University of Padova

Department of Department of Mathematics

Master Thesis in Big Data Management and Analytics

Cyclic-time granularity based decomposition approach for multi-seasonal time-series and its use on a data management and analysis backbone for Peruvian data about exports

Supervisor
PhD. Mariangela Guidolin
University of Padova

Co-supervisor
PhD. Alberto Abelló
Universidad Politécnica de Cataluña

Master Candidate
Sergio Postigo

Academic Year
2022-2023

Dedication.

A thesis, like any important endeavor in life, is a work that demands effort but especially consistency and dedication, as it requires many months of commitment. On certain days the process flows smoothly, while on others, obstacles arise. In the middle, there are young students and ultimately people trying their best, while also dealing with their particular lives. In my case, this process wouldn't have been possible without the support of my family and I want to dedicate this thesis to them. To my father for teaching me the value of hard work and that effort, in the end, always pays. To my mother for teaching me to pursue excellence in every life task, understanding that excellence isn't synonymous with perfection, but with the best version of oneself. To my brother for consistently listening to me through my highs and lows and being my role model of how to be a good professional and above all, a genuinely good person.

I would like to also mention my country. When choosing the topic for my thesis, I wanted to focus on something related to Peru, and I finally found the ideal project. Unlike more developed countries, the best opportunities in Peru are reserved for people with good economic positions. Usually, economic hierarchy is more relevant than meritocracy. I had the opportunity to study these last two years in Europe and therefore, as a privileged Peruvian, I felt the responsibility to make something that could benefit my country. Therefore, this thesis is also dedicated to Peru, not to any particular person, but to my homeland.

Abstract

The export of products constitutes 29.3% of Peru's GDP [1]. Therefore, a significant portion of the country's economy relies on this activity. SUNAT is the official customs agency in the country and every week, they release batches of data that contain the registers of products exported. While these data are very rich and can derive insightful conclusions for all the stakeholders, their use demand certain expertise that is not available to everyone. This work proposes the implementation of a data management and analytics pipeline that consumes these data and allows to generate forecasting models for some product categories. Specifically, exports of vegetables and fruits will be addressed. Two time series will be analyzed and forecasted for each category: the price per kilogram and the net weight exported.

Exports of vegetables and fruits are influenced by several calendar factors and as such, the time series of these products are multi-seasonal or complex-seasonal, meaning that they contain multiple seasonal patterns. This thesis proposes an approach to detect what seasonalities exist in the data, decompose the time series accordingly, and ultimately use the extracted components in forecasting models. A method called GMST decomposition (Granularity-based multi-seasonal trend decomposition) is proposed, as it uses the concept of cyclic-time granularities to define seasonalities. The results show that using the components extracted with this method tends to improve the accuracy of Facebook's Prophet model.

This work has two main contributions: the first one is the creation of forecasting models for each product's category that allow to make predictions useful for stakeholders in the Peruvian exports industry. This is possible through the second goal, which is the implementation of a methodology to address multi-seasonal time series.

Contents

Listing of figures

Listing of tables

Listing of acronyms

ACF . . . . . Autocorrelation Function

EDA . . . . . Exploratory Data Analysis

ETL . . . . . Extract, Transform and Load

EZ . . . . . Exploitation Zone

FFT . . . . . Fast Fourier Transform

FZ . . . . . Formatted Zone

GAM . . . . . Generalised Additive Models

GDP . . . . . Gross Domestic Product

GMST . . . . . Granularity-based multi-seasonal trend decomposition

HDFS . . . . . Hadoop Distributed Filesystem

JSD . . . . . Jensen-Shannon Distance

MAE . . . . . Mean Absolute Error

NQT . . . . . Normal Quantile Transform

PZ . . . . . Persistent Zone

SUNAT . . . . . National Superintendence of Customs and Tax Administration

TLZ . . . . . Temporal Landing Zone

USD . . . . . United States Dollar

1 Introduction

Since the 1990s, Peru's Gross Domestic Product has displayed a rising trend, coinciding with the implementation of free-market policies and global integration [3]. The exports of products play a leading role in that regard. The Peruvian territory, characterized by complex and diverse landscapes, elevations, and climatic conditions, provides optimal environments for a wide range of unique and highly demanded products worldwide. The export of a single product's category can impact many different stakeholders, from farmers to exporting companies.

SUNAT is the official customs entity in Peru and as such, they record every single product that is exported from the country [4]. Each register includes very rich information like the category, weight, or price of the good exported. Every week, SUNAT releases a new batch of data with new registers of exports. Many useful conclusions for the different stakeholders could be derived from the analysis of these data. However, this demands the implementation of a data management and analytics pipeline that of course implies an expertise that is not accessible to everyone. This was the motivation for creating a tool that consumes these data and that provides relevant metrics that can improve the stakeholders' activities. The decision was made to focus on the registers of vegetable and fruit exports, given that these encompass a wide range of categories yet share features that can be managed in a consistent manner.

Using the different attributes of the registers, time series can be built for each product's category. As it will be explained in Chapter 2, time series describing the product's price per kilogram and the total exported weight will be addressed in this thesis. Creating forecasts for these time series will be the final goal. An initial EDA revealed that these time series tend to

be multi-seasonal, meaning they have different seasonal patterns (yearly, monthly, weekly, etc). Figure 1.1 shows some examples of these time series where seasonal patterns can be observed.

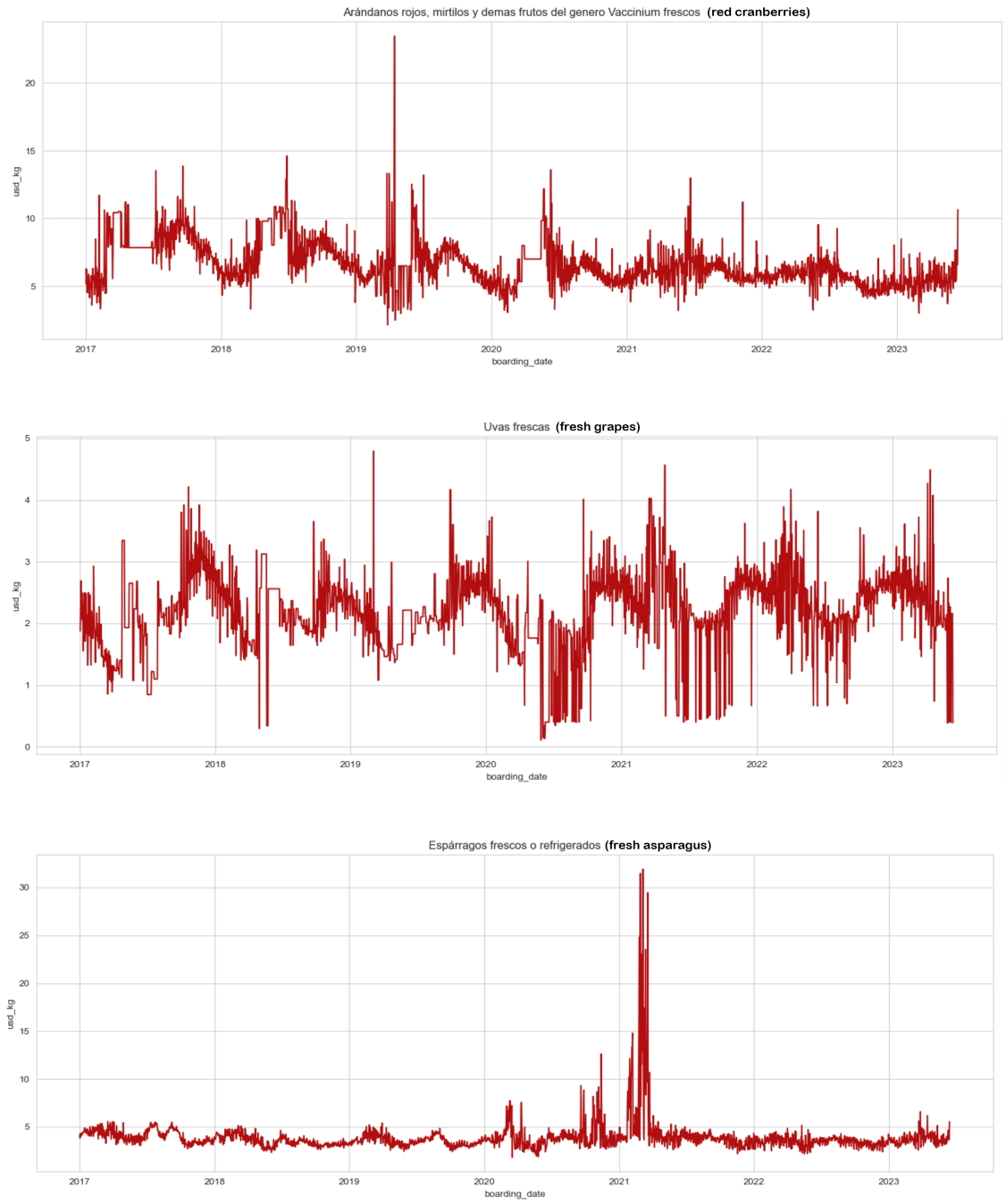

Figure 1.1: Examples of time-series built with the data displaying seasonal patterns.

The existence of multiple seasonal patterns within these time series makes sense considering that there are several calendar aspects that can influence them like weather or economic factors. For example, wine grapes tend to develop properly in environments with temperatures from 11° to 18° C [5], which depending on each country happens in a specific part of the year. Likewise, the overlapping periods of a product's availability in different countries can make international buyers opt for the most cost-effective option. Assessing which seasonality patterns exist in the data is then very important to understand the underlying dynamics of the time series and ultimately to improve forecasts. The multi-seasonal or complex-seasonal characteristic of the data demands the use of a special treatment, which will become the research core of this thesis.

In Data Science, the extraction of rich features from the data plays a key role in enhancing the performance and accuracy of models. Yet, a good feature is valuable not only for its potential to improve the model's predictions but also for its capacity to increase the overall explainability. Time series analysis is a clear example of this. The classical decomposition approach that splits a time series in trend-cycle, seasonality, and reminder [6] is a way to get features from the data that are easily interpreted and that can help to explain predictions' results. The trend-cycle and remainder are always single patterns, while the seasonality component can gather one or multiple seasonality patterns. Knowing what seasonality patterns exist in a time series isn't trivial, as this is highly dependent on the context of the data. This is important as many forecasting models ask the user to indicate the seasonality patterns that the data exhibits.

To detect what seasonality patterns exist in a time series, it is important to know *what* to look for. The remarkable work of Gupta et al. [2] introduce a methodology to test the existence of specific seasonality patterns using a statistical approach. Some improvements for this methodology are proposed in this thesis. Once, the seasonalities that exist in a time series are known, approaches to decompose it must be addressed. Unlike classical decomposition strategies, this thesis introduces a novel approach for decomposing a multi-seasonal time series into all the detected seasonality patterns and the trend-cycle. Afterward, proposals on how to integrate the extracted features into different forecasting models will be presented. Ultimately, an examination will be conducted to determine whether the extracted features positively influence a specific model's performance.

The structure of this thesis will include the implementation of a data management and analytics pipeline for the Peruvian exports data. The analytics block will include the complex seasonality decomposition and modeling strategies. The organization of this report goes as follows: Details of the data to use are presented in Chapter 2. The design and implementation

of the data management pipeline or backbone will be presented in Chapter 3. The complex seasonality analysis for time series will be introduced in Chapter 4. Then, the data analysis pipeline that consumes the implementations from the previous chapter will be showcased in Chapter 5. Finally, the concluding remarks are reported in Chapter 6.

The thesis-related code and implementations can be obtained from the repository `https://github.com/sergiopostigo/supertrade`.

2 The data

In this chapter, the general details of the data to use will be presented. As it was mentioned, time series will be built from these data and as such, the metrics that these series will describe are also defined here.

2.1 Details

This thesis aims to analyze data about the exportation of vegetables and fruits from Peru. These data are obtained from the official customs entity website in Peru [7]. The data of exports can be downloaded in weekly batches by making a get request to the following endpoint:

http://www.aduanet.gob.pe/aduanas/informae/xddDDmmYY.zip

The prefix xddDDmmYY determines the specific batch of data to download and it's defined as follows:

- *x*: refers to exports
- *dd*: refers to the date of the first day of the week
- *DD*: refers to the date of the last day of the week
- *mm*: refers to the month of the last day of the week
- *YY*: refers to the year of the last day of the week

Attribute	Details
CADU	customs office code
FANO	shipping order numbering date
NDCL	shipping order number
FNUM	shipping order numbering date
FEMB	shipping date
FECH_RECEP	declaration reception date
NDCLREG	export declaration number
FREG	regularization date
FANOREG	regularization year of export
CAGE	code of customs agent
TDOC	exporter's document type
NDOC	exporter's document number
DNOMBRE	company name of the exporter
CPAIDES	destination country code
CPUEDES	destination port code
CVIATRA	form of transportation code
CUNITRA	transportation unit code
CEMPTRA	transportation company code
DMAT	ship id
NCON	knowledge number
CENTFIN	financing entity code
CALM	warehouse code
DNOMPRO	name of supplier
DDIRPRO	address of supplier
DK	remote dispatch indicator for O.E
DK2	remote dispatch indicator for DUE
NSER	serial number
PART_NANDI	heading code
DCOM	commercial description 1
DMER2	commercial description 2
DMER3	commercial description 3
DMER4	commercial description 4
DMER5	commercial description 5
CEST	state of goods
VFOBSERDOL	value of goods at the time of board
VPESNET	net weight of goods
VPESBRU	gross weight of goods
QUNIFIS	amount exported in measurement units
TUNIFIS	measurement unit
QUNICOM	commercial measurement unit amount
TUNICOM	amount exported in commercial measurement units
UBIGEO	geographic location code
DNOMCON	name of buyer
DDIRCON	address of buyer

Table 2.1: Exports' data attributes

For example, the prefix x01070523 refers to the batch of data from May 1st to 7th, 2023. dd is always a Monday and DD is always a Sunday. Each batch includes a zipped folder with a .dbf file inside. Every new week, a new batch of data will have to be downloaded. As Table 2.1 shows, the exports' data is composed of 49 attributes. Some that are worth highlighting are PART_NANDI which contains a heading code to categorize each export depending on the type of product that is being traded (e.g. grapes, potatoes, etc), FEMB which denotes the shipment date of the good, VPESNET, which contains the net weight of the good and VFOB-SERDOL that contains the declared value of the goods exported in USD.

A preliminary EDA was performed, downloading all batches of data starting from February 2017. A plot of the number of exports published by week is shown in Figure 2.1. It is important to highlight that the graph shows an increasing trend, which suggests that the number of weekly exports will tend to grow. Additionally, the count of different product categories or headings, represented by the PART_NANDI attribute, is 6511.

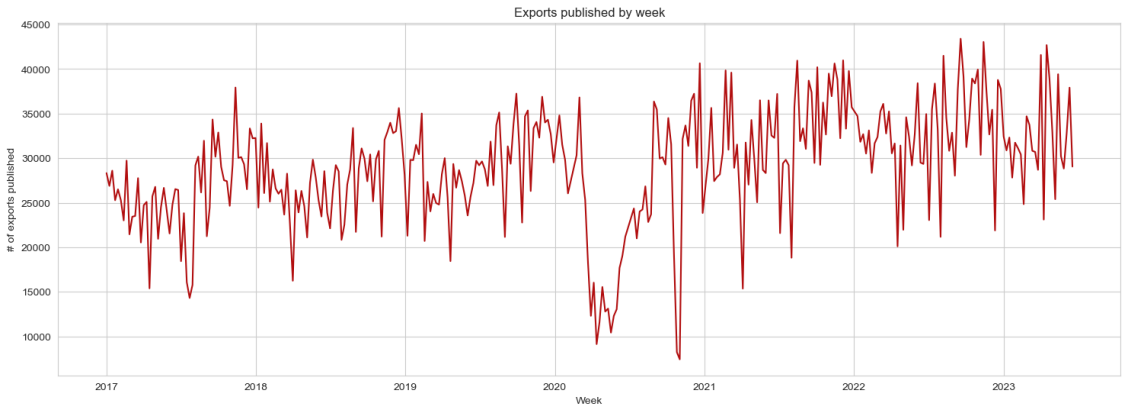

Figure 2.1: Exports per week

Another interesting observation is that each published weekly batch not only includes data from the corresponding week but also incorporates information from previous weeks, months, and even years, effectively filling in any missing gaps. For example, Figure 2.2 shows the most recent batch's data, denoting the number of exports per boarding date (FEMB attribute). As one can see, there are old exports, back to 5 years before. Of course, the majority of exports in the batch are closer to the actual corresponding week.

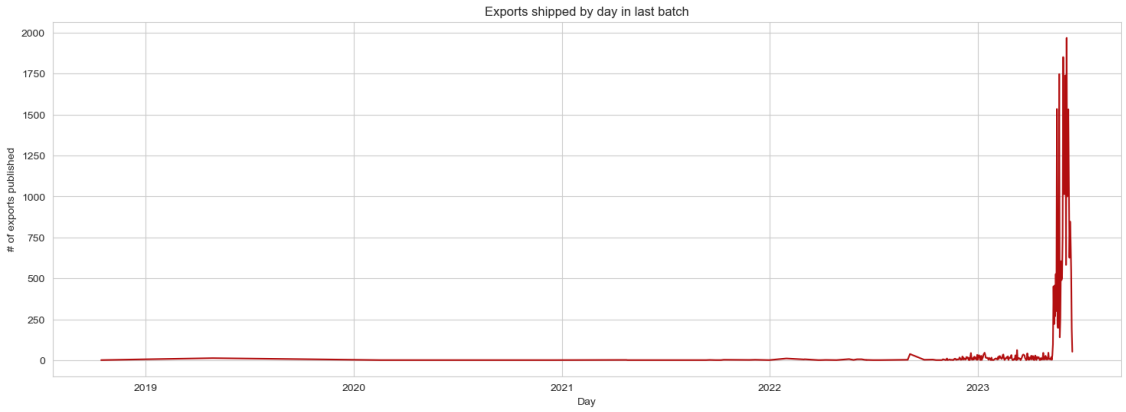

Figure 2.2: Exports shipped by day in the most recent batch

Attribute	Details
PARTIDA	heading code
DESCRIP	product's category description
...	associated taxes

Table 2.2: Headings' table attributes

Besides the exports data, an additional table containing the heading codes' details along with associated product-specific taxes will also be consumed [8], as it is presented in Table 2.2. Both tables can be related using the PART_NANDI and PARTIDA respectively.

2.2 Time-series

Given the data described in the previous section, it is necessary to define what time series can be built. The idea is to create something useful and relevant for the export industry. To achieve this, the approach was placing oneself in the position of the exporting companies and contemplating what metrics would be beneficial for their needs.

The variability in a product's price per kilogram is relevant as it allows companies to plan their cultivation activities and ensure for example, that their products are ready for shipments during specific weeks when prices reach their maximum. This proactive approach can maximize their profits and market opportunities. Another interesting metric is the total exported

weight of a product. For instance, analyzing daily exported weights might allow a company to anticipate future requirements and plan its production accordingly.

It was decided then to create a time series that represent these two metrics. The specific attributes to do so and the approach to follow will be described in Chapter 4.

3

Big Data Management Backbone

This chapter will address the design and implementation of the Big Data Management Backbone. It first describes the fundamentals of the methodology to use and then details each block of the pipeline.

3.1 Fundamentals

The Big Data Management Backbone is a methodology taught by the Polytechnic University of Catalonia (Universitat Politècnica de Catalunya) to create pipelines for big data lifecycles, specifically starting the collection of the data from the sources till the exposition of project-oriented datasets ready to be consumed by data scientists [9]. A schema of the backbone is shown in Figure 3.1 *.

Figure 3.1: Big Data Management Backbone schema.

It is composed of a series of layers through which the data flows sequentially. Some layers are used to store the data and some others are used to extract, transform and/or load the data into

*Originally, the backbone does not include the "Exploitation Datasets Generator" layer and shows the Formatted Zone besides the Exploitation Zone. However, for better understanding and homogeneity, it was decided to include it there for this work.

the next layer. For simplicity, the layers in charge of storing the data will be denominated as "storage layers" (light green boxes in Figure 3.1) and the others will be denominated as "transit layers" (black boxes in Figure 3.1). The input of the backbone is the data sources and the output is one or more datasets, each for one specific project.

The goal of this backbone's methodology is to provide a systematic approach to managing and monitoring the data lifecycle for better data governance and to ensure that the later analytical pipelines are ingested with high-quality and reliable data.

The next sections address each of the layers shown in Figure 3.1. The storage layers will be described before their preceding transit layer, as the implementation of the seconds depends on the first ones.

3.2 Temporal Landing Zone

Before collecting the data, it is important to define a short-term storage area, where all the retrieved files can land and from there be ingested to the next storage level in a uniform way. This first storage layer of the Data Management Backbone is called Temporal Landing Zone (TLZ) [9]. It's defined as "temporal" because the data that land here is deleted after the ingestion in the subsequent storage layer. The following subsections will discuss the selection and structure of the TLZ for this project.

3.2.1 Selection

As one can envision, the sole criterion used to choose the storage technology of this layer is that it can handle the size of the collected data. The exports' data, being the largest files, have been observed to reach a maximum size of 45 MB, with a new file released every week. For the initial collection, encompassing data from 2017 to February 2023 spanning 316 weeks, the TLZ needs to be capable of accommodating 45 x 316 MB, which amounts to 14.22 GB[†].

For this thesis, the TLZ will be the local file system where the project will be developed. It is a Solid-State Drive (SSD), with a designated partition of 25 GB allocated specifically for this purpose.

[†]As the headings data is very small compared to the exports data (2.2 MB) and it's released only once a year, it's not significant in the selection of the TLZ.

3.2.2 Structure

The data downloaded will be unzipped and stored temporarily in the local file system in a folder called *data* residing in the designated partition and it is organized as follows:

```
data
├── x
│   ├── xddDDmmYY.dbf
│   └── ...
└── headings
    └── NANDINA.txt (headings information table)
```

As seen, the goal is to structure the data in folders that gather different files of the same category. So, if in the future more data besides exports or headings are included, different folders should be created to allocate them.

3.3 Data Collector

The data retrieval from the sources is the first transit layer in the Data Management Backbone. Once the Temporal Landing Zone is defined and set, the process that brings the data from the sources into that first storage layer can be developed.

In the majority of cases, the data is dynamic rather than static, meaning that it will be regularly updated with new information. As a result, two data collection approaches are required: historical, which involves retrieving all available data prior to the project's starting point, and incremental, which focuses on retrieving data updates. Furthermore, depending on the data and their sources different collection implementations should be followed. Useful questions to ask here are: how often is the data going to be collected? what is the size of the data to be collected? what is the format of the data? how is the data downloaded?

Additionally, it is very important to keep a record of the data collected, for which it is customary to have a log.

The next subsection details the features of the data and its sources that are relevant to choose a collection implementation. Then, the historical and incremental collection approaches of every data category will be presented.

3.3.1 Data sources

In this project, two data categories will be consumed: exports and headings.

exports:

- Source: Website of SUNAT.
- Download approach: GET request to the endpoint, specifying the week
- Format: zipped DBF
- Size: Every file has up to 45 MB
- Frequency of updates: Every week a new file is published

Given all these features, it was decided that the collection of these data will be achieved with Python scripts. The scripts will allow us to make the GET requests to the endpoint to download the zipped files, unzip them and allocate them in the *x* folder of the TLZ. It will run every week to retrieve the updates.

headings:

- Source: Website of SUNAT.
- Download approach: Link-based
- Format: .txt
- Size: a file with 2.2 MB
- Frequency of updates: Every year

Taking these features and especially considering the frequency of updates, it was decided that the collection of this data will be done by a human, who will click the link every year and allocate the downloaded data in the *headings* folder of the TLZ.

3.3.2 Historical Collection

The historical retrieval for every data category is described in the following lines.

3.3.2.1 Exports

All the available weekly files of data starting from January 1st, 2017 to the present day are downloaded and stored in the TLZ. Every weekly batch's download is registered in a log file, detailing the prefix of the batch (xddDDmmYY), the response of the server (200 and 404 for successful and failed downloads respectively), the type of download approach (historical in this case), and the date of download.

Several download tests revealed that the custom's agency web server imposes restrictions on downloads during specific times of the day, delivering only batches of data starting from October 2020 onwards. However, after multiple attempts, it was observed that all the desired data can be downloaded from 7:00 am to 9:00 am (EST). This makes sense considering that at this time the local web traffic in Peru is very low. Certain batches consistently return a 404 code, indicating that they were not downloaded due to the unavailability of the requested source. It is possible that the customs agency did not publish a batch for that particular week. Consequently, the data in the batches includes not only the most recent week but also previous weeks to bridge the gaps caused by missing batches.

The scripts used for this purpose are detailed below:

- *utilities.py*
  - week_code_generator:
    A function that returns all the possible prefixes (xmmDDmmYY) for the batches of data given a start date and an end date.
  - batch_downloader:
    A function that downloads batches of data from the endpoint given a list of prefixes. It stores the data in the Temporal Landing Zone and registers the downloads in the log file

- *historical.py*
  This is the main script of the historical collection. Here, the start date of collection is set and data_collection is invoked to make the downloads.

3.3.2.2 Headings

The collection will be performed by a human, who will download the file and allocate it in the TLZ. Finally, he should register the collection in the log file.

3.3.3 Incremental Collection

3.3.3.1 Exports

The available data that isn't yet downloaded is obtained in this stage and stored in the Temporal Landing Zone. In this particular approach, a query is made in the Persistent Zone (to be

discussed later) to retrieve the prefix of the most recent batch. This prefix includes the end date of that batch, allowing for the subsequent download of all available batches after that specific date. As in the historical collection, every downloaded batch is registered in the log file. In this case, the type of download is shown as 'incremental'.

The scripts used in this stage are detailed below:

- *utilities.py* (same as historical)
- *incremental.py*

  The main script of the incremental collection. It makes a query on the Persistent Zone, asking for the most recent batch date. Subsequently, it initiates the downloads by invoking the data_collection function, setting the start date as the day immediately following the date obtained from the query.

3.3.3.2 Headings

Every year, a human manually downloads the .txt file and allocates it in the TLZ. The collection will be registered in the log file.

3.4 Persistent Zone

As explained before, the TLZ is a short-term storage layer, meaning that the data won't persist there but will be deleted when it is migrated to the second storage layer: the Persistent Zone (PZ). Here, all the data collected will be stored and will stay there for the entire lifetime of the project. As such, it is important to include a temporal track of the data or a versioning approach [9].

Selecting the storage technology for this layer is not trivial, as there are multiple considerations to make. Relevant features that will be used to choose a technology are the data size, data schema, queries, and the need for distribution. Precisely, the following subsections address the project's selection and structure of the technology used for this storage layer.

3.4.1 Selection

It was seen that the data available is in .dbf (exports) and .txt (headings) in tabular format. A fast EDA in the TLZ revealed that every batch of weekly exports data starting from 2017 till

February 2023 has maintained the same attributes. Likewise for the headings data. Nonetheless, there isn't any guarantee that new files will have exactly the same schema. Hence, there is a requirement for a data storage technology with a **flexible schema**.

As it was mentioned before, the historical data collection will retrieve around 14.5 GB of data. Then every week the data size will have an increment of 45 MB, which is a yearly growth of 2.3 GB. While the current data volume may be relatively small, there could be the need of including additional data sources in the future that could potentially improve the final forecasting models (e.g. import data, weather data, economic data, etc). Therefore, it is correct to anticipate future scalability requirements and consider a storage technology that supports sharding and replication. This approach ensures that the system is prepared to handle **data distribution** if the need arises.

While the queries to run over the PZ are not implemented at this stage (they will be described later), their general goal will be to extract and slightly transform the data to load into the Formatted Zone. Given the definition of the project, it can be anticipated that the exports files will be filtered by the heading code, as only those rows referring to vegetables and fruits will be analyzed. Furthermore, not all the 49 columns of these files will be sent to the Formatted Zone, as some are not relevant to the project's purpose. It will be necessary to execute these queries on a weekly basis, as this is the highest frequency at which new data is introduced. Hence, it is desirable for the storage approach to incorporate a structure that enables **efficient data querying**, allowing for selections based on headings and projections to choose specific columns[‡].

Considering all these needs, it was decided that the storage technology will be Hadoop Distributed Filesystem (HDFS) in combination with the Parquet file format. For each new batch of data, a separate Parquet file can be generated. The advantage of this approach is that if a batch contains a previously unseen column, it won't cause any issues: since each file is independent of the others, there is no requirement for all files to have the same columns or the same schema. Furthermore, Parquet files allow the creation of row groups, which serve as a logical horizontal partitioning of data inside a file, and that can increase the efficiency of query selection [10]. Inside a Parquet file, one row group for every heading could be created to improve the reading performance. Furthermore, using HDFS as the storage technology enables the option to use data distribution if the need arises.

In summary, using an HDFS + Parquet configuration will allow schema flexibility, handling large amounts of data with potential data distribution and efficient query approaches.

[‡]Do not confuse the headings codes in the exports' files with the headings' file that gives details about the codes. The headings' file is so small compared to the exports' data that it's irrelevant in this part.

3.4.2 Structure

As the size of data doesn't demand multiple nodes yet, an HDFS Single-Node cluster is used in this project [11]. Furthermore, it is customary to use a Single-Node cluster for the study and test phase of a project [12]. A relevant configuration to mention is that the block size was defined to be 128 KB.

Once the cluster is set up, the following folders hierarchy is defined:

```
thesis
└── peru
    ├── exports
    └── headings
```

As expected, the exports' data (.dbf files) will go to the *exports* folder and the headings' data (.txt file) to the *headings* folder. Of course, these files should be converted into Parquet format, which is done in the Data Persistent Loader.

Every export's file will be converted into Parquet and inside each file, the rows will be grouped according to the headings code. By utilizing row groups, querying based on the heading's code can be optimized by avoiding the need for a lookup operation on every individual row. Instead, the filtering process can be performed by referring to the metadata associated with the row groups.

In regard to the headings' file, as its size is very small, there isn't any significant query-wise improvement for using the Parquet format. However, the file will still be converted into Parquet, but without defining specific row groups.

To keep proper versioning of the data, every added row in the PZ should be enriched with two attributes: BATCH_WEEK indicating the week period to which that data belongs (only for exports) and LOAD_DATE (exports and headings) to indicate the date when the file was ingested into the PZ.

3.5 Data Persistent Loader

Once the PZ is defined and set, one can address the second transit layer. This is used to extract, transform and load the data from the TLZ into the PZ.

As for the Data Collector, it is common to have two approaches for this layer: historical and incremental. It is also crucial to maintain a log file during this stage, which aids in monitoring the data loaded into the PZ. Moreover, this layer enriches the data by incorporating attribute(s) that assist in maintaining versioning and temporal comprehension in the PZ because as it was

mentioned, all the data (even obsolete versions) will persist here. Lastly, upon data transfer, this layer is responsible for overseeing the deletion of the corresponding files in the TZ.

The next subsections will address the implementation of the historical and incremental persistent loaders.

3.5.1 Historical Persistent Loader

This part covers the historical extraction, transformation, and loading of data from the Temporary Landing Zone into the Persistent Zone. The implementation will be described for every data category.

Exports: A .dbf file is extracted from the TLZ and parsed into a Pandas data frame. Then, the data frame is enriched with two extra columns called BATCH_WEEK and LOAD_DATE, which indicate the publication's week to which data belong and the date of transfer to the PZ, respectively. Then, the data frame is converted into a PyArrow table which is then horizontally split by the heading code attribute, creating one subtable for every heading. Then, a Parquet file is created, where every row group is a subtable. According to Apache Parquet's documentation, it is desired that every row group fits entirely inside an HDFS block for an optimized read setup [13]. As in the HDFS setup the blocks were defined to be 128 MB and the maximum size of a .dbf file was 45 MB, it is clear that a row group, even if contained all the rows, will not exceed the HDFS block size, which as explained before is desired according to documentation to improve the reading performance [§]. Finally, the Parquet file is loaded into HDFS into the folder /thesis/peru/exports. This is repeated for every .dbf file in the TLZ. Every transfer is annotated in a log file.

Headings: The .txt file es extracted from the TLZ and parsed into a Pandas data frame. A LOAD_DATE attribute indicating the current date is added. Then, a Parquet file including a single row group with all the rows is created. As the amount of data here is so small (the .txt file contained 2.2 MB), reading optimization isn't critical and therefore, no horizontal partitioning strategy is set here. Finally, the Parquet file is sent to HDFS, into the folder /thesis/peru/headings. The transfer is annotated in a log file.

The scripts used in this layer and their functions are detailed below:

- *utilities.py*

[§]Of course, it could be the case that some row groups are be divided between two HDFS blocks. However, as the number of row groups per Parquet file is high (about 50), a single HDFS block will be able to allocate a large number of row groups. Thus, the presence of a few shared row groups will not pose a concern since the overall reading performance will be improved regardless.

- **exports_ingestion:** Performs the ETL process to transfer an exports' .dfb file from the TLZ into the PZ and registers the transfer in a log file.
- **headings_ingestion:** Performs the ETL process to transfer the headings' .txt file from the TLZ into the PZ and registers the transfer in a log file.

- *historical.py*

  Main script of the historical persistent loading.

  - **load_exports:** Gets the list of all files in the TLZ's *x* folder (.dbf files) and invokes the exports_ingest function per every file. The type of context "historical" is passed as an argument.
  - **load_headings:** Given the path of the headings' .txt file in the *headings* folder of the TLZ, invokes the headings_ingest function. The type of context "historical" is passed as an argument.

3.5.2 Incremental Persistent Loader

The incremental ETL process for transferring data from the TLZ to the PZ is detailed in the following lines. As in the previous subsection, the implementation will be described for every data category. Since a significant portion of the historical approach's implementation is reused here, the specific details will not be reiterated. However, when necessary it will be indicated that they are identical to the previous implementation.

Exports: Same as the historical approach.

Headings: Same as the historical approach.

The scripts used in this stage are detailed below:

- *utilities.py*

  Same as the historical approach.

- *incremental.py*

  Main script of the incremental persistent loading

  - **load_exports:** Same as the historical approach, but the context argument passed to exports_ingest is "incremental".

– **load_exports:** Same as the historical approach, but the context argument passed to headings_ingest is "incremental".

3.6 Formatted Zone

The data within the PZ was ingested "as it is", without undergoing any cleaning steps. While it has a unified format (Parquet), it remains in its raw state. Therefore, it is desired that the next storage layer contains a cleaned version of the data and also a consolidated data model. This new layer is called Formatted Zone (FZ) as it keeps the first "formatted" version of the data. Nevertheless, it is crucial that the cleaning performed in this stage is generic and not tailored to any particular project [9]. In other words, the data should be transformed in a way that makes it suitable for various purposes and can serve as input for multiple scenarios.

For the selection of the storage technology of the FZ, features like the data size, data structure, potential queries, and need for distribution will be taken into consideration. The following subsections address this matter.

3.6.1 Selection

The target of the FZ is to store a multipurpose dataset, ready to be consumed for different projects and objectives. This suggests that ideally, the FZ should contain all the attributes that might be useful for any scenario. However, for this thesis and the general goal, making a cleaning pipeline that includes attributes that won't be used at all will contribute to making a more complete FZ but not enhance the results. As such, the FZ implemented here will include only those attributes that might be useful for the purposes of this thesis. Similarly, only those rows that have information related to fruits and vegetables will be included in the FZ, as the future analysis will focus on only these data. As the amount of data of the headings.parquet file is tiny compared to the total export data, the selection of the technology will be driven considering only the second.

The FZ will gather a significantly smaller amount of data compared to the PZ. The headings of interest (related to vegetables and fruit products) are 174, while the total number of headings is 6511. As a big portion of Peruvian exports belongs to these categories, a comparison in the number of rows was performed, obtaining that 15.8% of the export data in the PZ are

associated with the headings of interest. Furthermore, considering that some columns won't be transferred to the FZ (this will be addressed in the next section), it is clear that there will be a significant reduction in the amount of data. As a result, the distribution of the data is no longer a crucial factor in this stage.

As the data in the TZ is in Parquet format, it is considered structured data. Furthermore, there isn't a need for a flexible schema, since the attributes to consider are known. So, using a fixed schema approach to store these data appears as an appropriate alternative. This will also allow us to set constraints on the data that is uploaded. Therefore, a data model that would be a good fit for the FZ is relational.

Although the specific queries to be executed on the FZ are not yet defined, it is evident that filtering based on headings is expected to occur frequently. Furthermore, since many different operations could be performed over these data when creating the final dataset for analysis (this will be discussed in the Exploitation Zone section), a very rich and well-documented query language will be needed. As the amount of data in this layer will be significantly less than in the previous one, there isn't a requirement for parallel processing.

Based on the reasons outlined above, relational database technology has been selected for the Formatted Zone, and specifically, PostgreSQL has been chosen. This is because PostgreSQL is an extensible, open-source database management system with thorough documentation. Additionally, its widely recognized SQL query language can be easily managed by other users. Furthermore, while there isn't a short-term need for distribution or parallel processing if new attributes and more rows are pipelined to the FZ, PostgreSQL has extensions like Citus, which would allow for distributed configurations [14].

3.6.2 Structure

An intermediate EDA in the PZ helped to choose those attributes that will be transfered to the FZ, both for the exports and the headings data. As such, the PostgreSQL database that will be used as FZ will have two tables: *peru_exports* and *peru_exports_headings*, whose details are presented in Tables 3.2 and 3.1 respectively.

For simplicity, this project utilized a Docker container featuring a PostgreSQL image. The database was named *formatted*. In order to create the tables, a script with the name *formatted_setup.py* was developed. Additionally, this script consumes a SQL file with the name *formatted.sql*. Both files are in the *database_settings* folder of the project.

Attribute	Description	Originally
HEADING	Product's category heading code	0
DESCRIPTION	Heading's description	1
MAPPED_TO	To map a heading's code to another	didn't exist before

Table 3.1: peru_exports_headings table details

Attribute	Description	Originally
HEADING	Product's category heading code	PART_NANDI
EXP_ID	Exporter's tax ID	NDOC
NET_WEIGHT	Net weight of exported good in kg	VPESNET
GROSS_WEIGHT	Gross weight of exported good in kg	VPESBRU
VALUE_USD	Value of good when boarding in USD	VFOBSERDOL
COUNTRY	Destination country code	CPAIDES
BOARDING_DATE	Boarding date	FEMB
DESCRIPTION	Description about the exported good	DCOM, DMER2, DMER3, DMER4 and DMER5
BATCH_WEEK	week of publication of the batch	BATCH_WEEK

Table 3.2: peru_exports table details

3.7 Data Formatter

The third transit layer manages the ETL process for moving data from the PZ into the FZ and it is called Data Formatter. The transformations performed in this stage correspond to a generic cleaning of the data, without targeting it to any specific project.

As it was commented before, given the nature of this thesis and its goals, making a generic cleaning of the data that includes the assessment of all the attributes is not worth it. While this would be ideal to have a more complete FZ, in this thesis only some of the attributes will be used and as such, it is better to focus only on them. Therefore, a semi-generic cleaning will be implemented. The idea is to have a multipurpose dataset in the FZ but target the time-series research.

The two classical approaches will be also implemented in this transit layer: historical and incremental. Each of them will be described in the following subsections.

3.7.1 Historical Formatter

This approach introduces two relevant challenges. The first one is the need to move around 15% of the data from the PZ to the FZ and apply transformations in the middle. Originally, the historical amount of data was 14.22 GB, which makes around 2.1 GB to be transferred. Depending on the resources available, saturating the memory is always a risk in these cases and thus, a special strategy needs to be defined[¶]. The second one is that, if this project scales up[‖], the amount of data to store in the PZ may require distribution, and then, a distributed data processing strategy could be employed for parallel processing.

Considering these challenges, it was decided that using a big data processing framework like Apache Spark was the best alternative. Given that the project is developed in Python, the library PySpark (Apache Spark's library for Python) will be used. Spark allows not having to load the entire dataset in memory, as it processes the data in partitions which are smaller subsets. The data in each partition is loaded into memory, processed, and then will be sent to the PostgreSQL database. This process will be performed iteratively for each partition, which allows Spark to handle large datasets without requiring all the data to fit into memory at once. Furthermore, if the need for distribution arises in the PZ, PySpark will facilitate the scalability of the code for parallel processing, allowing it to efficiently handle distributed data with slight changes.

[¶] Especially, if additional headings are included in the future.

[‖] For instance, adding more datasets.

The next lines will address the ETL procedure performed in the FZ. The tables need to be populated in the order that they appear here:

peru_exports_headings: The process for this table is performed in 3 steps.

- Step 1:
  - Get all the possible distinct headings' codes of interest (starting with 07 and 08) from the exports files in the TLZ.
  - Get all the headings' codes and their descriptions attributes from the headings' file in the TLZ.
  - Perform a left join on the heading code attribute. The idea is to have all the distinct headings **of interest** and their descriptions (if available).
  - Group by the heading attribute, concatenating the descriptions in case there are different ones of the same heading. That will be the raw_description attribute.
  - Create an empty curated_description attribute.
  - Generate a .csv file with the result containing columns heading, raw_description, curated_description and mapped_to.

- Step 2 (performed by human**):
  - With the PDF document "Arancel de Aduanas", perform a manual resolution of the data, checking if the raw_description matches the information there.
  - The final and cleaned description must be written in curated_description.
  - It may be the case that some headings in the list did not exist in the PDF document, so they must be mapped to headings that do exist according to the raw_description. The heading to which they are mapped is defined in mapped_to.

- Step 3:
  - Taking the .csv file created, send the data to the table peru_exports_headings in PostgreSQL. The attributes are heading, description and mapped_to

**This human-in-the-loop step is necessary only if a new batch of data has unseen headings. So far, since the very first resolution, there hasn't been the need to perform this step again (after about 12 updates). Furthermore, if an unseen heading appears, it will be only assessed once and never again.

The scripts employed in the aforementioned process are located in the folder *data_formatter/headings/historical* and are:

- *historical_1.py*
  Performs step 1.
- *historical_2.py*
  Performs step 3.

peru_exports: The approach for this table is implemented as follows:

- Filter those rows whose headings (PART_NANDI) start with 07 and 08.
- Concatenate the description attributes (DCOM, DMER2, DMER3, DMER4 and DMER5) into a single one called description.
- Rename the attributes of interest to be easy understandable.
- Finally take the following attributes and send them to the table PostgreSQL: heading, exp_id, net_weight, gross_weight, value_usd, country, boarding_date, description, batch_week.

The script performing the above tasks is *historical.py* and it is located in *data_formatter/exports/historical*.

3.7.2 Incremental Formatter

As in the historical version, PySpark will be the main tool here. As the incremental approach is very similar to the historical one, no reiterative details about the implementation will be provided but just the differences. Again, this will be described by table in the FZ.

peru_exports_headings: Same approach as in the historical formatter with a slight variation in step 1: Take the headings' codes of interest from the PZ that **do not exist** in the FZ yet. This way the entity resolution process is performed only for new headings (if any). If there is no new heading's code of interest in a recently added batch of data in the PZ, the procedure ends, as there are no updates to do in the FZ.

Located in the folder *data_formatter/headings/incremental*, the scripts used for the above process are:

- *incremental_1.py*
  Performs step 1 from the previously defined procedure (taking only new headings).

- *incremental_2.py*
  Performs step 3 from the previously defined procedure

peru_exports: Same approach as the historical with a variation. The most recent BATCH_WEEK code attribute is taken from the table in the FZ. Then, only data with the attribute later to that week is loaded.

The script performing this process is *historical.py* and its location has been designated to the folder *data_formatter/exports/incremental*.

3.8 Exploitation Zone

The FZ has a generically cleaned version of the data that is project agnostic and can be used for multiple purposes. Once a project is defined, target-oriented datasets can be generated from there. The layer containing these datasets is called Exploitation Zone (EZ) [9]. This is the last storage layer of the Data Management Backbone and serves as the input for analytics projects. Therefore, this layer is exposed to either internal data scientists or external tools that leverage the data for analysis [9].

The dataset built for a project can be generated in different data models depending on the specific needs. Tensors, relations or dataframes are commonly used [9]. Accordingly, a selection of the storage technology and the structure of the layer must be targeted. The next subsections address these matters.

3.8.1 Selection

The analytical tasks that will be described in future sections will use time series, each of which will be generated based on the heading's code[††]. Accordingly, a dataset that facilitates this task should be generated.

The FZ contains two tables from which the dataset should be generated. However, not all the data from these tables will be needed and this arises from the "business" specific knowledge of the project, as it will be described in the next section.

The consequent reduction of the data and the fact that the FZ's tables are in PostgreSQL led to the decision of implementing the dataset of the EZ as a view in the same database[‡‡]. This introduces advantages: The first one is that there won't be any need of implementing an

[††]Per heading, one or more time-series about different metrics will be generated
[‡‡]Note that a dataset could be formed from more than one view

Attribute	Description
HEADING	Product's category heading code
EXP_ID	Exporter's tax ID
NET_WEIGHT	Net weight of exported good in kg
GROSS_WEIGHT	Gross weight of exported good in kg
VALUE_USD	Value of good when boarding in USD
COUNTRY	Destination country code
BOARDING_DATE	Boarding date
DESCRIPTION	Description about the exported good
BATCH_WEEK	week of publication of the batch

Table 3.3: peru_exports_ts view details

incremental approach, as the view will be generated on demand. Second, removing the need and overhead of adding another storage technology to the pipeline. And finally, the view can be created and queried using SQL, as it can be treated as a normal table.

3.8.2 Structure

The EZ will contain a view with the dataset that will combine the data from *peru_exports* and *peru_exports_headings*. The name designated for this view will be *peru_exports_ts*. It will be structured as it is presented in Table 3.3.

3.9 Exploitation Datasets Generator

This section addresses the generation of the dataset in the EZ, based on the data available in the FZ. As it was stated before, for this project the EZ will reside in the same database as the FZ, but the dataset will be a view. This will remove the need of performing historical or incremental implementations, as the view can be generated on demand. The rationale behind this choice is due to the view's intended use on a weekly basis for analysis and model generation. There are no great advantages in persisting it within the database, as it would only result in unnecessary consumption of storage space. The view's structure was presented in Section 3.8.2.

To create the view, some business-specific details need to be addressed. They will define the

way this dataset will be generated. These are described below:

- Within the *peru_exports* table in the FZ, certain rows contain values in the net_weight column ranging between 0 and 100kg. Initially, it was uncertain whether this data should be considered as noisy. However, after conducting thorough research, it was discovered that this is the result of a common practice in the export of vegetables and fruits. Exporters typically send samples to potential clients before dispatching larger batches. Consequently, rows with low net weights are highly likely to represent these samples. Therefore, it has been decided not to include them in this research, since the focus is on the actual exports. A threshold must be set for this.

- Similarly, some rows in *peru_exports* contained very low values in the value_usd column ranging between 0 and 200 USD. Companies exporting goods usually send batches of thousands of dollars. It can be that during the registration the person adding the row forgot to add some digits or also that these rows correspond to samples. Therefore, it was decided not to include them in the analysis. Accordingly, a threshold must be set.

- Furthermore, it was seen that most of the data is concentrated from 2017 onwards. There are very few observations before 2017. So, the research will be done by taking data starting from 2017.

- Finally, it was seen that some headings have very few observations in *peru_exports*. Since the idea is to create time series by headings, we need them to have a considerable big amount of observations. Therefore, a threshold for this should also be set for the analysis.

Given all these considerations, the data from the tables *peru_exports* and *peru_exports_headings* is combined to generate the final dataset (represented as a view) as follows:

- Consider only those rows from peru_exports with a net_weight >= 100 kg, value_usd over 200 USD, boarding_date starting from 2017, and only headings with more than 1000 observations.

- Join with the peru_exports_headings table to have the heading descriptions. Map headings if the column mapped_to specifies it.

The script in charge of running this process is called *dataset_generator.py* and it is located in the folder *data_exploitation/datasets/time_series_analysis*. There, the filters above described are set and then a .sql file is called to be run in the FZ. That .sql file is in the folder *data_exploitation/datasets/time_series_analysis/queries*

4

Complex Seasonality Analysis

This chapter covers the complex seasonality analysis for time-series. First, a detailed description of seasonalities and their representation with cyclic-time granularities is showcased. The following section covers the state of the art regarding the use of cyclic-time granularities for seasonalities extraction. The last section introduces a series of functionalities for the decomposition a time-series using the concept of cyclic-time granularities.

4.1 A deeper look into seasonalities

Time is a continuous progression of events from the past, through the present to the future [15]. As continuous, one can't define an indivisible unit of time. Therefore, it is measured by dividing it into sequential chunks as shown in Figure 4.1. This way, time can be represented as a succession of discrete elements, each of which contains a continuous progression of events.

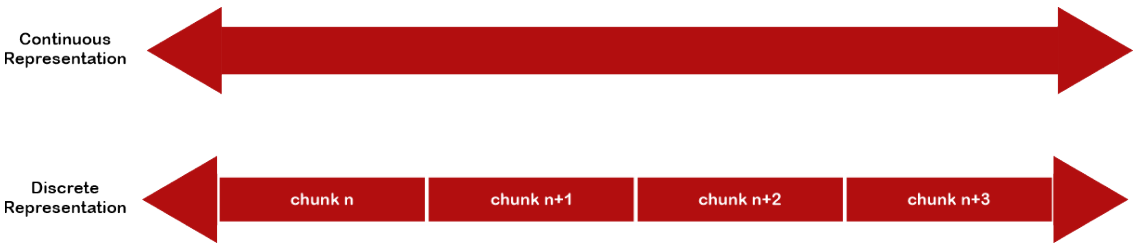

Figure 4.1: Time representations.

According to Gupta et al., [2], a specific deconstruction of time into chunks is defined as time granularity. Seconds, hours, days, or weeks are examples of time granularities. They differ between them in the length of their chunks as it is shown in Figure 4.2. Note that every chunk or time granule has a unique label, with digits being the most suitable option for highlighting them sequentially.

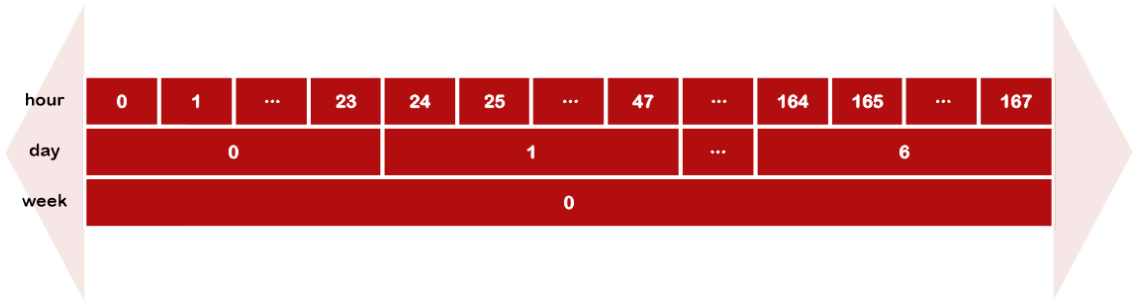

Figure 4.2: Examples of time granularities, inspired from Gutpta et al. work [2].

It is essential to mention that the chunks within a time granularity don't need to be equally spaced. For example, one could define a "weekday" time granularity where Mondays to Fridays are grouped in a chunk and Saturdays to Sundays are grouped in different ones, as shown in Figure 4.3

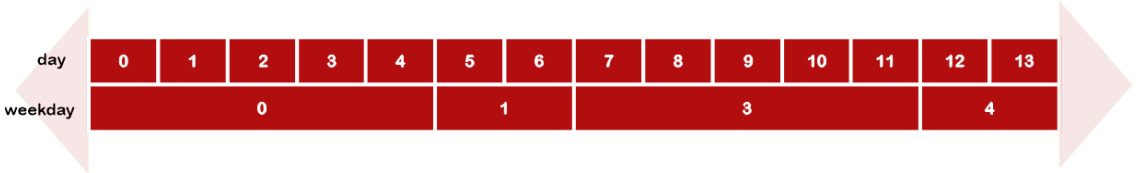

Figure 4.3: The "weekday" time granularity .

In general, one can define any type of time granularity by choosing smaller or larger chunks. Naturally, this decision relies on specific requirements and objectives, such as astronomical considerations, business demands, and so on. For example, hourly time granularity is related to the movement of the Earth. The weekday granularity can be useful for businesses as it splits the time into working and non-working chunks, proving advantageous for scheduling and operational purposes.

Gupta et al. [2] identify two types of time granularities: **linear time granularities**, which are the ones seen so far dividing the time in chunks with unique labels. The second type is denominated **cyclic time granularities**. These are the ones most commonly used and correspond to combinations of two linear time granularities. For instance, the day-in-week cyclic

time granularity shown in Figure 4.4 is built from day and week linear time granularities. Day-in-month or month-in-year granularities are further examples. In the cyclic time granularities, the labels of the chunks are repeated every certain period as shown in Figure 4.4. The day Monday (labeled as 0) happens every 7 days or every week, the month of January happens every 12 months or every year, etc. An important remark is that cyclic-time granularities can have regular or irregular periods. There are always 7 days in a week (day-in-week has a regular period), but some months may have 28,29,30, or 31 days (day-in-month has an irregular period).

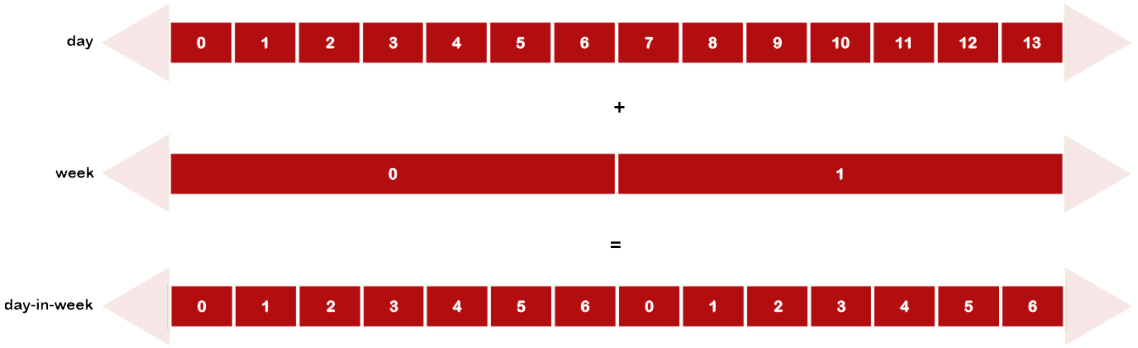

Figure 4.4: day-in-week granularity .

Incorporate the idea of cyclic time granularities within the time-series field. A seasonality in a time series is not more than a pattern within the chunk of a cyclic time granularity that repeats every same label. For instance, in Figure 4.5, there is a seasonal pattern repeating every 0-labeled chunk.

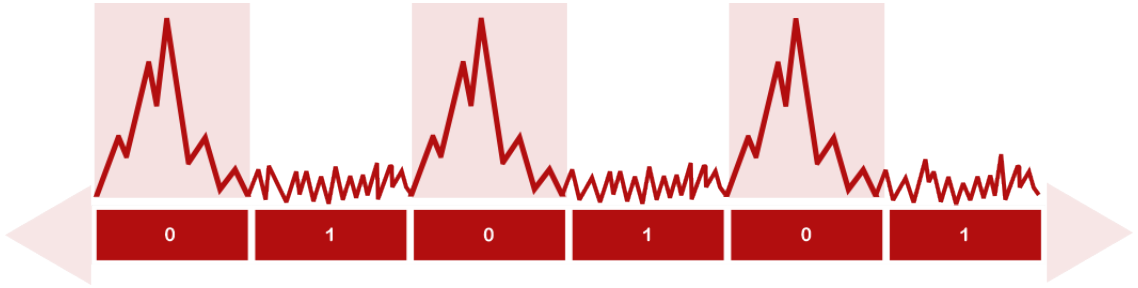

Figure 4.5: Seasonality pattern in a time granularity .

In many time-series related scenarios, it is common to hear the terms "yearly", "monthly" or "weekly" seasonalities, referring to seasonalities found in month-in-year, day-in-month, and day-in-week cyclic time granularities respectively. That is, if a data scientist is asked to check if a time series has a yearly seasonality, he would try to find patterns that repeat every 12 months.

However, it would be an oversimplified perspective to solely link the term "year" to month-in-year, as one could also find seasonality patterns in time granularities such as semester-in-year, week-in-year, or quarter-in-year. Therefore, when finding seasonalities in a time series it would be useful to extend the search using multiple cyclic time granularities. Of course, the time granularities to analyze must make sense with the context of the time series. For instance, consider a time series that records the daily sales of a shop operating exclusively from Monday to Friday: It wouldn't make sense to search a seasonality pattern in a weekday-in-week time granularity, as the shop does not have sales on the weekends.

Given the overall discussion, there are four questions that need to be addressed:

- How to implement a custom cyclic-time granularity labeled series?
- How to assess if a time series has a specific cyclic-time granularity's seasonality?
- How to obtain specific cyclic-time granularities' seasonal patterns from a time series?
- How to include extracted seasonal patterns in forecasting models?

The next subsection will explore existing proposals and solutions that target partially or completely the above-defined questions.

4.2 State of the Art

It is understood as an implementation of a cyclic-time granularity the labeling approach of a series representing time. To illustrate, consider a time series consisting of daily data spanning a period of 14 days and suppose the available data starts on a Wednesday. The implementation of a day-in-week granularity is shown in Figure 4.6. Here, the first day of the week is labeled as 1 and the last as 7. Similarly, a weekend-in-week granularity could be implemented by labeling the working days as 0 and the weekend days as 1, as shown in Figure 4.7

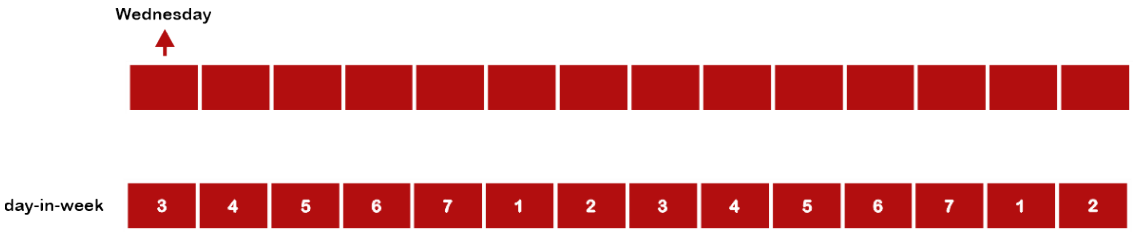

Figure 4.6: day-in-week labeling .

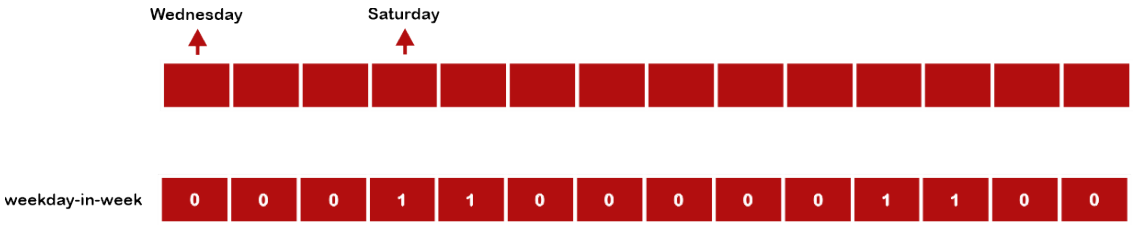

Figure 4.7: weekday-in-week labeling .

To achieve this, Gupta et al. [2] propose three approaches. All of them start by indexing all the available time points*. That is, the first time point has an index of 0, the second has an index of 1, and so on. Then, one of three formulas is applied, depending on the period of the desired cyclic-time granularity. For regular periods (called circular granularities by Gupta et al. [2]):

$$C_{G,H}(z) = [z/P(B,G)] \bmod P(G,H) \quad \forall z \in \mathbb{Z}_{\geq 0} \tag{4.1}$$

Where $C_{G,C}(z)$ denotes the label given to time point with index z when implementing a *G-in-C* granularity. As the original time points do not necessarily are represented in the linear granularity G, but in B, an initial conversion from B to G is needed. For example, if there is daily data available and week-in-year granularity is needed, the data needs to first be mapped to weeks. $P(B,G)$ indicates the period or how many chunks of B exist in a single chunk of C. For irregular granularities (called quasi-circular granularities by Gupta et al. [2]), they propose a more complex formula that takes into account the different periods that the chosen granularity can have. Also, they consider the period behind the irregularity of the period. For example, to build a day-in-years granularity, the periodicity of leap years (every how many years is a leap year) would be needed. Additionally, a further approach is presented when there isn't any defined logic behind the irregularity of the periods (imagine a granularity easterdays-in-year).

As seen in the previous subsection, approaches to finding a specific cyclic-time granularity's seasonal pattern in a time series need to be explored. The Autocorrelation function (ACF) shows the correlation between two lagged values of a time-series [6]. When a time series exhibits seasonality at a specific time granularity, the autocorrelations will tend to be higher for seasonal lags than for others [6]. Musbah et al. [16] present another approach that uses the Fast Fourier Transform (FFT) to convert the time-series into the frequency domain and then analyze the spikes and their location to determine the existence of seasonality patterns. According

*For instance, if there is a time series with daily data, the time points would be the days

to Musbah et al., [16], their method outperformed the ACF. Some visual methods to detect seasonalities at specific time granularities are the seasonal subseries plot, the spectral plot, and the box plot [17]. Hyndman et al. [18] propose a method in which they group data with the same time granule (label) and then compare their distributions using the Jensen-Shannon Distance metric. The idea is that if there is seasonality in a cyclic-time granularity, the distribution of the data at the different granules should vary between them.

After identifying which cyclic-time granularities associated seasonalities exist in a time-series, obtaining the seasonal patterns[†] from the original time-series is the next step. As many seasonalities could be found, approaches dealing with multi-seasonal time series will be addressed. Bandara et al. [19] introduce a method to decompose a multi-seasonal time series called MSTL. It is an extension of the Seasonal-Trend using Loess (STL) method to handle not only one but multiple seasonality patterns [20]. The idea is to apply a series of LOESS smoothings and subtract them from the time series for every seasonality. It includes an iterative refinement process in which every smoothed curve or seasonal pattern is added to the residuals, smoothed using LOESS again, and then subtracted from the residuals. Another interesting approach is used by Facebook in their modular regression model for scalable time-series forecasting called Prophet [21]. They propose an additive way to model a time series, in which the seasonal patterns are linear terms. The seasonality patterns are generated using Fourier series. Given a time series, the user can specify the periodicities that are known to exist in the data. So, it is expected that the user knows the seasonalities a priori. Then, multiple Fourier series will be generated, one for every period specified. The number of harmonics in each series is also customizable. This method is similar to the *dynamic harmonic regression with multiple seasonal patterns* presented by Hyndman et al. [6]. There, they generate seasonal patterns using Fourier series and then use them as external regressors in an ARMA model. De Livera et al. [22] proposed the TBATS model, where they also used Fourier series for the seasonal patterns but allowing them to change slowly over time. Dokumentov et al. [23] propose a time-series decomposition method called Seasonal-Trend decomposition using regression (STR), where they represent the seasonalities like two-dimensional surfaces resembling the topology of a cylinder, where one dimension is the time extending along the axis of the cylinder and the other dimension is "circular" and wraps around the cylinder [23]. It is circular because they resemble the cyclic-time granularities, where the labels repeat every certain period.

Once the seasonal patterns have been acquired, it is important to incorporate them into the forecasting process. It is crucial to establish a clear differentiation here because certain mod-

[†]Referring to the seasonal pattern as a time series itself.

els utilize seasonal patterns directly as predictors, while others employ alternative methods to represent seasonality for forecasting purposes. For example, SARIMA is an extension of the widely known ARIMA model that is able to forecast seasonal time series representing the seasonality with three components: the seasonal autoregressive component that models the relationship between the current observation and past observations at seasonal lags, the seasonal differencing component that removes the seasonal pattern by differencing the data at the seasonal lag, and the seasonal moving average component captures the relationship between the current observation and the residual errors at seasonal lags [6]. In contrast to SARIMA, alternative models incorporate a dependent variable (time series) that is subjected to regression with several independent variables, which encompass the seasonal pattern series, rather than using the same variable from previous time periods. Having discussed various approaches for acquiring the seasonal pattern series, the focus will now be directed toward examining models that utilize them. For example, a variation of ARIMA models that allow to include regressors are the ARIMAX and the SARIMAX models [24]. There, the regressors are linearly related to the target variable. Another relevant approach is the one proposed in the Prophet model [21]. They propose a modeling method that decomposes the time series into four components including trend, seasonality, holidays, and remainder:

$$y(t) = g(t) + s(t) + h(t) + \epsilon_t \tag{4.2}$$

The seasonal patterns $X(t)$ are adjusted with a parameter β, that define their contribution to $y(t)$:

$$s(t) = X(t)\beta \tag{4.3}$$

For different applications, the trend can be modeled using either a saturating growth model or a piecewise linear model. The choice between these models depends on whether the problem involves saturating or non-saturating growth[‡] [21]. The holidays are modeled by creating a matrix of regressors, where every regressor (vector with 1s in the date of the corresponding holiday) is multiplied by a parameter to adjust its contribution to the forecast [21]. The STR method introduced by Dokumentov et al. [23] decomposes the time series in three compo-

[‡]The Prophet model was implemented to cover many Facebook's applications, like modeling user's growth [21]

nents, including trend, seasonality, and remainder:

$$y(t) = T(t) + \sum_{i=1}^{I} S(t)^i + R(t) \tag{4.4}$$

Additionally, one could also add covariates to the formula. Similarly, as in the Prophet model, every component is multiplied by coefficients that adjust their contribution to $y(t)$. According to the article, their main contribution is to impose regularizations to the coefficients, resembling a ridge regression [23]. Of course, these methods assume that there is a linear relationship between the seasonal patterns and the target variable. A less biased approach is the Generalized Additive Models (GAM) [25]. Instead of using coefficients, the predictors and the target variable are related through smooth functions like splines. Scientific works for different fields like medical [26] or environmental [27] applications that model time series using GAM have been produced. Regarding ARIMA models, a way to integrate the extracted patterns could be achieved by using the ARIMAX configuration, which allows to add exogenous variables to improve the prediction [24]. Another effective multivariate regression model is Gradient Boosting which utilizes a combination of multiple weak learners, each of which is designed to correct the errors made by its predecessors, leading to a more accurate and robust overall prediction [28]. The weak learners are usually decision trees.

4.2.1 Remarks

Different approaches and solutions tackling the questions set in Section 4.1 have been addressed. A further assessment needs to be performed to determine what can be used for this project and also to find contribution opportunities. This will be done by analyzing each of the original questions.

Implementing custom cyclic-time granularities labeled series: Gupta et al. [2] approach introduces a formula-intensive implementation of the cyclic-time granularities. While their method is the most complete, detailed, and useful, some interesting adjustments could be made. For example, instead of using different formulas to tackle different regular and irregular periodicities, a fully programmatic approach could be achieved by taking the calendar's information. For example, if daily data is available and a semester_in_year granularity is needed, using the calendar one could obtain the month of a time point, divide it by 6 (as 6 months exist in every semester), and use a ceiling function, outputting 1 if it's the first semester and 2 if it's the second semester. Furthermore, a fully programmatic approach could explode some already ex-

isting functions from programming languages, like date.weekday() from Python, that gives the week_in_year label of a given date. Also, a fully programmatic approach could help to create complex cyclic-granularities whose logic to label data points depend on external factors (for example, a workingdays_in_month granularity of a shop that does not operate on rainy days). Additionally, it was observed that the approach proposed by Gupta et al. [2] uses the Standard Gregorian calendar instead of the ISO-8601 Week-Based calendar. In many cases, this could lead to inaccurate outcomes. For example, in the ISO-8601 calendar each week belongs exclusively to a single year, while in the Gregorian calendar, weeks can span across years [29]. As a reference, see Figure4.8: The week 1 of the year starts on a Wednesday in the Gregorian calendar, while in ISO-8601 it starts on a Monday. In fact, in the ISO-8601 calendar a week always starts on a Monday and the first week of the year is the one whose Monday is the closest to January 1st [29]. When someone uses the term "the next week", they are typically referring to it in the context of the ISO-8601 calendar rather than the Gregorian calendar. A week is conceived as a period starting on a Monday and finishing on a Sunday, that's how the world operates. Therefore the ISO-8601 calendar is widely used in industries like financial or retail [29]. As such, another interesting contribution would be considering the ISO-8601 calendar when creating the cyclic-time granularities.

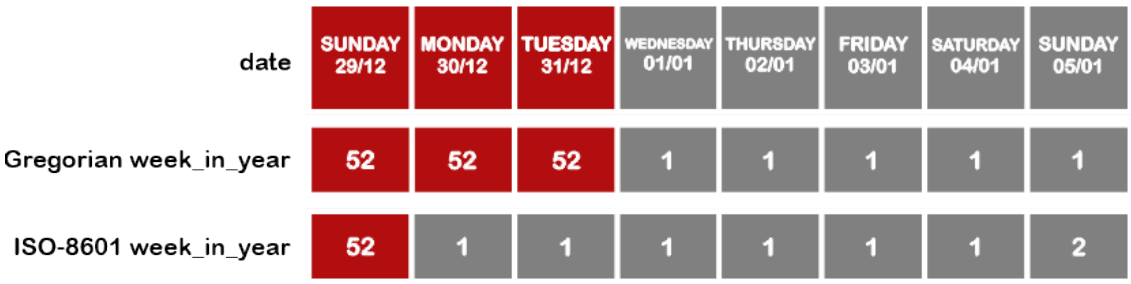

Figure 4.8: week_in_year in Gregorian vs ISO-8601 calendar .

Gupta et al. [2] approach is available for public use in R language in a package called *gravitas* [30]. As no implementation in Python exists, another interesting contribution would be to implement it in this language. For the current project's scope, it would be more consistent to do it in Python, as the rest of the scripts are in this language.

Assessing if a time series has a specific cyclic-time granularity's seasonality: To know if a time series is seasonal at a specific cyclic-granularity, approaches using and not using the cyclic-granularity labeled series were addressed. The ACF is the most straightforward approach, but detecting multi-seasonal patterns with simple correlations at individual lags can lead to inaccu-

rate conclusions. The approach using the FFT to convert the time-series to the frequency domain presented by Musbah et al. [16] seems very useful, but as the paper explains, "there are limitations that prevent the FFT technique from identifying the seasonality when the series contains a significant trend or insignificant swing along the periods of the time series". A method using the cyclic-time granularity labeled series is the one proposed by Hyndman et al. [18]. As explained, they proposed a statistical method to detect distributional differences within a cyclic-granularity and thus to say if a time series is seasonal at that time granularity or not. First, using the cyclic-time granularity labels, they group the time-series data in what they call "levels". Every level can be visualized using a box-plot graph. To gain a proper understanding, refer to Figure 4.9 for clarification. The figure depicts a time-series graph displaying monthly data where quarter_in_year labels are assigned to each time point and the shaded points are the ones that will be grouped into level 1. Subsequently, Figure 4.10 exhibits the box-plot representation of level 1, alongside the box-plots of the other levels of the quarter_in_year granularity. If there is a significant difference between the distributions of the levels, this would imply that the data have a differentiated behavior depending on the label they have, which would suggest the presence of seasonality.

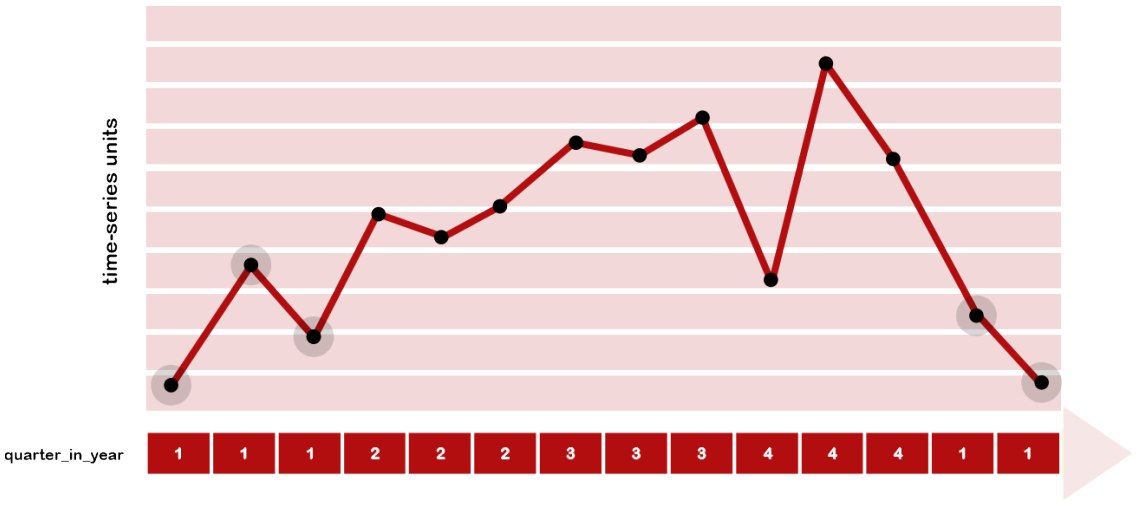

Figure 4.9: quarter_in_year granularity used to create levels (the time-points that are grouped in level 1 are shaded)

Hyndman et al. [18] follow a process in which they normalize the data, characterize each distribution and compute the distance between the distributions using Jensen-Shannon Distance. They have implemented this approach in the *gravitas* package in R. There is currently no existing Python implementation of this method and thus it would be a valuable contribution to do it in this language. Some specific adjustments will be proposed for this method, focusing on ar-

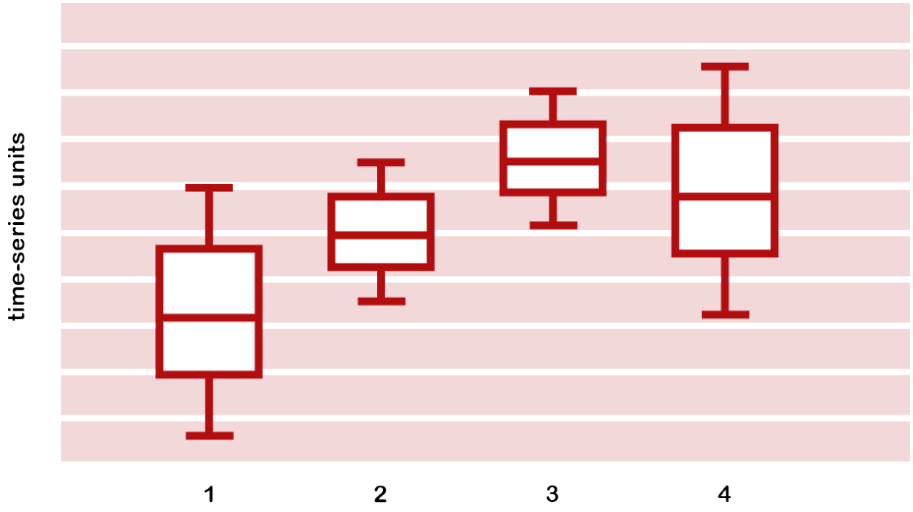

Figure 4.10: Box-plots of the different levels of the quarter_in_year granularity .

eas where potential contributions or improvements can be made, this will be clearly discussed in the next section.

Obtaining specific cyclic-time granularities' seasonal patterns from a time-series: Once the seasonalities associated with cyclic-time granularities in a time-series are identified, various methods for extracting them have been explored. As it was remarked, it was important to focus on methods dealing with multi-seasonal time-series. It was seen that methods like MSTL [19], Prophet [21] or STR [23] demand the user to add the periods of the seasonalities to extract, representing them with a float. As it was seen, not all cyclic-time granularities have regular periods and thus, representing their complexity with a single number is inaccurate. A common practice is to add decimals to represent irregular periods. For instance, in the documentation of the Prophet model, an example is presented where they define the period of a day_in_month granularity as 30.5 [31]. In essence, this is like saying that this granularity has a regular period of 30.5, while it actually has multiple periods that can be 28, 29, 30 or 31 days. So, setting it as 30.5 is an imprecise solution. This introduces a valuable opportunity to define a seasonality extraction method where all the possible periods are considered. For this, using the cyclic-time granularities series will be useful, as will be seen in the next section.

Including the seasonal patterns into forecasting models: Models using the seasonal patterns as predictors and others using alternative methods to integrate the seasonality were explored. As the previous step will provide the seasonality patterns, it makes sense to choose an approach that actually uses them. Although models like Prophet [21] or STR [23] model the seasonalities in a different way, they can integrate covariates in the forecasting task. So,

approaches using seasonal patterns as covariates could be tested. Likewise, it is worth exploring the application of General Additive Models or Gradient Boosting that incorporate these seasonal patterns as predictors.

4.3 Proposal

In this section, the implementation of three functionalities will be presented. The first one is a *cyclic-time granularity series generator*, the second one is a *seasonality analyzer* and the third one is a *cyclic-time granularity based time-series decomposer*. All of them will be used in the Data Analysis Backbone that will be addressed in the next chapter.

4.3.1 Cyclic-time granularity generator

The goal of this functionality is to create a series with labels for every time point of a time series according to a specific cyclic-time granularity specification. For example, if a time series with daily data is available and the day_in_week granularity needs to be created, every time-point will be labeled from 1 to 7 as it is shown in Figure 4.11. The labeling must be done in a logical way, meaning that Monday should be labeled as 1, Tuesday as 2, and so on. Additionally, the implementation will follow the ISO-8601 calendar.

The functionality should receive some inputs for creating the labels: the timestamps of starting and ending points of the time-series and the units of time used. Since the data collected for this thesis is in daily increments, the focus will be on establishing cyclic-time granularities down to this specific unit[§]. However, the same logic will be used if in the future even more specific cyclic-granularities are needed (i.e. minute_in_day or seconds_in_hour). Considering this, the only two inputs needed for the *cyclic-time granularity generator* will be the starting date and ending date of the time-series.

The functionality allows to define a programming approach for every custom desired cyclic-granularity. It was decided to include the following granularities:

- semester_in_year
- quarter_in_year
- month_in_year
- month_in_semester

[§]This means that the implementation to propose will work for daily data.

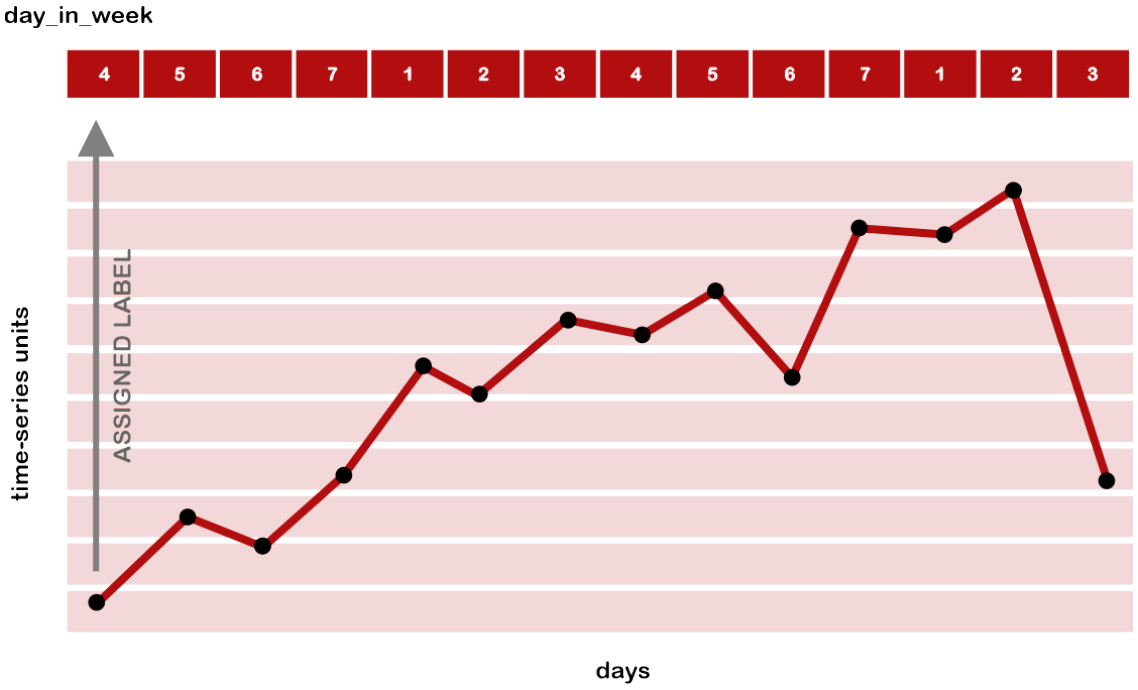

Figure 4.11: day_in_week granularity labeling .

- month_in_quarter
- quarter_in_semester
- day_in_month
- day_in_week
- week_in_year
- week_in_semester
- week_in_quarter
- week_in_month

For example, the quarter_in_year granularity takes the range of days between the given starting and ending dates, gets the month of every day, divides it by 3 (as there are 3 months in every quarter), and then makes a ceiling operation[¶]. Some implementations are more complex. For

[¶]The ceiling operation gives the first integer greater or equal to a given number

example, the week_in_month programming approach demands making an analysis determining if a day is the first Thursday of the month, as this is the reference used to determine if the week belongs to a new month or to the previous one. While ISO-8601 does not directly defines the first week of a month, the logic that it presents to determine the first week of a year can be used, and that is the week that contains the first Thursday. The idea is to consider the initial week of a month as the one in which the majority of days belong to the month, rather than the preceding month. Unlike the Gregorian calendar where a week does not necessarily start on Mondays [29], the presented approach is more suitable for real-life scenarios.

The decision to make the implementation for the above 12 listed granularities is based on the type of data that is used in this thesis and the types of seasonalities that it may have. Of course, many other granularities can be implemented (an unlimited number) but since the available data is not likely to present the corresponding seasonalities, they are not included. The final output of the functionality is a series of labels per granularity with the same length as the time series. Figure 4.12 shows the output dataframe, with the time-series values (denoted as usd_kg) and the related granularity labels depending on the date. The script containing this implementation is called *time_granularities_generator.py*.

	usd_kg	semester_in_year	quarter_in_year	month_in_year	month_in_semester	month_in_quarter	q
2017-03-09	0.402	1	1	3	3	3	
2017-03-10	0.157	1	1	3	3	3	
2017-03-11	0.168	1	1	3	3	3	
2017-03-12	0.168	1	1	3	3	3	
2017-03-13	0.360	1	1	3	3	3	
2017-03-14	0.232	1	1	3	3	3	
2017-03-15	0.200	1	1	3	3	3	
2017-03-16	0.200	1	1	3	3	3	
2017-03-17	0.077	1	1	3	3	3	
2017-03-18	0.168	1	1	3	3	3	
2017-03-19	0.168	1	1	3	3	3	
2017-03-20	0.168	1	1	3	3	3	
2017-03-21	0.221	1	1	3	3	3	
2017-03-22	0.380	1	1	3	3	3	
2017-03-23	0.074	1	1	3	3	3	
2017-03-24	0.107	1	1	3	3	3	
2017-03-25	0.221	1	1	3	3	3	
2017-03-26	0.221	1	1	3	3	3	
2017-03-27	0.200	1	1	3	3	3	
2017-03-28	0.200	1	1	3	3	3	
2017-03-29	0.200	1	1	3	3	3	
2017-03-30	0.326	1	1	3	3	3	
2017-03-31	0.170	1	1	3	3	3	
2017-04-01	0.200	1	2	4	4	1	
2017-04-02	0.207	1	2	4	4	1	
2017-04-03	3.294	1	2	4	4	1	
2017-04-04	0.200	1	2	4	4	1	
2017-04-05	0.356	1	2	4	4	1	

Figure 4.12: Example of the output of the time granularity generator functionality

4.3.2 Seasonality analyzer

The goal of this functionality is to determine if a time series is seasonal at specific cyclic-time granularities. As it was stated before, this will be implemented using the method developed by Hyndman et al. [18] with some specific adjustments.

The inputs of the functionality are the dataframe created with the granularities generator, the name of the column of the time series and the name of the granularity to analyze. The first step is to remove the trend. As it will be shown, the seasonality will be tested by comparing the different distributions across different time granules. If there is a trend, the distributions will for sure be different and this is a behavior that needs to be avoided. Although it is not explicitly mentioned in the paper as a trend mitigation approach, Hyndman et al. [18] methodology's first step addresses this by making an empirical Normal Quantile Transformation (NQT) of the time-series forces it to follow a normal standard distribution. The motivation behind their approach is to homogenize the type of distribution of all the levels inside a cyclic-time granularity for better comparison. As a reference take Figure 4.13, where a time series before and after the NQT is presented. For the implementation of the functionality in this thesis, the idea to use the NQT will be taken.

Once they have the transformed series, Hyndman et al. [18] group the data points according to their labels in the cyclic-time granularity and then characterize each distribution by taking the percentiles from each of them to then calculate the Jensen-Shannon Distance. According to Menendez et al., [32], the Jensen-Shannon Distance ‖ is a metric to measure the difference between two probability distributions. For example, Python [33] and R [34] implementations of the JSD function take as arguments probability vectors. Given that percentiles are not probabilities, there is a potential misapplication of the Jensen-Shannon Divergence in their approach that can lead to inaccurate outcomes. Therefore, an approach of characterizing the distributions with probability vectors instead of percentiles will be proposed for this functionality.

As it was mentioned, after the time series is normal quantile transformed, a distribution will be created for every label of the cyclic-granularity.

To clarify this, Figure 4.14 shows the boxen plot graph of every label's distribution at a month_in_year granularity of the time-series presented in Figure 4.13. To characterize these distributions the following approach is proposed:

1. Get the maximum and minimum value from the time series.
2. Define 10 equally spaced bins between the maximum and minimum values

‖Called Jensen-Shannon Divergence in the article.

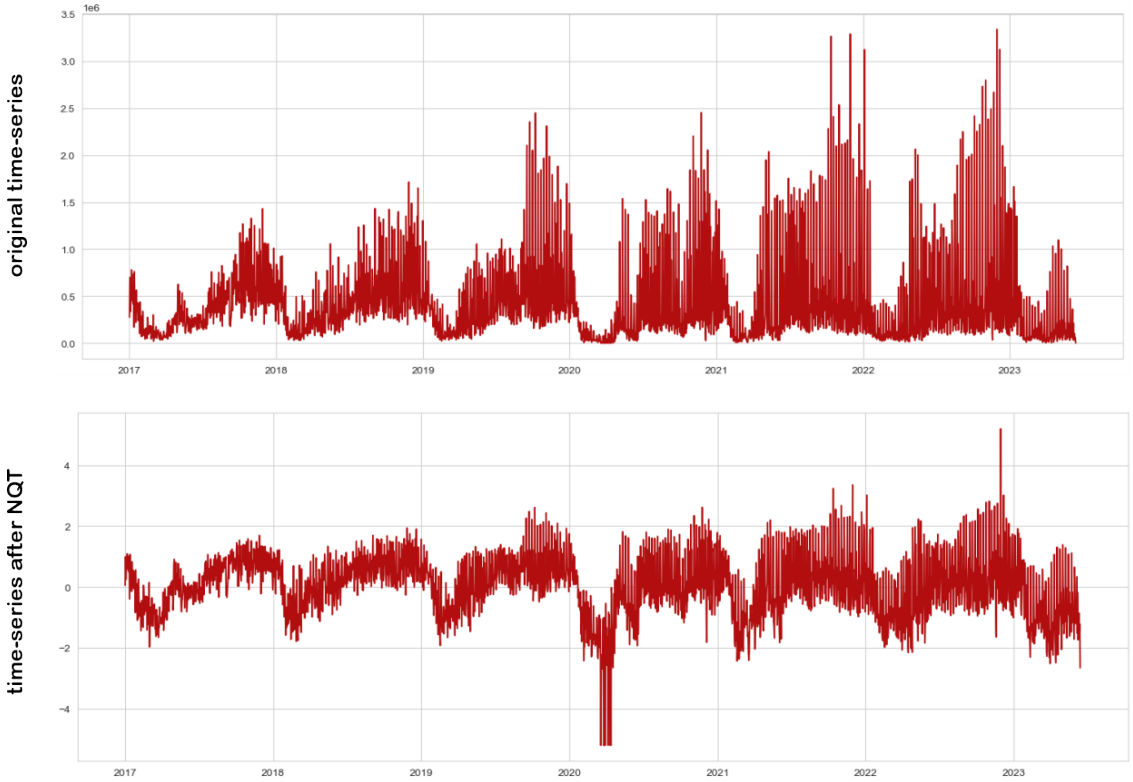

Figure 4.13: Applying a NQT to a time-series

3. Create a probability vector for each granularity label where every element is the proportion of points from a single label that fall inside a bin.

Figure 4.15 schematizes how the bins are defined for the month_in_year granularity example presented before. Every probability vector will have 10 elements, every element is the proportion of points inside the corresponding bin. There will be as many probability vectors as labels in the cyclic-granularity. For example, for label 1 in Figure 4.15, the probability vector will be:

$$P_1 = [0, 0, 0.02, 0.13, 0.45, 0.35, 0.5, 0, 0, 0]$$

Now that every distribution has been characterized using probability vectors, the Jensen-Shannon Distance can be used to compare them. As it was mentioned before, the idea is that if the distributions between labels are significantly different, this would suggest that the seasons have an effect on the distributions of the data and consequently that a seasonality exists at that cyclic-time granularity. According to Hyndman et al. [18] contributions, it is enough to make the comparison for consecutive labels, take the maximum obtained JSD value, and ana-

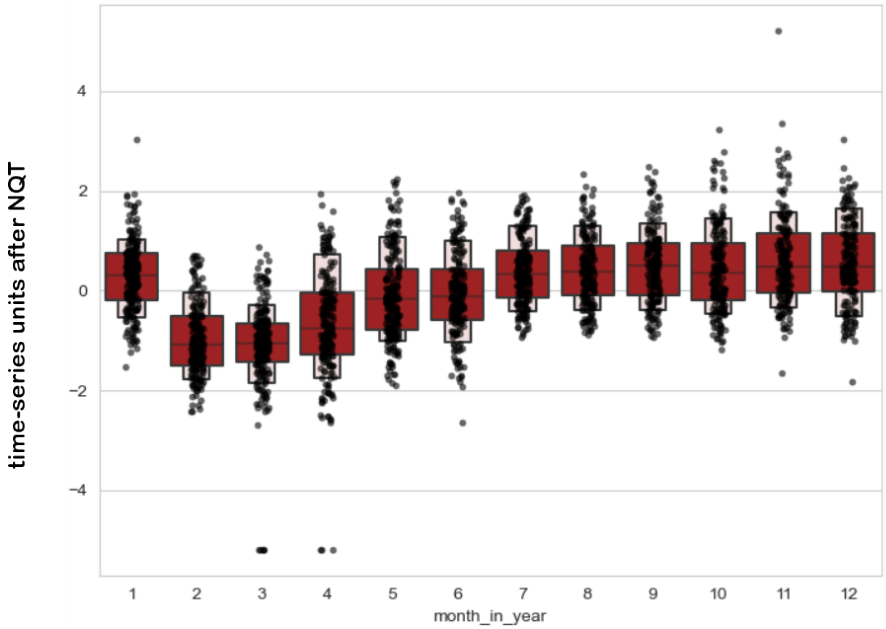

Figure 4.14: month_in_year granularity

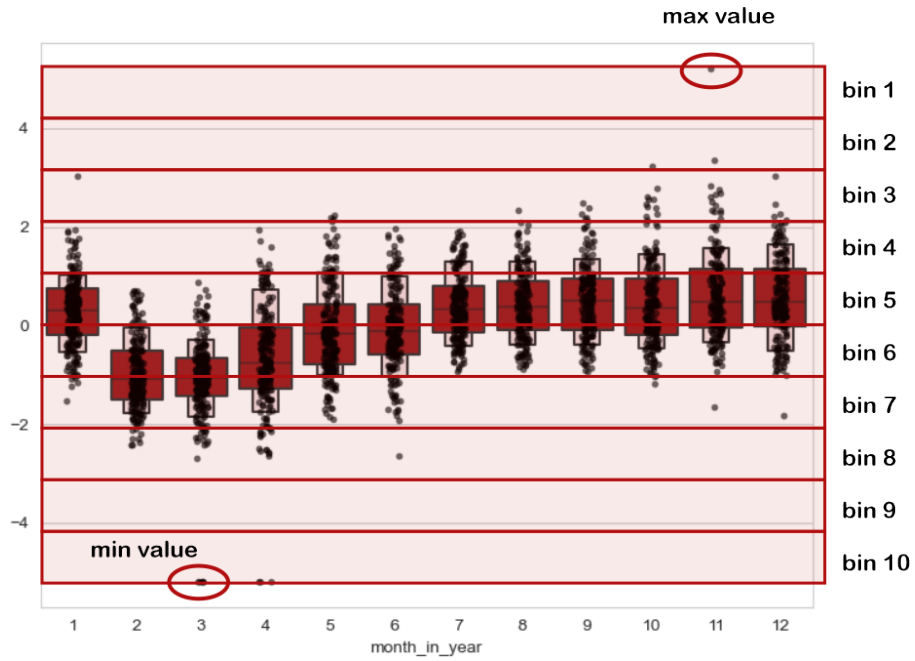

Figure 4.15: Bins definition example

lyze if the measure is significant enough. However, it was observed in this thesis that many time series exhibited small differences between consecutive pairs but large differences between non-

consecutive pairs. Hence, if only consecutive comparisons are made, one might erroneously conclude the absence of seasonality, despite its actual existence. Therefore, the functionality will run comparisons for all the possible pairs of labels and then take the maximum JSD obtained. However as Hyndman et al., [18] remarked, as the number of labels increases there is a higher chance to get a larger max JSD. To tackle this, a permutation test will be used. It is similar to the approach presented by Hyndman et al. [18] but without the strong assumption that the max JSD metric follows a normal distribution. The null hypothesis will be that the data across all the labels follows the same distribution which is translated into a low max JSD and the alternative hypothesis is that they are different which is represented by a high JSD. Therefore, a right-sided hypothesis test will be used. In every permutation, the data in every label will be randomly resampled and the max JSD will be calculated. After several tests, it was determined that 1000 iterations are adequate. If the observed max JSD's p-value is small enough, there would be enough evidence to reject the null hypothesis and to determine that there is a significant difference between the distributions and therefore, the existence of a seasonality. For example, a possible outcome is the one presented in Figure 4.16: the right graph shows the histogram of all the permuted max JSDs and the black line shows the observed max JSD. With a p-value of 0.2757, there seems to not be enough evidence to reject that the labels distributions are the same. This makes sense even visually, as there aren't significant distributional differences across the labels in the granularity.

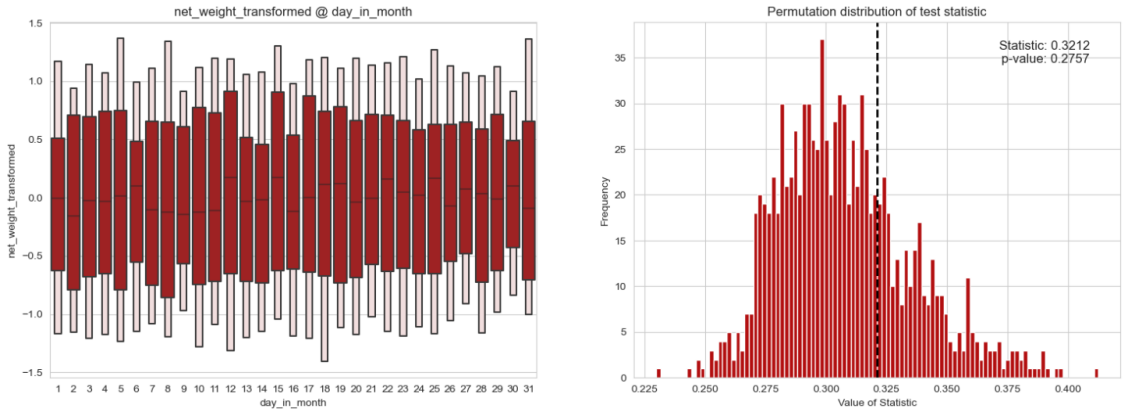

Figure 4.16: Permutation test for day_in_month granularity

A different outcome is presented in Figure 4.17, where the observed max JSD's p-value is very small, and therefore strong evidence to reject the null hypothesis exists. Again, this makes sense with what is visually concluded, as it is clear that months like February (2) or March (3)

have considerably different distributions than the rest.

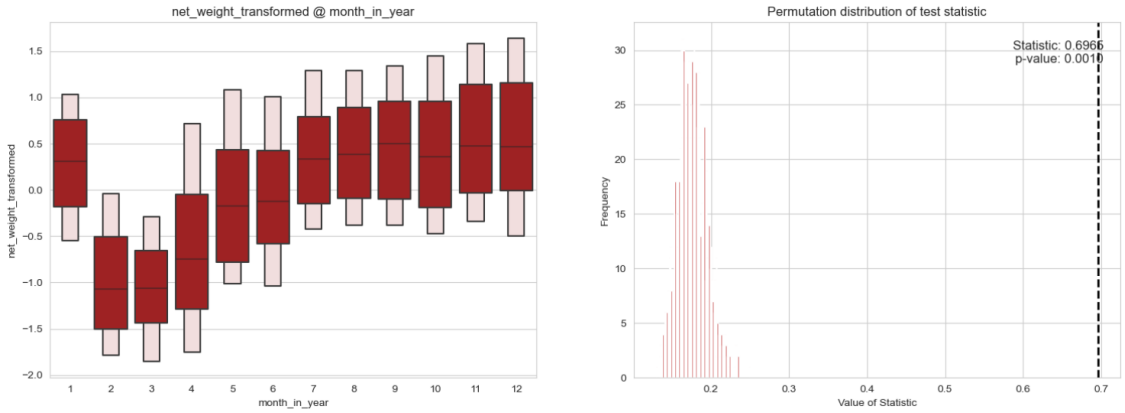

Figure 4.17: Permutation test for month_in_year granularity

As it is expected, a p-value threshold should be set to establish if a max JSD is significant enough. After several tests, it was decided that the 0.01 threshold gives adequate results.

This functionality can be used for every desired cyclic-granularity. For the time series of Figure 4.13, the results presented in Figure 4.18 were obtained.

granularity	statistic	pvalue	significant
semester_in_year	0.443543	0.000999	True
quarter_in_year	0.497914	0.000999	True
month_in_year	0.696495	0.000999	True
month_in_semester	0.387370	0.000999	True
month_in_quarter	0.126284	0.000999	True
quarter_in_semester	0.082189	0.008991	True
day_in_month	0.321193	0.299700	False
day_in_week	0.534367	0.000999	True
week_in_year	0.771687	0.004995	True
week_in_semester	0.487098	0.046953	False
week_in_quarter	0.379838	0.200799	False
week_in_month	0.131996	0.162837	False

Figure 4.18: Summary of seasonality analysis

This functionality is implemented in the script *time_granularities_generator.py*. It is a class

that receives as inputs the dataframe**, the name of the time-series column and the name of the cyclic-granularity series to analyze. It has three methods available for the user:

- **summary:** Returns a summary indicating the observed statistic (max JSD) and the corresponding p-value.
- **is_seasonal:** Returns True if the observed max JSD is significant enough to prove seasonality, False otherwise.
- **plot**: Return the boxen plot and the histogram like the ones shown in Figure 4.16 or Figure 4.17

4.3.3 Cyclic-time granularity based time-series decomposer

The goal of this functionality is to decompose the time series with the information obtained from the *seasonality analyzer*. It considers the widely used approach that a time series is composed of a seasonal, a trend-cycle, and a remainder component [6]. Unlike other decomposition strategies where the approach can be expressed in a mathematical formula (e.g. additive decomposition), the proposed functionality introduces a step-wise approach similar to MSTL [19] in which seasonal patterns and trend-cycle are extracted. It is built in two parts, the first one addresses the seasonalities, and the second one is the trend-cycle ("trend" will be used for simplicity from now on).

Once the *seasonality analyzer* provides the list of cyclic-time granularities that have a seasonal pattern in the time series, the seasonality extraction goes as follows:

1. Select one cyclic-time granularity
2. Select one label from the selected granularity
3. Take the average of all the values associated with that label
4. Assign to all the points from that label the calculated value
5. Repeat 2 to 4 for all the labels of the cyclic-granularity
6. Repeat 1 to 4 for every cyclic-granularity

**Defined in the previous section

Figure 4.19 shows an example of this process. There, the seasonal pattern for the quarter_in_year granularity is calculated. First, the points corresponding to label 1 are selected, averaged, and assigned with the obtained value. Then, the points corresponding to label 2 are selected, averaged, and assigned. This process continues till all the labels have been addressed and the final seasonal pattern is obtained, as shown in the bottom part of Figure 4.19.

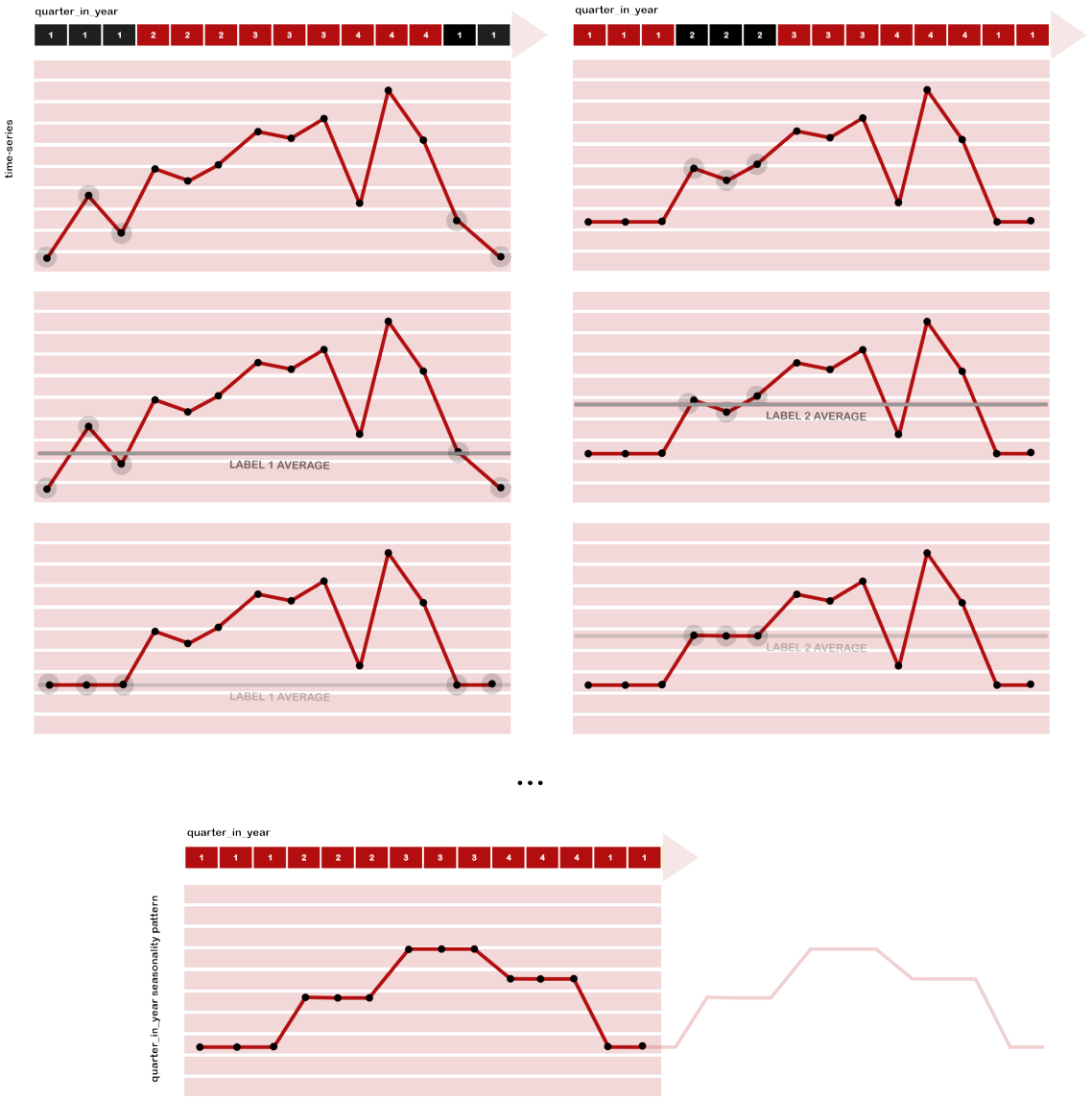

Figure 4.19: Seasonality extraction schema

It is important to highlight that all the patterns are independently extracted from the time series. This differs from MSTL, where a extracted pattern is subtracted from the time series (and

subsequently from the residuals) before making the next pattern extraction. Bandara et al. [19] work on MSTL explains that during the decomposition, seasonalities from larger period cyclic-time granularities can absorb the information from lower period ones if extracted before. That is, extracting a seasonal pattern from a semester_in_year granularity before a month_in_year granularity can make the first pattern "steal" information from the second one[††]. This can be clarified with an example: say that there is a time series with monthly data about retail sales where an increase always happens in December (due to Christmas), making the second semester always have more sales than the first one. It's clear that the increase is more related to a month's effect rather than to a semester's effect. However, if the semester_in_year seasonality pattern is extracted before, it will take the December increment as an increase related to the semester. Then when the month_in_year seasonality is extracted, part of that information will be not available cause it was captured by the previous pattern. However, one could also think of a situation where the opposite happens: say that there is another time series with sales data of a shop in monthly increments. Consider that the marketing team of the shop runs a campaign every first semester of the year to increase sales. When to launch the campaign depends on several factors and therefore, it doesn't have a specific month to be launched but it is always in the first semester. Note that here the increment is more related to a semester's initiative rather to a month's one. Extracting here the month_in_year seasonality first can make the first 6 labels (January to June) absorb part of the information that actually belongs to the semester_in_year seasonality. In summary, the order of extraction of seasonalities depends on the time series and its particular context. Therefore, in the approach proposed in this thesis all the seasonal patterns are extracted from the same time series (without any subtraction step) and as such, the order is not relevant. The downside of this approach is that potentially the extracted seasonality patterns will contain duplicated information and as such, they might be correlated. Nonetheless, this is the most generic approach that could have been proposed given the above explanation. For more accurate methods, one should know the context of the time series which is not always the case[‡‡]. For example, Figure 4.20 shows the time series of the daily price per kilogram of avocado exports and Figure4.21 shows the extracted seasonality patterns using the proposed method.

Once the seasonal patterns are extracted, the trend can be addressed. In MSTL method, which is a generalization of the STL method, the trend is obtained with a LOESS smoothing

[††]Under the MSTL approach, where an extracted pattern is subtracted from the time-series (or residuals) before making the next extraction.

[‡‡]However as it will be seen in the next chapter, this approach ends up generating good results

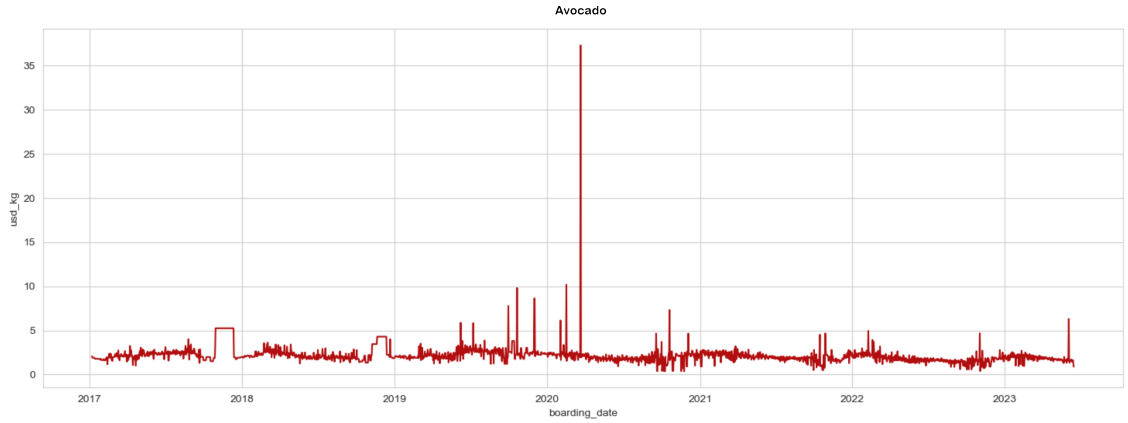

Figure 4.20: Avocado exports in usd/kg

[20]. Inspired by this, it was decided to use a smoothing technique to capture the trend in the functionality. The challenge lies in determining the appropriate fraction for LOESS smoothing.

It is desired that the trend captures long-term tendencies of the data [6], which of course are not related to seasonal patterns. If the trend is generated with LOESS smoothing that uses a tiny fraction of the data to define each point, it will be very similar to the time series. As the seasonal patterns are repetitive features extracted from the time series, they are correlated to it. So, the fraction to use in the LOESS should be large enough, so that the trend is enough different from the time series to not be significantly correlated with any of the seasonal patterns. That way it is ensured that the trend captures as much information without including seasonal features. To compare the trend and the seasonal patterns, the Pearson Correlation will be used. A two-tailed hypothesis test will be conducted, where the null hypothesis will be that the absolute value of the Pearson correlation is greater than the desired threshold. A rule of thumb is that correlations from 0.0 to $\pm$ 0.3 are considered negligible [35]. Considering this, the value taken as the threshold will be 0.1. The null hypothesis will be rejected if the p-value is below 5%. This means that if the absolute maximum correlation is less than 0.15 and the statistic is significant enough (p-value $<$ 0.05), there will be enough evidence to say that the correlation between the seasonal patterns and the trend is negligible. In summary, the trend extraction algorithm is proposed to be as follows:

1. Initialize the smoothing fraction in 0.01
2. Get the trend by applying a LOESS smoothing on the time series with that fraction
3. Calculate the Pearson Correlation between the trend and a seasonal pattern

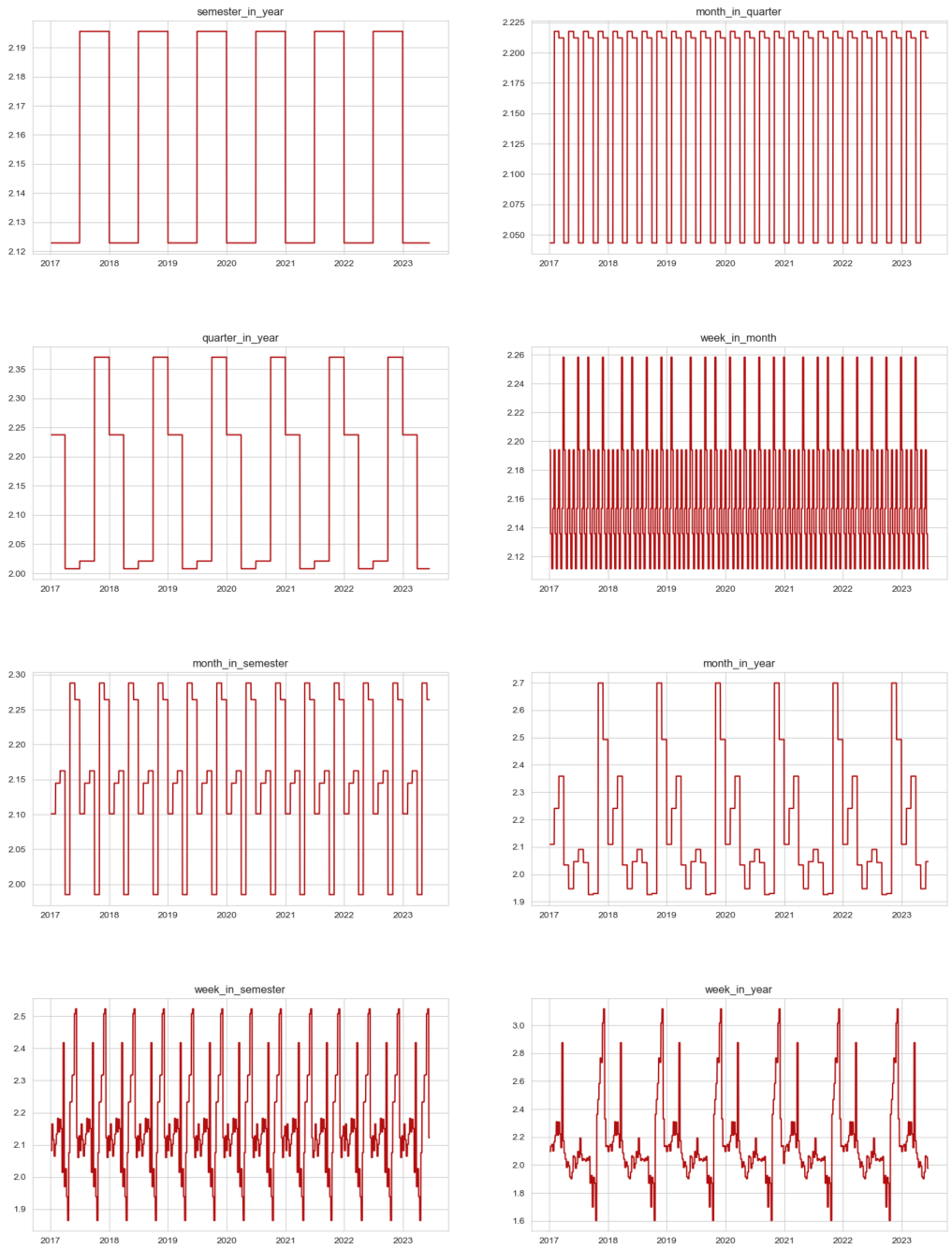

Figure 4.21: Seasonality patterns

4. Repeat step 3 for all the seasonal patterns

5. Get the maximum correlation value

6. If the correlation (absolute) is less than 0.15, proceed to step 7, else go back to step 1 adding an increment of 0.01 to the smoothing fraction.
7. Make the two-tailed hypothesis test to assess its significance
8. If the null hypothesis is not rejected, go back to step 1 adding an increment of 0.01 to the smoothing fraction, else return the obtained trend and finish the algorithm.

As a reference, this was tested with the time series from Figure 4.20 obtaining the trend displayed in Figure 4.22. In this case, the algorithm yielded a smoothing fraction of 14%.

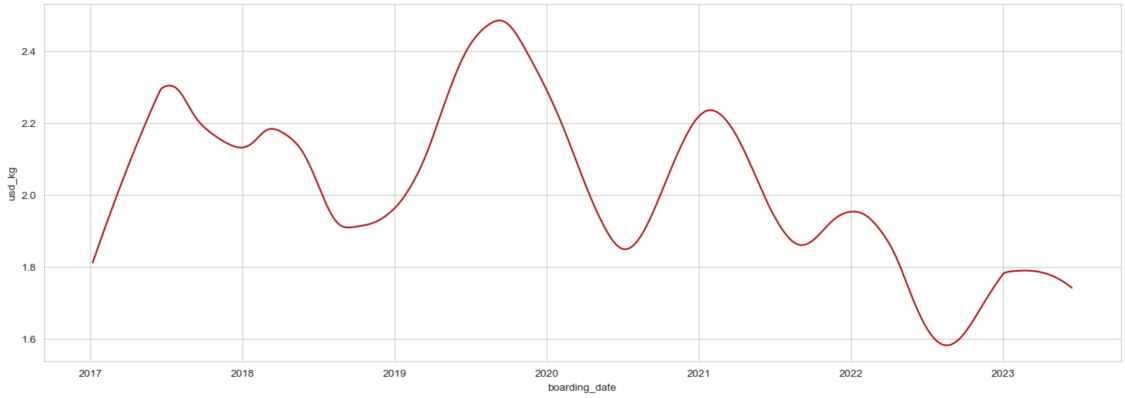

Figure 4.22: Extracted trend

The proposed approach to decompose the time series was denominated GMST decomposition, referring to (cyclic-time) granularity-based multi-seasonal trend decomposition. The functionality was implemented in the script *gmst_decomposition.py*. It receives, as input the cyclic-time granularities to extract (that are automatically obtained from the *seasonality analyzer* part) and the time-series. As expected, the output are the seasonal patterns and the trend.

5 Data Analysis Backbone

This chapter covers the data analysis pipeline that is the continuation of the data management backbone. First, the fundamentals of the methodology to use are presented. Then, the different layers of the pipeline are described. The complex seasonality analysis-related implementations will be used here.

5.1 Fundamentals

The Data Analysis Backbone is a methodology taught by the Polytechnic University of Catalonia (Universitat Politècnica de Catalunya) to create pipelines for data analysis. It is the continuation of the Big Data Management Backbone described in 3. Specifically, it addresses the analysis-related tasks from the retrieval of the data from the Exploitation Zone to the deploying of models [9]. A schema of the backbone is shown in Figure 5.1

As mentioned in 3, the Exploitation Zone (EZ) exposes a project's focused datasets to data scientists. In Figure 5.1, it is shown that the EZ exposes 3 different datasets, each for a different project. Therefore, every analytical project has its own analytical pipeline. A single pipeline is composed of a series of layers through which the data is leveraged sequentially. Similarly, as it was done with the Big Data Management Backbone, a distinction can be made between the layers according to the task they perform. The first layer is a "transit" layer (black) as it is used to generate the data input for the next ones that are denominated "analytical" layers (dark red).

In the next sections, the implementation of this backbone for this thesis will be described.

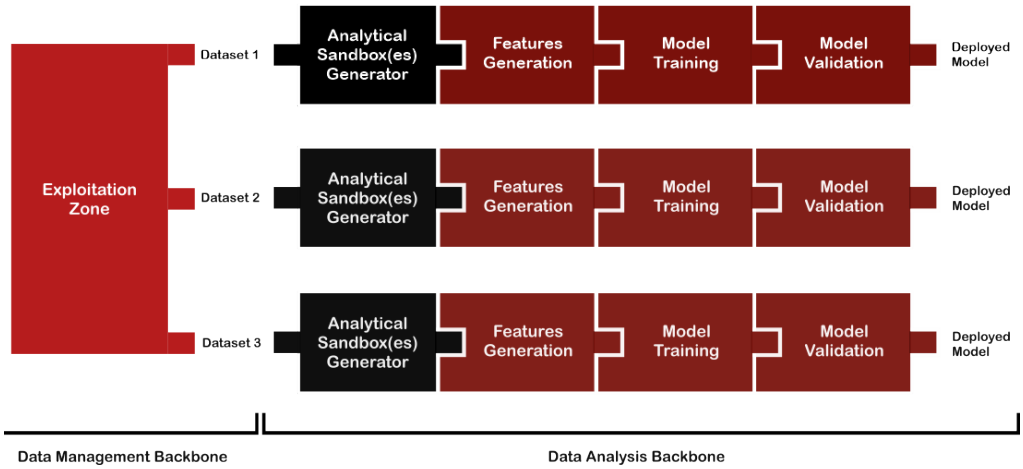

Figure 5.1: Data Analysis Backbone schema.

As in Chapter 3, each layer of the pipeline will be addressed.

5.2 Analytical Sandbox(es) Generator

The function of this first layer is to take subsets of the dataset provided by the EZ that will be used as inputs for analytical tasks inside a project. These subsets are called sandboxes [9]. In the Big Data Management Backbone implementation, it was defined that the dataset exposed by the EZ is a PostgreSQL view called *peru_exports_ts*, whose details are presented in Table 3.3. As it was mentioned before, it contains data on exports of vegetables and fruits from Peru. Every row in the dataset represents an export event on a particular day. Every row can be associated with a type of product according to the code in the attribute HEADING. Three numerical attributes exist in the dataset: NET_WEIGHT, GROSS_WEIGHT, and VALUE_USD. Furthermore, a BOARDING_DATE attribute exists to indicate the date of the corresponding export.

As it was stated in the introductory chapter of this thesis, two metrics regarding exports are worth analyzing: the daily price per kilogram and the daily total weight exported per product type. Then, the sandboxes will be implemented as time series from these two metrics for every product type. In other words, one group of sandboxes will gather time series of daily prices per kilogram and the other group will be composed of a time-series of daily total weights exported. Figure 5.2 schematizes this: for every heading (product type) two sandboxes are generated. These sandboxes can be created as follows:

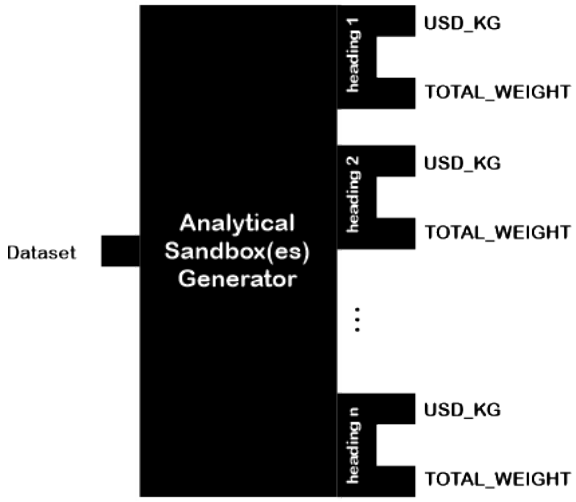

Figure 5.2: Sandboxes per heading schema

- **price by kilogram:** For this metric the VALUE_USD attribute can be used with the NET_WEIGHT or the GROSS_WEIGHT. As the NET_WEIGHT addresses the weight of the exported product without the packaging, this one will be considered. The price per kilogram is then defined as the division between the VALUE_USD and the NET_WEIGHT. A new attribute called USD_KG can be obtained per row. As the daily price per kilogram is needed, this new attribute needs to be aggregated using the BOARDING_DATE and taking the average of all the USD_KG per day. If there are days without observations, the USD_KG value is set with the price of the day before.

- **total weight exported:** This metric can be obtained using the NET_WEIGHT attribute. An aggregation needs to be made according to the BOARDING_DATE. As we need the total weight exported, the operation in the aggregation will be a sum of NET_WEIGHT. The aggregated column will be called TOTAL_WEIGHT. If there are days without observations, the metric is set to 0, meaning that 0 kg of that product where exported that day.

These implementations are done in the script *time_series_generator.py* which is located in the folder *data_analysis/sandbox_generators/scripts/*. It receives as input the heading code of the product and a string indicating the type of time series to create ('usd_kg' or 'total_weight'). This script consumes query files (.sql) located in *data_analysis/sandbox_generators/queries/*. The output is an array with the time-series and optionally, the user can use a method to plot it. Figure 5.3 shows a time series generated with this functionality.

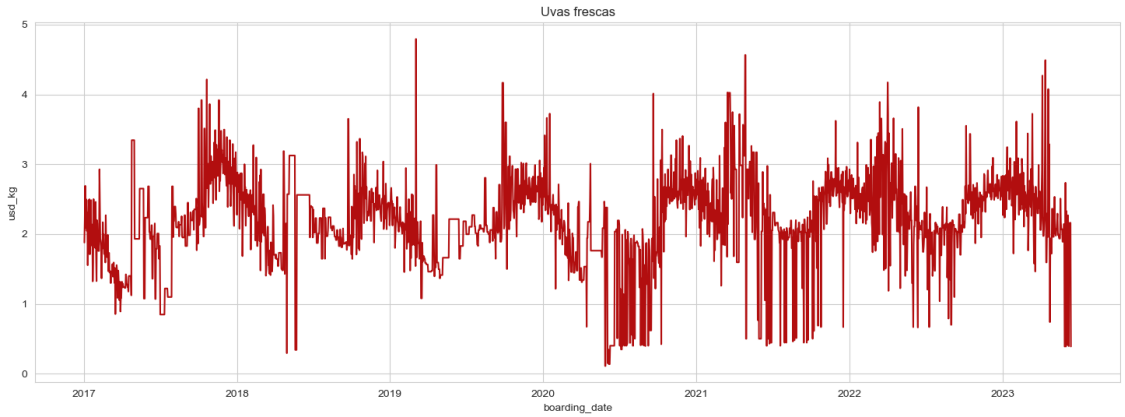

Figure 5.3: Time-series obtained in the sandbox generator

5.3 Features Generation

In this layer, the goal is to generate features from the sandboxes [9]. To achieve this, the data must be prepared considering specific models' needs. In the end, training and test datasets must be obtained.

In this thesis, the target is to create forecasting models for each time series, making use of the complex seasonality-related implementations described in Chapter 4. As such, the features to extract from each time series are the seasonality patterns and the trend-cycle. The idea is to use the same features in different models to see their performance. Considering this, the schema of the pipeline per product type can be seen in Figure 5.4. Note that a model selection layer was added since this will be the one choosing the best model.

Once a time series is generated as detailed in the previous section, the feature extraction process starts. First, the cyclic-time granularity generator is used to label each observation of the time series, once per granularity. For the time series in Figure 5.3, the dataframe from Figure 5.5 is generated in this first step. It has 12 cyclic-time granularities columns:

- semester_in_year
- quarter_in_year
- month_in_year
- month_in_semester
- month_in_quarter

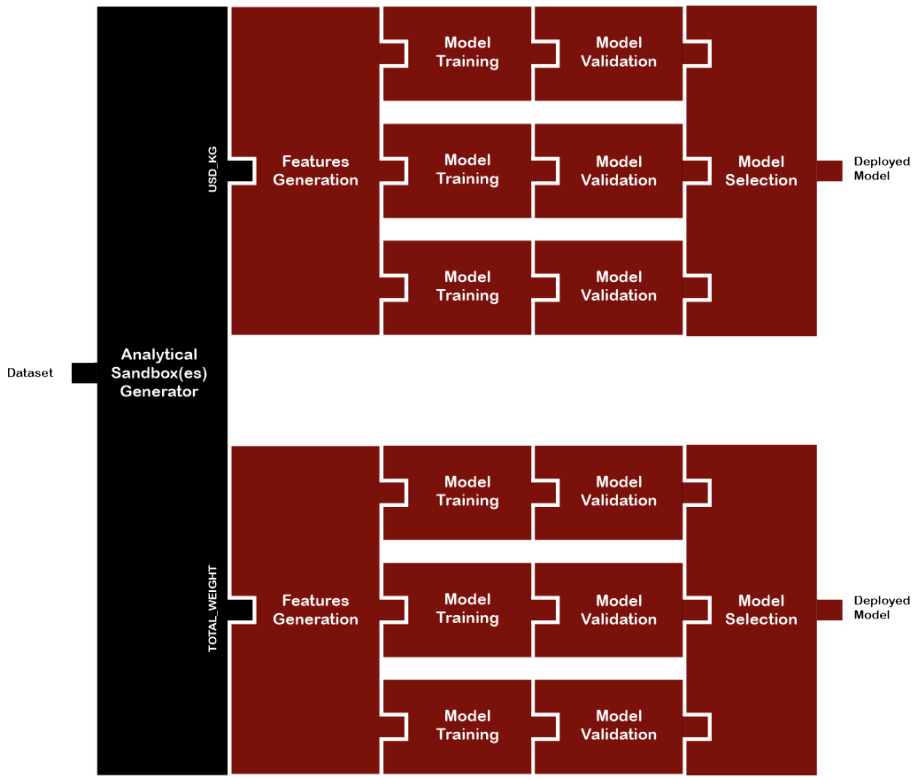

Figure 5.4: Data analysis pipeline schema per product type

- quarter_in_semester
- day_in_month
- day_in_week
- week_in_year
- week_in_semester
- week_in_quarter
- week_in_month

Then, the previously implemented seasonality analyzer is used. Every cyclic-granularity in the dataframe will be independently analyzed. All the seasonalities that are detected will be registered by adding the corresponding granularity names (e.g. semester_in_year) into a list. For example, for the dataframe shown in 5.5, the analyzer detected 8 seasonalities.

	usd_kg	semester_in_year	quarter_in_year	month_in_year	month_in_semester	month_in_quarter	quarter_in_semester	day_in_
2017-04-12	1.384	1	2	4	4	1	2	
2017-04-13	1.277	1	2	4	4	1	2	
2017-04-14	1.277	1	2	4	4	1	2	
2017-04-15	1.243	1	2	4	4	1	2	
2017-04-16	1.243	1	2	4	4	1	2	
2017-04-17	1.176	1	2	4	4	1	2	
2017-04-18	1.399	1	2	4	4	1	2	
2017-04-19	1.399	1	2	4	4	1	2	
2017-04-20	1.400	1	2	4	4	1	2	
2017-04-21	1.400	1	2	4	4	1	2	
2017-04-22	1.400	1	2	4	4	1	2	
2017-04-23	1.400	1	2	4	4	1	2	
2017-04-24	1.122	1	2	4	4	1	2	
2017-04-25	3.345	1	2	4	4	1	2	
2017-04-26	3.345	1	2	4	4	1	2	
2017-04-27	3.345	1	2	4	4	1	2	
2017-04-28	3.345	1	2	4	4	1	2	
2017-04-29	3.345	1	2	4	4	1	2	
2017-04-30	3.345	1	2	4	4	1	2	
2017-05-01	3.345	1	2	5	5	2	1	
2017-05-02	1.931	1	2	5	5	2	1	
2017-05-03	1.931	1	2	5	5	2	1	

Figure 5.5: Cyclic-granularities generated

At this stage, the seasonality patterns extraction process starts. The GMST decomposer implemented and described in the previous chapter is used to obtain all the seasonal patterns and trend from the time series. For the time series of the example seen so far, the GMST decomposer extracts the 8 seasonality patterns displayed in Figure 5.7 as well as the trend shown in Figure 5.6 (while the day_in_week seasonality is also extracted, it is not included in the figure as it's frequency is very high and its visualization is not ideal)*.

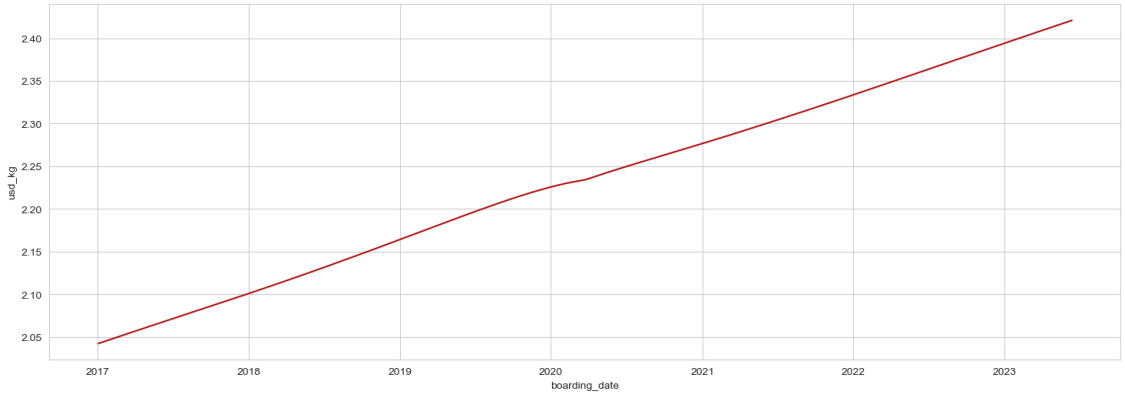

Figure 5.6: Extracted trend with the GMST decomposer

The described steps are implemented in the script *time_series_features_generation.py* located in the folder *data_analysis/features/scripts/*. The inputs are the dataframe with the observations

*An interesting remark to highlight from this example's decomposition is that the obtained fraction for the trend smoothing was 1.0. This means that the seasonality patterns capture much more information about the time series compared to the trend.

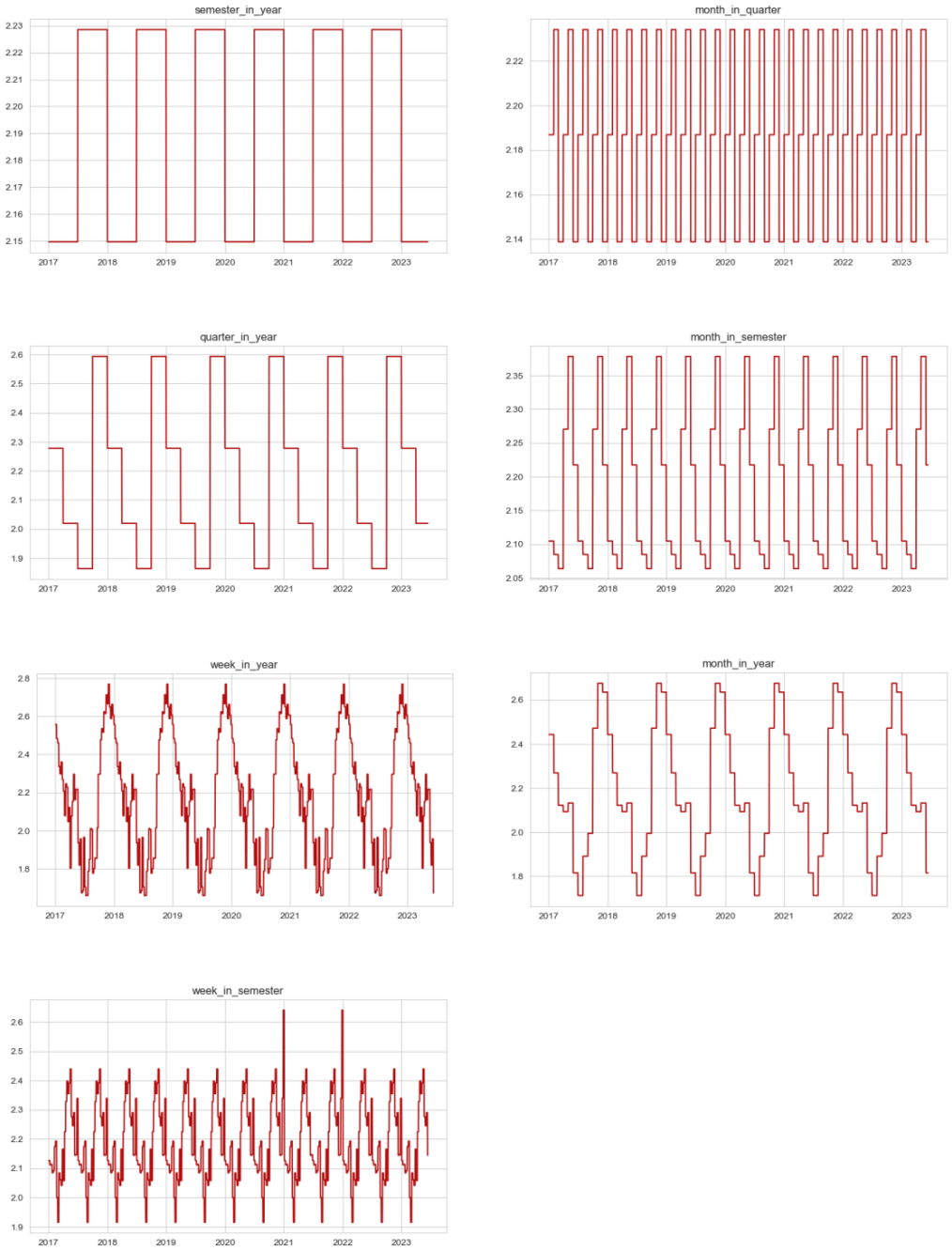

Figure 5.7: Extracted seasonality patterns with the GMST decomposer

with the cyclic-granularities and a list of the granularities to extract. The obtained arrays containing the original time series, the seasonal patterns, and the trend are split into two portions each: training and validation series. The training series will have 80% of the data and the vali-

dation series the remaining 20%. This is the output of the functionality.

5.4 Model(s) training

In this layer, the extracted features are used to train specific models. Thus, the training data previously generated will be consumed. The explanatory variables or predictors are the n_s seasonality patterns ($s_i(t)$) and the trend ($d(t)$). The target variable ($y(t)$) is the original time-series array. Forecasting models for predictions of 30, 60, and 90 days in advance will be trained. For this thesis, several models will be analyzed, specifically:

- Multivariate Linear Regression
- Generalised Additive Models (GAM)
- Prophet
- ARIMAX
- Gradient Boosting

The motivation to use each model and its configurations will be detailed in the next sections. In the case of the Prophet model, two different configurations will be studied.

5.4.1 Multivariate Linear Regression

A Multivariate Linear Regression model assumes that there is a linear relationship between the predictors and the target variable. It will be interesting to test if this assumption holds for some time series. Furthermore, it provides high explainability, as the model is very simple. This is desired to understand the real contribution of each seasonality pattern and the trend.

This model will be configured in a multi-step approach, which means utilizing past variables from a specific number of steps earlier to forecast future values. As explained before, steps 30, 60, and 90 days will be used. As such, the model can be written as follows:

$$\hat{y}(t) = \beta_0 + \sum_{i=1}^{n_s} \beta_{s,i} s_i(t - steps) + \beta_d d(t - steps) \tag{5.1}$$

The model to use is taken from the Statsmodels library and it will be fitted using the Ordinary Least Squares (OLS) method. [36].

5.4.2 Generalised Additive Models (GAM)

The motivation behind the use of the GAM model is the option to add non-linear relationships between the predictors and the target variable. This eliminates the rigid assumption of linearity, which may not necessarily be valid for all the time series.

This model will also follow a multi-step configuration using steps of 30, 60, and 90 days. Every predictor will be related to the target variable with a spline function $f(.)$. Therefore, the mathematical expression of the model is:

$$\hat{y}(t) = \beta_0 + \sum_{i=1}^{n_s} f_{s,i}(s_i(t - steps)) + f_d(d(t - steps)) \tag{5.2}$$

The training of the model is performed using the pyGAM library and specifically using the LinearGAM class [37].

5.4.3 Prophet

This model will be used to set two different configurations. To understand the motivation to do this, some context needs to be given. According to Taylor et al. [21], one of the features of Prophet is to provide a user-friendly model to analysts with deep domain expertise, but without wide experience in time-series forecasting. An analyst that knows the background of a time series could know a priori what seasonalities exist in the data and Prophet allows him to add this information as arguments of the model. However, specific time series may exhibit seasonal patterns that they do not necessarily know and/or that can't be easily expressed as arguments in the model[†]. Thus, a way to automatize the detection and use of seasonality patterns in the model should bring benefits from a user-experience perspective and potentially, from an improvement in the forecastings. This will be studied by proposing three different configurations for the model:

1. Prophet without regressors

$$\hat{y}(t) = d_{prophet}(t) + s_{prophet}(t) \tag{5.3}$$

[†]As it was explained before, the seasonality arguments in the Prophet models are numerical values that indicate the period of the seasonality. However, not all seasonalities have regular periods and as such, they can't be simply defined with a number.

2. Prophet using the extracted seasonal patterns

$$\hat{y}(t) = d_{prophet}(t) + s_{GMST}(t) \tag{5.4}$$

3. Prophet using the extracted seasonal patterns and trend as regressors

$$\hat{y}(t) = d_{prophet}(t - steps) + d_{GMST}(t - steps) + s_{GMST}(t - steps) \tag{5.5}$$

The first two configurations will be compared to see the effect of adding the seasonal patterns obtained with the GMST decomposition in the model. $d_{prophet}(t)$ refers to the trend extracted by the Prophet model. Prophet allows to add exogenous variables whose observations at time t are linearly related to the prediction at time t. Note that this is only possible because the future values of the seasonal patterns are known (as they are fluctuations). The third configuration includes the seasonal patterns but also the trend, whose future values are unknown. Therefore, this last configuration will be set in a multi-step approach, where differed observations of the exogenous variables ($t - steps$) will be used in the prediction. This last configuration will be compared to the other multi-step approaches seen so far.

According to its documentation, the Prophet model fits weekly and yearly seasonalities by default [31]. It is reasonable to think that an analyst with knowledge of exports of vegetables and fruits would know that these products tend to have yearly seasonality. As such, in the first configuration, only the default weekly seasonality will be excluded. In the second and third configurations, all the default seasonalities will be excluded, since the regressors already have that information. In the case of the Prophet's trend component, the documentation does not mention any way to exclude it. It would be ideal to do this for the second and third configurations as the trend is one of the regressors, but today that's unfeasible. However, it is reasonable to think that as both trends are being generated through different methods, they might be capturing different information. So having both might be beneficial. This will be discussed in the final results.

5.4.4 ARIMAX

As it was seen, in ARIMA models the prediction is performed considering a set of observations of the same variable that extend back to a certain point in the past. A type of these models called ARIMAX allows to add exogenous variables to improve the forecasting [24]. As it is desired to include the extracted seasonal patterns and trend as predictors, this is the model that will

be used[‡]. When adding additional predictors, the model makes use of a linear relationship between them at time t and the target variable at time t.

A multi-step approach for the ARIMAX model could be implemented by using past observations of the exogenous variables ($t - steps$) to make forecastings at time t. As such, the model's formula would be:

$$\begin{aligned}\hat{y}_t = \beta_0 + \sum_{i=1}^{n_s} \beta_{s,i} s_i(t - steps) + \beta_d d(t - steps) \\ +\phi_1 y'_{t-1} + \phi_2 y'_{t-2} + \ldots + \phi_p y'_{t-p} \\ +\theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \ldots + \theta_q \varepsilon_{t-q}\end{aligned} \tag{5.6}$$

The first line of the formula shows the linear relationship between differed observations of the exogenous variables and the target variable. The second and third lines show the autoregressive and moving average components respectively, that consider not one but a set of observations till point p and q in the past respectively. The y' symbolism is used to denote the differencing component d.

It is necessary now to define the orders of autoregressive, differencing, and moving average components (p, d, q respectively). As the exogenous variables include seasonal and trend information, short-term and long-term patterns are already being captured by the model there. Given this and in order to maintain the parsimony considering that exogenous variables added more parameters, it was decided to capture only first-order relationships for the autoregressive and moving average components. Additionally, a first-order differencing was chosen for the model to avoid introducing unnecessary distortions that could arise from higher-order approaches. Finally, the formula is:

$$\hat{y}_t = \beta_0 + \sum_{i=1}^{n_s} \beta_{s,i} s_i(t - steps) + \beta_d d(t - steps) + \phi_1 y'_{t-1} + \theta_1 \varepsilon_{t-1} \tag{5.7}$$

The library to use for implementing this model is Statsmodels.

[‡]Note that the SARIMAX configuration is not used, as the seasonal information is already in the exogenous variables.

5.4.5 Gradient Boosting

The motivation to use Gradient Boosting is to evaluate the performance of a model with fewer assumptions. As it is expected, this ensemble model will be configured in a multi-step approach for predictions 30, 60, and 90 days in advance. The number of boosting stages will be set to 2000. The maximum depth of a tree is 4 levels (not more to prevent overfitting) and the minimum number of samples to split a node is set to 10. Finally, the loss function chosen is the squared error and the learning rate is set to 0.01 to mitigate intense willingness.

To fit this model, the library Sci-Kit Learn will be used and specifically the module "ensemble" and the class "GradientBoostingRegressor".

5.5 Models' evaluation

The data contained information about exports of vegetables and fruits, specifically of 47 product categories in total. As it was mentioned before, the time series of two metrics by product were generated: *usd_kg* and *net_weight*. For every time series, 5 multi-step models and 2 non-multi-step models were fitted. For every multi-step model, 3 configurations were tested: 30, 60, and 90 steps.

In the previous section, it was mentioned that the two non-multi-step models were the Prophet without regressors and the Prophet using seasonal patterns as regressors. These models will be compared to see if the use of seasonal patterns positively impacts the prediction's accuracy. As these models are fitted for research purposes, they won't be deployed in the pipeline. The rest of the models (multi-step) will be compared between them, to get the best model by product. The idea is to select the best model by product. The best model by-product will be deployed in the pipeline.

The performance measure chosen for the models' evaluation is the Mean Absolute Error (MAE). The following sections will describe the results obtained for the non multi-step and multi-step models respectively.

5.5.1 Non multi-step models evaluation

This section addresses the evaluation of the models:

- Prophet without regressors (prophet_1)
- Prophet with seasonal patterns as regressors (prophet_2)

For every product (47 in total), the two models were fitted for the *usd_kg* and *net_weight* time series. By comparing the predictions and the real values in the test data, the MAE metric was obtained. For the group of time series about *usd_kg*, the best model among the two options was chosen. As it's shown in Figure 5.8, the Prophet model using the seasonal patterns as regressors had the smallest MAE in 89.4% of the time series.

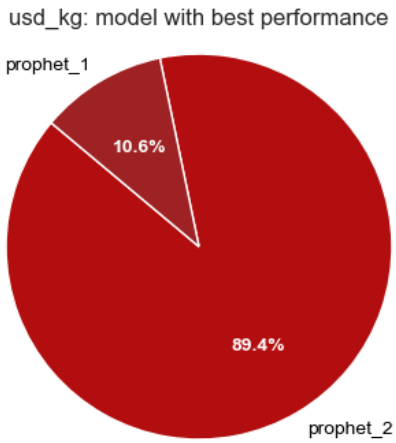

Figure 5.8: Comparison between models for all the products' time-series about *usd_kg*

The prophet_2 model was also the best performing one in the group of time-series about *net_weight*. As Figure 5.9 shows, this model had the smallest MAE in 59.6% of the cases. These results suggest that using the seasonal patterns extracted with the GMST decomposition method proposed in this thesis generally improves the forecasts of the Prophet model.

5.5.2 Multi-step models evaluation

The five models fitted by time series are:

- Linear Regression
- Generalized Additive Models (GAM)
- Prophet
- ARIMA
- Gradient Boosting

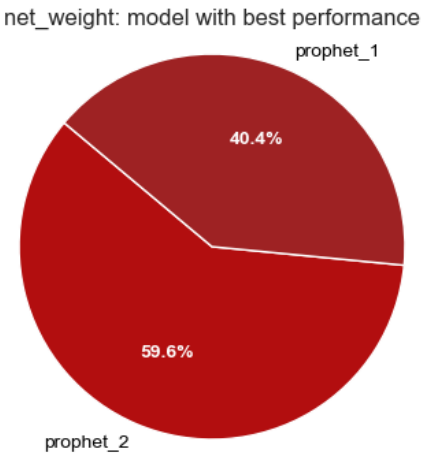

Figure 5.9: Comparison between models for all the products' time-series about *net_weight*

For each of the two time series by product (*usd_kg* and *net_weight*) and every option of steps (30, 60, and 90), the model with the smallest MAE was chosen. For example, Table 5.1 shows the performance summary for the time-series about *usd_kg* for the product with heading 7129090000. For predictions 30 steps before, the GAM model is the best option. For predictions 60 and 90 steps before, the Gradient Boosting and the GAM are the best-performing ones. As seen, the best model was dependent not only on the time series but also on the steps configuration. This behavior was observed for all the products (also in the group of time-series about *net_weight*).

Regarding the group of time series about *usd_kg*, no specific model stands out considerably in terms of performance for none of the steps' configurations. This can be observed from Figure 5.10, which illustrates the distribution of the best-performing models.

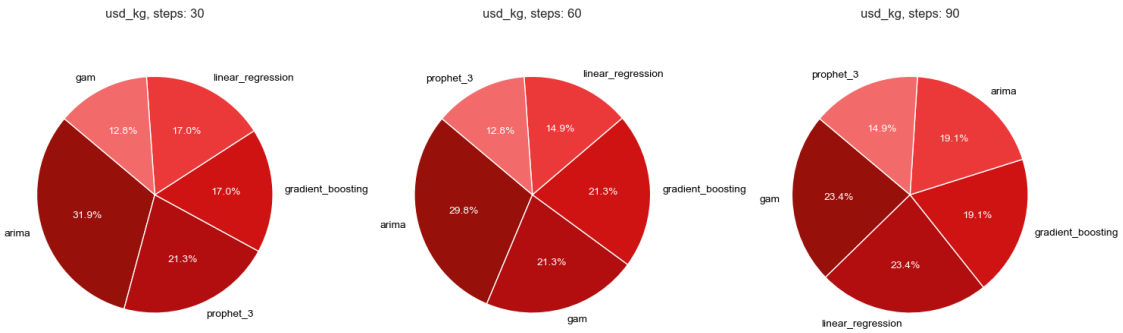

Figure 5.10: Best model among time-series of *usd_kg* for different steps configurations

Heading	Type	Steps	Model	MAE
712909000	usd_kg	30	linear_regression	4.472957
			gam	**4.071223**
			prophet_3	4.333122
			arima	4.432349
			gradient_boosting	9.880288
712909000	usd_kg	60	linear_regression	4.576499
			gam	4.377030
			prophet_3	4.490157
			arima	4.648773
			gradient_boosting	**3.997694**
712909000	usd_kg	90	linear_regression	4.802762
			gam	**4.493043**
			prophet_3	4.563068
			arima	4.847978
			gradient_boosting	7.142373

Table 5.1: Models performance for the product with heading 712909000 using the usd_kg time series and different steps

A different behavior can be observed in the group of time-series about *net_weight*, where the GAM model is generally the best performing one and the ARIMA and Prophet models the worse ones. This can be visualized in Figure 5.11, which also shows the distribution of the models with the least MAE.

These results suggest that the type of information that time series contains in some cases can be better explained with a specific model, but in other cases, the type of information isn't really relevant and no specific model stands out in terms of performance.

Following the pipeline design, for every product, the best model by type and steps was chosen and saved for deployment. In the end, 283 models (47 products x 2 time-series x 3 steps' configurations) are ready to be used to make forecasts 30, 60, and 90 days in advance. These models will be refitted every week, as new data will be ingested in the data management and analysis backbone and therefore, the time series will be updated.

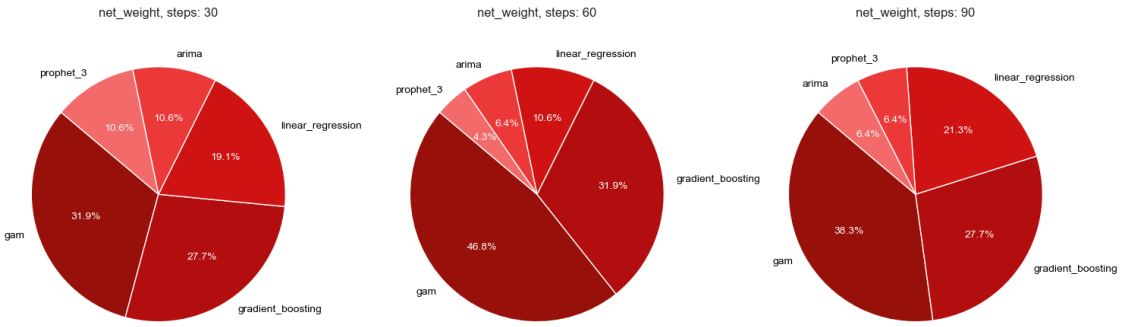

Figure 5.11: Best model among time-series of *net_weight* for different steps configurations

6 Conclusion

This thesis involved the proposal and implementation of a data management and analysis backbone addressing data on exports of vegetables and fruits in Peru. The multi-seasonal characteristics of the data motivated the proposal of a complex seasonality analysis approach to extract seasonal patterns and trend from the backbone's generated time-series.

The data management backbone is a key element not only to create a storage pipeline for the data, but also to manage the updates that happen in a weekly basis for this specific application. Furthermore, the use of HDFS in the Persistent Zone opens the possibility to horizontally scale up the storage capabilities in the future without big changes, as the data will only keep increasing.

The complex seasonality analysis related implementations and especially the proposed GMST decomposition method has shown encouraging results. It was shown that the programming approach to generate cyclic-granularities allows the definition of any time granularity that the user considers appropriate to explore. While the most suitable granularities for this application were implemented in this thesis, more granularities can be easily added. As explained, the granularities labeling approach adopts the ISO-8601 calendar, which aligns better with real-life scenarios than the Gregorian calendar. Furthermore, the extracted seasonal patterns through the GMST decomposition have shown to improve the prediction capabilities of the Prophet model in 89.4% of the time-series of *usd_kg* and in 59.6% of the time-series of *net_weight*. This evidence suggests that the implementations in this thesis can lead to improving the predictions of Prophet. Also, from a user perspective integrating the seasonal features into the Prophet

model removes the need for the user to define the seasonalities by himself, as he can rely on the automatic detection and extraction proposed in this thesis. This is remarkable because one of the key features of Prophet is its simplicity of use, which with the proposed approach becomes even more straightforward as defining the seasonalities isn't necessary anymore. Furthermore, several modeling approaches using the extracted features with the GMST decomposition were presented. It was observed that the most accurate model to adopt depends on each specific time series and not necessarily on the type of metric that it represents. More concisely, the best model isn't necessarily related to a specific type of time series, as is shown in Figure 5.10.

Finally, the data analysis backbone allows assigning the best possible model to each time series and to each multi-step approach. As it was seen, for a single time series, different models can be the best option for each step definition. Therefore, making specific models for each single case is the best approach. Furthermore, in the case of the time series about *usd_kg*, there wasn't a clearly more accurate model for all the products. A different behavior was observed when addressing the *net_weight* metric, where the GAM and the Gradient Boosting models were the most accurate ones in all the configurations.

These remarks are very insightful, as they suggest that when analyzing time series about exports of vegetables and fruits, the model that better explains an economic metric like the *usd_kg* is highly dependent on the product. However, in the case of a metric more related to the product's characteristics like the *net_weight*, specific models like GAM or Gradient Boosting tend to have the best results.

An essential aspect to highlight concerning the content of this thesis, evident to the reader, is the incorporation of teachings and technologies imparted by all three universities within the master's degree program. This wasn't on purpose but naturally arose, as this thesis was structured and executed as a complete data management and analytics project.

References

[1] The World Bank. Exports of goods and services (Available: https://data.worldbank.org/indicator/NE.EXP.GNFS.ZS?locations=PE

[2] S. Gupta, R. J. Hyndman, D. Cook, and A. Unwin, "Visualizing probability distributions across bivariate cyclic temporal granularities," *Journal of Computation of Graphical Statistics*, vol. 31, no. 1, pp. 14–25, 2022.

[3] ComexPeru. Importance of Foreign Trade in the Peruvian Economy. [Online]. Available: https://ucsp.edu.pe/archivos/comercioexterior/2017/Importancia-del-comercio-exterior-en-la-economia-peruana.pdf

[4] Superintendencia Nacional de Adimistracion Tributaria (SUNAT), *Arancel de Aduanas*, 1st ed. SUNAT, 2022.

[5] V. Macías-Carranza and A. Cabello-Pasini, "Climatology and evapotranspiration in winegrowing valleys of baja california," *Mexican Journal of Agricultural Sciences*, vol. 12, no. 5, 2022.

[6] R. Hyndman and G. Athanasopoulos, *Forecasting: Principles and Practice*, 3rd ed. OTexts, 2021.

[7] Superintendencia Nacional de Administración Tributaria (SUNAT). Databases of peruvian imports and exports. [Online]. Available: http://www.aduanet.gob.pe/aduanas/informae/presentacion_bases_web.htm

[8] ——. Complementary tables for databases of imports and exports. [Online]. Available: http://www.aduanet.gob.pe/ol-ad-tg/ServletTGConsultaTablas

[9] Database Technologies and Information Management Group, "Implementation of a (Big) Data Management Backbone," Universitat Politècnica de Catalunya, 2022.

[10] Apache Software Foundation. Apache Parquet: Concepts. [Online]. Available: https://parquet.apache.org/docs/concepts/

[11] The Apache Software Foundation. Hadoop: Setting up a Single Node Cluster. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html

[12] E. Zagan and M. Danubianu, "Hadoop: A comparative study between single-node and multi-node cluster," *International Journal of Advanced Computer Science and Applications*, vol. 12, no. 2, 2021.

[13] Apache Software Foundation. Apache Parquet: Configuration. [Online]. Available: https://parquet.apache.org/docs/concepts/

[14] Citus Data. Citus: Postgres at any scale. [Online]. Available: https://www.citusdata.com/product

[15] Oxford University Press. (2023) Oxford languages. [Online]. Available: https://languages.oup.com/

[16] H. Musbah, M. El-Hawary, and H. Aly, "Identifying seasonality in time series by applying fast fourier transform," *2019 IEEE Electrical Power and Energy Conference (EPEC)*, pp. 1–4, 2019.

[17] W. F. Guthrie, "Nist/sematech e-handbook of statistical methods (nist handbook 151)," *National Institute of Standards and Technology*, 2020.

[18] S. Gupta, R. Hyndman, and D. Cook, "Detecting distributional differences between temporal granularities for exploratory time series analysis," no. 20/21, 2021.

[19] K. Bandara, R. J. Hyndman, and C. Bergmeir, "MSTL: A Seasonal-Trend Decomposition Algorithm for Time Series with Multiple Seasonal Patterns," *International Journal of Operational Research*, 2021.

[20] R. B. Cleveland, W. S. Cleveland, J. E. McRae, and I. Terpenning, "STL: A seasonal-trend decomposition," *Journal of Official Statistics*, vol. 6, no. 1, pp. 3–73, 1990.

[21] S. Taylor and B. Letham, "Forecasting at scale," *The American Statistician*, vol. 72, 2017.

[22] A. M. D. Livera, R. J. Hyndman, and R. D. Snyder, "Forecasting time series with complex seasonal patterns using exponential smoothing," *Journal of the American Statistical Association*, vol. 106, no. 496, pp. 1513–1527, 2011.

[23] A. Dokumentov and R. J. Hyndman, "STR: Seasonal-Trend Decomposition Using Regression," 2021.

[24] G. Box, G. Jenkins, G. Reinsel, and G. Ljung, *Time Series Analysis: Forecasting and Control*, 5th ed. Wiley, 2015.

[25] T. Hastie and R. Tibshirani, "Generalized additive models," *Statistical Science*, vol. 1, no. 3, pp. 297–310, 1986.

[26] L. Yang, G. Qin, N. Zhao, C. Wang, and G. Song, "Using a generalized additive model with autoregressive terms to study the effects of daily temperature on mortality," *BMC medical research methodology*, vol. 12, p. 165, 2012.

[27] F. Dominici, A. McDermott, S. L. Zeger, and J. M. Samet, "On the Use of Generalized Additive Models in Time-Series Studies of Air Pollution and Health," *American Journal of Epidemiology*, vol. 156, no. 3, pp. 193–203, 2002.

[28] D. Witten, G. M. James, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning with Applications in Python*, 1st ed. Springer, 2023.

[29] Tableau Software LLC, "ISO-8601 Week-Based Calendar." [Online]. Available: https://tinyurl.com/mr3m3nt4

[30] D. C. Sayani Gupta, Rob Hyndman, *gravitas: Explore Probability Distributions for Bivariate Temporal Granularities*, 2020. [Online]. Available: https://cran.r-project.org/web/packages/gravitas/index.html

[31] (2023) Prophet documentation. [Online]. Available: https://facebook.github.io/prophet/

[32] M. Menéndez, J. Pardo, L. Pardo, and M. Pardo, "The Jensen-Shannon divergence," *Journal of the Franklin Institute*, vol. 334, no. 2, pp. 307–318, 1997.

[33] (2023) Scipy Jensen-Shannon Distance. [Online]. Available: https://tinyurl.com/5drpzubu

[34] H. N. Hajk-Georg Drost, *Jensen-Shannon Divergence*, 2022. [Online]. Available: https://search.r-project.org/CRAN/refmans/philentropy/html/JSD.html

[35] M. Mukaka, "Statistics corner: A guide to the appropriate use of correlation coefficient in medical research," *Journal of Medical Association of Malawi*, vol. 24, pp. 69–71, 09 2012.

[36] (2023) Statsmodels Linear Regression. [Online]. Available: https://www.statsmodels.org/stable/regression.html

[37] (2023) pyGAM. [Online]. Available: https://pygam.readthedocs.io/en/latest/index.html

Acknowledgments

Above all, I want to thank Professor Mariangela Guidolin from the University of Padua for multiple reasons. For her acceptance to supervise a thesis with data about my country and her invaluable guidance in defining a research objective. For all her provided inputs, knowledge, and expertise during the whole process of the thesis. For her promptness and clarity in addressing any question or uncertainty. I could not have chosen a better supervisor.

I would also like to thank Professor Alberto Abello from Universitat Politècnica de Catalunya for being my co-supervisor. His remarks and inputs especially in the data management part were fundamental and have brought a lot of value to this thesis.

I would also like to extend my acknowledgments to my fellow colleagues, with special mention to my friend Pietro Ferrazzi, for consistently being receptive to questions and particularly for his valuable input on some statistical aspects of this thesis.

I would like to also mention my specialization's university coordinator, Professor Massimiliano de Leoni for the academic organization and the support imparted this last year.

Finally, my gratitude to Professor Esteban Zimanyi, for leading and organizing the BDMA Master's Degree. It was one of the best experiences of my life.