Trabajo de Fin de Grado

Universidad de Zaragoza

Departamento de Métodos Estadísticos

MODELOS DE PREDICCIÓN DE LA DEMANDA CON MÉTODOS DE APRENDIZAJE MÁQUINA

Curso 2023 - 2024

Autor:
Inés Serrano Mayor

Tutor:
Dr. José Tomás Alcalá Nalvaiz

Undergraduate Dissertation in Mathematics

Universidad de Zaragoza

Statistical Methods Area

DEMAND FORECASTING MODELS WITH MACHINE LEARNING TECHNIQUES

Academic year 2023 - 2024

Autor:
Inés Serrano Mayor

Tutor:
Dr. José Tomás Alcalá Nalvaiz

I would like to thank Tomás for his wise guidance, his time and help, and for sharing his knowledge with me.

It is also the moment to thank my family for being my main stimulus to get here and for supporting me permanently and unconditionally.

1 Summary

El Aprendizaje Automático (ML, por sus siglas en inglés), una rama de la Inteligencia Artificial (IA), es considerado una revolución. En este proyecto hemos querido desafiar a esta nueva herramienta tecnológica aplicándola a un caso real en el mundo de la empresa. La pregunta que ha motivado este trabajo es: ¿es posible crear una herramienta de ML para ayudar a las empresas a mejorar su previsión de la demanda?

Con el objetivo de responder a esta pregunta, la empresa española de la industria alimenticia *Chocolates Lacasa*, conocida por sus famosos productos de chocolate *Lacasitos* y *Conguitos*, nos ha proporcionado datos de sus previsiones y ventas. Sin embargo, la gran pregunta es: ¿por qué esta herramienta podría servir de gran utilidad?

En esta empresa la producción se realiza antes de conocer la demanda y, por lo tanto, deben prever las ventas con antelación. Además, sus productos presentan una fuerte estacionalidad ya que tienen el pico de ventas durante la época de Navidad. Asimismo, juegan con la vida útil del producto. Como resultado de estas variables, la previsión de la demanda es un gran desafío. Los errores en la previsión pueden provocar ventas perdidas, exceso de inventario o incluso clientes insatisfechos.

Con el fin de evitar estas situaciones o de, al menos, mitigarlas, una herramienta que mejorase la predicción de la demanda sería de gran ayuda. En este proyecto entrenamos un algoritmo de ML con el objetivo de que sea capaz de predecir la demanda con mayor precisión que los métodos actualmente utilizados para ello. Para llevar a cabo esta tarea, se nos ha proporcionado datos históricos de 100 productos de la empresa Chocolates Lacasa. Estos datos incluyen tanto la demanda prevista por Lacasa como las ventas reales, con ellos entrenamos nuestro algoritmo.

Se comienza con un examen exhaustivo de los datos, empleando algoritmos como K-Nearest Neighbors (KNN) y K-Means para evaluar la similitud en la distribución de datos entre diferentes productos. Esto nos ayuda a seleccionar los datos de manera inteligente para alimentar el algoritmo de predicción de ML.

Una vez limpiados y seleccionados los datos, utilizamos el algoritmo de ML *XGBoost* para la predicción. Este algoritmo pasa por un proceso de entrenamiento, minimizando errores de predicción mediante el ajuste de los hiperparémtros del algoritmo. Este proceso de entrenamiento se realiza para múltiples productos. Los resultados que se obtienen probablemente harán que una empresa reconsidere su forma de prever la demanda, buscando minimizar su error de predicción para ahorrar costos y otras implicaciones negativas.

En cuanto a la estructura de la memoria, esta contiene cinco capítulos, siendo el primero y el útimo la introducción y conclusiones respectivamente. Además, incluye varios apéndices al final que desarrollan con más profundidad algunos conceptos mencionados a lo largo de la memoria.

El segundo capítulo comienza proporcionando el contexto de la empresa y los datos que se utilizan a lo largo del proyecto. Además, en él se indica el impacto que puede llegar a tener una mala previsión de la demanda, resaltando la importancia que tiene hacer una buena predicción.

En el tercer capítulo se presentan las herramientas más relevantes que se han utilizado a lo largo del proyecto. Se comienza hablando de las series de tiempo debido a la naturaleza de nuestros datos. Se sigue con los métodos de *clustering*, *KNN* y *K-Means*, utilizados para dividir los datos según la similitud entre sus series de tiempo. Por último, se explica el algoritmo *XGBoost* que es la herramienta final que nos permite hacer predicciones futuras de la demanda. Para ello, se comienza explicando los árboles de decisión, seguido del algoritmo *Gradient Boosting*, para terminar explicando el *XGBoost*.

El cuarto capítulo detalla el proceso completo para un producto, abordando todas sus etapas. Incluye la selección del producto, la identificación de sus vecinos más cercanos, la preparación de los datos, el entrenamiento del algoritmo y la generación de predicciones con sus respectivos errores. Estas predicciones se comparan con los errores producidos por las predicciones de Lacasa.

Finalmente, se incluyen apéndices que profundizan en ciertos conceptos presentados a lo largo de la memoria. Estos incluyen una explicación del concepto de *Time Dynamic Warping* para medir la similitud entre series de tiempo y del *Silhouette Score*, que es un índice para evaluar el rendimiento de un método de *clustering* para un número de *clusters* dado. Además, se proporciona una explicación de los diferentes hiperparámetros de nuestro algoritmo *XGBoost*, el procedimiento del algoritmo de validación cruzada (*Cross Validation*) utilizado para entrenar el algoritmo *XGBoost*, y el código de *Python* utilizado en el proceso de predicción de la demanda futura para un producto.

Contents

2 Introduction

Machine learning is the future of every business. "Ten years ago, we struggled to find 10 machine learning-based business applications. Now we struggle to find 10 that don't use it", Alexander Linden, research vice president at Gartner (2020).

Machine learning (ML), a branch of Artificial Intelligence (AI), is considered a revolution. In this project we wanted to challenge this new technological tool in the business world. The question that has motivated this work is: is it possible to create a ML tool to help businesses improve the forecast of the demand?

With the aim of answering this question we have been provided with real-world data from *Chocolates Lacasa*, a Spanish company in the food industry known for its famous chocolate pieces *Lacasitos* and *Conguitos*. However, how could this tool be something useful?

In this company the production is made before knowing the demand and therefore, sales should be forecast. On top of this, their products present a strong seasonality since they have the peak sales during Christmas time. In addition, they play with product shelf life. As a result of these things, demand forecast is a major challenge. Forecasting errors leave them with lost sales, excess inventory or even dissatisfied clients.

To this end, in this project we will train a machine learning algorithm with the goal of creating the tool mentioned that is able to predict demand more accurately than current methods. To accomplish this task we have been provided with historical data of 100 products of Chocolates Lacasa company. These data includes both Lacasa's predicted demand and the actual sales. We will use it to train our algorithm.

Our approach will begin with a thorough examination of the data, employing algorithms such as K-Nearest Neighbors (K-NN) and K-Means to assess the similarity in data distribution across different products. This will help us choose data smartly to feed the prediction algorithm.

Once the data is curated and cleaned, we will use the XGBoost algorithm for prediction. Our objective is to optimize the training process, minimizing prediction errors by fine-tuning the algorithm parameters. We will conduct this process across multiple products to draw conclusions. The results obtained will probably make a company rethink its way of forecasting demand, looking for less error to save costs.

3 Preliminaries

3.1 Case Study: Chocolates Lacasa

Chocolates Lacasa is one of the leading confectionery producers in Spain with a significant presence in international markets. This company has a wide range of products that are sold through a variety of channels including supermarkets, specialty stores and online retailers, a broad client base.

Since Lacasa's products consists of sweets, candies, and chocolates, their sales present a strong seasonality, peaking during the Christmas season. Consequently, significant orders must be placed in the months leading up to December to prepare for this increased demand. To determine the quantities required from suppliers, Lacasa must first accurately estimate the sales of each product to its clients.

Figure 1 illustrates an example of the data used in this project. There is historical data spanning four years, from 2019 to 2022, encompassing 100 products. The dataset includes an initial prediction made by an analytical computer program (VPRU) and a second prediction, which is the first prediction adjusted by human input (VPRD). The latter represents Lacasa's final prediction. Additionally, the dataset contains information about the actual sales (VREA). A limitation of this data is the absence of actual product demand information. If the demand exceeded the number of units produced, the recorded sales data only reflect the units that were available to sale.

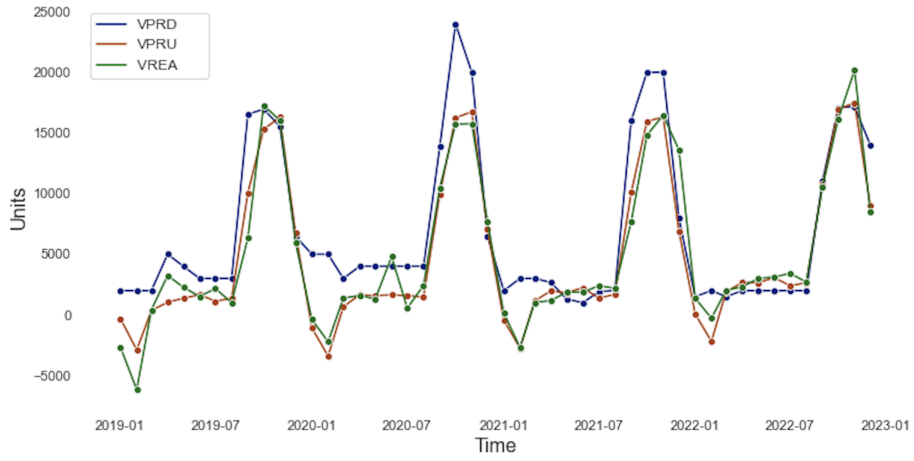


Figure 1: Sales over time

3.2 Overstocking and Understocking

We have highlighted the emphasis on this project on enhancing demand forecasting because of the typical disparity between predictions and actual outcomes. However, why is this important? Let us delve into the repercussions of an inaccurate forecast through an illustrative example.

Consider a hypothetical company which sells product A. To ensure product availability when customers demand it, the company must take place orders in advance based on sales forecast. Suppose the initial prediction suggests selling 1,000 units of product A, and orders are placed accordingly. However, actual sales later amount to only 800 units, resulting in an *Overstock* situation. This inaccuracy triggers several impacts Logistic [2023]:

- Financial Impact: excess inventory entails holding costs, maintenance expenses, and potential markdowns to move outdated stock.
- Storage Costs: overstocking needs additional warehouse space, incurring storage expenses.
- Risk of Obsolescence: excess inventory increases the likelihood of products becoming obsolete before they are sold.

Conversely, imagine that the actual demand for product A is 1,200 units, but only 1,000 units were produced, leading to an *Understock* scenario. This situation introduces different impacts:

- Missed Sales Opportunities: insufficient inventory leads to missed sales opportunities, impacting revenue.
- Customer Dissatisfaction: unfulfilled orders result in dissatisfies customers and potential damage to brand reputation.
- Lost Market Share: inability to meet demand may prompt customers to seek alternatives, resulting in lost market share.
- Supplier Relationships: inconsistent demand forecasts strain relationships with suppliers who may struggle to meet fluctuating demands.

In both scenarios, inaccurate demand forecasting leads to operational inefficiencies, financial losses, potential damage to customer relationships, and market competitiveness. Therefore, enhancing demand forecasting accuracy is crucial for mitigating these adverse effects.

4 Relevant tools

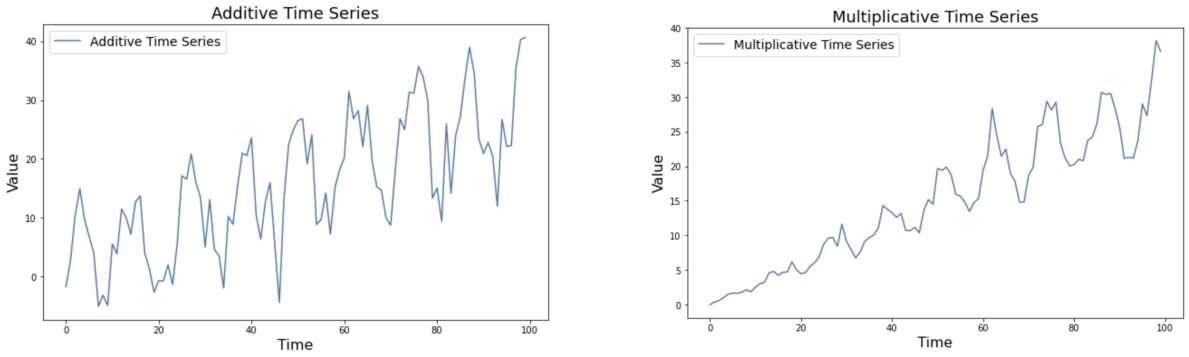
4.1 Time Series

A time series Song [2017] is a collection of observations made sequentially in time. Mathematically, a time series can be represented as a set of data points y_t, where t denotes time. The observations y_t are usually collected at discrete and equally spaced time intervals.

Time series can be decomposed into three parts: the trend, the stationary component, and noise. The trend (T_t) represents the long-term progression of the data. It can be a linear or a non-linear function reflecting the movement of the data. The stationary has to do with the Seasonality (S_t) and the Cycle (C_t). The seasonality captures periodic fluctuations and the cycle captures fluctuations occurring at irregular intervals. Finally, the noise (N_t) represents variations in the data that do not follow a predictable pattern. The general form of a time series model can be written as Equation 4.1 for the additive model or Equation 4.2 in the multiplication model.

$$y_t = T_t + S_t + C_t + N_t \tag{4.1}$$

$$y_t = T_t \cdot S_t \cdot C_t \cdot N_t \tag{4.2}$$



(a) Additive (b) Multiplicative

Figure 2: Randomly Generated Time Series Examples

As we can deduce from Figure 1 our data generally presents additive time series with a linear horizontal trend, a strong yearly seasonality, and more or less noise depending on the product.

We have chosen for this project boosting algorithms to solve the problem of future predictions. However, there exists many classical methods that aims to accomplish this task Hamilton [2020]. Some examples of these are Exponential Weighted Moving Average (EWMA), Autoregressive Moving Average (ARMA), and Autoregressive Integrated Moving Average (ARIMA).

4.2 Data Clustering

How do we prepare the algorithm to forecast next month's demand? Should we use all available data, or do we select specific subsets? To address this, it is crucial to consider that predicting the demand for a particular product may require feeding algorithm with its own historical data, and possibly supplemented by data from products performing similar. However, the challenge lies in determining which products to include.

There exists many ways of tackle this. In this project, our focus revolves around two algorithms: *K-Nearest Neighbors (K-NN)* and *K-Means*. K-Nearest Neighbors Algorithm aims to identify the K most similar products to a given one, while K-Means Algorithm partitions the data into K distinct clusters based on similarity.

In the upcoming sections, we will delve into the mechanisms of these algorithms and discuss how to employ them for selecting products to train the XGBoost algorithm.

4.2.1 K-Means Algorithm

Clustering is one of the most popular data-mining methods and consists of partitioning n observations into K clusters, where a cluster is characterized with the notions of homogeneity. For example, if we had a set of images of cats and dogs, a clustering algorithm would discover the animal types and cluster the images in two clusters, one for each animal.

For clustering algorithms we need a notion of distance for our data. If our data live in some space X, we need a function that computes the distance between two points on X. If we have $\mathbf{x}$, $\mathbf{x'} \in X$, we call the distance between them: $||\mathbf{x} - \mathbf{x'}||$. In the case of X being $\mathbb{R}^D$, a possible choice could be the Euclidean distance (ED):

$$||\mathbf{x} - \mathbf{x'}||_2 = \sqrt{\sum_{i=1}^{D}(x_i - x_i')^2} \tag{4.3}$$

There are many metrics that can be used depending on your data and on what *similarity*

means for the problem to solve. In fact, we will use the distance called *DTW*. But we will come back to this later.

Another important decision is selecting the number of clusters k in which we want to divide our data. We can choose a smaller k to have a more compressed representation but losing information, or a bigger k to have a less compressed representation but keeps more information about the data. In our case, we will evaluate the algorithm using different numbers of clusters and judge them using a metric called *Silhouette score* (Anex B).

Assume now that we have already chosen the number of clusters k and our metric for the distance (which will be the Euclidean for easy explanations). Let us understand how the algorithm works. Suppose that we have N points on our space X and we denote each like $\boldsymbol{x_n}$, then our data set is $\{\mathbf{x}_n\}_{n=1}^{N}$. The clustering algorithm will assign each of these $\boldsymbol{x_n}$ to a cluster. We represent these assignments with a vector $\boldsymbol{r_n}$ for each $\boldsymbol{x_n}$. This vector is all zeros expect for the cluster it is assigned to and we will call it the *responsibility vector*. In addition, we can denote $\boldsymbol{\mu_k}$ to the *mean vectors*. We can represent each cluster by the point in data space that is the average of the data assigned to it. Since each cluster is represented by an average, this approach is called *k-means*. With this notation the algorithm can be written as follows Adams [2018]:

Algorithm 1 k-means algorithm

Data: Data vectors $\{\mathbf{x}_n\}_{n=1}^N$, number of clusters K
for $n = 1$ *to* N **do**
    $\mathbf{r}_n \leftarrow [0, 0, \cdots, 0]$ `// Zero out the responsibilities`
    $k' \leftarrow \text{RandomInteger}(1, K)$ `// Make one of them randomly one to initialize`
    $r_{nk'} \leftarrow 1$
end
repeat
    `// Loop over the clusters`
    for $k = 1$ *to* K **do**
        $N_k \leftarrow \sum_{n=1}^N r_{nk}$ `// Compute the number assigned to cluster` k
        $\mu_k \leftarrow \frac{1}{N_k} \sum_{n=1}^N r_{nk} \mathbf{x}_n$ `// Compute the mean of the` k`th cluster`
    end
    `// Loop over the data`
    for $n = 1$ *to* N **do**
        $\mathbf{r}_n \leftarrow [0, 0, \cdots, 0]$ `// Zero out the responsibilities`
        $k' \leftarrow \arg\min_k \|\mathbf{x}_n - \mu_k\|^2$ `// Find the closest mean`
        $r_{nk'} \leftarrow 1$
    end
until;
until none of the $\mathbf{r}_n$ change
return $\{\mathbf{r}_n\}_{n=1}^N$ for each datum, and cluster means $\{\mu_k\}_{k=1}^K$

K-Means Algorithm iterates until it converges to a solution. In each loop it makes two updates: it loops over the responsibility vectors $\boldsymbol{r_n}$ and changes them until they get to the closest cluster, and it loops over the mean vectors $\boldsymbol{\mu_k}$ and changes them to be the mean of the data that currently belong to it. The idea is that each of the $\boldsymbol{x_n}$ can be replaced by the appropriate $\boldsymbol{\mu_k}$.

As any other Machine Learning algorithm we define a loss function to evaluate how well or poorly the algorithm is doing. Our objective function is:

$$C(\{\boldsymbol{r_n}\}_{r=1}^N, \{\boldsymbol{\mu_k}\}_{k=1}^K) = \sum_{n=1}^N \sum_{k=1}^K r_{nk} ||\boldsymbol{x_n} - \boldsymbol{\mu_k}||_2^2 \tag{4.4}$$

The K-Means Algorithm alternates minimizing with respect to $\boldsymbol{r_n}$ and with $\boldsymbol{x_n}$. If we first think about $\boldsymbol{r_n}$, we realize there only exists k possible values and we can minimize by choosing $r_{nk} = 1$ for the cluster that has the smaller distance:

$$r_{nk} = \begin{cases} 1, & \text{if } k = argmin_{k'} ||\boldsymbol{x_n} - \boldsymbol{\mu_{k'}}||_2^2 \\ 0, & \text{otherwise} \end{cases} \tag{4.5}$$

Thinking now about minimizing the μ_k, we can write (4.4) such as:

$$C(\boldsymbol{\mu_k}) = \sum_{n=1}^{N} r_{nk} ||\boldsymbol{x_n} - \boldsymbol{\mu_k}||_2^2 = \sum_{n=1}^{N} r_{nk} (\boldsymbol{x_n} - \boldsymbol{\mu_k})^\intercal (\boldsymbol{x_n} - \boldsymbol{\mu_k}) \tag{4.6}$$

To minimize, we first differentiate with respect to $\boldsymbol{\mu_k}$:

$$\nabla_{\boldsymbol{\mu_k}} C(\boldsymbol{\mu_k}) = \nabla_{\boldsymbol{\mu_k}} \sum_{n=1}^{N} r_{nk} (\boldsymbol{x_n} - \boldsymbol{\mu_k})^\intercal (\boldsymbol{x_n} - \boldsymbol{\mu_k}) = \sum_{n=1}^{N} r_{nk} \nabla_{\boldsymbol{\mu_k}} (\boldsymbol{x_n} - \boldsymbol{\mu_k})^\intercal (\boldsymbol{x_n} - \boldsymbol{\mu_k}) \tag{4.7}$$

We set the gradient to zero (we use $\nabla_a \boldsymbol{a}^\intercal \boldsymbol{a} = 2\boldsymbol{a}$) and solve for $\boldsymbol{\mu_k}$:

$$\nabla_{\boldsymbol{\mu_k}} C(\boldsymbol{\mu_k}) = -2 \sum_{n=1}^{N} r_{nk} (\boldsymbol{x_n} - \boldsymbol{\mu_k}) \tag{4.8}$$

$$\sum_{n=1}^{N} r_{nk} \boldsymbol{x_n} = \boldsymbol{\mu_k} \sum_{n=1}^{N} r_{nk} \tag{4.9}$$

$$\boldsymbol{\mu_k} = \frac{\sum_{n=1}^{N} r_{nk} \boldsymbol{x_n}}{\sum_{n=1}^{N} r_{nk}} \tag{4.10}$$

And this will be the K-Means Algorithm explained for Euclidean distance. However, we said before that we are going to use DTW (Dynamic Time Warping) instead (Anex A). This is a highly accurate but computationally more expensive distance measure. DTW can be seen as an extension of Euclidean distance (4.3) that offers a local non-linear alignment. Euclidean distance offers a comparison between two time series $\boldsymbol{x} = (x_1, ..., x_m)$ and $\boldsymbol{x'} = (x'_1, ..., x'_m)$. DTW searches for the "nearest point" between each point of the two series, thus allowing to discern similar shapes that may be deformed or out of phase. To do this, DTW algorithm constructs and m-by-m matrix M with the euclidean distances between any two points $\boldsymbol{x}$ and $\boldsymbol{x'}$, which compares the two series and calculates the distances between all the points of one and all the points of the other; and, on that matrix, the "most optimal path" is selected, i.e., the shortest distance between each pair of points.

The *warping path* $W = w_1, w_2, ..., w_r$, with $r \geq m$, is a contiguous set of matrix elements that defines a mapping between $\boldsymbol{x}$ and $\boldsymbol{x'}$ under several constraints Paparrizos and Gravano [2017].

$$DTW(\boldsymbol{x}, \boldsymbol{x'}) = min \sqrt{\sum_{i=1}^{r} w_i} \tag{4.11}$$

This path can be computed on matrix M with dynamic programming as follows Senin [2008]:

$$\gamma(i,j) = ED(i,j) + min(\gamma(i-1,j-1), \gamma(i-1,j), \gamma(i,j-1)) \tag{4.12}$$

This has proven that this way of computing the path captures the phase lags or shape deformations. Check Senin [2008] for more detail of γ and ED.

4.2.2 KNN Algorithm

As mentioned before, we use the K-Nearest Neighbors (KNN) algorithm to find the K closest products to a given product. Closeness here is understood in terms of distance, and as discussed already in the K-means algorithm section 4.2.1, distance is chosen depending on what similarity means. In this case, we will use again the Dynamic Time Warping (DTW) distance metric. Therefore, similarity is determined by the shape of the sales evolution over time.

We can easy describe how the algorithm works:

Algorithm 2 K-NN Algorithm

Data: Data vectors $\{\mathbf{x}_i\}_{i=1}^{N}$, number of nearest neighbours K

1. Compute the distance $d(\boldsymbol{x}, \boldsymbol{x_i})$ for $i = 1, 2, ..., N$, where $\boldsymbol{x}$ is the input point and $\boldsymbol{x_i}$ the rest of the points.
2. Order the distances in ascendant order so the closest points to the input point come first.
3. Take the first k points corresponding to the first k distances.

4.3 XGBoost Algorithm

In order to explain the XGBoost algorithm we are going to start by explaining the basic ideas of how decision trees and then gradient boosting works.

4.3.1 Decision Tree

Decision Trees can perform both classification and regression tasks. To understand how they make predictions let us just build one.

Suppose you have a dataset with different types of fruits each with their corresponding features such as color, diameter or texture. Imagine you want to classify fruits that are either an orange, an apple or a pear. You start at the *root node* (depth 0), this node asks whether the fruit's color is orange. If it is, then you move down to the root's left child node (depth 1). Since this node does not have children nodes, you directly classify this as an *Orange*.

Imagine now that you want to classify another fruit, but in this case it is green. You move now to the right root's child node and ask another question: is the fruit's diameter more than or equal to 6 cm? If it is, then your fruit is most likely an *apple*. If not, it is likely a *pear*. And there can be as many nodes as needed.

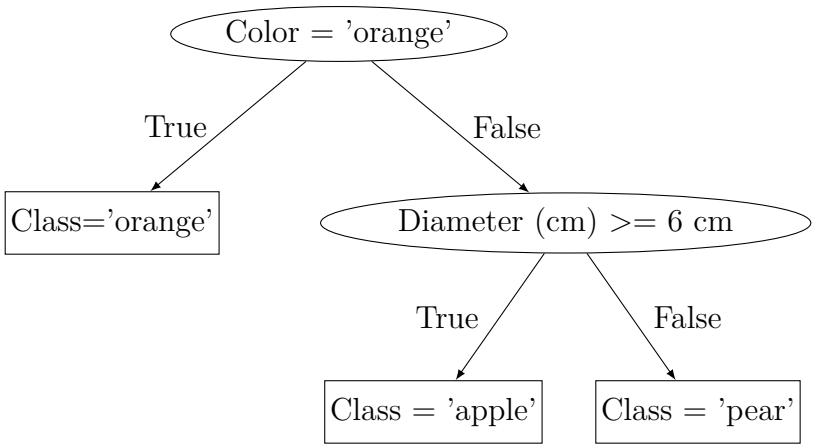


Figure 3: Decision Tree simple example.

A very important concept to understand how decision trees work is the *impurity* or *gini score*, G_i . A node is *pure* ($G_i = 0$) if all training instances it applies to belong to the same class Géron [2019]. A node's samples attribute counts how many training instances it applies to. For example, since the depht-1 left node applies only to "Orange" training instances, it is pure and its gini score is 0. Equation 4.13 shows how the training algorithm computes the gini score G_i of the i^{th} node.

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2 \tag{4.13}$$

where $p_{i,k}$ is the ratio of class k instances among the training instances in the i^{th} node.

The idea of the algorithm to train decision trees is simple: the algorithm first splits the training set in two subsets using a single feature k and a threshold t_k (e.g. "Diameter >= 6 cm"). It searches for the pair (k, t_k) that produces the purest subsets. The cost function that the algorithm tries to minimize is:

$$l(k, t_k) = \frac{m_{left}}{m} G_{left} + \frac{m_{right}}{m} G_{right} \tag{4.14}$$

where $G_{left/right}$ measures the impurity of the left/right subset and $m_{left/right}$ is the number of instances in the left/right subset.

Once it has successfully split the training set in two, it splits the subsets using the same logic, the the sub-subsets and so on, recursively. It stops once it reaches the maximum depth (Anex C), or it cannot find a split that will reduce impurity.

We have described here the decision tree for clustering. However, as we already mentioned we can also use decision trees for regression. The main difference is that instead of predicting a class in each node, it predicts a value. In addition, the algorithm works the same way as earlier, except that instead of trying to split the training set in a way that minimizes impurity, it tries to split the training set in a way that minimizes de MSE. The function that tries to minimize is:

$$l(k, t_k) = \frac{m_{left}}{m} MSE_{left} + \frac{m_{right}}{m} MSE_{right} \tag{4.15}$$

where:

$$MSE_{node} = \sum_{i \in node} (\hat{y}_{node} - \hat{y}^{(i)})^2 \tag{4.16}$$

and

$$\hat{y}_{node} = \frac{1}{m_{node}} \sum_{i \in node} y^{(i)} \tag{4.17}$$

4.3.2 Gradient Boosting

The Gradient Boosting algorithm works by sequentially adding predictors to an ensemble, each one correcting its predecessor. This method tries to fit the new predictor to the residual errors made by the previous predictor. A common type of predictor here are Decision Trees (4.3.1).

For a given data set with n examples and m features $\mathcal{D} = (\boldsymbol{x_i}, y_i)$ ($|\mathcal{D}| = n, \boldsymbol{x_i} \in \mathbb{R}^m, y_i \in \mathbb{R}$), a tree ensemble model uses k additive functions to predict the output **?**.

$$\hat{y}_i = \phi(\boldsymbol{x_i}) = \sum f_k(\boldsymbol{x_i}), \;\; f_k \in \mathcal{F} \tag{4.18}$$

where $\mathcal{F} = \left\{ f(\boldsymbol{x}) = w_{q(\boldsymbol{x})} \right\}$ ($q : \mathbb{R}^m \to T, w \in \mathbb{R}^T$) is the space of regression trees. Here q represents the structure of each tree that maps an example to corresponding leaf index. T is the number of leaves in the tree. Each f_k corresponds to an independent tree structure q and leaf weights w. Each tree has a continuous score on each of the leaf, w_i.

The model uses the following function that needs to minimize:

$$\mathcal{L}(\phi) = \sum_i l(y_i, \hat{y}_i) + \sum_k \Omega(f_k) \tag{4.19}$$

where $\Omega(f) = \gamma T + \frac{1}{2}\lambda||w||^2$, l is a loss function that measures the difference between the prediction, $\hat{y}_i$, and the target, y_i. The term Ω penalizes the complexity of the model and helps avoid over-fitting (Anex C).

The Gradient Tree Boosting model is trained in an additive manner. If $\hat{y_i}^{(t)}$ is the prediction of the *i-th* instance at the *t-th* iteration, we will need to add f_t to minimize the following objective:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y_i}^{(t-1)} + f_t(x_i)) + \sum_{k} \Omega(f_k) \tag{4.20}$$

this means we greedily add the f_t that most improves our model.

The basic idea of how the Gradient Boosting algorithm works is ilustrated in figure (4) and follows the following algorithm:

Algorithm 3 Gradient Boosting Algorithm

Data: Training set $\{(\boldsymbol{x_i}, y_i)\}_{i=1}^{n}$, loss function $l(y, \hat{y})$, number of iterations M
Initialize model with a constant value:

$$\hat{y_0} = \sum_{i=1}^{n} l(y_i, \mathcal{L}) \tag{4.21}$$

for $m = 1$ *to* M **do**

1. Compute the *pseudo-residuals*:

$$r_{im} = - \left. \frac{\partial l(y_i, \hat{y}_i)}{\partial \hat{y}_i} \right|_{\hat{y}_i = \hat{y_i}^{(m-1)}} \tag{4.22}$$

for $i = 1, ..., n$

2. Fit a learner (e.g. a tree) $f_m(x_i)$ to pseudo-residuals; i.e., train it using the training set $\{(\boldsymbol{x_i}, r_{im})\}_{i=1}^{n}$.
3. Compute multiplier $\mathcal{L}^{(m)}$ by solving the following one-dimensional optimization problem:

$$\mathcal{L}^{(m)} = \sum_{i=1}^{n} l(y_i, \hat{y_i}^{(m-1)} + f_m(x_i)) + \Omega(f_m) \tag{4.23}$$

4. Update the model:

$$\hat{y_i}^{(m)} = \hat{y_i}^{(m-1)} + f_m(x_i) \tag{4.24}$$

end
endfor
Output: $\hat{y_i}^{(M)}$

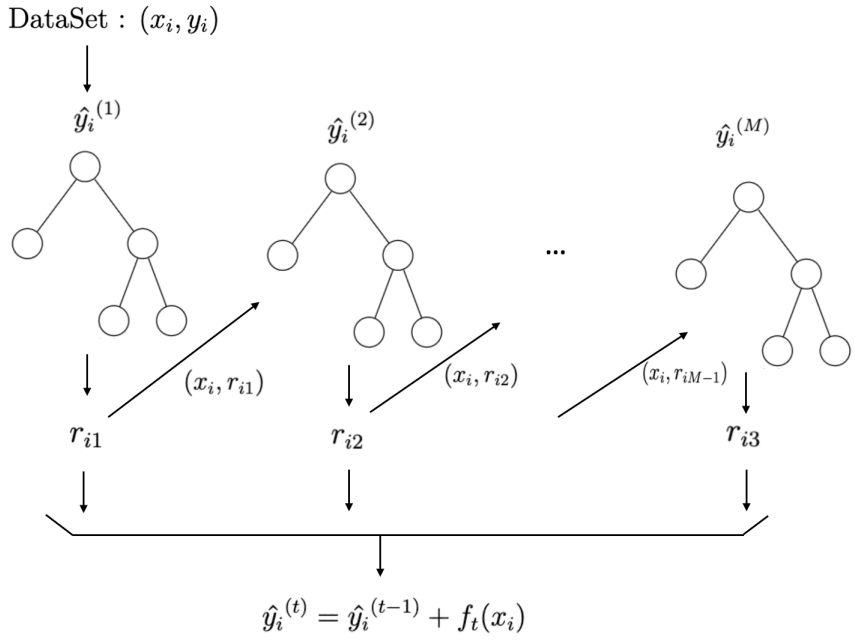


Figure 4: Gradient Boosting algorithm

4.3.3 XGBoost Algorithm

XGBoost, or Extreme Gradient Boosting, is an advanced implementation of the gradient boosting algorithm designed to enhance its performance and efficiency.

We can see in Algorithm 4 the structure of the algorithm. At each iteration m, the algorithm seek to minimize Nielsen [2016]:

$$J_m(\phi_m) = \sum_{i=1}^{n} l(y_i, \hat{y}^{(m-1)} + \phi_m(x_i))$$

Since it is a tree boosting algorithm, the basis are trees and therefore:

$$\phi_m(x) = \sum_{j=1}^{T} w_{jm} I(x \in R_{jm})$$

where T is the number of leaves in the tree and we call these regions $R_1, ..., R_T$.

XGBoost learns the tree structure and leaf weights at each iteration. We can say it has three stages. First, determine the leaf weights w_{jm} for a proposed (fixed) structure. Then, different structures are proposed with weights determined from the previous stage. Here the tree structure and thus the regions $\hat{R}_{jm}$, $j = 1, ..., T$ are determined. Finally, once a tree is

settled upon, the final leaf weights w_{jm}, $j = 1, ..., T$ in each terminal node are determined.

For this algorithm, we can write Nielsen [2016]:

$$J_m(\phi_m) = \sum_{i=1}^{n} \left(\hat{g}_m(x_i)\phi_m(x_i) + \frac{1}{2}\hat{h}_m(x_i)\phi_m(x_i)^2 \right) \tag{4.25}$$

If wee rewrite the Equation 4.25 with trees as basis functions:

$$J_m(\phi_m) = \sum_{i=1}^{n} \left(\hat{g}_m(x_i) \sum_{j=1}^{T} w_{jm} I(x_i \in R_{jm}) + \frac{1}{2}\hat{h}_m(x_i) \left(\sum_{j=1}^{T} w_{jm} I(x_i \in R_{jm}) \right)^2 \right) \tag{4.26}$$

Thanks to the disjoint nature of the regions of the terminal nodes, we can rewrite this as:

$$\begin{aligned} J_m(\phi_m) &= \sum_{i=1}^{n} \left(\hat{g}_m(x_i) \sum_{j=1}^{T} w_{jm} I(x_i \in R_{jm}) + \frac{1}{2}\hat{h}_m(x_i) \sum_{j=1}^{T} w_{jm}^2 I(x_i \in R_{jm}) \right) \\ &= \sum_{j=1}^{T} \sum_{i \in I_{jm}} \left(\hat{g}_m(x_i) w_{jm} + \frac{1}{2}\hat{h}_m(x_i) w_{jm}^2 \right) \end{aligned} \tag{4.27}$$

where I_{jm} is the set of indices of the x_i falling in the region R_{jm}. Letting $G_{jm} = \sum_{i \in I_{jm}} \hat{g}_m(x_i)$ and $H_{jm} = \sum_{i \in I_{jm}} \hat{h}_m(x_i)$, we can rewrite this as:

$$J_m(\phi_m) = \sum_{j=1}^{T} \left(G_{jm} w_{jm} + \frac{1}{2} H_{jm} w_{jm}^2 \right) \tag{4.28}$$

For a proposed fixed structure, the weights are thus given by:

$$w_{ij} = -\frac{G_{jm}}{H_{jm}}, \ j = 1, ..., T \tag{4.29}$$

Pluggin the weights from Equation 4.29 into Equation 4.28, we find for a fixed structure:

$$J_m(\phi_m) = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_{jm}^2}{H_{jm}} \tag{4.30}$$

When searching for the optimal split, that is the function to minimize. That is, the splits are determined by maximizing the gain given by:

$$Gain = \frac{1}{2} \left(\frac{G_l^2}{H_l^2} + \frac{G_R^2}{H_R^2} - \frac{G_{jm}^2}{H_{jm}^2} \right) \tag{4.31}$$

Algorithm 4 XGBoost Algorithm

Data: Data set D, loss function l, number of iterations M, learning rate η, number of terminal nodes T

Initialize $y_0 = \arg\min_\theta \sum_{i=1}^{n} l(y_i, \theta)$

for $m = 1$ *to* M **do**

1. Compute the first derivative:

$$g^{(m)}(x) = \sum_{i=1}^{n} \frac{\partial l(y_i, \hat{y}_i)}{\partial \hat{y}_i}\bigg|_{\hat{y}_i = \hat{y}_i{}^{(m-1)}} \tag{4.32}$$

2. Compute the second derivative:

$$h^{(m)}(x) = \sum_{i=1}^{n} \frac{\partial^2 l(y_i, \hat{y}_i)}{\partial \hat{y}_i{}^2}\bigg|_{\hat{y}_i = \hat{y}_i{}^{(m-1)}} \tag{4.33}$$

3. Determine the structure $\{\hat{R}_{jm}\}_{j=1}^{T}$ by selecting splits which maximize the gain:

$$\text{Gain} = \frac{1}{2}\left(\frac{G_L^2}{H_L} + \frac{G_R^2}{H_R} - \frac{G_{jm}^2}{H_{jm}}\right) \tag{4.34}$$

4. Determine the leaf weights $\{\hat{w}_{jm}\}_{j=1}^{T}$ for the learned structure by:

$$\hat{w}_{jm} = -\frac{G_{jm}}{H_{jm}} \tag{4.35}$$

5. Update the model:

$$\hat{y}_m = \eta \sum_{j=1}^{T} \hat{w}_{jm} I(x \in \hat{R}_{jm}) \tag{4.36}$$

$$\hat{y}^{(m)} = \hat{y}^{(m-1)} + \hat{y}_m \tag{4.37}$$

end

endfor

Result: $\hat{y} \equiv \hat{y}^{(M)}$

5 XGBoost Algorithm Feeding and Training

Let's walk through our process with one of the selected products.

5.1 Clusters

First, we are going to use the *k-means* algorithm to divide our data into clusters or groups. We start standardizing our data to avoid confusion when computing distances in our algo-

rithm. Then we construct our algorithm using the DTW metric for the distance. But in how many clusters k should we divide the data? To make this decision we run several trials with different number of clusters, from 2 to 10, and use the *Silhouette Score* as a measure of goodness (Anex B). We obtain the following result:

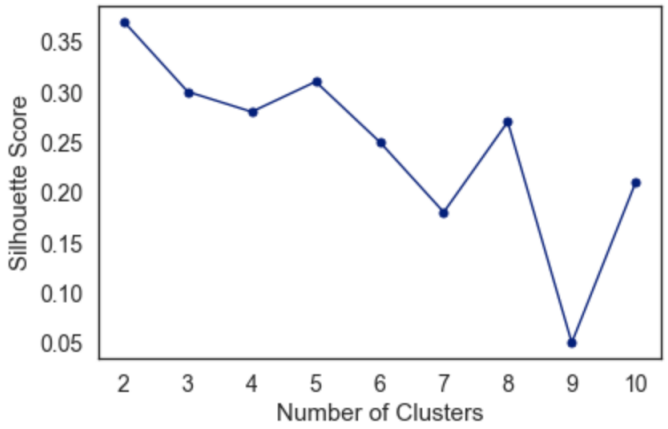


Figure 5: Silhouette Score for different number of clusters for the K-Means Algorithm with DTW as a metric for the distance

The figure shows that the best number of clusters in which we should divide the data is 2. However, as we explained before dividing our data in just a few clusters provides little information. This is why we finally decide to choose $k = 8$, since its Silhouette Score is relatively high and it will give us more information about the closeness between the various products. We do inspection now on the shape of the graphs of the different clusters and select a cluster with a specific pattern (Figure 6) for the evolution of sales over time.

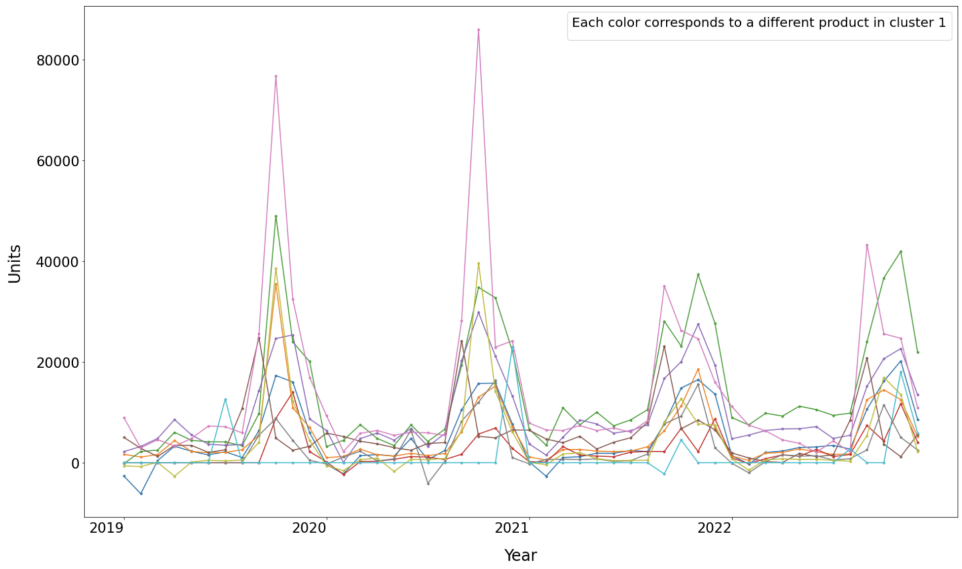


Figure 6: Sales (in units) over time of the products that belongs to Cluster 1

We select now a product from this cluster and introduce it in the constructed K-NN Algorithm to get its closest neighbours. We select 4 neighbours since we believe this could give us enough data to train the XGBoost algorithm later but without overtraining (Anex C). After getting the nearest neighbours of this product, we realize they both are in the same cluster of the clusters generated by the K-Means Algorithm. This is just a coincidence, since in general this fact is not true. However, we think this may benefit the training and stay with this product for the example.

5.2 Data Processing

After selecting the data to feed our algorithm and conducting initial trials, we must prepare it. We have chosen five products to train the algorithm, using their historical sales data spanning the past 4 years. Our goal is to forecast the future demand for one of this products. We divide the dataset into subsets of five. Each subset will consist of vectors containing the historical data from the preceding four months for the five products (including the target product and its four neighboring products). Additionally, we will incorporate two time variables representing the month and year for each prediction. The output will be the actual sales for the next month (fifth month) of the target product. The algorithm will be trained using all the input vectors (X) paired with their corresponding output (y).

	p06_2019	p07_2019	p08_2019	p09_2019	p10_2019
5	1524.0	2167.0	1011.0	6331.0	17289.0
6	1973.0	2040.0	2547.0	5426.0	35449.0
13	2040.0	2536.0	10726.0	24843.0	4869.0
44	472.0	354.0	512.0	4010.0	38590.0
99	0.0	12600.0	0.0	0.0	0.0

Figure 7: Subset of five months obtained for products 5,6,13,44 and 99

For instance, let us consider products number '5', '6', '13', '44', and '99' over the 100 products we have data from. Being product number '5' the target product. We divide the data in groups of five months and obtain subsets such as the one in Figure 7 ('p06-2019' refers to June 2019, etc). Taking this example, our input vector X will be of the form:

$$[1524, 2167, 1011, 6331, 1973, 2040, ..., 0, 12600, 0, 0]$$

with the goal of predicting y = 17289.

To enhance our predictive model, we add categorical variables for the month and year at the beginning of vector X. In this case it will be November of 2019, since it is the month of which we want to make the prediction of the sales with this training vector X. The modified vector will look like this:

$$[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1524, 2157, 1011, 6331, 1973, 2040, ..., 0, 12600, 0, 0,]$$

The first twelve entries represent the months (January to December). A '1' indicates the month of interest, November (11th position). The next four entries represent the years (2019 to 2022). A '1' in the 2019 position (first of the four year slots) indicates the year of interest.

Once we have our data ready we construct our XGBoost algorithm. We first split the data into train (80% of the data) and test (20% of the data). The training data is the one that will be used to train the algorithm and the test data is the one that will be used to test the goodness of learning. Now we need to select the hyperparameters that the algorithm will use for the training (Anex C). In order to do this we uses the method Cross Validation (Anex D) for help. This method automatically chooses the best parameters between a given range. We make our predictions and calculate the error for the test samples. We have chosen the Root Mean Squared Error (RMSE):

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{x_i})^2} \tag{5.1}$$

where N is the number of observations in the test set, $\hat{x_i}$ are the predicted values of the sales and x_i the actual sales.

We compare now the errors obtained with XGBoost algorithm and the ones of Chocolates Lacasa. RMSE(XGBoost) = 270.8, RMSE(Lacasa) = 1,278.9. XGBoost algorithm clearly has improved the forecast of the demand for this product. As shown in Figure 8 XGBoost predictions are nearer to the red line (which will be the perfect prediction) than Lacasa's predictions. Note that we have used 'VPRD' as Lacasa's prediction.

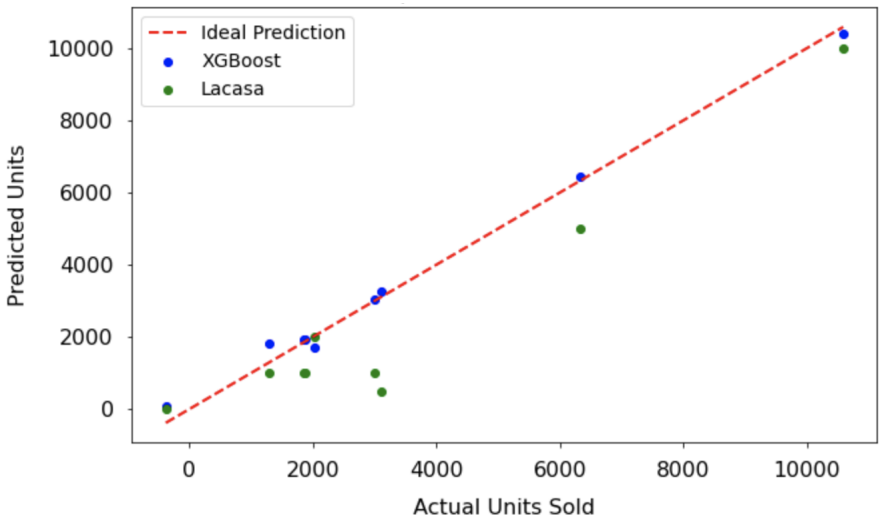
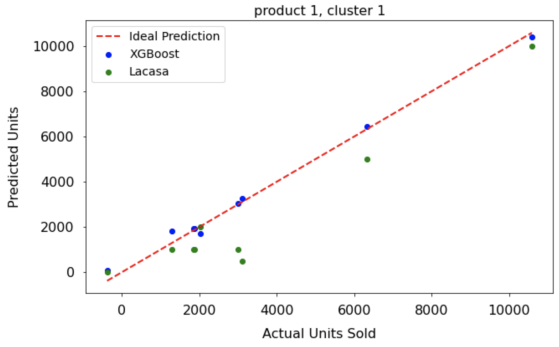


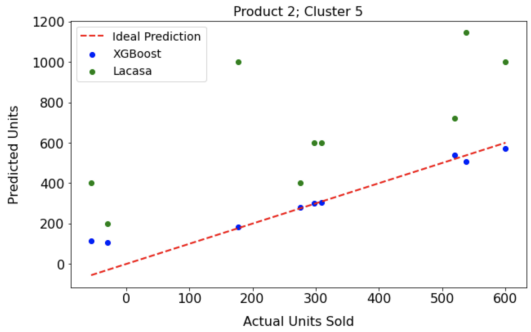
Figure 8: Actual sales vs XGBoost predictions and vs Lacasa predictions

6 Results

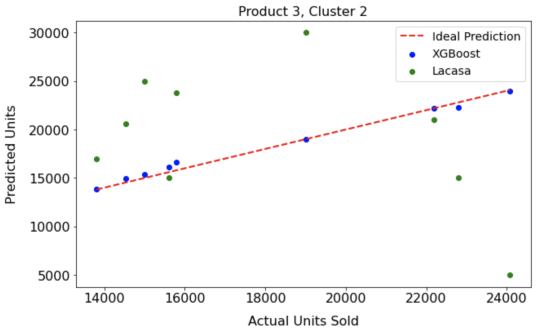
After the success of our training with the first choice product, we want to validate the result of XGBoost predicting better demand by doing this process across different products. We repeat then the whole process for another 9 products belonging to different clusters. By observing the results obtained in Table 1, we see that with this method we have managed to reduce the RMSE significantly by an average of 88 %.



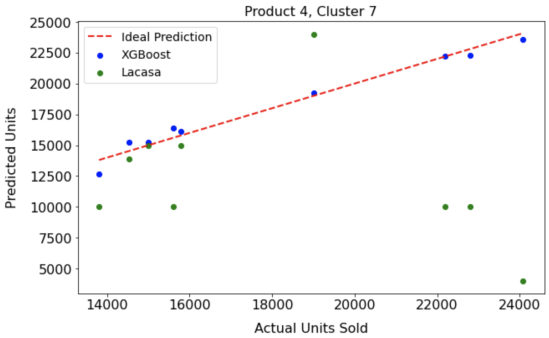
(a) Product 1, Cluster 1



(b) Product 2, Cluster 5



(c) Product 3, Cluster 2



(d) Product 4, Cluster 7

Figure 9: Actual units sold vs Lacasa predictions and XGBoost predictions

Reducing the discrepancy between demand and forecast offers numerous advantages. Firstly, financial savings arise from decreased inventory levels. With more accurate forecasts the need for excessive Safety Stock, the extra production to safeguard against market fluctuations, diminishes. This means less inventory to hold in the warehouse and consequently, they would need less storage capacity with the induced savings (financial, storage, insurance, theft, deterioration, obsolescence, etc).

Moreover, this improved forecasting allows for better resource utilization. Manufacturing facilities have limited production capacity, and inaccurate forecasts can leave with no reaction time to respond to actual demand. Enhanced forecasting precision minimizes the need for reaction time, enabling the factory to operate more efficiently.

Having the product readily available provides a significant competitive advantage, as customers are less likely to seek out alternative suppliers or turn to competitors. This enhances customer satisfaction and loyalty. Additionally, accurate forecasting reduces opportunity costs and fosters a more stable operational environment. The decreased need for reactive

decision-making allows for greater focus on critical areas such as process improvement and innovation.

Product	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Cluster	1	5	2	7	1	7	0	3	4	6
RMSE Lacasa	1,279	434	9,185	9,354	7,347	1,873	2,179	1,441	22,977	27,06
RMSE XGBoost	271	74	425	594	782	301	128	92	7,587	89
Relative Accuracy	79 %	83 %	95 %	94 %	89 %	84 %	94 %	94 %	67 %	97 %

Table 1: RMSE comparison across different products.

In Table 1 Relative Accuracy refers to:

$$\left(1 - \frac{RMSEXGBoost}{RMSELacasa}\right) \cdot 100 \tag{6.1}$$

7 Conclusions, Limitations, and Further Improvements

In this project we have been able to apply Machine Learning Algorithms to the business world. We have succeeded in making the XGBoost algorithm learn to predict future demand for a time series type data. The algorithm has thus been able to identify patterns needed to learn automatically.

The goal of this project was to see if we could develop a tool making use of the ML techniques to improve the actual demand prediction of Chocolates Lacasa. To accomplish this task we were provided with data of 100 products. But we needed to chose data smartly to feed the XGBoost later. Then, we first did a study of the data, identifying strong seasonality and similar behaviours between several products. Then, decided to cluster the data using K-Means Algorithm to split the data according to similarities. We used also K-NN Algorithm to keep only the nearest neighbours to the product we are interested in predict the demand. Once we selected these data we added to the input data of the XGBoost algorithm the years and months we want to predict with each data vector in order to provide more information and obtain a better forecast. Finally, we constructed the XGBoost model and we trained it choosing the best hyperparameters with Cross Validation method. This procedure was done for several products belonging to different clusters to compare RMSE in each of them between our new method and the old one.

The results that we obtained strongly suggest that ML methods are able to improve the forecast demand of this company and probably the forecast of many other companies in the

world. The implementation of a tool like this in a company can help you in many ways. It has very relevant impacts such as financial savings coming from the reduction of inventory in the warehouse, the avoid of obsolescence, keeping the customer happy by meeting their demand, avoiding losing market share, reducing the need of reaction time, etc.

To enhance this study, several improvements could be implemented. Recording actual demand instead of sales will help us understand the potential sales we could have achieved without supply constraints. Incorporating price data for each product at different times will allow us to analyze how price changes impact demand. Additionally, we used 'VPRD' data, which corresponds to the forecast of the demand predicted by a computer tool ('VPRU') "fixed" by the human expertise, as Lacasa's prediction. However, by doing this we are assuming that those final predictions are better than the initial ones without the modification made by humans. But this may not be always true. Therefore, we could also do this study with the 'VPRU' data. On the other hand, Extending the XGBoost training to all 100 products, instead of just 10, will provide a more comprehensive analysis. Exploring different clustering methods and performing hyperparameter tuning can further optimize the algorithm. On top of this, I believe it could be very interesting to put this tool in practice in real time. Predict the demand with the XGBoost Algorithm and with the actual method for the next month for all products, compare it later with the actual demand and see what the potential saving costs will be between both methods. These enhancements will lead to a more accurate and comprehensive understanding of demand patterns and improve the study's overall predictive performance.

After carrying out this work we can see how powerful this tool can be. The development of these algorithms is continuously improving and this technology is already revolutionizing the business world.

References

R. P. Adams. K-means clustering and related algorithms. *Princeton University*, 2018.

D. Berrar. *Cross-Validation*. 01 2018. ISBN 9780128096338.

A. Géron. *Hands-On Machine Learning with Scikit-Learn & TensorFlow*. 2019.

J. D. Hamilton. *Time series analysis*. Princeton University, 2020.

L. Logistic. Overstocking vs understocking: Finding the balance for effective inventory management, 2023. Accessed: 2024-06-10.

D. Nielsen. Tree boosting with xgboost-why does xgboost win "every" machine learning competition? Master's thesis, NTNU, 2016.

J. Paparrizos and L. Gravano. Fast and accurate time-series clustering. *ACM Transactions on Database Systems (TODS)*, 42(2):1–49, 2017.

P. Senin. Dynamic time warping algorithm review. *Information and Computer Science Department University of Hawaii at Manoa Honolulu, USA*, 855(1-23):40, 2008.

M. Shutaywi and N. N. Kachouie. Silhouette analysis for performance evaluation in machine learning with applications to clustering. *Entropy*, 2021.

H. Song. Review of time series analysis and its applications with r examples (3rd edition), by robert h. shumway & david s. stoffer. *Structural Equation Modeling: A Multidisciplinary Journal*, 24(5):800–802, 2017. doi: 10.1080/10705511.2017.1299578.

Appendix

A Dinamic Time Warping

In the time series analysis, Dynamic Time Warping (DTW) Senin [2008] is an algorithm for measuring the similarity between two time sequences that provides a good fit even in the face of a time or velocity tag.

Unlike the Euclidean method, which is based on the direct comparison of each point of a series with its equivalent in a different series, the DTW searches for the "nearest point" between each point of the two series, thus allowing to discern similar shapes that may be deformed or out of phase. To do this, a similarity matrix is generated, which compares the two series and calculates the distances between all the points of one and all the points of the other; and, on that matrix, the "most optimal path" is selected, i.e., the shortest distance between each pair of points.

DTW follows certain rules and restrictions to find the optimal match between two time series:

- Every index from the first sequence must be matched with one or more indices from the other sequence, and vice versa
- The first index from the first sequence must be matched with the first index from the other sequence (but it does not have to be its only match)
- The last index from the first sequence must be matched with the last index from the other sequence (but it does not have to be its only match)
- The mapping of the indices from the first sequence to indices from the other sequence must be monotonically increasing, and vice versa, i.e. if $j > i$ are indices from the first sequence, then there must not be two indices $l > k$ in the other sequence, such that index i is matched with index l and index j is matched with index k, and vice versa

The optimal match is the match that satisfies all the restrictions and the rules and that has the minimal cost, where the cost is computed as the sum of absolute differences, for each matched pair of indices, between their values.

The sequences are "warped" non-linearly in the time dimension to determine a measure of their similarity independent of certain non-linear variations in the time dimension. In addition to a similarity measure between the two sequences, a so called "warping path" is produced. By warping according to this path the two signals may be aligned in time. The

signal with an original set of points X(original), Y(original) is transformed to X(warped), Y(warped).

B Silhouette Score

Silhouette index Shutaywi and Kachouie [2021] is an unsupervised method for evaluating the performance of a clustering method. There are other methods to evaluate clustering results, such as Rand Index or Distortion Score. However, while most of the performance evaluation methods need a training set, the Silhouette index does not need a training set to evaluate the clustering results. The silhouette width $s(x_i)$ for the point x_i is defined as:

$$s(x_i) = \frac{b(x_i) - a(x_i)}{max\{b(x_i), a(x_i)\}} \tag{B.1}$$

where x_i is an element in the cluster r_k, $a(x_i)$ is the average of the distance of x_i to all other elements in the cluster r_k (within dissimilarity), and

$$b(x_i) = min\{d_l(x_i)\},\ with\ l \neq k$$

where $d_l(x_i)$ is the average distance from x_i to all point in the cluster r_l for $l \neq k$ (between dissimilarity).

The value of the Silhouette width, equation (B.1), can vary between -1 and 1. A negative value is related to a case in which $a(x_i) > b(x_i)$ which is undesirable because that means that the distance within dissimilarity is greater than between dissimilarity. A positive value is obtained when $a(x_i) < b(x_i)$, and the Silhouette width reaches its maximum $s(x_i)$ for $a(x_i) = 0$. The greater the $s(x_i)$, the higher the likelihood to be clustered in the correct group.

C Hyperparameters

The ML algorithms present several parameters, usually called *hyperparameters*, that need to be set manually or smartly choose by the algorithm in some way. These parameters need to be adjusted to help the model give the best outcomes. Playing with this parameters can also leave you in a situation of *Underfitting* or *Overfitting*.

Underfitting occurs when the algorithm is too simple to capture the underlying patterns in the data. As a result, the model performs poorly on both the training and the test set.

This is often a consequence of insufficient data or a very simple model. When a model underfits, it fails to learn the training data adequately and thus, performs poorly in predictions.

Overfitting occurs when your model is overtrained, meaning it is too complex for the given data or you have too much data. This implies that your model performs well with the training data but poorly with the test data, which is the new data it has never seen before. It is as if the model learns the training data by heart, including noise and outliers, which hampers its performance on new data. One way to reduce overfitting is by simplifying the model, for example by increasing the learning rate, reducing the number of features, or using regularization techniques.

The hyperparameters of our XGBoost algorithm Nielsen [2016] are the following (note that not all hyperparameters were included explicitly in our algorithm explanations for simplicity):

- **Number of Trees** (M) : the parameter M corresponds with the number of boosting iterations, which in our case is the number of trees in the model. More trees may help the algorithm capture more complexity, but increases the risk of overfitting.
- **Learning Rate** (η) : the learning rate or shrinkage parameter shrinks the added basis function at each iteration. For tree boosting, η can be seen to shrink the leaf weights of each of the individual trees learnt at each iteration. If η is set too high, the model will fit a lot of the structure in the data during the early iterations, thereby quickly increasing variance. We mentioned that using a larger number of trees increases the representational ability. Thus, by lowering the learning rate η, a larger number of trees can be added before the additive tree model will start to overfit the data. This will allow the model to have greater representational ability, with smoother fits, before overfitting the data.

$$f^{(m)}(x) = f^{(m-1)}(x) + \eta \sum_{j=1}^{T} w_j I(x \in R_j)$$

- **Max Depth** : Maximum number of split levels in each tree. It is used to control overfitting. This affects the structure $\{\hat{R}_{jm}\}_{j=1}^{T}$
- **Subsample** : size of the subsets of the training data used to fit each tree. Subsampling can help prevent overfitting.
- **Colsample bytree**: The fraction of features to be used for fitting each tree. It helps in preventing overfitting, similar to subsample, but applied to the feature space.

- **Gamma** (γ): It penalizes the gain of a split. Higher values make the algorithm more conservative, avoiding splits that may be due to noise.

$$\text{Gain} = \frac{1}{2}\left(\frac{G_L^2}{H_L} + \frac{G_R^2}{H_R} - \frac{G_{jm}^2}{H_{jm}}\right) - \gamma \tag{C.1}$$

- **Reg** α: L1 regularization term that adds a penalty to the objective function.

$$\mathcal{L} = l(\Theta) + \alpha \sum_{j=1}^{n} |\theta_j| \tag{C.2}$$

  where l is the loss function, Θ the model parameters, and θ_j are the weights in the model.

- **Reg** λ: L2 regularization term on weights (analogous to Ridge regression).

$$\mathcal{L} = l(\Theta) + \frac{\lambda}{2} \sum_{j=1}^{n} \theta_j^2 \tag{C.3}$$

  where l is the loss function, Θ the model parameters, and θ_j are the weights in the model.

D Cross Validation

Cross-validation Berrar [2018] is one of the most used methods to estimate the true prediction error and to tune model parameters. It consists in splitting the data into training and validation sets which help the model to generalize well and avoid overfitting. This methods follows the following logic:

Algorithm 5 Cross Validation Algorithm

Data: Learning Data Set

1. The available learning set is partitioned into k disjoint subsets.
2. The model is trained using $k-1$ subsets, which, together, represent the training set.
3. Then the model is applied to the remaining subset, which is denoted as the *validation set*, and the performance is measured.
4. This procedure is repeated until each of the k validation sets has served as validation set.

The average of the k performance measurement on the k validation sets is the cross-validated performance.

The Cross Validation method repeats the algorithm as many times as there are different possible combinations of hyperparameters. The possible values for each hyperparameter are determined when constructed the algorithm. In our case we used the following possible values for each parameter:

```
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [5, 10, 15],
    'learning_rate': [0.1, 0.01, 0.001],
    'gamma': [0, 0.1, 0.2],
    'subsample': [0.8, 0.9],
    'colsample_bytree': [0.8, 0.9],
    'reg_alpha': [0, 0.1, 0.5],
    'reg_lambda': [1, 2, 5]
}
```

Therefore, in this case the algorithm looked for the best combination of hyperparameters between $3^6 \cdot 2^2 = 2,916$ possible combinations. Repeating then $2,916$ times the procedure of Algorithm 5. The hyperparameters chosen for each product are shown in Table 2.

Product	Cluster	colsample	γ	η	Max Depth	Number of Estimators	reg α	reg λ	Subsample	RMSE Lacasa	RMSE XGBoost	Relative Accuracy
1	1	0.9	0.1	0.1	15	150	0.1	5	0.8	1,278	270	78 %
2	5	0.9	0.2	0.1	5	150	0	2	0.9	434	73	83 %
3	2	0.9	0.1	0.1	5	150	0.5	5	0.8	9,185	424	95 %
4	7	0.9	0.1	0.1	5	150	0.1	5	0.9	9,354	593	93 %
5	1	0.8	0	0.1	10	100	0.5	5	0.9	7,347	781	89 %
6	7	0.9	0	0.1	10	150	0	2	0.9	1,873	301	83 %
7	0	0.9	0.1	0.1	10	150	0	5	0.8	2,179	128	94 %
8	3	0.9	0.2	0.1	5	150	0.5	1	0.8	1,441	91	93 %
9	4	0.9	0.1	0.1	10	100	0	5	0.9	22,977	7,587	66 %
10	10	6	0.9	0.1	15	150	0	1	0.9	2,705	89	97 %

Table 2: Hyperparameters values and RMSE results of the XGBoost algorithm training of each product.

E Python Code

```

#import python libraries
import pandas as pd
import numpy as np
from tslearn.preprocessing import TimeSeriesScalerMeanVariance
from sklearn.impute import SimpleImputer
```

```
from sklearn.preprocessing import MinMaxScaler
from tslearn.clustering import TimeSeriesKMeans
import seaborn as sns
from tslearn.clustering import silhouette_score
from sklearn.cluster import KMeans
from sklearn.metrics import pairwise_distances
import matplotlib.pyplot as plt
from tslearn.metrics import soft_dtw

#read file
df= pd.read_csv('merge_19202122.csv')

#Silhouette score to select number of clusters k
from sklearn.metrics import silhouette_score

# df is the DataFrame with the product sales data
# Selecting relevant columns
X = df.iloc[:, 2:].values  # Time series start from the third column

# Scale the data to avoid very large values
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# Normalize the time series
X_normalized = TimeSeriesScalerMeanVariance().fit_transform(X_scaled)

# Range of clusters to test
min_clusters = 1
max_clusters = 12 # Adjust the range as needed

silhouette_scores = []

# Iterate over different numbers of clusters
for n_clusters in tqdm(range(min_clusters, max_clusters + 1)):
    # Soft-DTW k-means clustering with a custom metric wrapper
    kmeans = TimeSeriesKMeans(n_clusters=n_clusters, metric="softdtw",
    verbose=True)
    y_pred = kmeans.fit_predict(X_normalized)

    # Calculate silhouette score
    silhouette_avg = silhouette_score(X_normalized.reshape((X_normalized.
    shape[0], -1)), y_pred, metric="euclidean")
    silhouette_scores.append(silhouette_avg)

# Plot the silhouette scores
plt.plot(range(min_clusters, max_clusters + 1), silhouette_scores, marker=
```

```
    'o')
plt.title('Silhouette Score for Optimal Number of Clusters')
plt.xlabel('Number of Clusters')
plt.ylabel('Silhouette Score')
plt.show()

# Find the number of clusters with the highest silhouette score
best_n_clusters = silhouette_scores.index(max(silhouette_scores)) +
    min_clusters
print(f"Best Number of Clusters: {best_n_clusters}")

# Selecting relevant columns
X = df.iloc[:, 2:].values  #Time series start from the third column

# Scale the data to avoid very large values
scaler = MinMaxScaler()
X = scaler.fit_transform(X)

# Normalize the time series
X = TimeSeriesScalerMeanVariance().fit_transform(X)

# Number of clusters (adjust as needed)
n_clusters = 8

# Soft-DTW k-means clustering
kmeans = TimeSeriesKMeans(n_clusters=n_clusters, metric="softdtw", verbose
    =True)
y_pred = kmeans.fit_predict(X)

# Add the cluster labels to the DataFrame
df['cluster_softdtw'] = y_pred

cluster_counts = df['cluster_softdtw'].value_counts()
len(y_pred), y_pred #print the cluster of each product

products_by_cluster = {}

# Iterate over unique cluster labels
for cluster_label in df['cluster_softdtw'].unique():
    # Filter DataFrame to get products for the current cluster
    products_for_cluster = df[df['cluster_softdtw'] == cluster_label]['
    product'].tolist()

    # Store the list of products in the dictionary
    products_by_cluster[cluster_label] = products_for_cluster

```

```
# Print or use the dictionary as needed
for cluster_label, products_list in products_by_cluster.items():
    print(f"Cluster {cluster_label}: {products_list}") #print the clusters
    with the products in each
```

Listing 1: Time Series k-means Clustering with Soft-DTW

```

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import NearestNeighbors
from tslearn.metrics import dtw as ts_dtw

df= pd.read_csv('merge_19202122.csv')
df = df[df['cod_con'] == 'VREA']
df[df['product'] == 'Product1'] #filter by the product which we want to
    get the closest neighbours

from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import NearestNeighbors

# Selecting a product by its index in the DataFrame
selected_product_index = 135 # Change this index to the product of
    interest

# Extract columns of time series, assuming they start from the third
    column
time_series_columns = df.columns[2:]

# Prepare data for k-NN search
X = df[time_series_columns].values

# Standardize data
scaler = StandardScaler()
X_standardized = scaler.fit_transform(X)

# Initializing the NearestNeighbors model with the custom distance
    function
n_neighbors = 5 # You can adjust the number of neighbors
knn_model = NearestNeighbors(n_neighbors=n_neighbors, algorithm='auto',
    metric=ts_dtw)

# Fit the model to standardized data
knn_model.fit(X_standardized)

# Use the data of the selected product for the query
```

```
selected_product_data = df.loc[selected_product_index, time_series_columns
    ].values.reshape(1, -1)

# Standardize query data before searching for nearby neighbors
selected_product_data_standardized = scaler.transform(
    selected_product_data)

# Find indexes and distances of nearest neighbors
distances, indices = knn_model.kneighbors(
    selected_product_data_standardized)

# Show nearest neighbors
nearest_neighbors = df.iloc[indices[0]]

# Display only the index and name
nearest_neighbors_info = nearest_neighbors[['product']]
print(nearest_neighbors_info)
```

Listing 2: Finding Nearest Neighbors with k-nn algorithm using DTW

```

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

df= pd.read_csv('merge_19202122.csv')
df_vrea = df[df['cod_con'] == 'VREA']
df_vrea.reset_index(drop=True, inplace=True)

#write the names of the 5 neighbours
products_to_keep = ['product',
                    'neighbour 1',
                    'neighbour 2',
                    'neighbour 3',
                    'neighbour 4']

# Filtering the DataFrame
df_vrea = df_vrea[df_vrea['product'].isin(products_to_keep)]

# Extracting only the columns we need
subsets_5 = [df_vrea.iloc[:, i:i+5] for i in range(2, len(df_vrea.columns)
    -4, 1)]

# Dropping the last column from each subset
subsets_4 = [subset.drop(subset.columns[-1], axis=1) for subset in
    subsets_5]

```

```
# Reshaping the data to have 21 columns
reshaped_data = []
last_column_values = []  # Store the last column values from the first row of each subset

for subset in subsets_4:
    rows = [subset.iloc[:, i:i+4].values.flatten() for i in range(0, len(subset.columns), 4)]
    for row in rows:
        reshaped_data.append(row)

for subset in subsets_5:
    rows = [subset.iloc[:, i:i+5].values.flatten() for i in range(0, len(subset.columns), 5)]
    for row in rows:
        last_column_values.append(subset.iloc[0, -1])

# Creating a new DataFrame with reshaped data
num_rows = len(reshaped_data)
num_columns = len(reshaped_data[0])

# Ensure that the last column values are aligned with the reshaped data
last_column_values = last_column_values[:num_rows]

new_df = pd.DataFrame(reshaped_data, columns=[f'col_{i}' for i in range(num_columns)])

# Adding the last column with values from the first row of deleted columns
new_df['col_21'] = last_column_values

# Create a list of month names and years
months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December']
years = [2019, 2020, 2021, 2022]

# Initialize lists to store month and year values for each row
month_values = []
year_values = []

# Calculate the starting index for the month and year
start_month_index = 4  # May is index 4 in the months list
start_year_index = 2019

# Populate the lists with month and year values
for i in range(len(new_df)):
    month_index = (start_month_index + i) % 12  # Calculate the month
```

```
    index (0-11)
67      year_index = start_year_index + (start_month_index + i) // 12  #
    Calculate the year
68      month_values.append(months[month_index])
69      year_values.append(year_index)
70
71 # Insert the month and year columns at the beginning of the DataFrame
72 new_df.insert(0, 'Month', month_values)
73 new_df.insert(1, 'Year', year_values)
74
75 # Encode the month and year columns as categorical variables
76 new_df['Month'] = pd.Categorical(new_df['Month'], categories=months,
    ordered=True)
77 new_df['Year'] = pd.Categorical(new_df['Year'], categories=years, ordered=
    True)
78
79 # Convert categorical variables to dummy variables
80 new_df = pd.get_dummies(new_df, columns=['Month', 'Year'])
81
82 new_dff = new_df.iloc[:, -16:].join(new_df.iloc[:, :21])
83
84 from sklearn.model_selection import train_test_split
85 import xgboost as xgb
86 from sklearn.metrics import mean_squared_error
87 from sklearn.model_selection import GridSearchCV, KFold
88
89 # Splitting the data into features (X) and target variable (y)
90 X = new_dff.iloc[:, :37]
91 y = new_dff['col_21']
92
93 # Splitting the data into train and test sets (80% train, 20% test)
94 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
95
96 # Define the XGBoost model
97 xgb_model = xgb.XGBRegressor()
98
99 # Define the parameters grid
100 param_grid = {
101     'n_estimators': [50, 100, 150],
102     'max_depth': [5, 10, 15],
103     'learning_rate': [0.1, 0.01, 0.001],
104     'gamma': [0, 0.1, 0.2],
105     'subsample': [0.8, 0.9],
106     'colsample_bytree': [0.8, 0.9],
107     'reg_alpha': [0, 0.1, 0.5],
```

```
    'reg_lambda': [1, 2, 5]
}

# Define cross-validation strategy
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Perform GridSearchCV
grid_search = GridSearchCV(estimator=xgb_model, param_grid=param_grid, cv=
    kf, scoring='neg_mean_squared_error', n_jobs=-1)
grid_search.fit(X_train, y_train)

# Get the best parameters
best_params = grid_search.best_params_
print("Best Parameters:", best_params)

# Train a new model using the best parameters
best_xgb_model = xgb.XGBRegressor(**best_params, random_state=42)
best_xgb_model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = best_xgb_model.predict(X_test)

# Calculating the Mean Squared Error (MSE) on the test data
mse_pred = mean_squared_error(y_test, y_pred)
rmse_pred = np.sqrt(mse_pred)
print(f"Root Mean Squared Error on Test Data: {rmse_pred}")

# Calculating the Mean Squared Error (MSE) from Lacasa
y_pred_lacasa = df[df['product'] == 'Product 1']
y_pred_lacasa = y_pred_lacasa[y_pred_lacasa['cod_con'] == 'VPRD']
#Take the test data
y_pred_lacasa = y_pred_lacasa.iloc[:, [37+6, 24+6, 25+6, 36+6, 34+6, 40+6,
    4+6, 12+6, 8+6]]
y_pred_lacasa = y_pred_lacasa.values.flatten()
mse_lacasa= mean_squared_error(y_test,y_pred_lacasa)
rmse_lacasa = np.sqrt(mse_lacasa)
print(f"Root Mean Squared Error of Lacasa: {rmse_lacasa}")

import matplotlib.pyplot as plt

# Plotting real values vs predicted values
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred, color='blue', label='XGBoost')
plt.scatter(y_test, y_pred_lacasa, color='green', label='Lacasa')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--'
    , lw=2, color='red', label='Ideal Prediction')
```

```
plt.xlabel('Actual Units Sold',fontsize=16, labelpad= 13)
plt.ylabel('Predicted Units',fontsize=16, labelpad =13)
plt.yticks(fontsize=16)
plt.xticks(fontsize=16)
plt.legend(fontsize=14)
plt.grid(False)
plt.title('product 1, cluster 1', fontsize=16)
plt.tick_params(axis='x', pad=10)
plt.tick_params(axis='y', pad=10)
plt.show()
```

Listing 3: XGBoost Algorithm Training