMSW06], reinforcement learning and Q-learning (Kutschinski et al., 2003) [KUP03], simulated annealing (Xia and Dube, 2009) [XD09], Markov chain Monte Carlo methods (Chung et al., 2012) [CLY$^+$12], goal-directed and derivative-following strategies in simulation (Vázquez-Gallo et al., 2014) [VGEE14], neural networks (Shakya et al. 2012) [SKOC12], and direct search methods (Brooks et al., 2002) [BGD$^+$02].

4.4 Tree models

Lately, one of the machine learning algorithms providing best results in a variety of tasks belong to the tree models family.

Classification and Regression Trees (CART) were introduced by Breiman et al. [BFOS83] in 1984. This method uses a decision tree (as a predictive model) to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves). Tree models where the target variable can take a discrete set of values are called classification trees; in these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. Decision trees where the target variable can take continuous values (typically real numbers) are called regression trees.

A tree is built by splitting the source set, constituting the root node of the tree, into subsets—which constitute the successor children. The splitting is based on a set of splitting rules based on classification features. This process is repeated on each derived subset in a recursive manner called recursive partitioning. The recursion is completed when the subset at a node has all the same values of the target variable, or when splitting no longer adds value to the predictions.

When an observation has reach a leaf, its value can be assigned be mean or majority voting, whether if they are a Regression Tree (the output is a continuous number) or a Categorical one (the output is a discrete class).

There are some techniques, called *ensemble methods* that construct more than one decision tree. Among them we have:

- **Bagging** or bootstrap aggregated. This technique was also introduced by Breiman [BFOS83]. It's one of the first ensemble techniques. Ensemble means to use one than one model to combine their predictions and obtain a better one, achieving a better performance. Bagging utilizes bootstrap re-sampled versions of the training

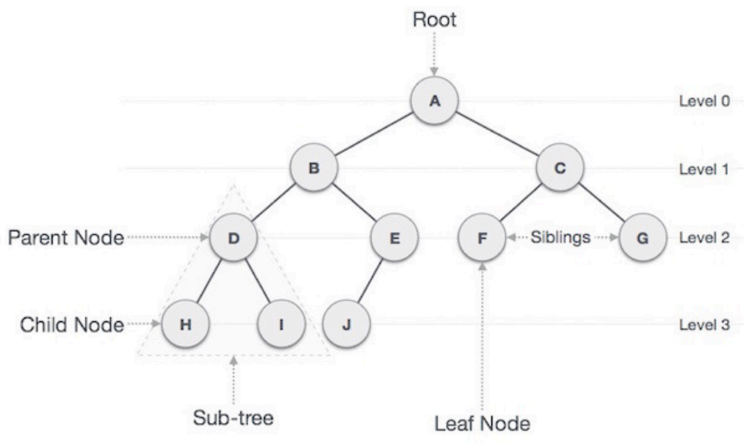


Figure 4.1: Structure of a tree algorithm

data and builds a regression or classification tree from each of that versions and then predicts the final result by majority vote

- **Boosting** It was first implemented by Freund and Schapire in 1996 [FS96] as the AdaBoost algorithm. It's an iterative process that starts with a weak classifier and fits over the last model to improve its performance. After fitting one tree the algorithm assign weights in the previous error and generates a new tree, so it takes into account where mistakes were made and adjusts the tree according to them.

- **Random forest** It is a special case of bagging. Also introduced by Breiman in order to diminish the high correlation introduced by the bagging algorithm. In bagging, all samples are using the same variables, so most of the trees will have the same structure. To reduce this correlation randomness is introduced into the algorithm. The process is (1) generate bootstrap re-sampled data-set, (2) randomly choose a subset of predictors or features, (3) Select the best feature maximizing the gain to generate the decision splits and continue to build the tree, (4) repeat the process several times to average the results.

  The main difference between boosting and random forest is how the ensemble procedure is done. In Random Forest the trees are independent, can reach their maximum depth and contribute equally. In boosting it is quite different, trees depend on past trees, have limited depth and contribute differently to the final model, in this case the final classifier a is weighted average of classifiers

4.4.1 XGBoost

XGBoost is short for eXtreme gradient boosting. It is a library designed and optimized for boosted tree algorithms. It's main goal is to push the extreme of the computation limits of machines to provide a scalable, portable and accurate solution for large scale tree boosting.

Gradient boosting was proposed by Friedman [Fri01] after the previous ones had been studied. This method became very popular due to its increased performance compared to the existing models and lead to the development of XGBoost. XGBoost is a supervised learning approach, so it need an objective function to be evaluated and a training cycle to adapt to the dataset.

Objective function The objective function is divided in two parts: Training Loss $+$ Regularization.

$$Obj(\Theta) = L(\theta) + \Omega(\Theta)$$

where L is the training loss function, and Ω is the regularization term. The training loss measures how predictive the model is on training data. For example, a commonly used training loss is mean squared error.

$$L(\theta) = \sum_i (y_i - \hat{y}_i)^2$$

The regularization term controls the complexity of the model, and helps in avoiding overfitting, which means the algorithm learns the training set specifically and limits it to generalize to new data points. It is a penalty for the coefficients of the algorithm that shrinks them to zero in order to avoid capturing the noise of the data. The general principle is that the goal is to achieve both a simple and predictive model. The trade-off between the two is also referred as bias-variance trade-off in machine learning.

Now that the main idea has been displayed, for simplicity, the process of how the algorithm learns will be avoided. But there are important things to know about the model.

Learning The idea behind Gradient Boosting is similar to the previously explained Adaboost. Select tree depth, P and number of iterations, B. Compute the average response, and use this as the initial predicted value for each sample. First select a tree depth P, and a number of iterations B. For $b = 1$ to B :

1. A weak learner is fitted by a Classification or Regression Tree.
2. Compute the residuals, difference between the observed value and the predicted value for each sample.
3. Fit a regression tree with depth P using the residuals as the response.
4. Predict each sample using the regression tree fitted in the previous step.
5. Update the predicted value of each sample by adding the previous iteration's predicted value to the predicted value generated in the current step.

4.4.2 LightGBM

Proposed by Microsoft Research in 2017 [KMF$^+$17], this algorithm introduces two novel techniques that help reducing the high computational and memory cost of other gradient boosting algorithms as the previously mentioned XGBoost.

This and another gradient boosting decision trees implementations must, for every feature, scan all the data instances to estimate the information gain of all the possible split points. Therefore, their computational complexities will be proportional to both the number of features and the number of instances. This makes these implementations very time consuming when handling big data.

To tackle this problem, LightGBM introduces two techniques that lead to the reduction of that complexity. Those are *Gradient-based One-Sida Sampling (GOSS)* and *Exclusive Feature Bundling (EFB)*

Gradient-based One-Side Sampling (GOSS). While there is no native weight for data instance in GBDT, we notice that data instances with different gradients play different roles in the computation of information gain. In particular, according to the definition of information gain, those instances with larger gradients [1] (i.e., under-trained instances) will contribute more to the information gain. Therefore, when down sampling the data instances, in order to retain the accuracy of information gain estimation, we should better keep those instances with large gradients (e.g., larger than a pre-defined threshold, or among the top percentiles), and only randomly drop those instances with small gradients. We prove that such a treatment can lead to a more accurate gain estimation than uniformly random sampling, with the same target sampling rate, especially when the value of information gain has a large range.

Exclusive Feature Bundling (EFB). Usually in real applications, although there are a large number of features, the feature space is quite sparse, which provides us a possibility of designing a nearly lossless approach to reduce the number of effective features. Specifically, in a sparse feature space, many features are (almost) exclusive, i.e., they rarely take nonzero values simultaneously. Examples include the one-hot features (e.g., one-hot word representation in text mining). We can safely bundle such exclusive features. To this end, we design an efficient algorithm by reducing the optimal bundling problem to a graph coloring problem (by taking features as vertices and adding edges for every two features if they are not mutually exclusive), and solving it by a greedy algorithm with a constant approximation ratio.

When adding these improvements, the new algorithm, *LightGBM* speeds up the training process by up to 20 times compared to *XGBoost* while maintaining almost the same level of accuracy. This algorithm is the one chosen to use during this project, as the performance of another gradient boosting tree classifiers has been previously proved in dynamic pricing problems. It does not improve the performance of another models but it reduces drastically the training time, and therefore, the costs of computing, which is very valuable for a solution that will be used inside a business environment.

Chapter 5

Design of our solution

5.1 High level view

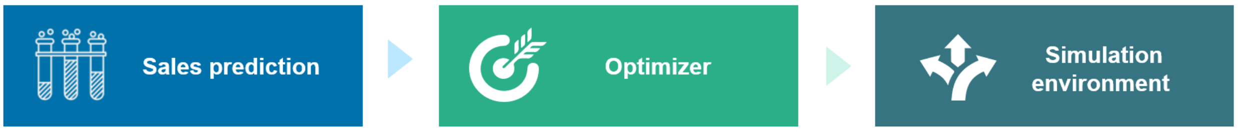


Figure 5.1: The three main components

- **Sales prediction:** Development of an advanced analytic model which allows to predict the sales of a model-color given a set of variables, including price elasticity
  - One productive model for each subfamily-country
  - Reference level (model-color)
  - Weekly aggregated data
- **Optimizer:** Development of a mathematical model for obtaining the combination of discounts that maximizes the income of the client
  - Predefined range of input prices
  - Definition of the income function
  - Consideration of the stock within the computation
- **Simulation environment:** Development of a simulation environment from which the final user is able of executing the optimizer for different configurations and validate the optimal prices obtained
  - Optimal prices combination
  - Price simulator restricted to the set of predefined prices

5.2 Architecture

The general workflow is represented in the following functional diagram

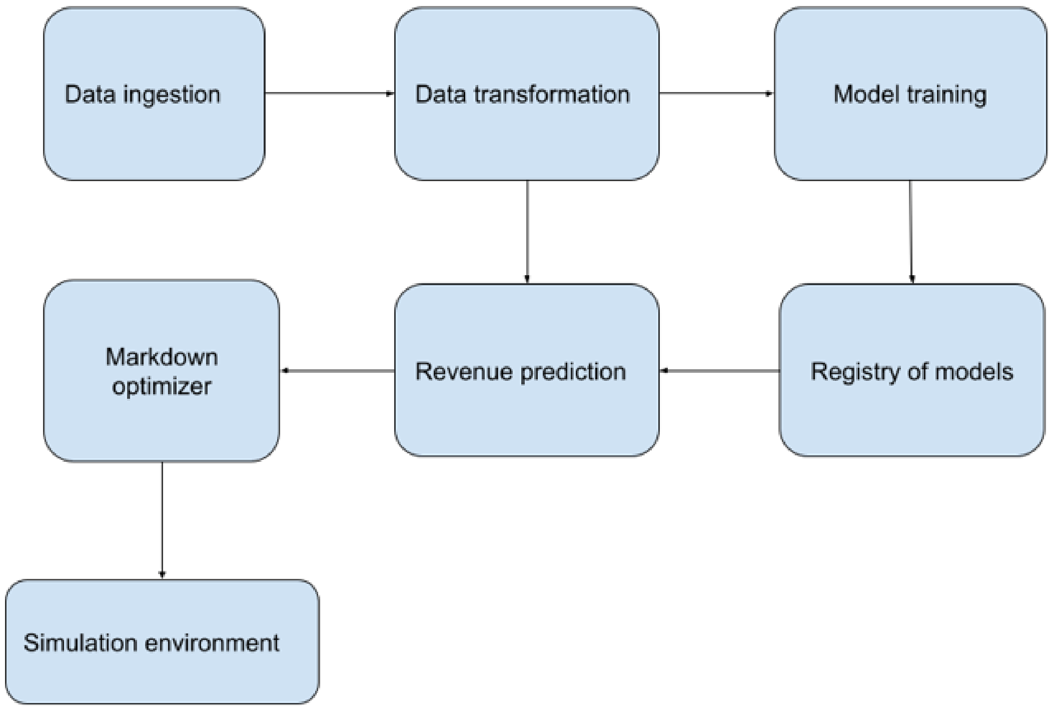


Figure 5.2: Functional diagram

The workflow starts with the data ingestion and transformation. Historic data will be used to train models, and a registry of the trained models will be stored in a registry. New data will be the input for the trained models to predict the revenue, which will be then be used to optimize the markdown process. Finally, the results can be tested inside a simulation environment

The architecture needed to sustain this entire system is composed by several parts. Among them we have:

- Database service
- Service to execute our model in
- Deployment service
- Code repository
- Secrets guardian

Given the growing usage and popularity of the cloud-based systems, and the advantages they provide, such as the warranted scalability and availability, increased speed and efficiency, we chose to base our solution in a cloud architecture.

Specifically, we chose the Amazon Web Services cloud to build the required infrastructure, using the following technologies.

- **Amazon Sagemaker** to train and test the models
- **Amazon AWS** where the system will be deployed and the predictions made
- **Bitbucket** as a web repository where the code kept
- **Secured S3 buckets** to protect data and secrets

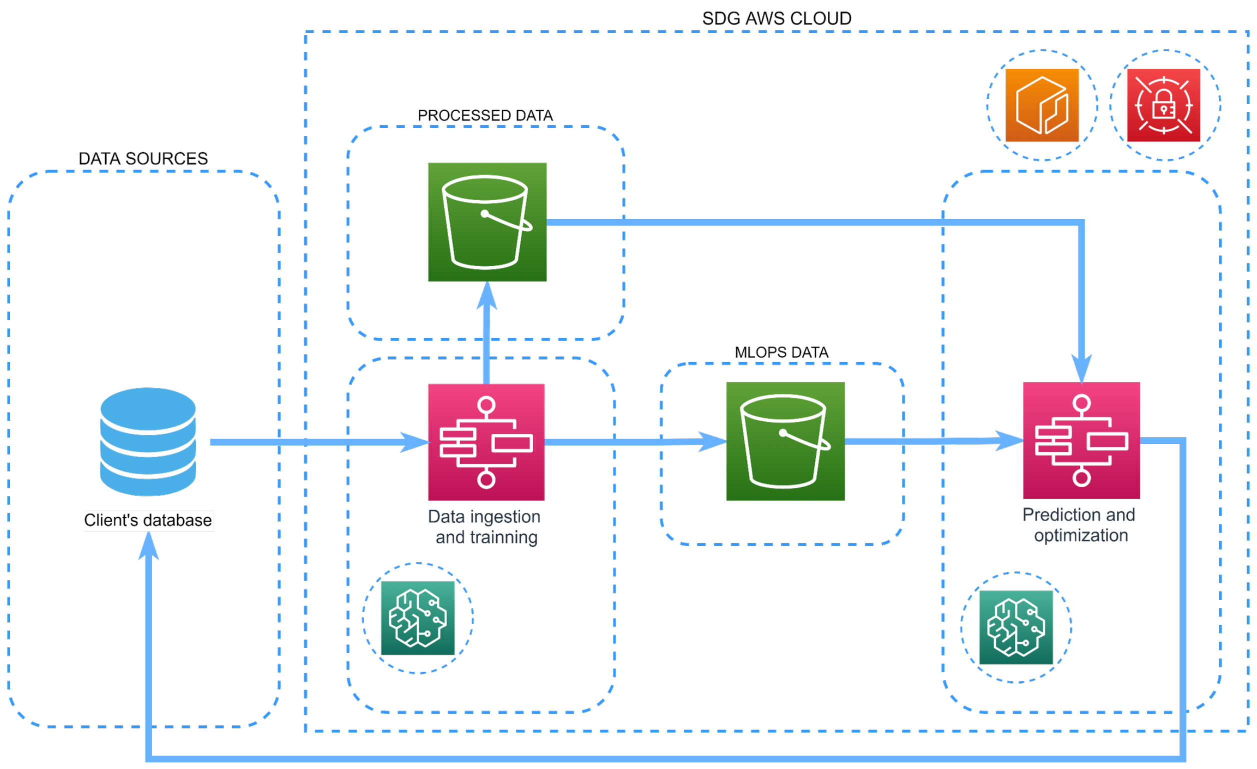


Figure 5.3: Architecture schema

The above figure illustrates the chosen architecture to develop this solution. Amazon Sagemaker is an automated learning tool that facilitates the development of a model and the posterior training of it. It provides the hardware infrastructure to carry on the training, and computes several metrics which allow the team to evaluate the performance of the model.

Given the compatibility between the previous technologies and the Amazon Web Services, the easiest solution came through using the Amazon cloud to provide the deployment infrastructure and also the security part. The resulting architecture is then a robust one, with largely tested compatibility between its different parts while being decoupled from each other.

Finally, for developing the software itself Bitbucket has been used. This service inherits most of its capacities from the well know Git and adds some other features.

This architecture is implemented with Terraform. Terraform is an open-source infrastructure as code software tool that provides a consistent CLI workflow to manage hundreds of cloud services. Terraform codifies cloud APIs into declarative configuration files. By using this framework we can easily configure the whole infrastructure as it was designed with code.

Next, a more detailed view of the architecture is presented.

The data sources are allocated into the client's cloud, ideally being composed of two different schemas.

The data schema is where historic product data from different products is stored, and is the information which will be retrieved by the processing layer to train the predictive models. The MlOps Schema will storage data about the trained models (parameters, metrics, other registries) which will later be retrieved by the predictor/optimizer module, whose results will also be stored here

The processing layer ingests raw data from the client's database, transforms it and train models with it. In a second cycle, it recovers the prediction dataset and the previous model and parameters from the respectives data sources and stores and predicts the best possible markdown, which will later be exported back to the client's database

All the interactions within the processing layer are secured by an external secrets guardian which assures only the allowed users can access to the different parts of the processing

The storage layer is where the code of the project itself and other data derived from intermediate processing stages is stored. This includes preprocessed data for training and the resulting model artifacts used for making predictions.

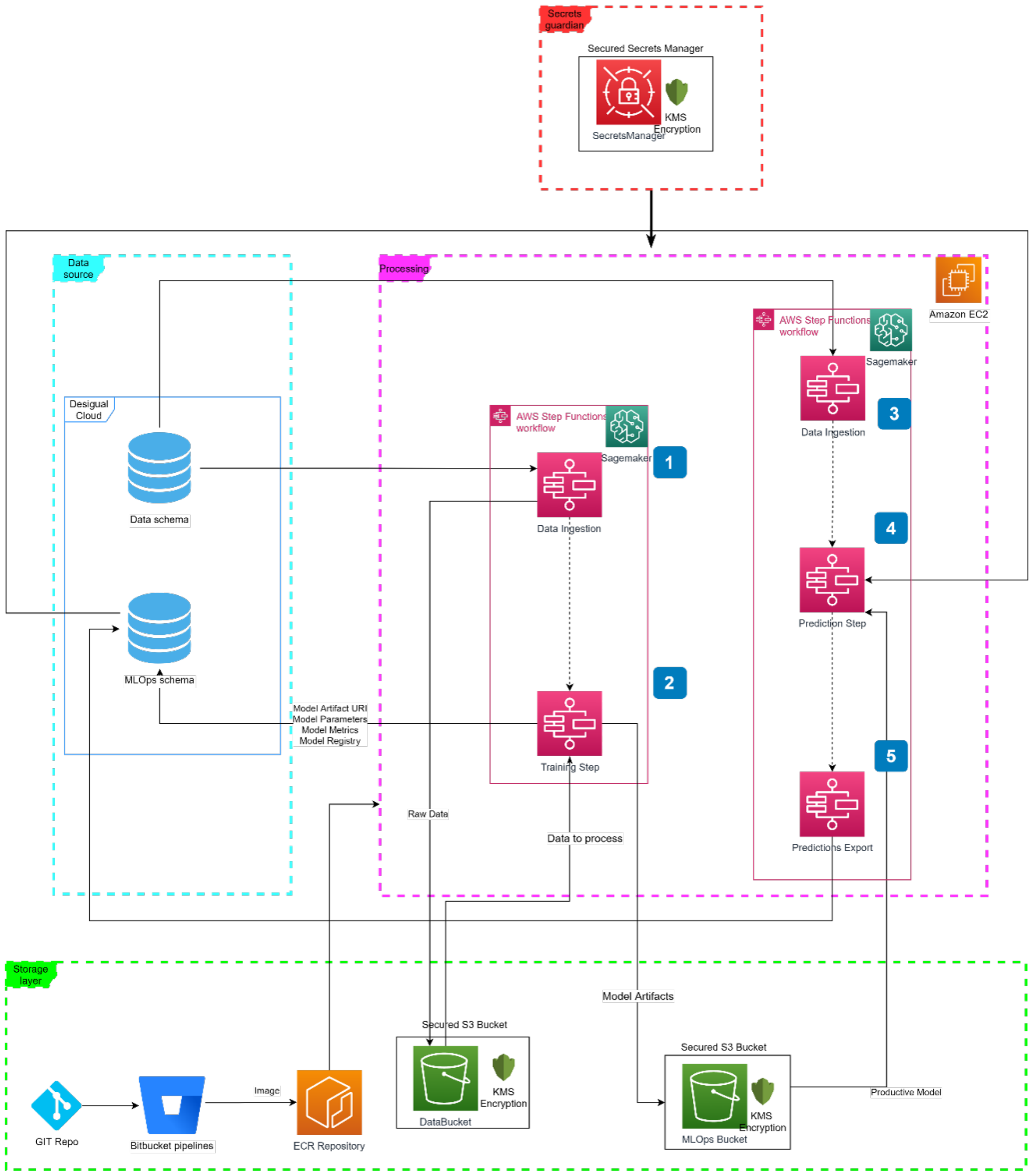


Figure 5.4: Architecture detailed schema

Main steps

1. The first step is to retrieve the training data from the database, and dump it in an S3 bucket

2. The second step, raises a Sagemaker instance, which retrieve the preprocessed data and trains the models, finally stores the trained models in the mlops bucket and saves the metrics in the mlops schema in the client's database

3. The data for prediction is retrieved from the client's database

4. In order to do the prediction, we retrieve the productive model from each family and run the predictions for the different discount configurations, after that, we select the optimal

5. Finally, the results are all exported to the client's data schema

5.3 Pipeline

5.3.1 Data mapping and pre-processing

The first phase after defining the architecture of the system is getting the data and apply some preprocessing.

PHASE	TASKS
Data mapping	Identify principal entities needed for the development of the use case in the client's database, schemas, etc
Integrity	Check the integrity of each of the tables and the crosses between them
Statistics by table	Count of temporal registries by table
Statistics by field	Compute basic statistic datafor the differend fields of the relevant tables, validating that their values keep being coherent
Distributions	Study the distributions of the principal metrics depending of the most relevant dimensions
Temporal series	Study the temporal evolution of the objective metric, and other relevant metrics

Table 5.1: Data analysis

Data is retrieved from the client's database and, after some preprocessing, stored in secured temporal files. There is an independent process for retrieving and reading those files, inside the StorageHandler module. This way, file management and database retrieval are two different processes and can be independently run or modified.

5.3.2 Training

For each family, a model is trained to predict sales volume according to a series of variables, always containing historic sales volume, previous discount and some other indicators.

This algorithm is the previously explained *LightGBM*. A grid search is performed for each model to find the best combination of hyper-parameters.

Variable groups:

- **Calendar:** indicator that helps the model to include external factor due to special dates
- **Prices:** All the information relative to the mirror reference's price and/or complementary/replacement references. Base price, discounts...
- **Stock:** variables that indicate the latest stock value and previous values.

- **Sales:** number of units sold

Over the variables of price, stock and sales, some other synthetic variables will be computed

5.3.3 Optimization

The optimizer will be executed week after week, updating the markdown proposal with the information available from the last week. The purpose of the optimizer is to evaluate the different outcomes produced by different combinations of discount levels, and choosing the combination that provides the higher total income

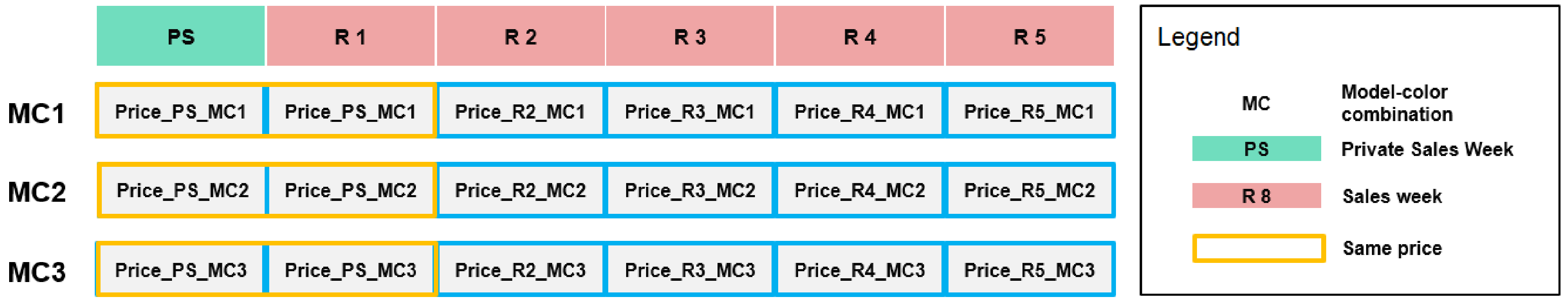


Figure 5.5: Basic weeks schema

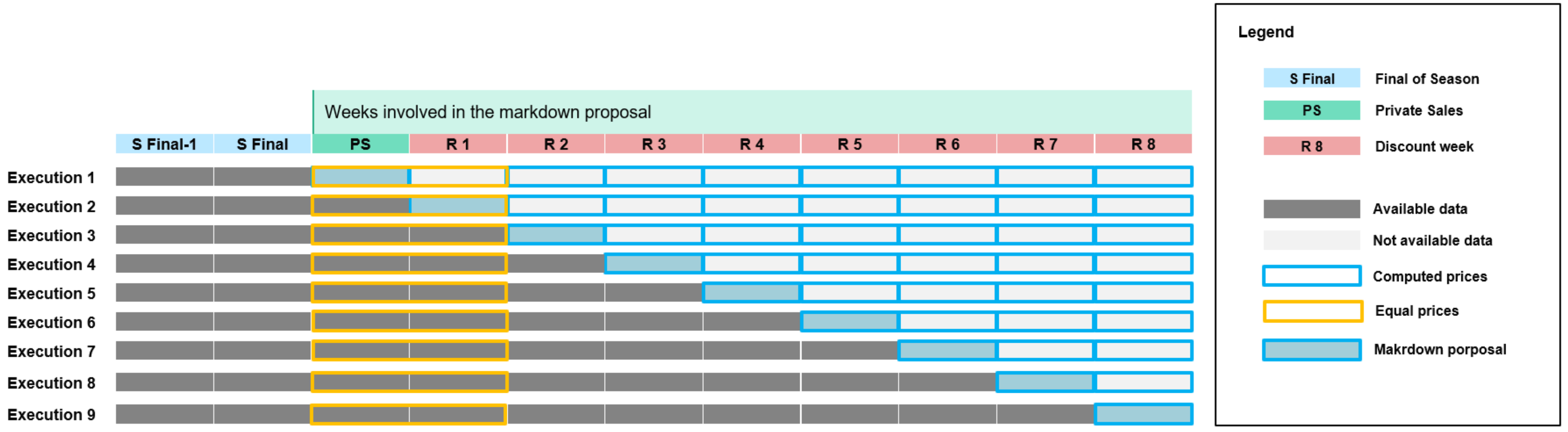


Figure 5.6: Optimizer flow

Each week can have a discount level equal or minor to the previous one, but once the level decreases it can not increase to a previous level.

That fact can be represented as a tree that grows as its depth increases.

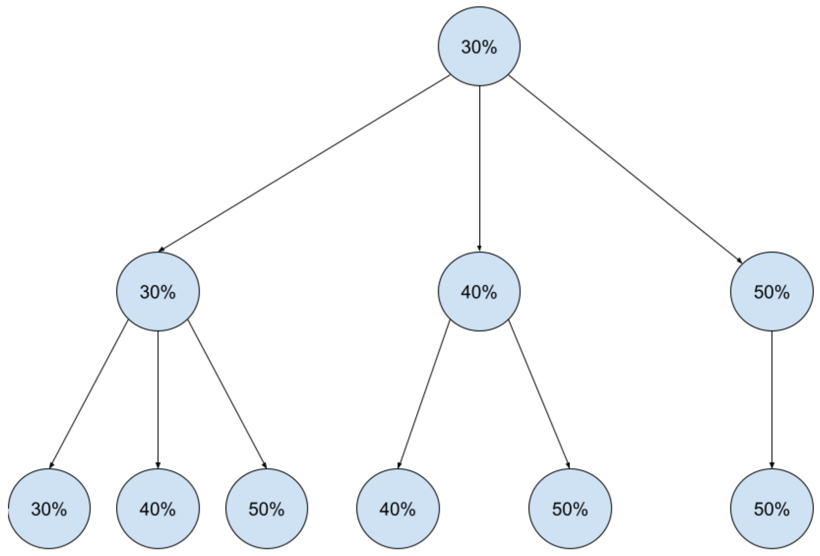
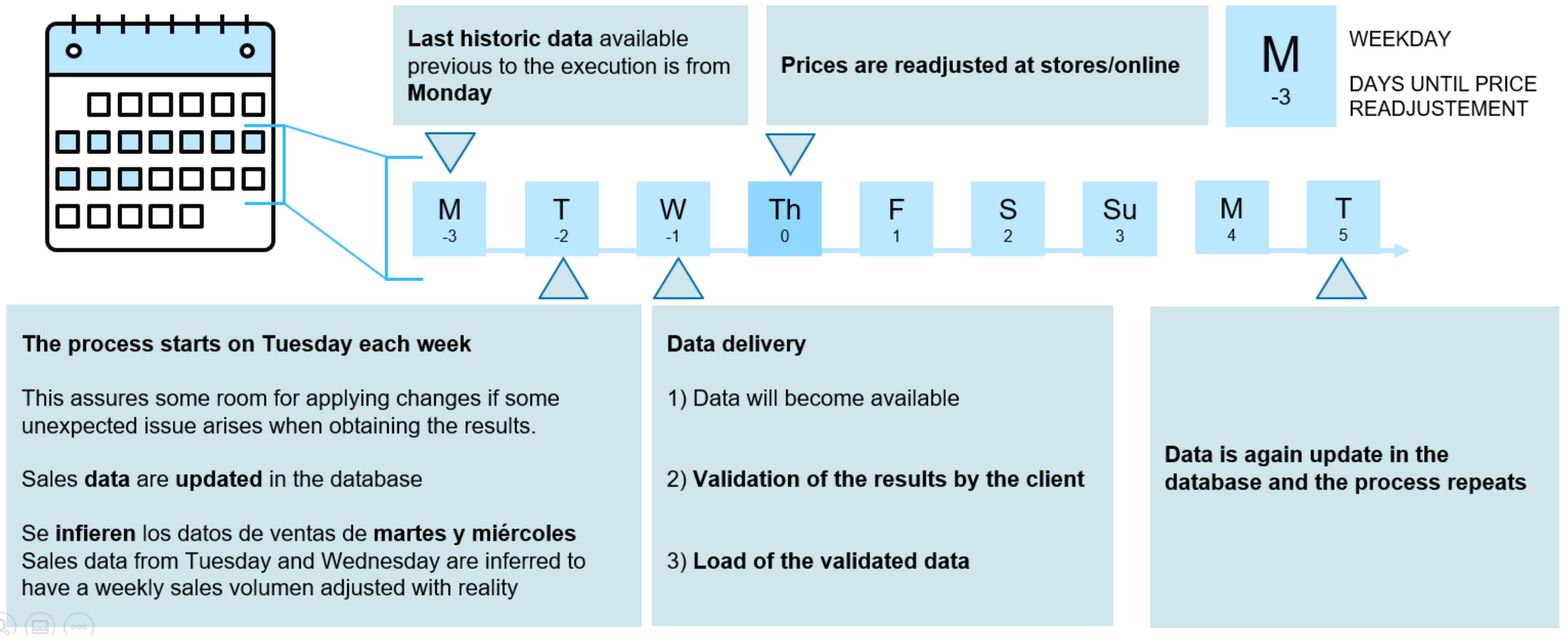


Figure 5.7: Discount level tree

5.3.4 Data update

Once a week, the historic data will be updated with the new available data from the last week sales, and new predictions will be obtained according to the following figure.



Last historic data available previous to the execution is from **Monday**
Prices are readjusted at stores/online
M
-3
WEEKDAY
DAYS UNTIL PRICE READJUSTEMENT
M -3
T -2
W -1
Th 0
F 1
S 2
Su 3
M 4
T 5
The process starts on Tuesday each week
This assures some room for applying changes if some unexpected issue arises when obtaining the results.
Sales **data** are **updated** in the database
Se **infieren** los datos de ventas de **martes y miércoles**
Sales data from Tuesday and Wednesday are inferred to have a weekly sales volumen adjusted with reality
Data delivery
1) Data will become available
2) **Validation of the results by the client**
3) **Load of the validated data**
Data is again update in the database and the process repeats

Figure 5.8: Data updating cycle (weekdays are illustrative)

Each week the system will predict the demand until the end of the season and select the best path possible, but each week more real data is fed into the system and the models can be fined tuned with it, so it the solution can be re-driven at the same time the season advances.

Chapter 6

Implementation

6.1 Architecture deployment

As stated in the previous chapter, the general architecture of the system will be specified through the Terraform framework. This framework allows to define each of the modules that compose the project and the outputs of those modules in an easy way.

The main Terraform file goes as follows:

```
    terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.27"
    }
  }
  required_version = ">=0.14.9"
  backend "gcs" {}

}

provider "aws" {
  profile = "default"
  region  = "eu-west-1"
}

module "aws_caller" {
  source = "./modules/aws-caller"
}
module "ecr" {
  source       = "./modules/ecr"
  project_tags = var.project_tags
}

module "s3" {
  source       = "./modules/s3"
  project_tags = var.project_tags
}
module "secrets_manager" {
  source       = "./modules/secrets-manager"
  project_tags = var.project_tags
}

module "iam" {
  source       = "./modules/iam"
  project_tags = var.project_tags
  bucket       = module.s3.bucket_arn
  mlops_bucket = module.s3.mlops_bucket_arn
  ecr          = module.ecr.ecr_arn
  account_id   = module.aws_caller.account_id
  secret_db    = module.secrets_manager.db_secret_arn
  secret_orm   = module.secrets_manager.orm_secret_arn
  kms_secrets  = module.secrets_manager.kms_key
  kms_buckets  = module.s3.kms_key
}

module "lambda" {
  source       = "./modules/lambda"
  project_tags = var.project_tags
  role         = module.iam.lambda_role_arn
}
module "step_functions" {
  source                     = "./modules/step-functions"
  project_tags               = var.project_tags
  ecr                        = module.ecr.ecr_uri
  bucket                     = module.s3.bucket_name
  execution_role             = module.iam.execution_role_arn
  training_instances         = var.training_instances
  max_wait_time              = var.max_wait_time
  max_train_time             = var.max_train_time
  db_password_secret_arn     = module.secrets_manager.db_secret_arn
  mlops_bucket               = module.s3.mlops_bucket_name
  snowflake_mlops_secret_arn = module.secrets_manager.orm_secret_arn
  timestamp_lambda           = module.lambda.function_arn
}
```

The Terrraform project it’s easily configurable so only changing the variables named ”client” and ”project” (already exiting in AWS) it will be ready to work with any client. The specific servers used for composing the architecture and other resoruce details are defined inside the var files, and they can be adjusted according to the client’s budget.

6.2 Class diagram

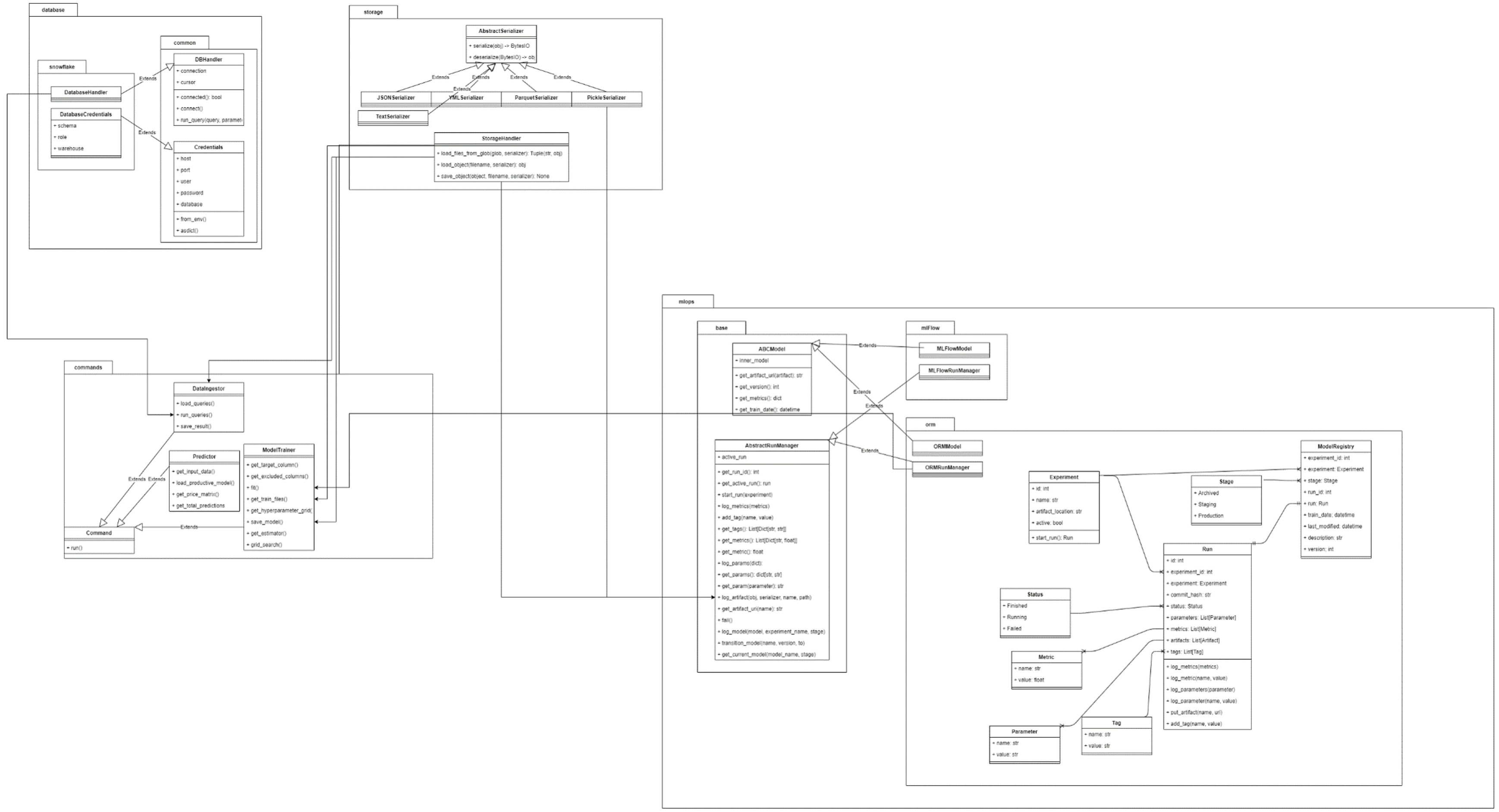


Figure 6.1: There are 4 principal groups of classes, each of them representing a structural block of the proposed solution: database, storage, commands and mlops

6.2.1 Commands

For each of the following commands, we provide a default implementation that in most of the cases should be enough to carry out the dynamic pricing process, and also an abstract interface defining the main methods and expected behaviour, so alternative implementations can be built extending those interfaces if needed.

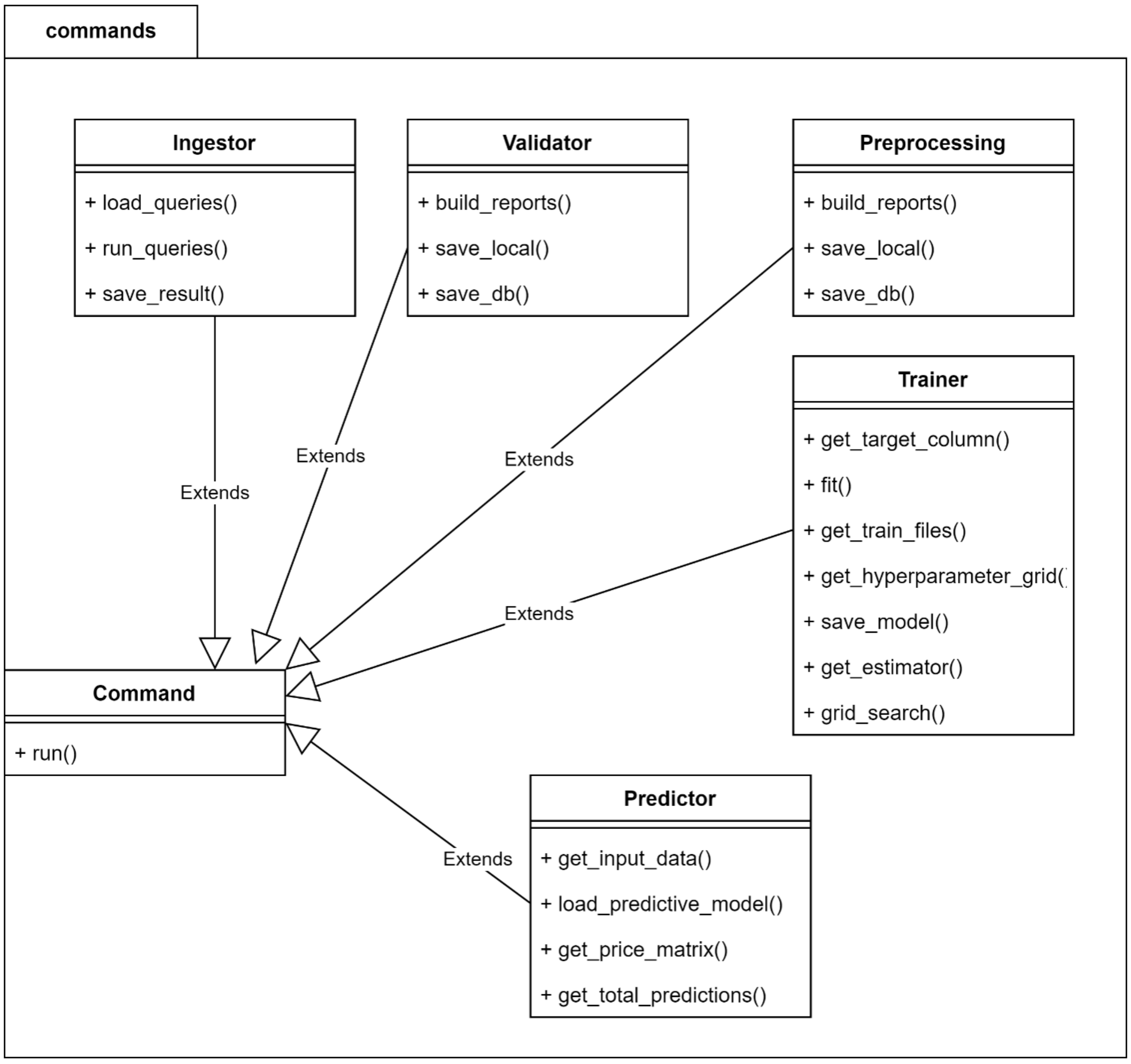


Figure 6.2: Command classes

Commands carry out the principal tasks that compose the main pipeline.
The DataIngestor retrieves the required data for the training and the prediction.
The DataValidator validates the ingested data end checks its integrity and corrrectness
The DataPreprocessing module carries out the specified preprocessing and leaves the input data ready for the training
The ModelTrainer carries out the training of the models and the fine tunning of the hyper-parameters.

The Predictor loads the information from the previous two modules and gets the optimal markdown process based on predictions from the trained models.

6.2.2 Database

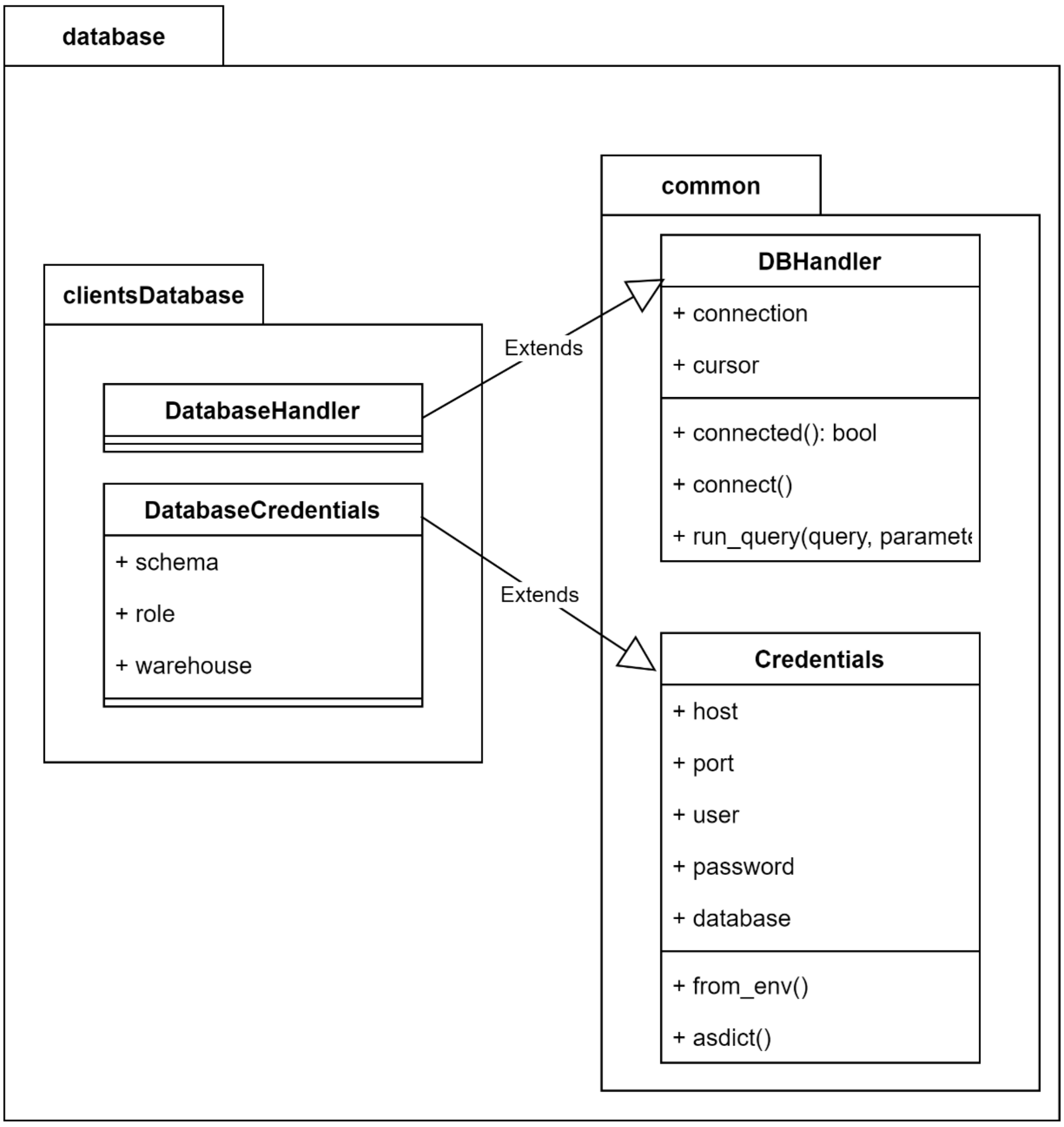
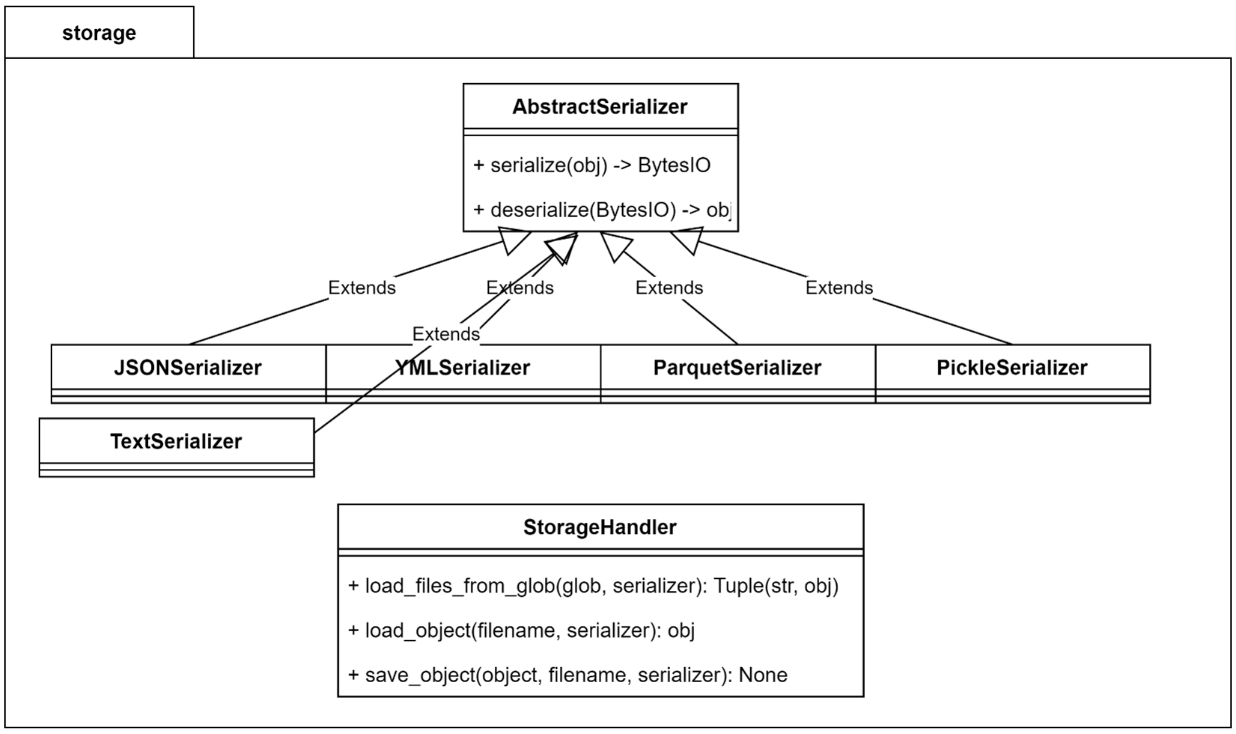


Figure 6.3: DB classes

The database module manages the data flow between the client's storage and our infrastructure



storage
AbstractSerializer
+ serialize(obj) -> BytesIO
+ deserialize(BytesIO) -> obj
Extends
Extends
Extends
Extends
Extends
JSONSerializer
YMLSerializer
ParquetSerializer
PickleSerializer
TextSerializer
StorageHandler
+ load_files_from_glob(glob, serializer): Tuple(str, obj)
+ load_object(filename, serializer): obj
+ save_object(object, filename, serializer): None

Figure 6.4: Storage classes

6.2.3 Storage

The storage module allows the storage of different data (historic data from the client, trained models) and the access to this data from other parts of the application (as the predictor or the simulation environment)

6.2.4 Models

Both ORM and MlOps modules are designed to store the metrics obtained and the params used during the training, to have a registry of the performance and characteristics of the different models and params combinations. They are two different implementations of the interface defined for the model registry
Internally, the registry is composed of several different classes that facilitate to identify key aspects of each concrete experiment

6.3 Persistence layer

6.3.1 DBHandler

The DBHandler interface provides the required methods to be implemented according to the client's database

```
class DBHandler(ABC):
```

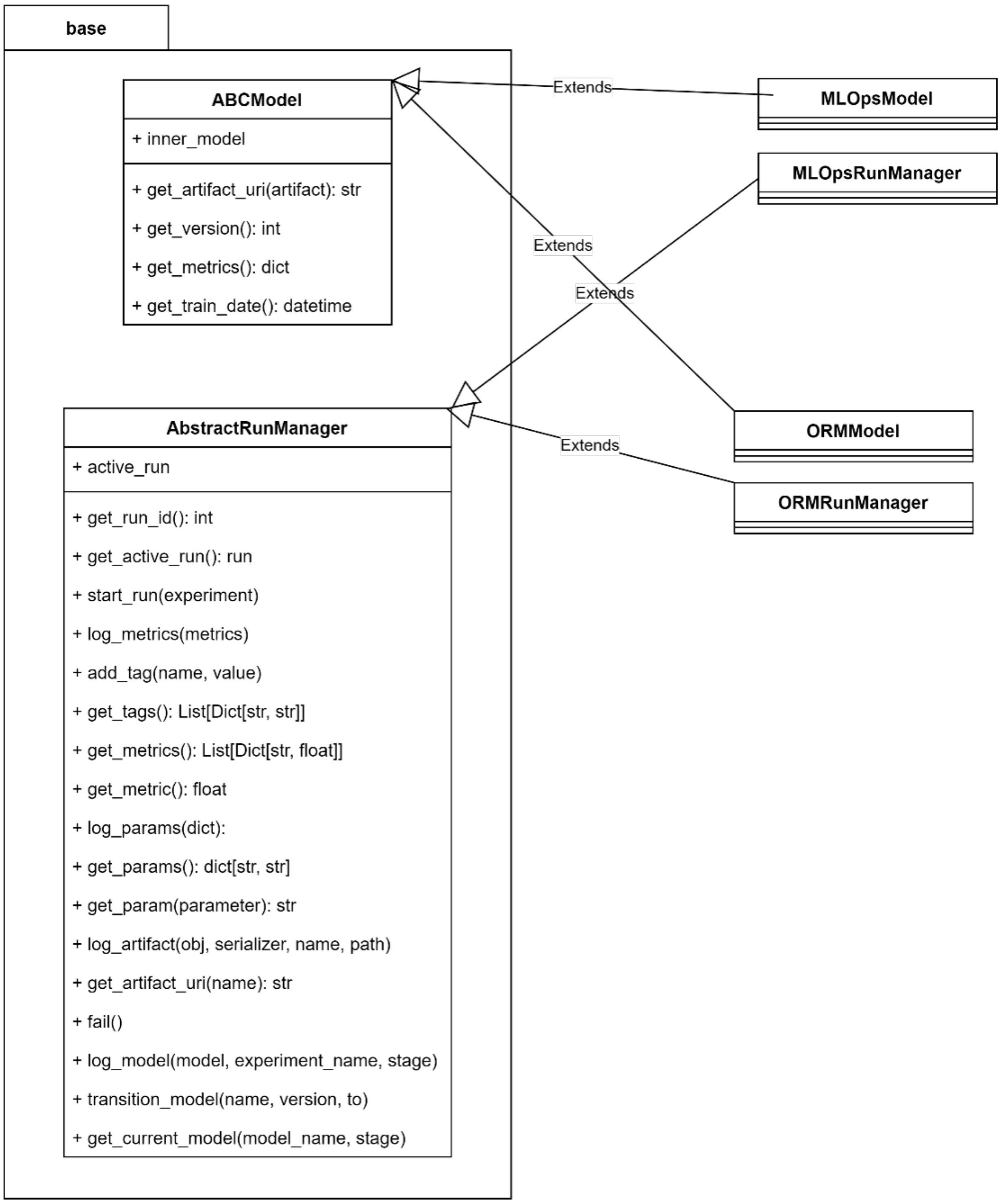

base
ABCModel
+ inner_model
+ get_artifact_uri(artifact): str
+ get_version(): int
+ get_metrics(): dict
+ get_train_date(): datetime
AbstractRunManager
+ active_run
+ get_run_id(): int
+ get_active_run(): run
+ start_run(experiment)
+ log_metrics(metrics)
+ add_tag(name, value)
+ get_tags(): List[Dict[str, str]]
+ get_metrics(): List[Dict[str, float]]
+ get_metric(): float
+ log_params(dict):
+ get_params(): dict[str, str]
+ get_param(parameter): str
+ log_artifact(obj, serializer, name, path)
+ get_artifact_uri(name): str
+ fail()
+ log_model(model, experiment_name, stage)
+ transition_model(name, version, to)
+ get_current_model(model_name, stage)
Extends
Extends
Extends
Extends
MLOpsModel
MLOpsRunManager
ORMModel
ORMRunManager

Figure 6.5: Base model classes

```
Credentials = DBCredentials

def __init__(self, from_env=False):
    if from_env:
        self.credentials = self.Credentials.from_env()
    self.connection = None
```

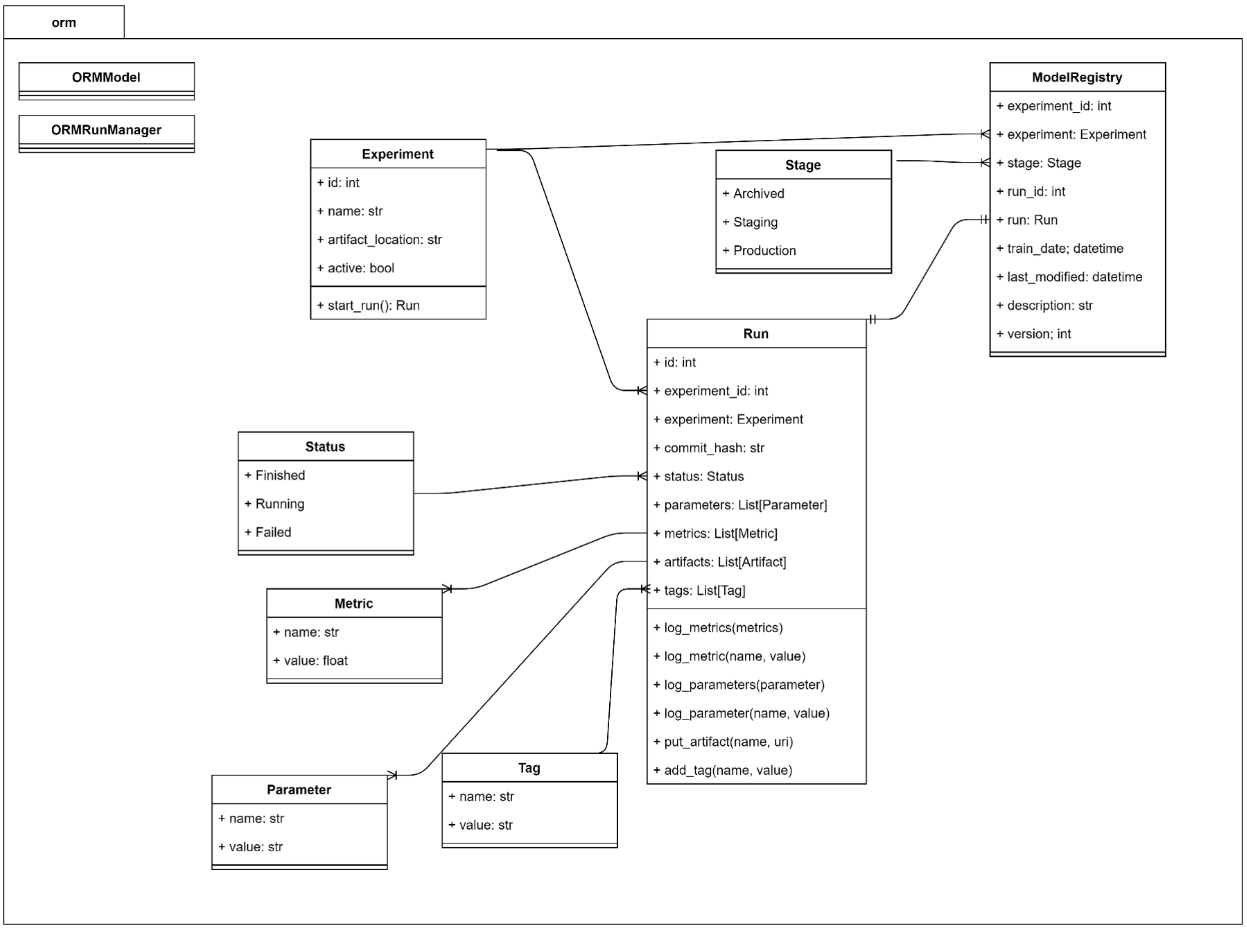


Figure 6.6: Internal model classes

```
    @property
    def connected(self):
        return not self.connection is None

    def connect(self):
        ...

    def execute_script(self, query):
        ...

    def execute(self, query, params):
        ...

    def run_pandas_script(self, script, index_ret):
        ...

    def run_pandas_query(self, query, **params):
        ...

    def run_query(self, query, **params):
        ...

    def save_pandas_df(self, df, table):
        ...
```

6.3.2 Storage handler

The module in charge of storing data read from the database and other data, such as the trained model artifacts, is the StorageHandler. The implementation of this module consists of three main methods: two for loading object (single or multiple files) and another one for saving an object to a file.

```
class AbstractStorageHandler(ABC):
    @classmethod
    def load_files_from_glob(cls, glob_regex, serializer) -> List[Tuple[str, str]]:
        ...

    @classmethod
    def load_object(cls, filename, serializer):
        ...

    @classmethod
    def save_object(cls, obj, filename, serializer):
        ...
```

These methods receive a Serializer, which is not more than a class with methods to read from a given format.

6.4 Data ingestion

The functioning of this module is simple. It is initialized in either *train* or *predict* mode, which determines which queries will be loaded. After that there is a method for loading the required queries and another method for executing them. After the queries are executed, the results are stored via the StorageHandler module inside a *.parquet* file. The DBHandler used for ingesting the data mst be specified through the ingestor's constructor

```
from abc import ABC
from .command import Command
from dynamic_pricing.persistence.storage.base.handler import AbstractStorageHandler
from dynamic_pricing.persistence.database.base.handler import DBHandler


class Ingestor(Command, ABC):
    def __init__(
        self,
        db_handler_cls: DBHandler,
        storage_handler_cls: AbstractStorageHandler,
        query_list,
        groupby_columns,
        ingest_queries_dir,
        output_path,
    ):
        self.storage_handler = storage_handler_cls
        self.ingest_queries_dir = ingest_queries_dir
        self.query_list = query_list
        self.groupby_columns = groupby_columns
        self.output_path = output_path
        self.db_manager = db_handler_cls(from_env=True)
        self.loaded_queries = []
        self.result = None

    def load_queries(self):
        '''Loads queries from the ingest queries dir,
        (using the provided storage handler) and saves them
        inside the loaded_queries variable'''
        ...

    def run_queries(self):
        '''Runs the queries inside loaded_queries'''
        ...

    def save_result(self):
        '''Saves the result of running the queries'''
        ...
    def run(self):
        self.load_queries()
        self.run_queries()
        self.save_result()
```

6.5 Data validation

After the data ingestion is completed, a validation of the data is carried on. It checks if the values of the different tables and columns are in the range they should be or are the type they are supposed to be. These validations are specified within the constructor of the project and will be very different depending on the client using the tool and their database. This module generates a report containing the results of the performed validation.

```
class Validator(Command, ABC):
    def __init__(self, report_config, report_destiny, db_handler_cls: DBHandler):
        self.report_config = report_config
        self.report_destiny = report_destiny
        self.db_handler_cls = db_handler_cls
        self.reports = dict()

    def build_reports(self):
        '''Builds a report checking the integrity of the data'''
        ...

    def save_local(self):
        '''Saves the generated reports to the report_destinys'''
        ...

    def save_db(self):
        '''Saves the generated reports to a database'''
        ...

    def run(self):
        self.build_reports()
        self.save_local()
        self.save_db()
```

Additionally, there is a preprocessing class that performs some transformation just before the training or the prediction steps. It usually one-hot-encodes categorical values, as our model only accepts numerical ones.

6.6 Data preprocessing

The preprocessing module performs the transformations provided (should extend the preprocessor interface) and leaves the data ready for the training step

```
class Preprocessing(Command, ABC):
def __init__(
    self,
    preprocessors: List[PreprocessorABC],
    input_path,
    output_path,
    storage_handler_cls: AbstractStorageHandler,
):
    self.preprocessors = preprocessors
    self.input_path = input_path
    self.output_path = output_path
    self.storage_handler = storage_handler_cls
    self.train_files = []
    self.processed_files = []

def get_train_files():
    '''Retrieves train files from input_paths
    (using the provided storage handler)'''
    ...

def preprocess_files(self):
    '''Applies the provided preprocessors'''
    ...

def save_result(self):
    '''Saves the preprocessed data ready for trainning
    (into the output path)'''
    ...

def run(self):
    self.get_train_files()
```

```
        self.preprocess_files()
        self.save_result()

class PreprocessorABC(PipelineComposerABC):
    name: str = None
    pipeline: Union[Pipeline, ColumnTransformer, TransformedTargetRegressor] = None

    def __init__(self, name=None):
        self.name = name
        self.build()

    def build(self, *args, **kwargs):
        pass

    def __call__(self):
        return self.pipeline

    def fit(self, *args, **kwargs):
        return self.pipeline.fit(*args, **kwargs)

    def fit_transform(self, X, *args, **kwargs):
        return self.pipeline.fit_transform(X, *args, **kwargs)
```

6.7 Model selection and training

The ModelTrainer class implements the training of the model, which is an instance of a *LGBMRegressor* adapted to the features fed to it.

```
class Trainer(Command, ABC):
    def __init__(
        self,
        estimator,
        validator_cls,
        run_manager_cls: AbstractRunManager,
        storage_handler_cls: AbstractStorageHandler,
        column_config,
        fit_config,
        input_dir,
        checkpoint_dir,
        experiment_prefix,
        hyperparameter_path,
    ):
        self.estimator = estimator
        self.validator_cls = validator_cls
        self.run_manager = run_manager_cls()
        self.storage_handler = storage_handler_cls
        self.column_config = column_config
        self.input_dir = input_dir
        self.checkpoint_dir = checkpoint_dir
        self.train_files = []
        self.experiment_prefix = experiment_prefix
        self.hyperparameter_path = hyperparameter_path
        self.fit_config = fit_config
        self.target_column = self.column_config["target_column"]
        self.excluded_columns = self.column_config.get("excluded_columns")
        self.included_columns = self.column_config.get("included_columns")
        self.cv_group_columns = self.column_config["cv"].get("group_columns")
        self.cv_n = self.column_config["cv"].get("n")
        self.gs_scoring = self.column_config["cv"].get("scoring")


    def get_train_files():
        '''Retrieves train files from input_dir
        (using the provided storage handler)'''
        ...

    def get_hyperparameter_grid(self):
        '''Retrieves a hyperparameter grid from hyperparameter_path'''
        ...

    def fit(self):
        '''Fits a model to predict the target column using the
        rest of the column configuration and the hyperparameter grid.
        Saves the resulting model into the checkpoint dir'''
        ...


    def run(self):
        self.get_train_files()
        self.get_hyperparameter_grid()
        self.fit()
```

The model trainer tries to predicts the target column defined in the column configuration. The columns included and excluded for training the model and other parameters needed for the training are also defined in that dictionary.

The training core step performs a grid search and trains the regressor with different combinations of hyperparameters to find the best combination of them.

The models can be fine tuned every time new data is available when executing in a real scenario. Also, the predictions are made for a closer horizon as it approaches the end of the sales period, so they are expected to be more accurate.

6.8 Model registry

The interface *AbstractRunManager* defines the methods for registering data about the different executions of a model. The one that achieves better performance is going to get the ”Production” tag, and the one that will be retrieved during the prediciton step.

```
class AbstractRunManager(ABC):
def __init__(self):
    self.active_run = None

def get_run_id(self):
    """ Returns the id of the current active run  """
    ...

def get_active_run(self, model_name, run_type):
    """ Returns the current active run """
    ...

def start_run(self, model_name):
    """ Starts a new run for the Experiment {model_name} """
    ...

def log_metrics(self, metrics: Dict[str, float]):
    """  Logs a series of metrics for the current run  """
    ...
```

```
def add_tag(self, tag_name: str, tag: str):
    """ Tags the current run with a key value pair """
    ...

def get_tags(self) -> List[Dict[str, str]]:
    """ Retrieves the tags for the current run """
    ...

def get_metrics(self) -> List[Dict[str, str]]:
    """ Retrieves the metrics for the current run """
    ...

def get_metric( self, name: str) -> Union[float, None]:
    """ Returns the value of a given metric, or none if not exists """
    ...

def log_params(self, params: Dict[str, str]):
    """ Logs parameters in the current run """
    ...

def get_params(self) -> List[Dict[str, str]]:
    """ Retrieves the parameters of the current run """
    ...

def get_param(self, name) -> Union[str, None]:
    """ Retrieves a parameter or None if not exists """
    ...

def log_artifact( self, artifact, serializer, name, path=""):
    """ Logs an artifact for the current run """
    ...

def get_artifact_uri(self, name: str) -> str:
    """ Returns the URI of a given artifact for the current run """
    ...

def fail(self):
    """  Marks the run as failed """
    ...
def log_model(self, model: Any, name: AnyStr, stage="Staging") -> ABCModel:
    """ Registers the model in the model registry, associated with the current run,
    additionnally, it uploads {model} as an artifact with name model/model.pkl  """
    ...

def transition_model(self, name, version, to):
    ...

def get_current_model(self, model_name, stage="Production" ):
    ...
```

6.9 Predictor

The predictor module, as its name states, is in charge of predicting the evolution of the demand through the sales season, using for that the individual predictions obtained from the trained model. It evaluates how the market will evolve and chooses the best alternative between the possible ones.

For implementing this evaluation procedure, we compute the results for all different paths, starting from the combinatory matrix computed from the number of different level of discounts. Before starting computing each possible case, some criteria can be enforced and the combinations not complying those criteria are removed. For every legal combination, the optimizer uses the predictions of the trained model and simulates the sales of each week, along with the other sales dependant variables (margin, revenue...) and other dynamic variables which computation can be provided thanks to the *OperationParser* During the process, a continuous prune is carried to remove combinations whose results violate other bussiness rules (like discounts going backwards), so only the potentially successful combinations are computed. We enrich this implementation with the already implemented cache mechanism of python.

This approach is similar to the *Backtracking* technique, which can be defined as a general algorithm that considers searching every possible combination in order to solve a computational problem

In our case, our algorithm performs its computation forward, but pruning the illegal branches before developing them. Once all the legal combinations are obtained, a maximum search can be performed, this can be simply the highest benefit or another heuristic chosen by the client. The function whose maximum will be searched is provided in the predictor's constructor.

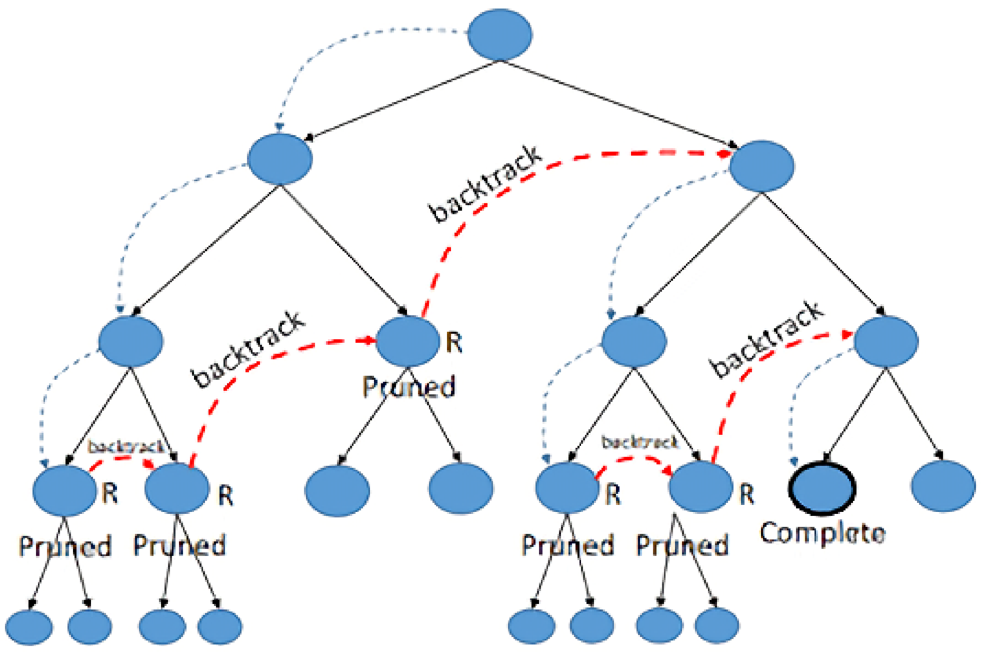


Figure 6.7: Backtracking scheme

Through the predictor's constructor, it can be set the first week of sales to start predicting the demand, the last week of the season, the discount levels and their respective real percentages, the week from which the discount should start being optimized, dynamic variables and other configurations.

```
class Predictor(Command, ABC):
    def __init__(
        self,
        run_manager_cls: AbstractRunManager,
        db_handler_cls: DBHandler,
        storage_handler_cls: AbstractStorageHandler,
        optimizer: AbstractOptimizer,
        parser: AbstractOperationParser,
        input_path,
        optim_folder,
        optimize_function,
        config,
    ):
        self.input_data = []
        self.input_path = input_path
        self.run_manager = run_manager_cls()
        self.storage_handler = storage_handler_cls
        self.folder = optim_folder
        self.db_handler = db_handler_cls(from_env=True)
        self.optimize = optimize_function
        self.optimizer = optimizer
        self.parser = parser
        self.config = config
        self.principal_columns = config["principal_columns"]
        self.experiment_prefix = config["experiment_prefix"]
        self.output_table = config["output_table"]

    def get_input_data(self):
        '''Gets the data needed to predict from the
        input_path (using the provided storage handler)'''
        ...

    def load_productive_model(self, hyerarchy_name):
        '''Loads the stored productive model belonging to the specfied hyerarchy
        (Can use the provided run_manager to retrieve run configurations)'''
        ...

    def run(self):
        '''Prediction and optimization of the desired target feature, using the Optimiz
        ...



class AbstractOptimizer(ABC):
    @staticmethod
    def load_dynamic_columns(config, operation_parser):
        ...

    @classmethod
    def get_predictions(cls, data, model, operation_parser, config):
        ...

    @cache
    def compute_case(data, case, operations, model, included_columns, config, current_w
        ...

    @classmethod
    def get_discount_matrix(cls, levels: Iterable, steps: int, criteria_equality=None,
        ...
```

6.9.1 Operation Parser

One of the requirements of the project was that the optimizer should take into account additional features that should be configurable via configuration file. This is possible thanks to the class OperationParser. This class is able to parse a configuration file and extract the desired operation and the arguments involved (features or not), and perform that computation, returning it under the specified name (also from the configuration file).

```
class OperationParser:

@staticmethod
def substract(df, c1, c2):
    return df.iloc[len(df.index) - 1][c1] - df.iloc[len(df.index) - 1][c2]

@staticmethod
def lag(df, column, level=1):
    level = int(level)
    return df.iloc[len(df.index) - 1 - level][column]

...

@classmethod
def parse(cls, op_str: str):
    arg_array = op_str.split(".")
    column_name = arg_array[0]
    operation = arg_array[1]
    arg_1 = arg_array[2]
    arg_2 = arg_array[3]

    return [column_name, getattr(cls, operation), arg_1, arg_2]
```

Using this class, different operations can be specified and new features can be computed from already existing ones. For example:

```
- NUM_DISCOUNT_PERC_DELTA_0_1.delta_lag.NUM_DISCOUNT_PERC.1
- NUM_UNITS_SOLD_LAG1.lag.NUM_UNITS_SOLD.1
- NUM_UNITS_SOLD_LAG2.lag.NUM_UNITS_SOLD.2
- NUM_UNITS_SOLD_DELTA_1_2.substract.NUM_UNITS_SOLD_LAG1.NUM_UNITS_SOLD_LAG2
- NUM_STOCK_LAG1.lag.NUM_STOCK.1
- NUM_STOCK_COVERAGE.divide.NUM_STOCK.NUM_UNITS_SOLD_LAG1
```

Chapter 7

Concrete use case: FashionRetail

The defined system was implemented for a specific client which name we'll keep anonymized. From now on, we will refer to this client, a Spanish important fashion retailer, as FashionRetail.

7.1 Initial assumptions

- During the first project phase, countries and product families to work in will be defined via collaboration between FashionRetail and SDG. The sales prediction model will work to a subfamily, week and country level.

- It will be necessary to work into *mirror references* to grow the sales history database. We expect our products to behave in a similar way to their mirror reference, a product from a previous season which shares several characteristics. As every season almost the catalog is renewed, these references will help to imagine how the current products would have been sold during previous seasons or years.

7.2 Data mapping

After an auditory with the client the origins of the required data are determined and validated as it can be seen in the following figure

The groups are generated to the extent of subfamily and independently from the campaign of each model-color. Mirror references are also assigned and other historic data is introduced in our data base. All this preprocessed data is stored into an Snowflake database, which constitutes one of the principal modules of the architecture.

7.2.1 Data Ingestion

Data is retrieved from the Snowflake database through a specific Snowflake implementation of our defined DBHandler interface and the data needed for training and/or predicting is stored as parquet files inside the correspondent folder.

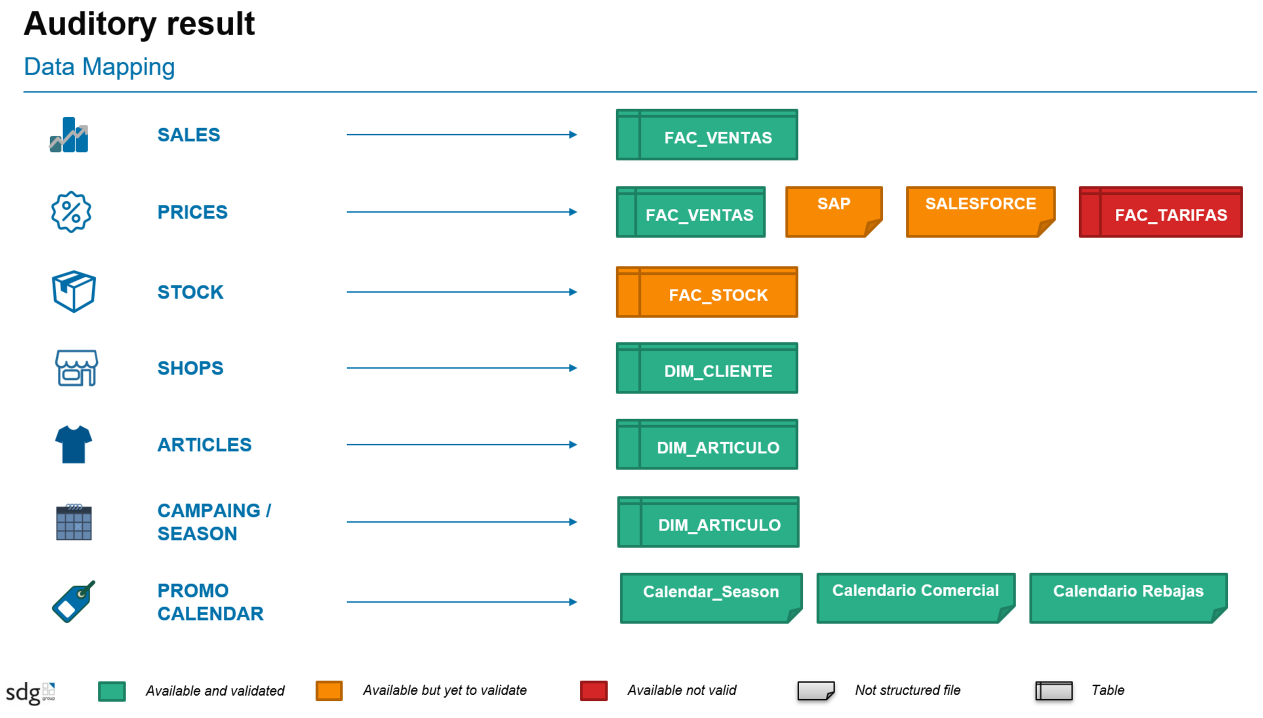


Figure 7.1: Origin tables for the required data and their status

7.2.1.1 Data quality

The data quality module in this case performs simple checks about the data integrity, mainly checking if the numerical values are indeed number and if they are inside the range they are supposed to be.

Models design:

- **1 model for each family/subfamily/mirror reference:** During the modeling phase it will be determined which is the best hierarchy
- Each model inside a subfamily will be able to be determined to a model-color
- Retail and e-commerce sales will be processed jointly

7.3 Prediction

7.3.1 Price and discount

Price Price in physical stores and web will identical and a new price will be assigned every Thursday. Prices will be rounded to one of the the digits (.49 or .99).

Discount

- **Granularity:** The level of detail when assigning discount will be **model-color and week**
- **Discount range:** Prices can have an applied discount within the interval **20-50%** and steps of **5%**
- **Private sales:** The proposed price for the private sales and the first week of discounts will be the same. This parameter (number of weeks when the discount level must be constant) can be configured.
- **Price changes:** During the discount weeks the discount can be maintained or increased, but never decreased
- The proposal of price will be done in order to maximize the total income once finalized the discount season. The duration of it can be configured.
- It will be possible to specify a condition about the minimum stock surplus

Cross validation Our client requires a cross-validation system that allows the training of models using a temporal window ranging from a given year to a total of N given years. It should also allow the sampling of random combinations according to different combinations of categories.

7.3.2 Optimization

The objective function the client wants to optimize is the total income at the end of the sales period, which follows the equation:

$$Income = (SalePrice * UnitsSold) + (ResidualValue * RemainingStock) \quad (7.1)$$

In this specific case the predictor modules optimizes this function while also making sure some other restrictions are met. This criteria are given to the predictor through its constructor and only the combinations that meet the specified criteria are considered valid.

First of all, all possible combination are computed using a backtracking algorithm. After that, combinations are asked to comply the additional criteria. Once the ones that do not comply them are removed, the best combination is chosen, also following an stability criteria, so the solution is inside a stable zone and its proximities keep being good solutions.

7.4 Configuration

For this specific client, all configurable aspects are gathered into a configuration file which is configured in the following way, specifying aspects about the ingestor, the data validation, the preprocessing, the model trainer and the predictor. This file is parsed into python objects and fed to the defined interfaces and implementations to configure the project.

```
train:
  groupby_columns:
    - ID_COUNTRY
    - ID_SUBFAMILY
  models:
    - input: null
      applies_to:
        - [ES, WWT63]
        - [ES, WWT64]
    - input:
        - [IT, WWV12]
        - [IT, WWV15]
      applies_to:
        - [IT, WWV12]
        - [IT, WWV15]
ingestor:
  groupby_columns:
    - ID_COUNTRY
    - ID_SUBFAMILY
  queries-train:
    - &step1
      file: T_COUNTRY_SUBFAMILY.sql
      parameters: {}
    - &step2
      file: T_VENTAS.sql
      parameters:
        HIST_START_DATE: 20170101
        CHANNELS: "'[\"EC\",\"RT\"]'"
        IDS_VENTA: "'[\"Sell-out\",\"Sell-Out\"]'"
        TRANSACTION_TYPE: "'[\"Sell\"]'"
    - &step3
      file: T_MARKDOWN_WEEKLY.sql
      parameters:
        WEEK_BEGINNING: 4
        SEASON: 4
    - file: query_train.sql
      parameters:
        EXEC_CAMPAIGN: "'OI20'"
        SALES_START: -1
        TRAIN_IS_PROD: 0

  queries-predict:
    - *step1
    - *step2
    - *step
    - file: T_MARKDOWN_RESULTS
      parameters: {}
    - file: query_predict.sql
      parameters:
        EXEC_SEASON: "'OI20'"
        EXEC_WOY4_MARKDOWN: -2
        LAGS_DEPTH: 4
model_trainer:
  experiment_prefix: markdown
  target_column: NUM_UNITS_SOLD
  included_columns: &inccols
    - NUM_DISCOUNT_PERC
    - NUM_DISCOUNT_PERC_DELTA_0_1
    - DT_WOY4_MARKDOWN
    - NUM_STOCK_LAG1
    - NUM_UNITS_SOLD_LAG1
    - NUM_STOCK_COVERAGE
    - MR_CLUSTER_TS
    - IND_CHRISTMAS
    - ID_YEAR
  duplicate_data:
    cases: [-1]
    params:
      - [NUM_UNITS_SOLD, 2]
      - [NUM_UNITS_SOLD_LAG1, 2]
      - [NUM_UNITS_SOLD_LAG4, 2]
      - [NUM_UNITS_SOLD_DELTA_1_2, 2]
      - [NUM_STOCK_LAG1, 2]
  cv:
    n: 8
    group_columns:
      - ID_CAMPAIGN
      - DT_WOY4_MARKDOWN
    scoring: "neg_mean_absolute_error"
  hyperparameters: /cfg/hyperparams
  model:
    kwargs:
      categorical:
        ["MR_CLUSTER_TS", "IND_BLACK_FRIDAY", "IND_CHRISTMAS", "IND_MID_SEASON"]
      logcols: ["NUM_UNITS_SOLD_LAG1", "NUM_UNITS_SOLD_LAG4", "NUM_STOCK_LAG1"]
  fit:
    args: []
    kwargs:
      regressor__categorical_feature:
        ["MR_CLUSTER_TS", "IND_BLACK_FRIDAY", "IND_CHRISTMAS", "IND_MID_SEASON"]
predictor:
  GLOBAL: &global
    first_week: -2
    output_table: T_MARKDOWN_RESULTS
    experiment_prefix: markdown
    discount: &discount
      levels: [0.20, 0.30, 0.40, 0.50]
      steps: 9
      criteria_equality: [[0, 1]]
    principal_columns:
      price: PRICE
      sales: NUM_UNITS_SOLD
      week: DT_WOY4_MARKDOWN
      stock: NUM_STOCK
      discount_perc: NUM_DISCOUNT_PERC
      subfamily_id: ID_COLOR_MODEL
      residual_value: NUM_RESIDUAL_VALUE
      cost: NUM_COST
    included_columns: *inccols
    prediction_hierarchy_dynamic_columns:
      - NUM_DISCOUNT_PERC_DELTA_0_1.delta_lag.NUM_DISCOUNT_PERC.1
      - NUM_UNITS_SOLD_LAG1.lag.NUM_UNITS_SOLD.1
      - NUM_UNITS_SOLD_LAG2.lag.NUM_UNITS_SOLD.2
      - NUM_UNITS_SOLD_DELTA_1_2.substract.NUM_UNITS_SOLD_LAG1.NUM_UNITS_SOLD_LAG2
      - NUM_STOCK_LAG1.lag.NUM_STOCK.1
      - NUM_STOCK_COVERAGE.divide.NUM_STOCK.NUM_UNITS_SOLD_LAG1

    classification_hierarchy_dynamic_columns: []
    comparison_hierarchy_dynamic_columns: []
    target_function:
      function: REVENUE = NUM_UNITS_SOLD * ((PRICE * (1 - NUM_DISCOUNT_PERC)) - NUM_COS
      group:
        - ID_COUNTRY
        - ID_FAMILY
        - ID_SUBFAMILY
        - ID_COLOR_MODEL
        - ID_CAMPAIGN
      agg: sum
    score_function: simple_margin_score
```

7.5 Results

The developed framework was fed with real historic data from our client, containing information of several families and subfamilies and some hundreds models. The whole pipeline, including training and predicting, could be executed in about three hours, paralleling each family in a different server.

At first glance, the model seems to give worse results (sales) when compared with historical data during the same period. This is because the model tends to underestimate sales, and this error tends to accumulate if we run a complete simulation instead of predicting week after week

The fact that the model underestimates can be better noticed when we ask the model to predict the sales volume using the real discounts that were applied during the campaign.

If we compared the sales predicted using the real used discounts with the one obtained using the optimized discounts, the results end up being better (yellow line) in most of the cases

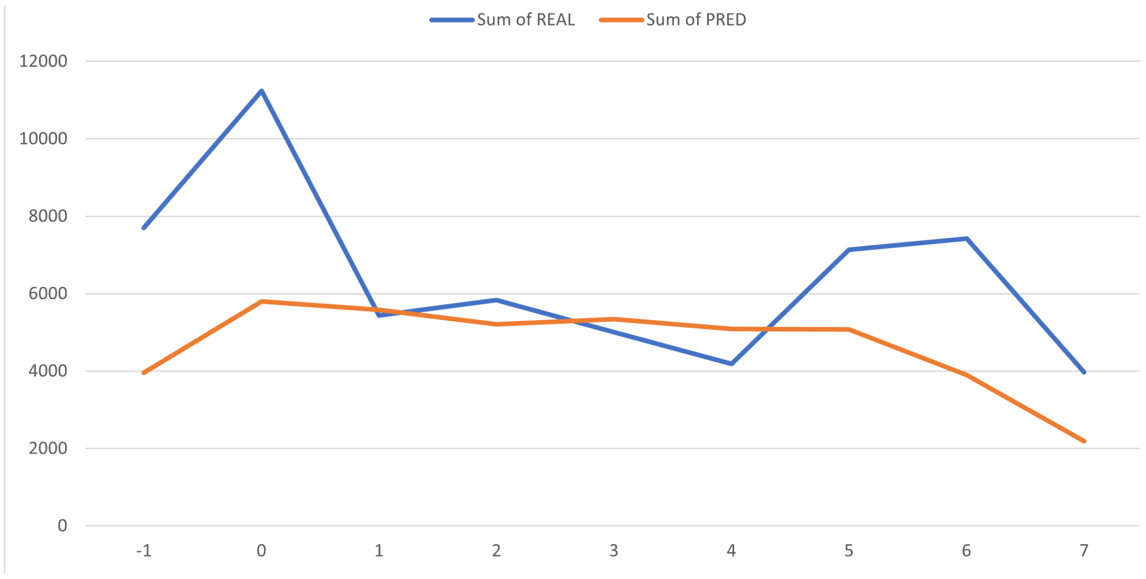


Figure 7.2: Real sales and predicted sales for country A

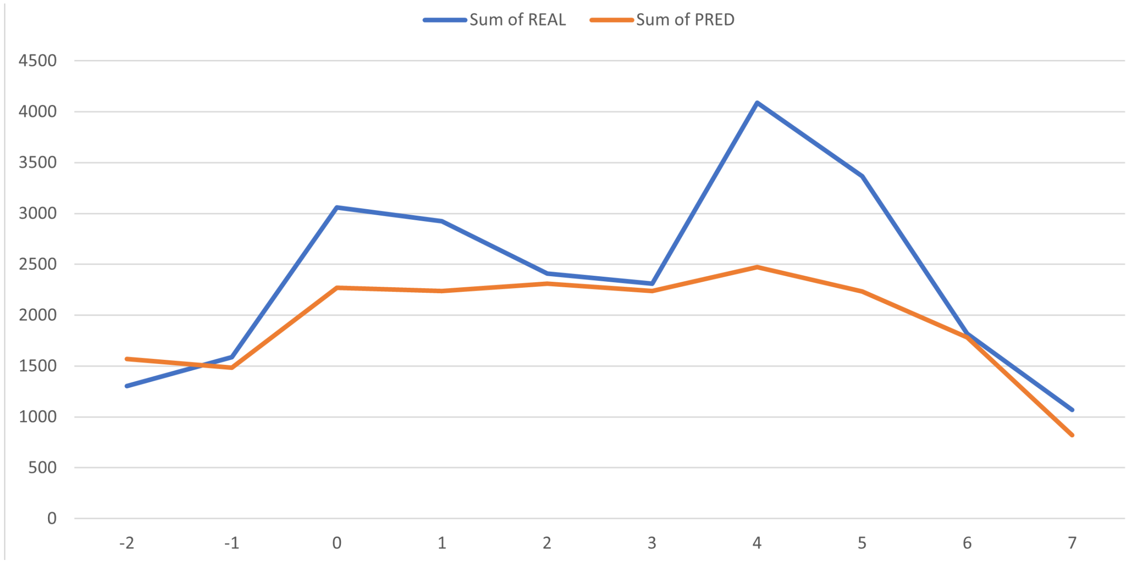


Figure 7.3: Real sales and predicted sales for country B

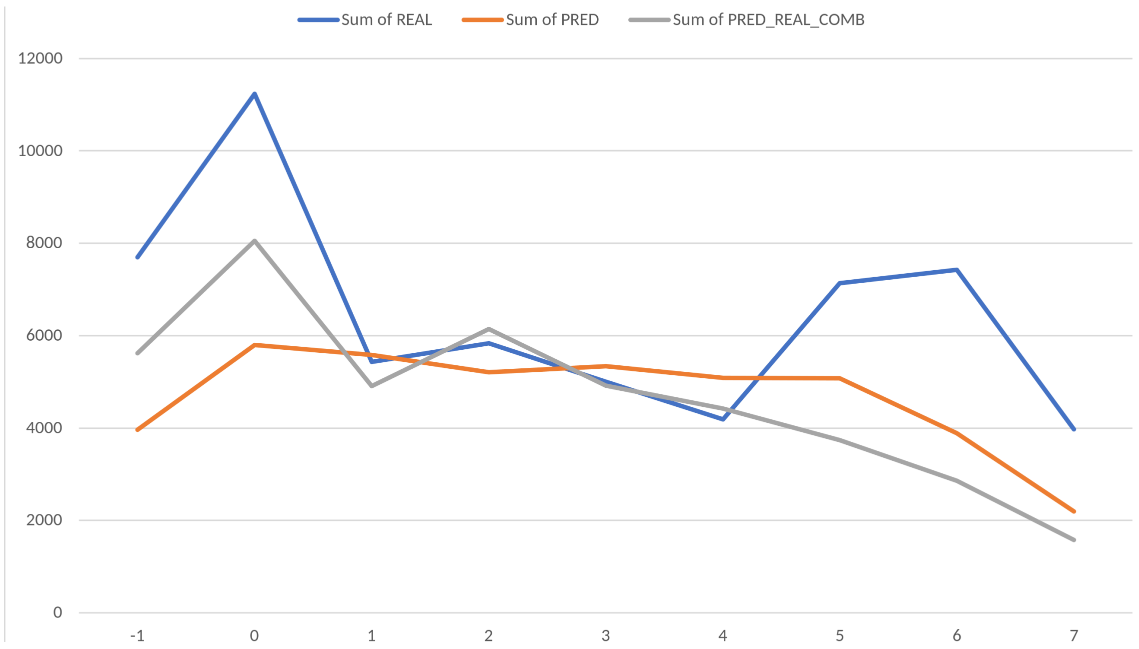


Figure 7.4: Real sales and predicted sales for country A

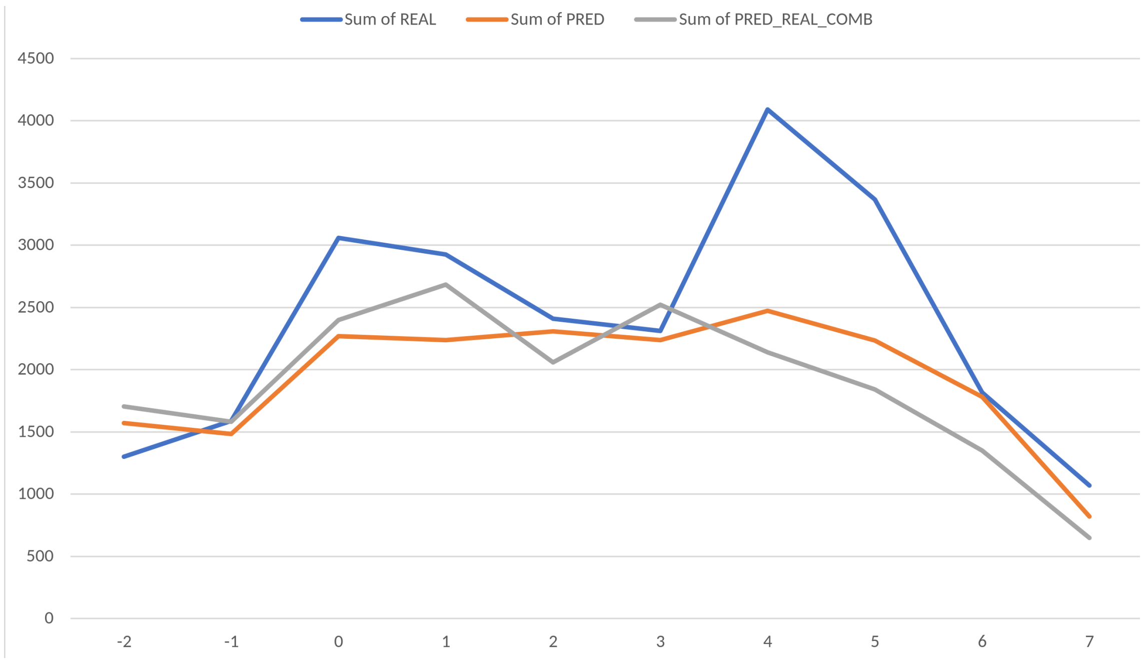


Figure 7.5: Real sales and predicted sales for country B

The described behaviour is better observed inside each subfamily

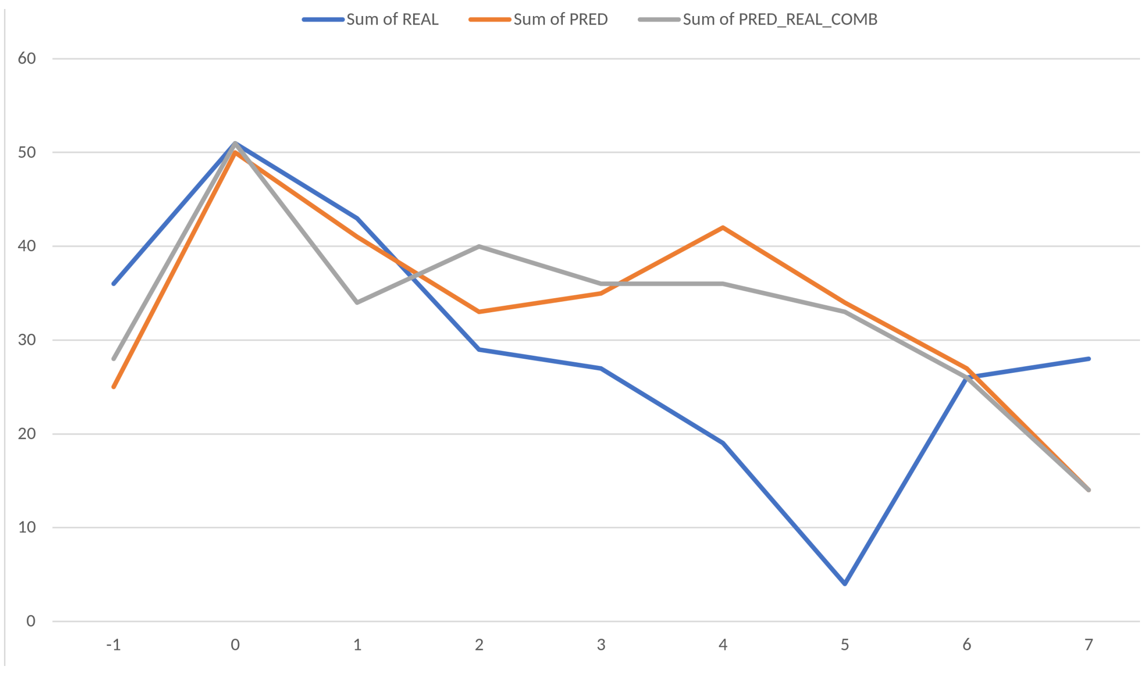


Figure 7.6: Real sales and predicted sales for subfamily A

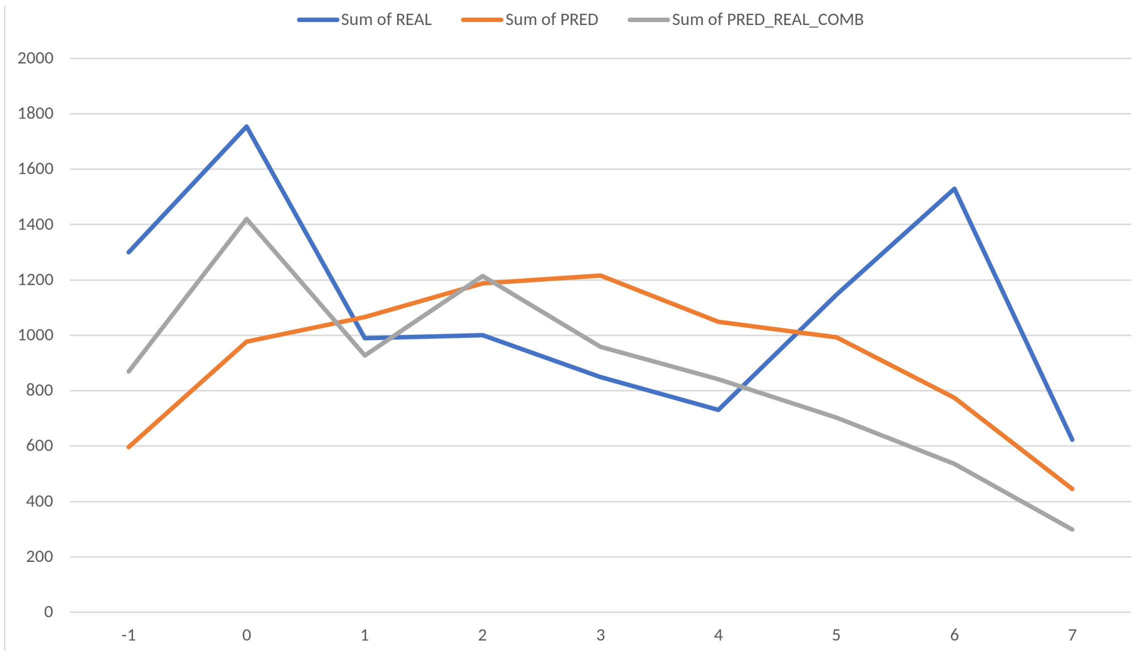


Figure 7.7: Real sales and predicted sales for subfamily B

Discounts are observed to behave different than historical data, overall starting lower but also climbing up faster. Combining this data with sales the company's revenue can be computed

Sum of predictions	Sum of predictions with real discounts
8176	**9169**
4329	**4517**
8506	7766
301	298
9825	9382
11205	11102
42342	42234

Table 7.1: Comparative between predictions for different product subfamilies

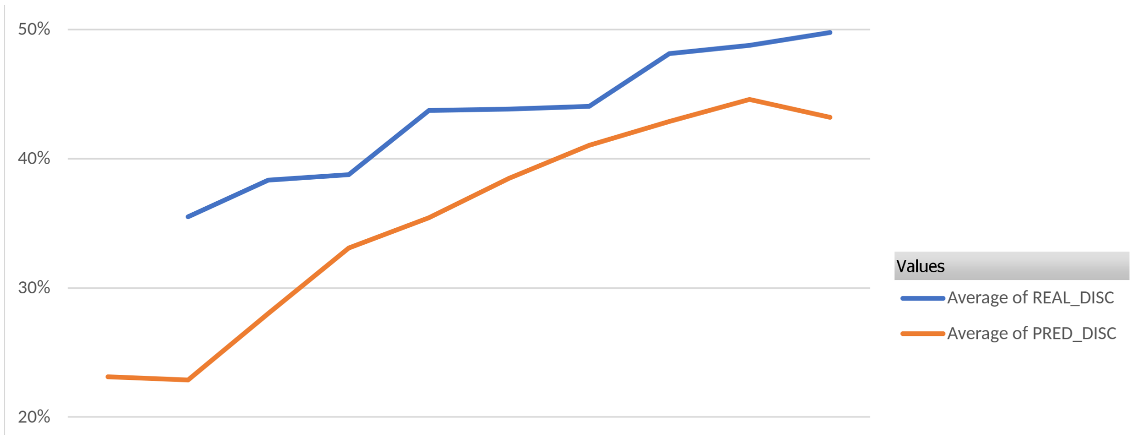


Figure 7.8: Real discounts and predicted discounts for country A

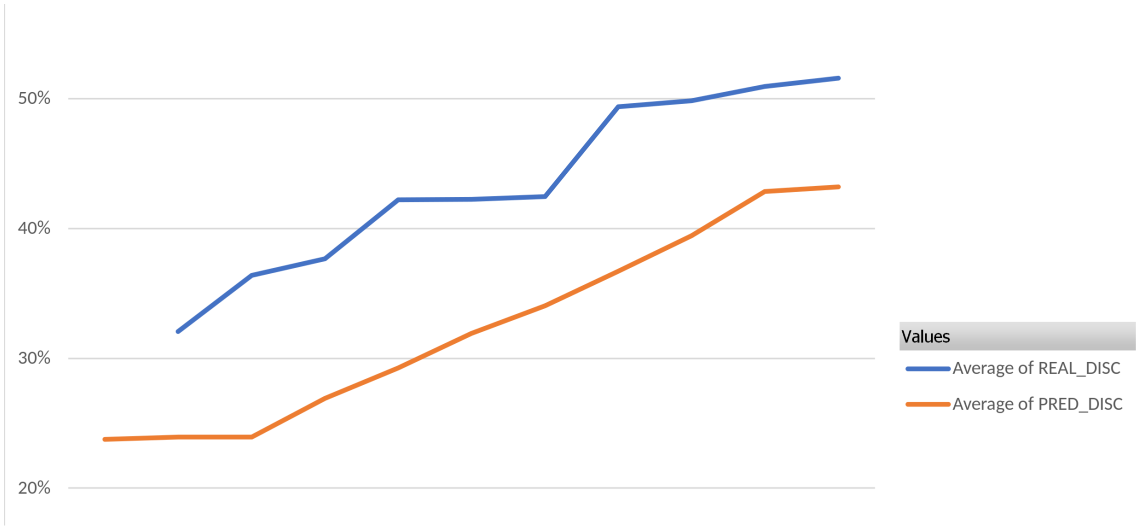


Figure 7.9: Real discounts and predicted discounts for country B

Results are better when we look at the generated revenue, since the difference between the predicted revenue and the revenue predicted with the real applied discounts is clear and indicates the model really maximizes profit and the difference with the real revenue is due to the underestimation and the error accumulation across the simulation.

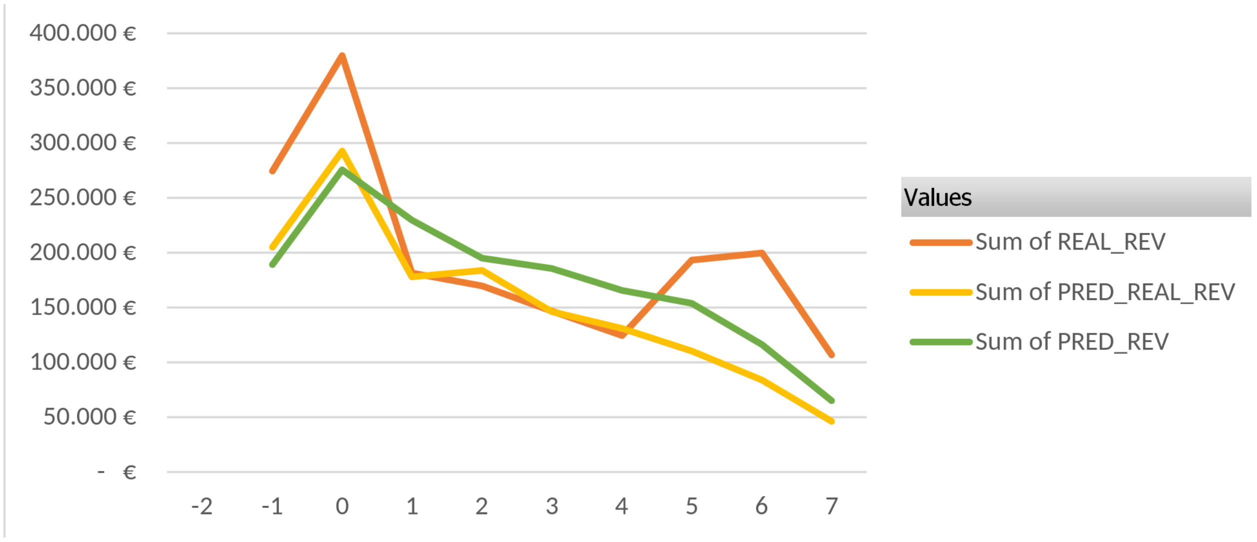


Figure 7.10: Real revenue and predicted revenue for country A

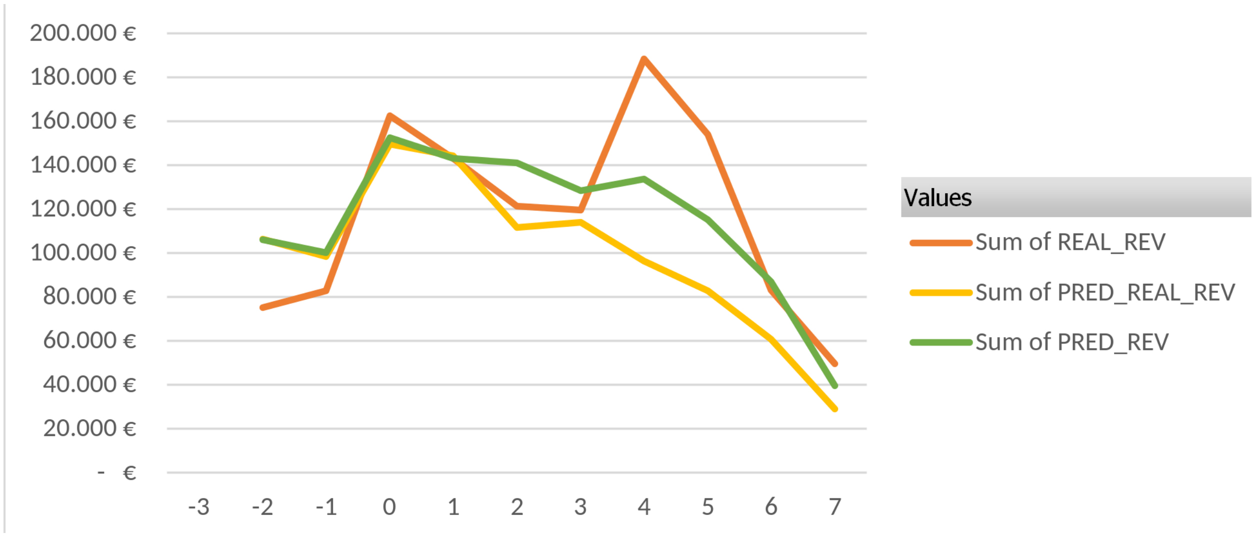


Figure 7.11: Real revenue and predicted revenue for country B

Chapter 8

Conclusions

The framework has successfully been used with a SDG's client which provided real historic data and the results obtained are coherent.

The design chosen and the architecture built for the project have been proven to work with real data from a real client, being able to make predictions on sales volume and using those predictions to find the best combination of discounts.

The predictions made from the trained *LightGBM* model usually subestimates future sales volume, as it can be seen when comparing the 2020 simulation with the real sales volume. For training the model data from 3 different years has been used, so this subestimation may become lighter as more real data is fed to the model. However, the model seems to improve the company's profit if predicted discounts are applied instead of the usual ones.The discordance between the real sales volume and the predicted ones is due to the model subestimating, which, indeed, is preferred to an overestimating model as results are expected to be better than simulations.

A real comparative would not be possible until the pilot is carried on during this season, when the predictor will be run each week, considering all past real data, including the real sales volume generated after the suggested discounts are applied, and allowing ourselves to measure the real revenue generated during the entire process. The results obtained when executing it this way, are expected to be better than our simulations.

Our framework has been proven to be easily adaptable to FashionRetail, and able to take into account every specific business requirement by only implementing a pair of interfaces and using the default implementations for carrying on the rest of the functionalities.

As future work, the framework could also be extended to be used with other cloud providers and not only AWS. That should not be too difficult as the deployment infrastructure is defined as *Terraform* code and is separated from the python code that defines the functionality. As long as the new infrastructure supports the execution of python code, that should be enough.

I would like to remark the difficulty to obtain exact and numeric results from solving this type of problems where so many factors interfere. There is a correlation between prices and sales volume, but factors like the actual tendencies, the country’s economy, competitor’s sales, changes in the design of the products compared with the previous season, pandemic episodes and much more things that are outside our control can influence the results. That said, our simulation can work as an ”ideal” baseline, which indeed tells us the goals of the project are accomplished.

Bibliography

[BFOS83] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and C. J. Stone. Classification and regression trees. 1983. 18

[BGD$^+$02] Christopher H. Brooks, Robert S. Gazzale, Rajarshi Das, Jeffrey O. Kephart, Jeffrey K. MacKie-Mason, and Edmund H. Durfee. Model selection in an information economy: Choosing what to learn. *Computational Intelligence*, 18(4):566–582, November 2002. Available from: `https://doi.org/10.1111/1467-8640.t01-2-00204`. 18

[BK21] Gah-Yi Ban and N. Bora Keskin. Personalized dynamic pricing with machine learning: High-dimensional features and heterogeneous elasticity. *Management Science*, 67(9):5549–5568, September 2021. Available from: `https://doi.org/10.1287/mnsc.2020.3680`. 17

[BZ15] Omar Besbes and Assaf Zeevi. On the (surprising) sufficiency of linear models for dynamic pricing with demand learning. *Management Science*, 61(4):723–739, April 2015. Available from: `https://doi.org/10.1287/mnsc.2014.2031`. 17

[CG18] Ningyuan Chen and Guillermo Gallego. A primal-dual learning algorithm for personalized dynamic pricing with an inventory constraint. *SSRN Electronic Journal*, 2018. Available from: `https://doi.org/10.2139/ssrn.3301153`. 17

[CLY$^+$12] Byung Do Chung, Jiahan Li, Tao Yao, Changhyun Kwon, and Terry L. Friesz. Demand learning and dynamic pricing under competition in a state-space framework. *IEEE Transactions on Engineering Management*, 59(2):240–249, May 2012. Available from: `https://doi.org/10.1109/tem.2011.2140323`. 18

[DBZ14] Arnoud V. Den Boer and Bert Zwart. Simultaneously learning and optimizing using controlled variance pricing. *Management Science*, 60, 03 2014. 17

[Fri01] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2001. 20

[FS96] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm, 1996. 19

[HM15] Benjamin R. Handel and Kanishka Misra. Robust new product pricing. *Marketing Science*, 34(6):864–881, November 2015. Available from: `https://doi.org/10.1287/mksc.2015.0914`. 17

[KMF$^+$17] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Proceedings of the 31st International Conference on Neural*

Information Processing Systems, NIPS'17, page 3149–3157, Red Hook, NY, USA, 2017. Curran Associates Inc. 21

[KUP03] Erich Kutschinski, Thomas Uthmann, and Daniel Polani. Learning competitive pricing strategies by multi-agent reinforcement learning. *Journal of Economic Dynamics and Control*, 27:2207–2218, 09 2003. 18

[KZ14] N. Bora Keskin and Assaf Zeevi. Dynamic pricing with an unknown demand model: Asymptotically optimal semi-myopic policies. *Operations Research*, 62(5):1142–1167, October 2014. Available from: `https://doi.org/10.1287/opre.2014.1294`. 17

[MMSW06] P.B. Mullen, C.K. Monson, K.D. Seppi, and S.C. Warnick. Particle swarm optimization in dynamic pricing. In *2006 IEEE International Conference on Evolutionary Computation*, pages 1232–1239, 2006. 18

[SKOC12] S. Shakya, M. Kern, G. Owusu, and C.M. Chin. Neural network demand models and evolutionary optimisers for dynamic pricing. *Knowledge-Based Systems*, 29:44–53, May 2012. Available from: `https://doi.org/10.1016/j.knosys.2011.06.023`. 18

[SOO09] Siddhartha Shakya, Fernando Oliveira, and Gilbert Owusu. Analysing the effect of demand uncertainty in dynamic pricing with EAs. In *Research and Development in Intelligent Systems XXV*, pages 77–90. Springer London, 2009. Available from: `https://doi.org/10.1007/978-1-84882-171-2_6`. 18

[VGEE14] María-Jesús Vázquez-Gallo, Macarena Estévez, and Santiago Egido. Active learning and dynamic pricing policies. *American Journal of Operations Research*, 04(02):90–100, 2014. Available from: `https://doi.org/10.4236/ajor.2014.42009`. 18

[XD09] Cathy H. Xia and Parijat Dube. Dynamic pricing in e-services under demand uncertainty. *Production and Operations Management*, 16(6):701–712, January 2009. Available from: `https://doi.org/10.1111/j.1937-5956.2007.tb00290.x`. 18