”LUCIAN BLAGA” UNIVERSITY OF SIBIU

ENGINEERING FACULTY

DEPARTAMENT OF COMPUTER SCIENCE, ELECTRICAL AND ELECTRONICS ENGINEERING

DISSERTATION

Scientific Coordinator:

Assoc. Prof. Arpad Gellert

Graduate: Burghelea Zaharia

Master Program: Advanced Computing Systems

- Sibiu, 2023 -

”LUCIAN BLAGA” UNIVERSITY OF SIBIU
ENGINEERING FACULTY
DEPARTAMENT OF COMPUTER SCIENCE, ELECTRICAL AND ELECTRONICS
ENGINEERING

Machine Learning Methods for Monitoring and Controlling the Quality of PET Bottle Manufacturing Process

Scientific Coordinator:
Assoc. Prof. Arpad Gellert

Graduate: Burghelea Zaharia
Master Program: Advanced Computing Systems

- Sibiu, 2023 -

Abstract

This dissertation addresses the need of developing a sustainable PET bottle manufacturing process in industrial factories, as those desire to remain competitive while are challenged in the same time to adopt eco-friendly practices. Therefore, in order to deal with the environmental issue, this study proposes a Machine Learning-based software which aims to facilitate the monitoring and analysis of the PET bottle production.

The focus is on two aspects that determine the environmental impact, namely energy consumption and toxic emissions - which are collected as time series data from the PET bottle machines and further analyzed within the Machine Learning software.

This software consists of a pipeline with two main components: the first one aims to categorize the operating modes of the PET bottle machines (using time series clustering methods, such as Hidden Markov Models), while the second one focuses on flagging huge deviations within the environmental data (using anomaly detection techniques, such as Deep Learning-based Autoencoders). Those two components build a close connection, as the clustering part is applied first in order to enable an easier flow in detecting anomalies.

Through the application of those techniques, this study aims to enable more sustainability and to reduce the toxic environmental impact in PET bottle production. Additionally, while some parts of the software are specific to the PET industry, the project is also designed to be versatile, which paves the way for the adoption of its main components in other industries.

Contents

Chapter 1

Introduction

Plastic pollution has become one of the most important environmental problems and it begs for eco-friendly practices in its various subfields, notably also in PET (Polyethylene Terephthalate) bottle manufacturing. As an example, making a 500 ml. PET bottle generates more than 100 times the toxic emissions than producing an equally sized bottle out of glass. The situation gets more serious given the significant amount of CO2 that PET bottle production generates each year - this has the aftermath of contaminating the air and further leads to health issues among the local residents [1].

Industrial factories might often present insufficient knowledge when it comes to enable eco-friendliness in production, henceforth it is essential to identify and come up with efficient methods to improve the manufacturing process in order to minimize the toxic environmental impact.

This dissertation focuses on the development and application of machine learning techniques - specifically, time series clustering and anomaly detection algorithms. The clustering stage aims to identify patterns within the time series, where the final goal is to categorize the operating modes of the machines that produce PET bottles. Furthermore, those clusters are meant to aid the process of flagging anomalies in order to obtain valuable insights regarding the manufacturing process.

Since this project is developed with a general approach in mind, it also expands the applicability beyond that of PET bottle production. While some aspects of the software are particular for this specific industry, the core idea of fusing clustering and anomaly detection algorithms can be further applied to promote eco-friendliness in other fields too.

In what's to come, the structure of this study goes as follows: Chapter 2 outlines the PET bottle manufacturing process alongside some of the current issues and existing solutions. Afterwards, it continues by reviewing the common approaches found in literature applied to time series clustering and anomaly detection.

Chapter 3 describes in detail the theoretical aspects of the algorithms that this project will be using. Mainly: Hidden Markov Models, Time Series Decomposition and LSTM-based Autoencoders can be found. The focus is onto the mathematical part, but also targeted towards which components of the algorithms will be further implemented practically.

Chapter 4 focuses mainly on the implementation of the outlined algorithms, but it also describes the way those techniques were applied in order to enable a pipeline that can promote eco-friendliness in the PET industry.

Chapter 5 presents the obtained experimental results through the application of the implemented machine learning techniques.

Lastly, Chapter 6 finishes the work by the drawing overall conclusions about this project and gives some ideas about potential future studies.

Chapter 2

Background and Literature Review

This chapter starts by describing the process of PET Bottle manufacturing, followed by an outline of the current issues and existing solutions. Afterwards, an overview of what other researchers used in order to perform clustering on time series is provided. The chapter ends with a review of some of the existing techniques that are employed in literature for detecting anomalies.

2.1 PET Bottle Manufacturing Process

While the PET bottle manufacturing procedure may differ from one factory to another, a general approach can be provided by focusing on how individual machines handle the process. The procedure involves two steps: preform production and bottle forming and can be further divided into single or two stage blow molding techniques. Figure 2.1 illustrates the general process.

On a detailed level, preform production starts by heating and melting the PET resin. While the resulting fluid is hot and homogenized, it is colored and stirred until an uniform color is obtained. Afterwards, the result is introduced into a cavity from within an injection

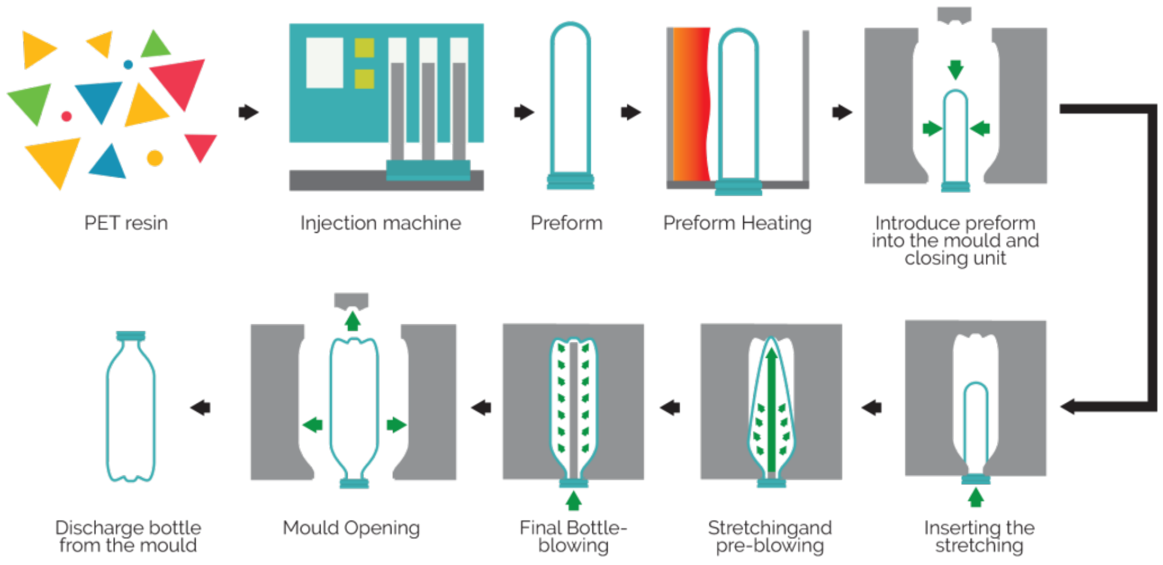
PET resin
Injection machine
Preform
Preform Heating
Introduce preform into the mould and closing unit
Inserting the stretching
Stretchingand pre-blowing
Final Bottle-blowing
Mould Opening
Discharge bottle from the mould

Figure 2.1: PET Bottle Manufacturing Process [2]

molding machine (operating at around 280°C), thus creating the preform - this preform is a cylindrical tube with a threaded neck, but significantly smaller than the final product. In the end, the preform is rapidly cooled in order to maintain its shape [3] [4].

During bottle forming, the preforms are heated again (up to 110°C) so that they become pliable and then those are placed into a cavity with the desired shape and size. Finally, clean and cool compressed is blown into the preform causing it to expand and stretch creating the bottle structure - this part also cools the preform and the bottles keep their final shape [4].

In contrast to the two phase process outlined above, a single stage procedure skips the first cooling part and directly shapes the preforms into the final desired product (all steps are performed within the same machine). As an advantage, this minimizes the energy consumption, however the two-stage alternative is more flexible - especially when the demand for the types of bottle varies, or when those bottles are meant to be transported at different locations [5].

2.2 Current Issues and Solutions

One of the Major defects in the production of PET bottles is the generation of Acetaldehyde (AA), which gives an acidic taste in most citrus fruits. This compound can significantly degrade the taste of the drinks found in PET bottles, particularly in bottled water as other flavoured drinks can mask the unwanted taste. The production of AA appears when the PET material's temperature surpasses 260°C - typically occurring within the injection molding of the preforms, if the process isn't properly managed. Additionally, excessive temperature also increases the energy consumption [6].

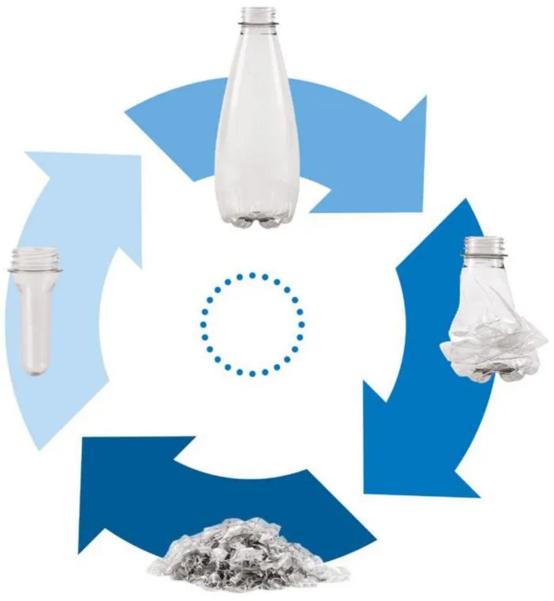

Figure 2.2: "Zero Waste" model [7]

Such issues, alongside the toxic environmental problems caused by plastic pollution (described in Chapter 1) have garnered significant attention from various researchers. Currently, it is widely accepted that the only path to achieve absolutely no waste is through the means of a circular system - as shown (specifically for PET bottles) in Figure 2.2.

In the present day, there hardly exist such circular systems, which can take a no longer usable product and recreate it anew - without releasing more toxic emissions in the environment. Unfortunately, that's not even close to be the case for the plastics industry, as only a small portion of the PET bottles are collected for recycling and even a much smaller number of those manage to find their way back into new bottles [7].

As a result, smaller steps are usually taken in order to address this issue. Solutions based on machine learning techniques are found to be discussed and implemented in various forms.

Although not related specifically to anomaly detection, an automatic sorting system for plastic recycling is proposed in [8], where a machine vision approach is employed to distinguish PET bottles from non-PET bottles. More specifically a Self Organizing Map is used to extract features from images representing the plastic items, followed by an application of a neural network which is trained to classify the resulting features.

After China's prohibition of importing plastic waste, researchers in [9] developed a Deep Learning approach to determine the status of PET bottles - e.g. bottles that include the presence of a seal, a cap, both items or none of the items. Furthermore, they deployed their neural network (a SqueezeNet model) on a Raspberry PI aiming to promote only high quality bottles for recycling.

A closely related application (to this dissertation) of anomaly detection within the PET industry is presented in [10], where the authors propose a two-phase anomaly detection system for the preform manufacturing process. In contrast to this project, the flow is reversed as the first stage starts with a Long Short-Term Memory network (LSTM) employed to flag anomalies within various features (temperature, pressure etc). The second stage takes the detected anomaly data (to which the defect label is unknown) and clusters it, so that it can further alarm defects to the decision makers inside the factory.

2.3 Time Series Clustering

Clustering is a technique where similar data is grouped into homogeneous categories to which there is no prior knowledge or information - in contrast, classification follows the same process, however with the groups being already defined.

Specifically for this project, one type of clustering that will be performed is time series clustering - a time series is a sequence of values recorded at successive timesteps. The main difference over the classical clustering arises due to the fact that the data is manifesting temporal dependencies over time [11].

To elaborate further, the goal is to group the operating modes of the PET bottle machine. While the exact number of possible clusters is unknown, as we don't even know the specific machine used, we are aware that there must be some working modes (e.g. the machine can be shut down, can heat, can cool and so on). Such modes might be observed for a longer period of time (e.g. if the machine is turned off over the night) or they can switch frequently (e.g. in case the cooling comes immediately after heating one product and then this processes is repeated).

One common algorithm used for time series clustering is K-means, which falls into the category of Partitioning Clustering methods. K-means divides the data into *k* distinct clusters by looking at the similarity between the data points and each cluster. While this is a simple, efficient and a fast technique, the disadvantage lies in the fact that inherently it neglects the temporal dependencies [12]. The issue can however be overcomed by mapping the time series to the frequency domain (as an example), however this also adds an extra layer of complexity.

On the opposite spectrum, there are Hierarchical Clustering methods, such as the Agglomerative Hierarchical Clustering (AHC) algorithm. In this case, each data point is considered a cluster initially and then the clusters are gradually merged until only the most relevant ones remain. Like previously, the temporal information isn't taken inherently into account here either [11] [12].

In order to overcome the previous mentioned limitation, other algorithms can be employed, such as the Model-Based Clustering methods. Within this category falls also the main algorithm used in this project, namely the Hidden Markov Model (HMM) which does take into account the temporal dependencies. HMM will be further described in Chapter 3.

The idea of using HMM to categorize sequences of data can also be seen in a related field, namely applied to music genre classification. Without going into much details, this is described in [13], where the audio files are divided using a sliding window and then each frame is weighted by a Hamming window to limit edge effects. Furthermore, the energy, the derivatives, the acceleration coefficients and the first 12 Mel-Frequency Cepstral Coefficients (MFCCs) are computed to build a feature vector for each frame, which is then used to train a HMM in order to automatically determine the music genre.

While the music part isn't directly relevant for this dissertation, the important part that should be highlighted is the application of a model-based algorithm on time series. Moreover, this article also shows a way on how to select the number of hidden states (or groups) within the HMM, which is problematic as we are not aware of the number of possible operating modes within the PET bottle machines. The paper suggests increasing the number of hidden states until no noticeable improvement is seen in performance [13].

An alternative for choosing the number of states in HMM application is described in [14], were a Monte-Carlo based cross-validation approach was proposed. Also [15] goes in depth to discuss this notorious problem of determining the optimal number of hidden states and inclusively mentions the usage of AIC (Akaike Information Criterion) and BIC (Bayesian Information Criterion). A particular case of those will also be employed in this study, as it will be described in Chapter 4.

2.4 Anomaly Detection

Anomaly detection (or outlier detection) is the process of flagging data points which significantly deviate from the expected pattern of the other items within the whole dataset. These inconsistent data points are known as anomalies or outliers and identifying them can provide valuable information in many applications. From detecting fraud transactions, diagnosing medical risks or identifying environmental problems, the usage of anomaly detection is widespread across many fields [16].

With regard to this project, anomaly detection is performed with the final goal to flag huge deviations within the PET bottle machine data, which could indicate potential mechanical or environmental problems. Flagging such anomalies early can also prevent more serious issues down the line and make the process an eco-friendly one.

Furthermore, a prior application of time series clustering can provide valuable information for the anomaly detection process - as by splitting the data into different working modes it can simplify the detection procedure for each operating mode and make the flow go smoother. This step is not strictly necessary, but besides the mentioned advantages, it can also give more reliability to the process.

One approach to identify anomalies is described in [17] and it suggests the application of joint statistical moments. By an analogy to the principal component analysis (PCA), the principal kurtosis vectors are used to signal the principal directions along which the outliers are most likely to appear - the kurtosis is serving as a reliable metric to detect outliers, since it represents the ”tailedness” of the probability distribution (or in other words, how often the outliers occur).

Hidden Markov Models, which were previously discussed in the context of time series clustering, can also be adapted for anomaly detection. More details can be found in Section 3.1.3, however as a short introduction, one application of HMM is to compute the likelihood of observing a particular sequence given a model. By taking advantage of this, anomalies can be detected by identifying the sequences which are unlikely to belong to any of the hidden states (or operating modes in the context of this dissertation) [18].

Modern state-of-the-art methods for anomaly detection are often based on Deep Learning techniques, as described in [16]. One of them being the Autoencoders (AE), which aim to learn a representation space of a lower dimension for the input and accurately reconstruct it. The same procedure can be found applied in data compression, where an AE can learn to encode and decode a particular signal or image. The core idea of using this technique in anomaly detection, is that the learned representation is enforced to be remembered in order to minimize the reconstruction error during the decoding part. Thus, since anomalies are much harder to be reconstructed, those decoded data points which significantly deviate from the original points are flagged as anomalies.

A special type of Autoencoders, based on Recurrent Neural Networks - specifically Long Short-Term Memory Networks (LSTMs), can be applied to identify anomalies to ECG time signals, as described in [19]. The advantages of using LSTMs over classical neural networks are due the fact that they take into account temporal dependencies and are capable of remembering information over longer periods, making them more suitable for time series.

Furthermore, the authors in [10] employ a LSTM network to detect possible anomalies within the injection molding stage of the PET bottle production. This closely resembles this project, however the flow is reversed - as there the clustering step isn't used to aid the anomaly detection phase, but to further cluster the detected anomaly data in order to improve the process. Additionally, the current study doesn't focus on a specific stage within the PET bottle manufacturing process, but as a whole.

Chapter 3

Theoretical Aspects

This chapter describes the algorithms that will be further implemented in this project. At first, Hidden Markov Models (HMMs) will be explored, followed by an examination of Time Series Decomposition, which is relevant since the PET bottle manufacturing data presents unwanted fluctuations. Lastly, in the context anomaly detection, LSTM-based Autoencoders will be presented.

3.1 Hidden Markov Models

A hidden Markov model (HMM) is a statistical model which represents systems directed in the background by hidden (or unobservable) states. Even though those states are hidden, the emissions (or the observable outputs) are dependent on them and this leads to the possibility to analyze the hidden aspects, based on the observable outputs.

Each of the states posses a probability of transitioning to another state (which can also mean that it can remain in the same state, or in other words, perform a self-transition). Furthermore, those are subject to an important constraint, in which the upcoming states depend solely on the current ones. As an example, if we were to model weather patterns, then a hidden Markov model assumes that if we desire to forecast tomorrow's weather - then we will do that based on today's weather, ignoring any information prior to this day. This principle is more concisely called the Markov Property, which is at the core of Markov Chains [20].

3.1.1 Markov Chains

Markov Chains (or Markov Processes) are memory-less stochastic processes that model the transition probabilities between sequences of random variables, each of those variables representing a state in the system [21]. Formally, if we define the sequence of states $Q = q_i,\ i \in \overline{1,n}$, then for any state k, the Markov Property can be described as:

$$P(q_k|q_{k-1},...,q_2,q_1) = P(q_k|q_{k-1})$$

In addition to the states Q, a mathematical description of a Markov chain also include a matrix holding the transition probabilities, $A = a_{ij},\ i,j \in \overline{1,n}$, where each a_{ij} represents the probability of switching from the state i to the state j. Initially, the system can be found according to a certain probability in any of the system's state, as given by the initial probability distribution: $P = p_i,\ i \in \overline{1,n}$.

Figure 3.1 illustrates this by modelling a simple weather pattern. Each circle represents a possible state (including the initial pseudo-state "START") and the arrows represents the transition probabilities. By denoting r as "rainy" and s as "sunny", then $p_r = 0.6$ (the chance to start with a rainy day) and $p_s = 0.4$ (the chance to start with a sunny day). Also, the transition probabilities can be written as: $a_{rr} = 0.7,\ a_{rs} = 0.3,\ a_{sr} = 0.4$ and $a_{ss} = 0.6$.

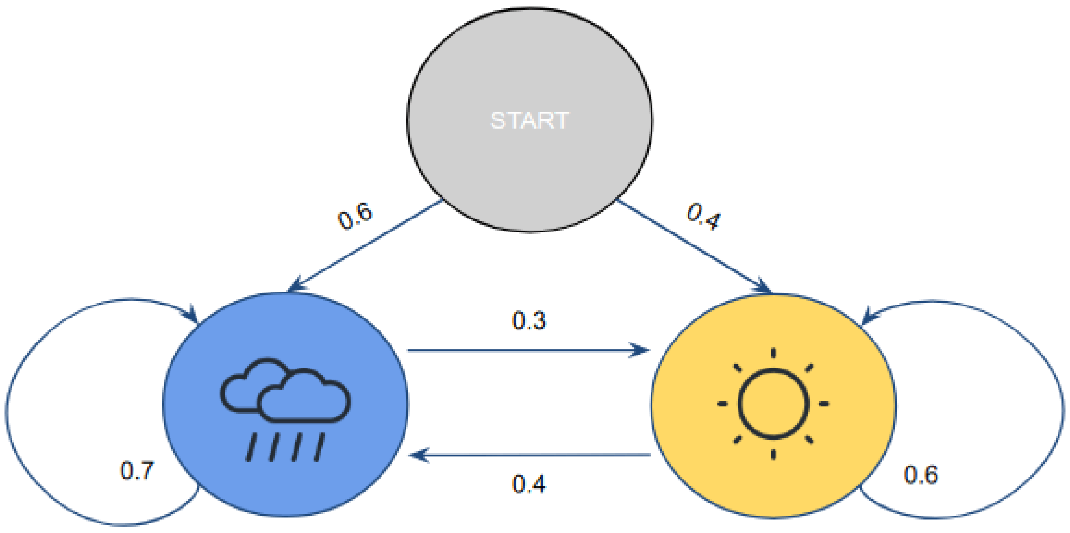

Figure 3.1: Markov Chain illustrated [22]

3.1.2 The Hidden Markov Model

In contrast, the Hidden Markov Model is an extended variant of the Markov Chain, in which the chain is augmented with an additional "layer" of emissions $B = b_j(o_k)$, each of which represents the likelihood of emitting the observation o_k while being in the q_j state. Above $O = o_k, k \in \overline{1,m}$ is the set of all possible observations.

On top of the Markov Property, the nature of the model also brings out the Output Independence property:

$$P(o_k|q_1,...,q_n,o_1,...,o_m) = P(o_k|q_i)$$

That is, the probability of observing the output o_k is based only on the state that produced the output (in this case q_i) and not on any other state or observations [20].

Figure 3.2 continues the previous example about Markovv Chains by including an additional layer with emissions, in order to illustrate a simple Hidden Markov Model. Furthermore, the previous notations are kept, but additionally J is added to denote "jogging",

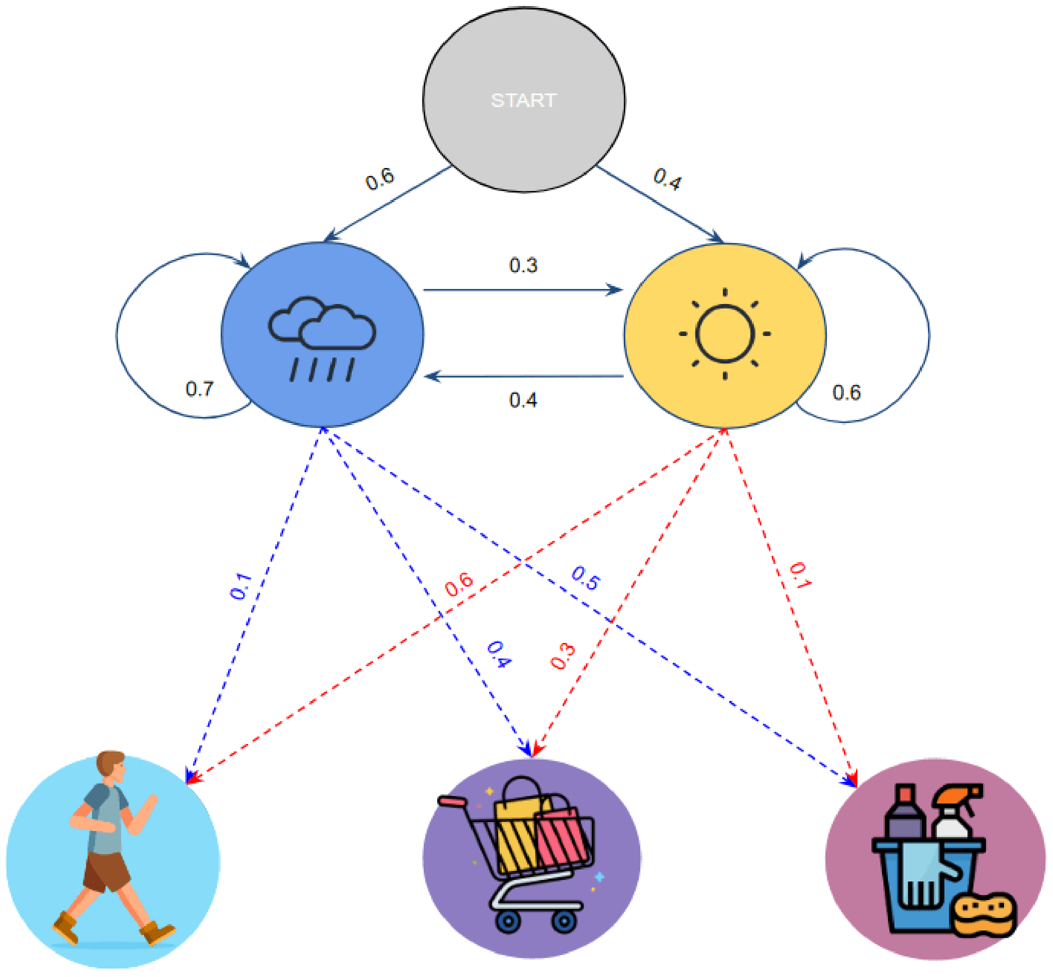

Figure 3.2: Hidden Markov Model illustrated [22]

S to denote ”shopping” and C to denote ”cleaning”. Then, in case of a rainy day we have: $b_r(o_J) = 0.1, b_r(o_S) = 0.4$ and $b_r(o_C) = 0.5$, whereas in case there is a sunny day, we have: $b_s(o_J) = 0.6, b_s(o_S) = 0.3$ and $b_s(o_C) = 0.1$.

In [23], Rabiner characterized Hidden Markov Models by three fundamental problems: computing the likelihood of observing a specific sequence, decoding the sequences and learning the model parameters. Moreover, in [20] Daniel Jurafsky and James H. Martin offers a description to all three HMM applications, which are also summarised below.

3.1.3 Computing the Likelihood

The first mentioned problem that a HMM can tackle is to compute the probability of observing a specific sequence, given a model.

In order to illustrate this, we can take as an example a student which is about to hold their dissertation followed by the admission to some post-graduate studies. In this case, the model's observable states would be the student's marks, while the hidden sequences are resembled by the student's understanding of the subjects as well as the preparedness. Even though the grade's influence would largely be based on the student's knowledge (a high grade is direct proportional to a high display of understanding), what makes this model hidden is the uncertainty that comes from other factors, such as stress, exam difficulty, level of burnout and so on, which contribute to the hidden nature of this model.

If we knew the sequence of hidden states this would have been straightforward to compute, namely:

$$P(O|Q) = \prod_{i=1}^{t} P(o_i|q_i)$$

Above we are trying to find the probability of the observable sequence O (of length t) while the system is in the corresponding states Q (assuming that in this specific case we know exactly what the sequence is).

However, as the uncertainty arises, we are unaware of the exact sequence Q, therefore we must take into consideration every possible sequence. Namely, for n hidden states, n^t computations would be required:

$$P(O) = \sum_{Q} P(O,Q) = \sum_{Q} P(O|Q)P(Q)$$

Given the exponential complexity, this is impractical to use for large sequences, hence in practice the Forward Algorithm is used in order to compute the likelihood more efficiently [24].

The improvement that the Forward Algorithm brings to the table is due to the usage of dynamic programming. As an intuitive example of why this can be performed more optimally, we can think of two sequence which differ only at a single timestep. Then, using the previous mentioned formula those would be computed separately, however the Forward Algorithm makes sure that the common parts are computed only once.

The Forward Algorithm

The Forward Algorithm is a kind of algorithm (which takes advantage of dynamic programming) that stores the intermediate values (or the forward probabilities), denoted as α, while it builds up the calculation of the desired probability [20] [24]. The algorithm contains three main steps:

1. Initialization: The algorithm starts by initializing the forward probabilities at the first timestep (α_1) for each possible state. This is obtained as the product of the initial state probability (p_i) and the emission probability of observing the first output while being in the state i, namely $b_i(o_1)$.

$$\alpha_1(i) = p_i b_i(o_1), \quad i \in \overline{1,n}$$

As a small note, since the observations at each timestep are known (we are trying to compute the likelihood of observing them), above and in what's to come those are simply indexed using their respective timestep.

2. Recursion: For the remaining timesteps, the α values are computed for each state j as the sum over all the possible previous states i. This is obtained as the product of the previous intermediate value α_{t-1}, the likelihood of switching from each state i to the state j (a_{ij}) and the emission probability of observing the next observation, namely $b_j(o_{t+1})$.

$$\alpha_t(j) = \sum_{j=1}^{n} \alpha_{t-1}(j) a_{ij} b_j(o_{t+1}), \quad j \in \overline{1,n}$$

This step is illustrated for a better understanding in Figure 3.3. In other words, the probability of ending up in state j is equal to the sum of all the different ways of arriving in

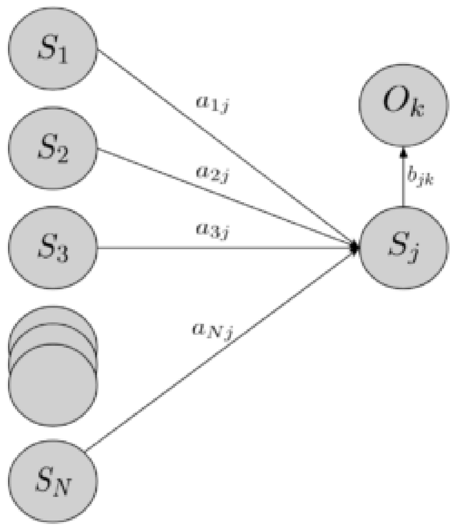

Figure 3.3: Computing the forward probabilities illustrated [22]

the state j from the previous timestep, while seeing the next observation and also keeping track of the full "path" trough the forwards probabilities (α) at each timestep [22].

3. Termination: As there are no further possible transitions, this step finishes the algorithm by summing the final forward probabilities (denoted as T in the last timestep) over all states j - which the probability of the sequence O being observed.

$$P(O) = \sum_{j=1}^{n} \alpha_T(j)$$

3.1.4 Decoding

The second problem that HMM can deal with is to decode which sequence of hidden states most likely led to an observed sequence of outputs, given a model.

Sticking with the same example as for the previous problem, in this case if we already have the student's results, then this decoding procedure aims to find for each grade the underlying cause that led to it (such as the level of understanding, preparedness, stress and so on).

One simple, but inefficient way of obtain the most likely sequence, is to apply the forward algorithm for each possible sequence and then just pick which is the most probable sequence of hidden states that caused the outputs. However, like in the previous section we will face a high complexity and in practice the Viterbi algorithm is used instead.

The Viterbi Algorithm

Similarly to the Forward Algorithm, the Viterbi Algorithm also takes advantage of dynamic programming, however instead of computing the likelihood of observing a specific sequence, it computes the most probable sequence of hidden states that led to the observed sequence. Additionally, the Viterbi algorithm contains one more component, namely backpointers, which holds the best path that led to a certain state [20].

The Viterbi Algorithm goes as follows:

1. Initialization: The algorithm starts by initializing the maximum probability matrix (v) for each state i, by multiplying the initial state probabilities (p_i) with the emission probability of the observation at the first timestep, namely $b_i(o_1)$. Also, the backpointers for the first timestep are initialized so that they don't point to any state (as there doesn't exists any previous state).

$$v_1(i) = p_i b_i(o_1), \quad i \in \overline{1,n}$$

$$bp_1(i) = 0, \quad i \in \overline{1,n}$$

2. Recursion: This step computes the likelihood of the path ending up in the state i at time t. This is achieved by finding the maximum possible probability of transitioning from the previous state j to the state i (a_{ji}) while taking into account which state gave the maximum probability for the previous timestep, namely $v_{t-1}(j)$. Additionally, this is multiplied by the corresponding emission probability of observing the next known observation. The backpointer will record which state is giving the maximum probability at the current step.

$$v_t(i) = \max_{j=1}^{n} \left[a_{ji} v_{t-1}(j) b_i(o_t) \right], \quad i \in \overline{1,n}$$

$$bp_t(i) = \operatorname*{argmax}_{j=1}^{n} \left[a_{ji} v_{t-1}(j) b_i(o_t)\right], \quad i \in \overline{1,n}$$

3. Termination: Lastly, the algorithm identifies the state (q_T) which gives the best possible score at the last timestep (T).

$$q_T = \operatorname*{argmax}_{i=1}^{n} [v_T(i)]$$

To finish the algorithm, there is also the need to backtrack through the recorded states - by following the backpointers. This is done, so that the most probable sequence of hidden states is obtained, starting from the final state (q_T) and going back to the first timestep. The obtained sequence is the Viterbi path, or the most probable sequence of hidden states that could produce the observed sequence [20] [23] [24].

3.1.5 Learning

The final problem that HMM can deal with is to find the optimal model parameters (the transition probabilities and the emission probabilities), given an observed sequence of outputs.

If we consider again the student example, we may imagine that based on the test scores from a student, obtained over the course of a semester, we can try to determine the influence of the hidden states (which includes the level of understanding, preparedness etc) which led to these results. The learning task in this case would be to effectively model the student's performance over the semester.

The standard approach for this task is the Forward-Backward Algorithm, also knows as the Baum-Welch Algorithm (a specific instance of the Expectation Maximization algorithm). The algorithm lets us train the parameters (mentioned above) through an iterative process. Essentially, it initially estimates the probabilities and further employ those to derive a better estimate across multiple steps - thus, iteratively obtaining a better model [23].

The Backward Probability

Beside the forward probability (introduced alongside the forward algorithm), this algorithm further adds the backward probability (β), which is the likelihood of encountering the known observations from the $t+1$ timestep until the end (assuming that we are at time t) [20]. This is calculated similarly to the forward probability:

1. Initialization: We start with the initialization of the backward probabilities for each state in the last timestep (T). Since we know that we are already there, those probabilities are simply equal to 1.

$$\beta_T(i) = 1, \quad i \in \overline{1,n}$$

2. Recursion: For the prior timesteps ($t \in \overline{1,T}$), the backward probability is computed by multiplying the upcoming values (previously computed as β_{t+1}), the probability of switching from each state i to the state j (a_{ij}) and the emission probability of observing the next observation, namely $b_j(o_{t+1})$.

$$\beta_t(i) = \sum_{j=1}^{n} \beta_{t+1}(j) a_{ij} b_j(o_{t+1}), \quad i \in \overline{1,n};\ t \in \overline{1,T}$$

In the same style as for the forward probabilities, this step is illustrated for a better understanding in Figure 3.4. In other words, the probability of ending up in each state i is calculated by taking into account the possibility of priorly being in every possible state, while seeing the next observation and also keeping track of the full "path" trough the backwards probabilities (β) at each timestep [22].

3. Termination: In the last step, the probability of the observed sequence is calculated by summing up the products of the initial state probabilities, the emission probabilities of the first observation and the backward probabilities at the first timestep:

$$P(O) = \sum_{i=1}^{n} p_i b_i(o_1) \beta_1(i)$$

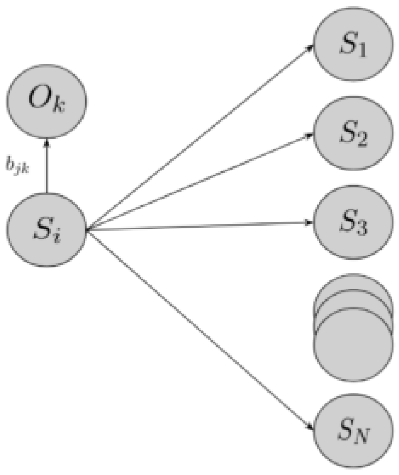

Figure 3.4: Computing the backward probabilities illustrated [22]

The Baum-Welch Algorithm

To go further with the Baum-Welch algorithm, where the aim is to find the transition and the emission probabilities, we can start by intuitively estimate the transition probabilities as:

$$\hat{a}_{ij} = \frac{\text{expected switches from state i to state j}}{\text{expected switches from state i anywhere}}$$

The estimated number of shifting from state i to state j, further denoted as $\xi_t(i,j)$, can be calculated using the following formula:

$$\xi_t(i,j) = \frac{a_{ij} b_j(o_{t+1}) \alpha_t(i) \beta_{t+1}(j)}{\sum_{k=1}^{n} \alpha_t(k) \beta_t(k)}, \quad i,j \in \overline{1,n}$$

That is, the numerator represents the probability of being in the state i at time t and then in the next step to switch to the state j. Since ξ is a probability, the probabilities of all transitioning from state i to any other state must sum up to 1 and this is where the denominator ensures that this happens - basically the denominator holds the place of a normalization constant [20].

In the end, the anticipated number of switches from the state i to the state j, is the sum of all t belonging to ξ:

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T} \xi_t(i,j)}{\sum_{t=1}^{T} \sum_{k=1}^{n} \xi_t(i,k)}, \quad i,j \in \overline{1,n}$$

In the same style, the emission probabilities can be estimated as:

$$\hat{b}_i(o_k) = \frac{\text{expected count of seeing the observation k in state i}}{\text{expected count of times being in state i}}$$

The expected count of being in the state i at time t, namely $\gamma_t(i)$, can be obtained by following a similar approach to the one used for finding the transition probabilities, but by keeping in mind that those are concerned with the likelihood of being in a particular state at a specific time according to the observable sequence.

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{k=1}^{n} \alpha_t(k)\beta_t(k)}, \quad i \in \overline{1,n}$$

The numerator from above represents the joint probability of seeing the sequence up to the t timestep (while ending in state i) and that of noticing the upcoming known observations from time $t+1$ to the end. The denominator ensures that the sum of all $\gamma_t(i)$ equals 1. Using this, the estimated emission probabilities follows by summing up $\gamma_t(i)$ in the cases when the observed output at the time t is o_k.

$$\hat{b}_i(o_k) = \frac{\sum_{t=1, O_t=o_k}^{T} \gamma_t(i)}{\sum_{t=1}^{T} \gamma_t(i)}, \quad i \in \overline{1,n}$$

The denominator is the expected number of times that the process is in state i, summed over all timesteps. [23]

By using those estimations, the Baum-Welch algorithm iteratively improves the transition and the emission probabilities until a stopping criteria is reached (according to a convergence threshold or a predefined number of iterations), in order to find a model that fits the given data.

3.2 Time Series Decomposition

3.2.1 Time Series Components

In general, a time series is made up of the following three components: the trend, the seasonality and the remainder (or residual) - as illustrated in Figure 3.5.

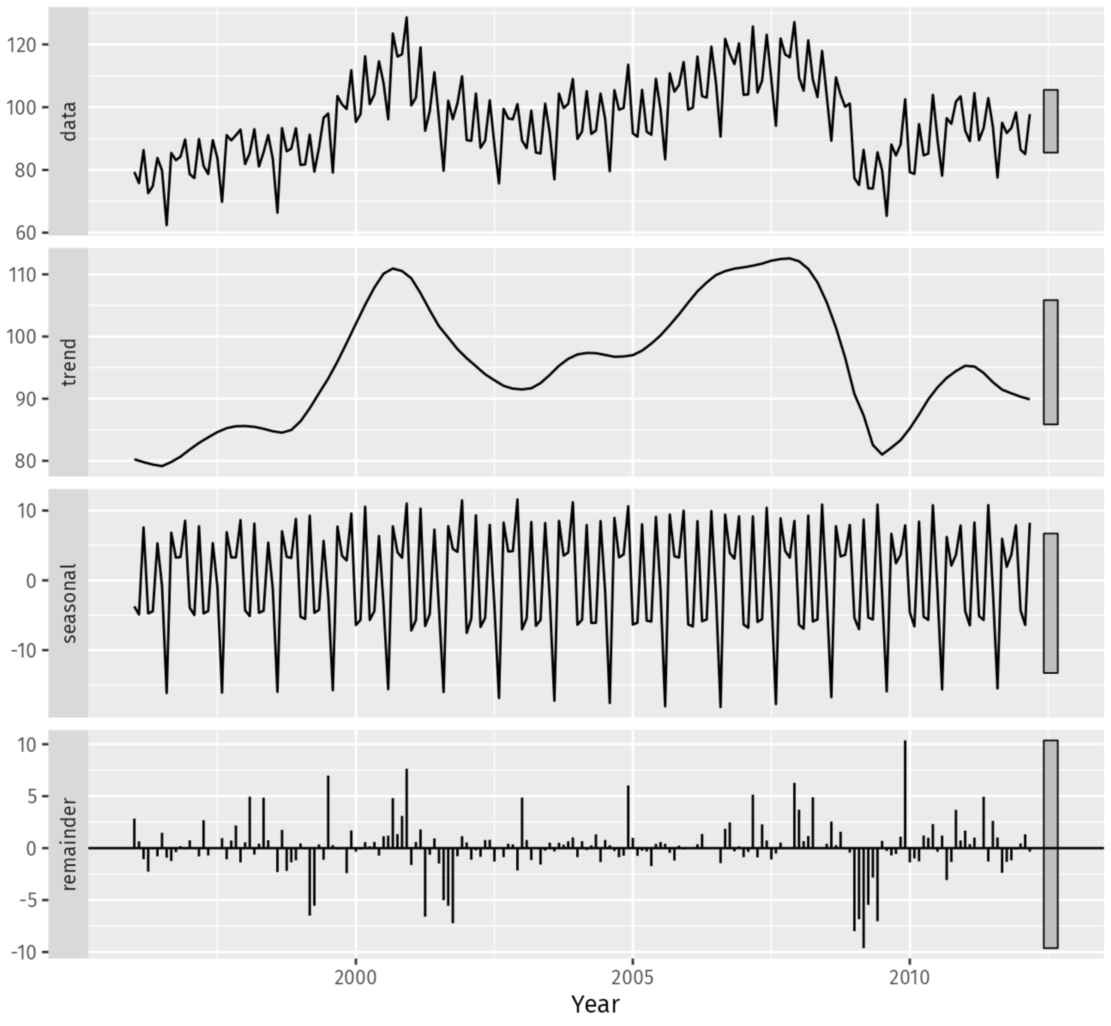

Figure 3.5: The components of a time series[25]

Furthermore, mathematically we can split a time series (y) at the t timestep as:

$$y_t = T_t + S_t + R_t$$

Where the initials (T,S,R) represent each component mentioned above. In short, adding up the three parts gives back the original data, if an additive composition is present. A multiplicative composition can also happen, however, in that case logarithmating the time series results in an additive one [25].

The trend catches the overall direction towards which the data is heading over in time, whereas the seasonality aspect represent regular patterns that repeat at ”seasonal” intervals - more exactly, a season is a repeating cycle within the data. The residual is essentially the remainder when the two components from above are subtracted from the data - it also has a meaning of capturing irregular influences not found within the trend, nor in the regular patterns [26].

3.2.2 Classical Decomposition

A classical method of obtaining the aforementioned components is to perform time series decomposition using the Moving Averages procedure. The moving average of order n, at time t, for the time series y, can be computed by averaging the previous and the upcoming $\lfloor \frac{n}{2} \rfloor$ values. Namely:

$$\mathrm{MA}_t(y,n) = \frac{1}{n} \sum_{i=-\lfloor \frac{n}{2} \rfloor}^{\lfloor \frac{n}{2} \rfloor} y_{t+i}$$

Furthermore, the process of Classical Decomposition (specifically for the additive variant) goes as described in [27]. By summarising, the following steps are to be found:

- First, the trend component is computed for every single timestep as:

$$\hat{T}_t = \mathrm{MA}_t(n)$$

In case of missing values, a particular padding (e.g. with zero values) or interpolation can be performed.

- Afterwards, the so called detrended series is obtained using the following formula:

$$d_t = y_t - \hat{T}_t$$

- The seasonal component is estimated next for each season. Assuming that its length is k (timesteps), then $\hat{S}_t$ is computed by averaging all k values within that season. All $\hat{S}_t$ belonging to a specific season are equal.

  Additionally, $\hat{S}_t$ is also adjusted so that summing each of them up equals 0 (for all possible seasons). This makes sure that the seasonal component does not include any trend fluctuations.

- Lastly, the residuals are found by subtracting the previous two components from the original data, namely by using:

$$\hat{R}_t = y_t - \hat{T}_t - \hat{S}_t$$

As mentioned previously, in case the data presents a multiplicative composition (e.g. when the magnitude of the season or of the trend increases or decreases in time - for an additive composition the magnitude repeats seasonally in contrast) taking the logarithm of the time series can transform it into an additive one again - thus the aforementioned procedure can be applied again. This typically can be found in the economic field, where each year the market value can increase, making seasonal repetitions less likely to occur [25], [26].

3.3 LSTM-based Autoencoders

3.3.1 Recurrent Neural Networks

The temporal nature of a Hidden Markov Model is reflected through the fact that the information gleaned along the way is carried forward in time. In contrast, other machine learning algorithms, such as K-means or feedforward neural networks, don't inherently take into account this temporal information - as those algorithms assume simultaneous access to the whole data [28]. An extra layer of complexity however can deal with this limitation, for example by mapping the data to the frequency domain using the Fourier transform.

Deep Learning offers a modern alternative to represent time in its architecture, specifically with the introduction of Recurrent Neural Networks (RNNs). Such networks are characterized by the fact that they contain cycles within their connections - meaning that RNNs considers previous outputs as inputs in their computations.

In other words, RNNs are neural networks with hidden states. Each hidden state at a given time can be computed based on the current input and the previous hidden state, directly integrating the time dependencies in the model and also drawing a close parallel to the HMM algorithm [29].

To illustrate this concept, we can consider the so called Simple Recurrent Network (also known as the Elman Network), like shown in Figure 3.6.

The process goes as with a classical neural network, namely an input vector (x_t) is multiplied by a matrix (which holds the associated weights) and then dragged through an activation function (f) which gives the final result for the hidden layer. The output from this hidden layer is further used to calculate the network output (y_t). Above, the indices are used to represent the vector values at the t timestep.

However, in the aforementioned figure, a cycle is additionally present within the architecture. This incorporates, beside the input, also the previous values of the hidden layer's

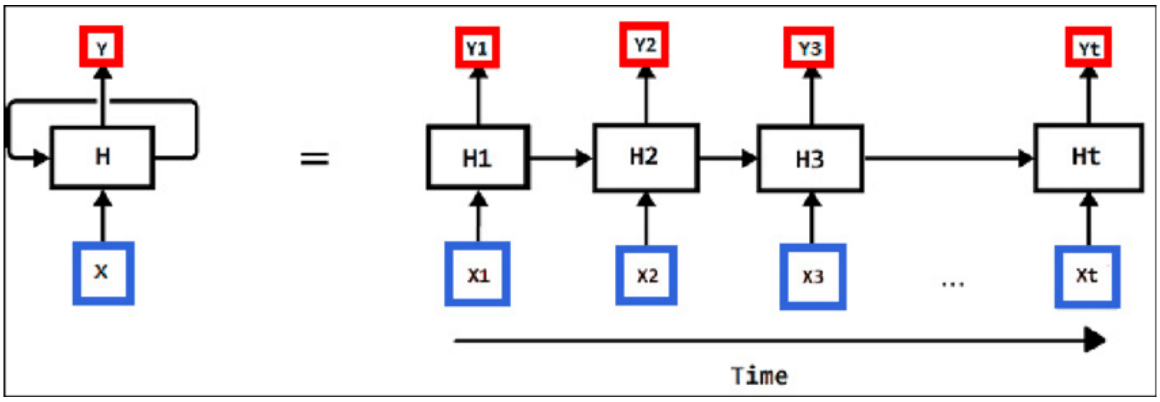

Figure 3.6: The Simple Recurrent Network being unrolled in time [30]

output (from the $t-1$ timestep) when computing the current value for the hidden state - which creates some sort of memory.

In practice, for the forward step, the only modification necessary is to include (or extend) the weight matrix for the recurrent values. Thus the hidden layer output would be:

$$h_t = f\left(x_t W_i + h_{t-1} W_h + b_h\right)$$

Where W_i is the weight matrix associated with the input layer, W_h is the new part which represents the weight matrix associated with the recurrent cycle and b_h is the bias vector[29].

In order to find the optimal weights, backpropagation is used like for the traditional neural networks, however this time the weights associated with the recurrent connection should be taken into consideration. Embedding this creates the concept of backpropagation through time (BPTT).

To achieve BPTT, we need to unroll all timesteps of the network (like it was shown in Figure 3.6), then apply backpropagation to the unrolled network and sum up all the computed gradients. From a high-level perspective this is done by saving the outputs of each layer at all timesteps and also saving the error terms in the hidden layers for all timesteps.

In practice, typically only a few number of successive timesteps are taken into consideration, while the rest are clipped. This is done since it is difficult to train RNNs due to problems such as the vanishing gradient - where repeated multiplication with subunitary values (which is essentially how the weights are updated) eventually leads to zero-values for the weights. Additionally, the layers associated with recurrent cycles are trying to make an accurate decision in the current timestep, while also trying to remember distant information - and performing two task at once adds too much complexity to the process [28].

3.3.2 Long Short-Term Memory Networks

To address the previous limitations, more advanced Recurrent Neural Networks are being employed, such as the Long Short-Term Memory (LSTM) network. Those networks resemble vanilla (traditional) RNNs with the difference that the recurrent cycle is replaced by a memory cell [29].

The name ”long short-term memory” comes from the original paper [31] and reflects how the networks can use their recurrent cycles to memorize representations of recent input events in the form of outputs (seen as the short-term memory) for a long-term period - embodied within the process of slowly changing weights.

As with vanilla RNNs, the data flowing into those memory cells (or LSTM cells), are the input at the current timestep alongside the value of the hidden state (the output of the memory cell) from the previous timestep. The process of how LSTMs work is being detailed in [29] and further summarized below.

LSTM cells are formed out of an internal state (which is the memory that the cell holds through processing of the data) and three gates that control the flow of the information into, within and out of the cells. This is illustrated in Figure 3.7, but more specifically there is:

- the input gate decides how much of an input should be stored in memory
- the forget gate ponders whether some part of the information should be flushed or not

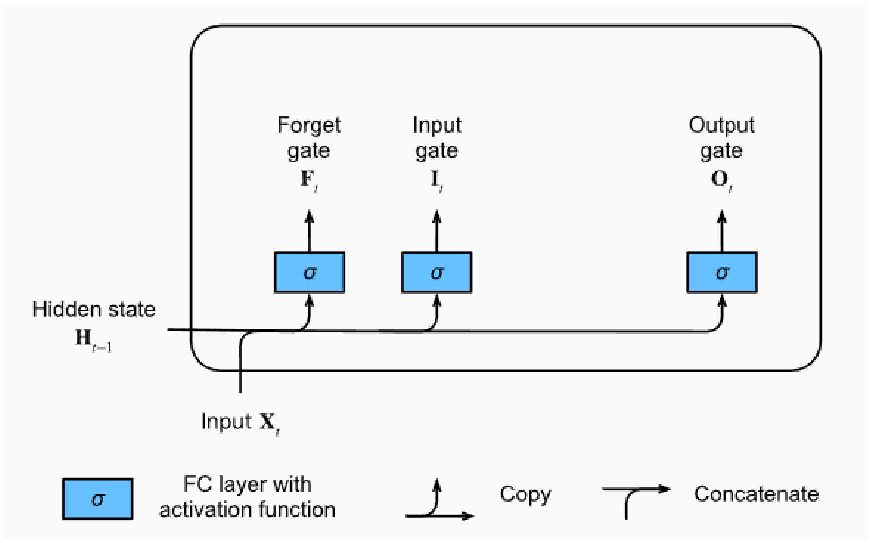

Figure 3.7: The architecture of a LSTM memory cell [29]

- the output gate decides how much the memory information should impact the cell's output.

Essentially, the three gates are some separate network layers with their outputs being dragged through an activation function - typically the sigmoid function is used (denoted as σ) as its output is in the $(0,1)$ interval and this is ideal for creating a gate (or rather, it's easier for each gate to take a decision this way).

Mathematically, the output of each gate is computed as:

$$g_t = \sigma\left(x_t W_x + h_{t-1} W_h\right)$$

The above formula is repeated three times (for each gate) and W_x are the weights associated with the input that is being fed in and W_h are the weights associated with the hidden state values found at the previous timesteps. Each initial letter i, f, o denotes their corresponding gate name. The biases were omitted for simplicity.

So far we have seen how the outputs of each gate were computed. Next, we will describe the rest of the flow within the LSTM memory cell - this is also shown in Figure 3.8.

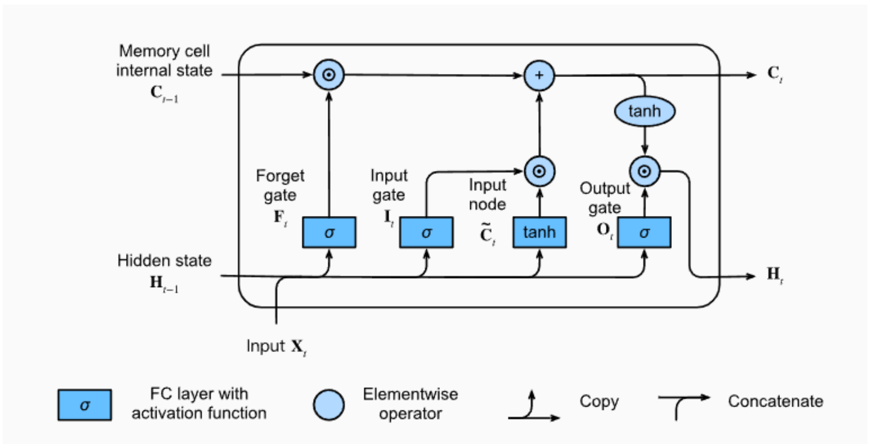
Memory cell internal state $\mathbf{C}_{t-1}$
$\mathbf{C}_t$
tanh
Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Input node $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
Hidden state $\mathbf{H}_{t-1}$
$\mathbf{H}_t$
Input $\mathbf{X}_t$
FC layer with activation function
Elementwise operator
Copy
Concatenate

Figure 3.8: The flow within the LSTM memory cell [29]

At first, the input is transformed into the candidate state, denoted as $\tilde{\mathbf{C}}_t$ (which is the value that the input gate further works with). This is calculated similarly as for the gates outputs, however a hyperbolic tangent function is used instead:

$$\tilde{\mathbf{C}}_t = \tanh\left(x_t W_{xc} + h_{t-1} W_{hc}\right)$$

Where W_{xc} are the weights associated with the input that is being fed in and W_{hc} are the weights associated with the previous hidden state. The main reason for using the hyperbolic tangent function instead of the sigmoid one, is that it is symmetric around 0, as its output is in the $(-1,1)$ interval - and centering the data around 0 can make the learning easier for the next layers.

Furthermore, the internal state is being updated. Essentially, the forget gate determines how much of the previous internal memory (C_{t-1}) should follow to the next timestep, whereas the input gate is used to clip the value of the candidate state ($\tilde{\mathbf{C}}_t$). Mathematically this can be written as:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{\mathbf{C}}_t$$

Above, $\odot$ represents the Hadamard product, namely:

$$(a, b, ..., z) \odot (A, B, ..., Z) = (aA, bB, ..., zZ)$$

Lastly, the hidden state (at the current timestep) is being computed - this is basically the output of the LSTM memory cell.

$$h_t = o_t \odot \tanh(C_t)$$

Above, the cell's internal state is being dragged through the tanh activation function and afterwards the output gate filters out which parts of the cell state should be let as the final output.

This summarises the forward part of a LSTM cell. In the backward phase, all the associated weights are being updated using BPTT - as described from a high-level perspective in the previous section.

A single LSTM cell might not achieve that much on its own, but interconnected with other similar cells it builds up the structure of the layers and further that of the entire recurrent neural network.

A thing that makes RNNs so special, is their ability to operate over sequences of vectors in multiple ways, as shown in Figure 3.9. This includes structures such as: one-to-one (with fixed inputs and outputs, as for image classification); one-to-many (with a sequence vector output, as when the input is an image and the output is a language description of it); many-to-one (the reverse of the one-to-many structure) and many-to-many (the most generalized variant, where both the input and the output are sequences of vectors) [32].

The flow within such a general structure goes shown in Figure 3.9. Specifically for a time series, the process start with the first timestep being the input for the first LSTM cell, then the second timestep alongside the output of the first LSTM cell becomes the input for the second LSTM cell. The process goes further like this until the very end, but taking into account the different types of outputs (mentioned previously).

Additionally, since those cells are encapsulated so that from an external perspective those will appear as basic units (similarly as with the concept of objects within object-oriented programming) it enables easily to experiment with different types of RNN archi-

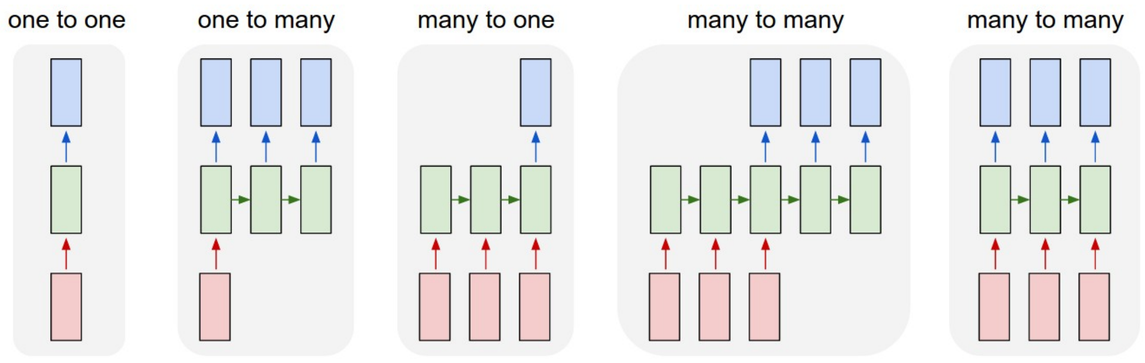

Figure 3.9: Different Structures of RNNs [32]

tectures [28].

One such possible different architecture is the Bidirectional LSTM (Bi-LSTM), which essentially consists of two LSTMs: one that processes the data in the natural direction, whereas the other one reverses the direction of the sequences that are being processed - with both the outputs from each network being concatenated at each timestep.

While this is not feasible for online learning (where new data continuously appears), it is a major advantage for offline training (meaning that we are in possession of all the data) as there is simultaneous access to both the future and the past data from the network's perspective.

3.3.3 The Autoencoder

An Autoencoder is a special type of a neural network that can be used to compress efficiently some input data. The architecture is composed out of an Encoder (which tries to obtain a lower dimensional representation of the input data) and a Decoder (which tries reconstructs the original data based on the encoded representation) [33].

As an illustrative example, if we consider a simple feedforward neural network with a single hidden layer and the size of the output layer is the same as for the input layer, then the network can be used as an Autoencoder by training the network to minimize the error

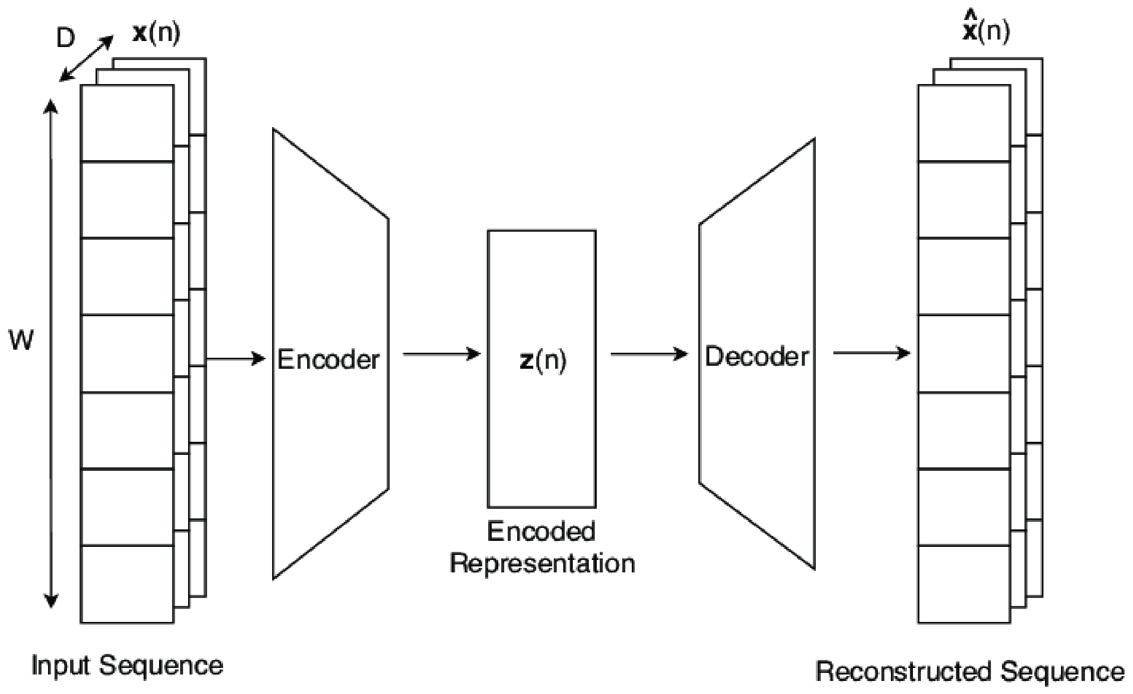

Figure 3.10: The Architecture of a LSTM Autoencoder [34]

between the output of the network and the original input itself - making an encoder-decoder process.

More exactly, the output of the hidden layer is the resulting encoded representation and in case the hidden layer contains much fewer neurons than the input or the output layer then this representation can be also viewed as the compressed data - assuming that the programming language represents the resulting encoded representation using the same datatype as for the original input, which often might not be the case.

A LSTM Autoencoder is similar to the feedforward network exemplified above, however instead of the classical layers the architecture is built out of layers containing LSTM cells - as described in the previous section [34]. Figure 3.10 illustrates the structure belonging to such an Autoencoder.

Chapter 4

Implementation

This chapter outlines the practical implementation of the proposed software, describing how the theoretical aspects were adapted specifically for the PET bottle production. The most relevant parts of the code that implements the algorithms are also discussed.

4.1 Project Setup

This project was developed entirely in Python, using state-of-the-art frameworks and libraries adapted accordingly. Furthermore, the software was also tested using Python 3.9.13 inside a Jupiter Notebook (in Visual Studio Code) on a Windows 10 operating system.

The reason why Python was chosen, even though other languages (such as Julia) were shown to be much faster [35], is due to its multitude libraries which doesn't constrain us to use some specific algorithms, thus shifting the focus away from writing manual code to the design and implementation of the core concepts. Additionally, compared to other programming languages Python provides a much more intuitive and easy to use graphical interface.

4.2 Data Preprocessing

After the initial setup, the next step within this project was to preprocess the time series data, which was already gathered in CSV files from a PET bottle machine - more exactly, the data consisted of two time series related to energy consumption and environment emissions. The exact type of the machine and its operating modes is however unknown.

Specifically, the energy data contained: the Active Power, the Current, the Power Factor, the Reactive Power, the THDI (total harmonic intensity distortion), the THDU (total harmonic voltage distortion) and the Voltage - three of them for each phase (A, B and C) as the machine was using a three-phase electric power system.

Meanwhile, the environmental data consisted of: the level of CO2 in the air, the humidity, the particulate matter (pm) measurements - three of them with the diameters of 1, 2.5 and 10 micrometers, the atmospheric pressure, the temperature (surrounding the machine) and a measure of the volatile organic compounds (voc).

For preprocessing, *pandas* - a Python popular library, was utilized to load and merge the time series into *Pandas* dataframes. Given that the initial data had a frequency of 1 second, those time series were resampled by averaging them over 5 seconds. This was mainly performed in order to ignore the unnecessary high-frequency fluctuations and thus focusing on longer patterns. After this step, any missing data was filled with the nearest non-null value within the dataframe.

Lastly, as the energy data came from a three-phase electric power system, each of the three phases were averaged in order to obtain a single representative value for each feature. Thus, preserving the essential information while in the same time reducing the high-dimensionality of the data.

Figure 4.1 shows over a short period some relevant features obtained after the preprocessing step. The data was mapped to the $[0, 1]$ interval for a better view.

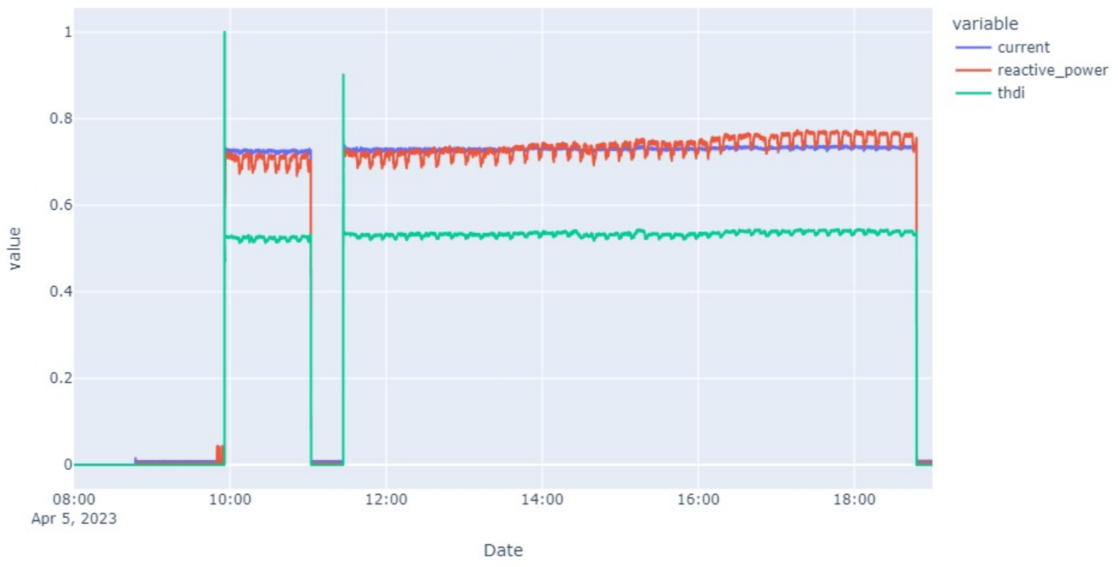

Figure 4.1: A small part of the preprocessed data

4.3 Implementing the Time Series Clustering

4.3.1 Scaling the Data

Before applying the Hidden Markov Model (HMM) to cluster the timeseries, it was necessary to ensure that all features contribute equally to the model. This was achieved by scaling the data using the *StandardScaler* function found within the *scikit-learn* library. More precisely, standardizing the data maps the values so that the new mean is 0 and the new standard deviation is 1.

The relevant code performing this operation is shown below in a general form that can be adapted for other time series as well - with the mention that in this case *data* contains the already preprocessed time series.

```
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data[data.columns])
```

This step is also essential to apply before training the HMM algorithm, since the emissions are assumed to belong to a Gaussian distribution and having the values standardized can make the model more stable. To be exact, the *hmmlearn* library is used and the most appropriate model available for the PET data was *GaussianHMM* - which works with Gaussian emissions, as opposed to *CategoricalHMM* - which assumes that the emissions are discrete.

4.3.2 Determining the optimal number of clusters

The next step after standardizing the data was to determine the optimal number of hidden states, or in other words to find the number of clusters that the HMM algorithm should work with - a parameter that must be predefined.

This can be found visually by looking at how HMM performs when changing the number of hidden states. However, in order to enable an automatic process we desired to find the optimal number dynamically - without harcoding the value.

Some variants for determining the optimal number of states automatically were already presented in Section 2.3 - within the literature review of the Time Series Clustering.

Specifically, a common criterion that deals with this issue is the Bayesian Information Criterion (BIC), which is defined as:

$$BIC = k\ln(n) - 2\ln\left(\hat{L}\right)$$

Above, k is the number of parameters in the model, n is the data size (or the number of timesteps) and $\hat{L}$ is the maximized value of the likelihood of the model given the observed data (in simpler terms this likelihood is a measure of how accurately the model represents the data) [15].

However, for the data originating from the PET bottle machines, BIC had the unfortunate drawback of not penalizing the model enough - as it led to the situation where increasing the number of hidden states resulted in a decrease for the BIC value (and a smaller value of BIC means that the model performs better). In short, BIC always recommended to choose the largest possible number of states, but visually we could see that this should not be the case.

Alternatives variants to BIC, such as applying the Akaike Information Criterion (AIC) or performing cross-validation didn't resolve this issue either (for the current dataset), as those also continued to suggest the maximum possible number of hidden states.

In order to address this limitation, a modified version of BIC was further proposed in this study. More precisely, an Adjusted Bayesian Information Criterion (ABIC) was considered - which penalizes harder models with a higher number of parameters. This criterion is further defined as:

$$ABIC = \ln\left(\hat{L}\right) - k^2 \ln(n)$$

The main difference appears in the fact that the number of parameters (k) is squared, in order to give a harder penalization. The minus sign was also reversed so that a higher valuer for *ABIC* now suggests that the model performs better (which felt more natural, as usually metrics with higher values tend to represent a better performance).

The relevant code which computes the ABIC coefficient is shown below in an generalized manner, thus being applicable for other time series as well - in this case, *data* represents the already scaled data and shape[0] represents its size, whereas shape[1] holds the number of features. The parameter *model* is assumed to be an already trained model on the given data.

```
def compute_abic(model, data):
    n_features = data.shape[1]
    n_states = model.n_components

    n_transition_params = n_states * (n_states - 1)
    n_emission_params = n_states * n_features
    n_initial_state_params = n_states - 1
    n_params = n_transition_params + n_emission_params +
                                        n_initial_state_params

    adjusted_bic = model.score(data) - np.square(n_params) * np.log(data
                                        .shape[0])
    return adjusted_bic
```

4.3.3 Introducing the Hierarchical Hidden Markov Model

When further inspecting the time series in order to apply the Hidden Markov Model, we observed that the data fluctuates over time - and those fluctuations can unfortunately mask the working modes of the PET bottle machine, making it challenging to differentiate between them.

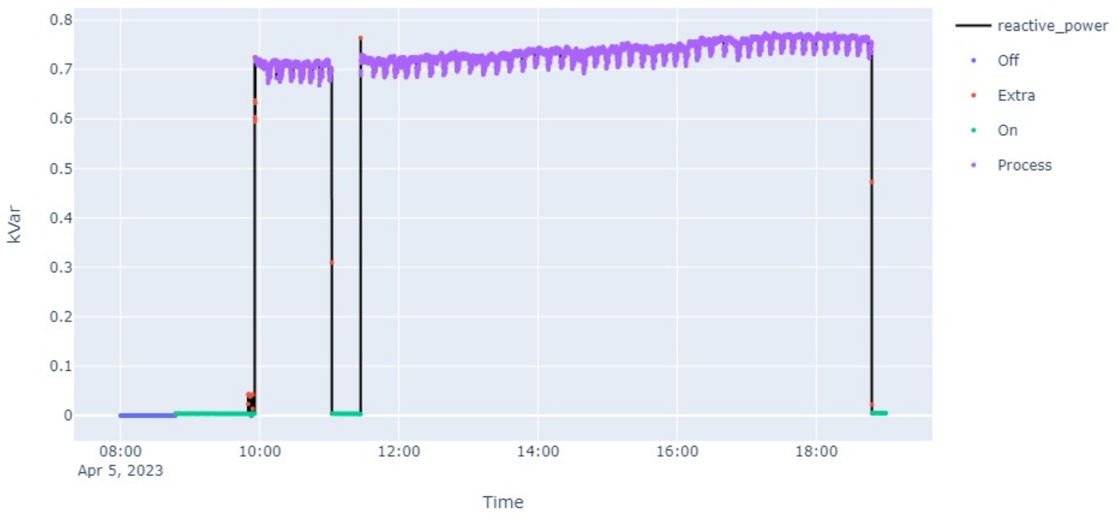

Figure 4.2: Fluctuations occurring within the data

More exactly, the fluctuations appear within features such as the *reactive power* and gradually changes the overall trend through the day. Even though those features also change frequently (which gives valuable information in order to distinguish between the working modes), for some operating modes this information is masked by the fluctuating trend.

This drawback can be observed from a close up perspective in Figure 4.2 over a small portion of time. The reactive power was exemplified in this case as it was the most relevant feature in further splitting the purple sequence (which represents one cluster).

In order to counter this issue, a three-step Hierarchical Hidden Markov Model (HHMM) is further proposed. This process starts by using a HMM to cluster the data as it is (affected by the fluctuations), then an application of Time Series Decomposition is used in order to get rid of the fluctuations (found in some clusters of the data) and finally another instance of the HMM is employed in order to perform a further splitting on the resulting data.

4.3.4 The first clustering part of the Hidden Markov Model

Going further with the proposed three-step HHMM, the first phase of training the Hidden Markov Model is implemented as shown in the code from below.

```
from hmmlearn import hmm


model = hmm.GaussianHMM(n_components=np.argmax(model_scores)-1)
model.fit(scaled_data)
first_clusters = model.predict(scaled_data)
```

Above, argmax(model_scores) is basically the optimal number of hidden states that ABIC suggests. However, this coefficient doesn't overcome the fluctuation issue, as for Figure 4.2 it actually recommends to use 5 clusters (which is also visually the right choice in the end). However, due to the unwanted trend changes, with 5 clusters the HMM result would be as seen in Figure 4.3 - which is clearly not the desired one.

Taking these into account, for the first phase of the HHMM we assumed the number of

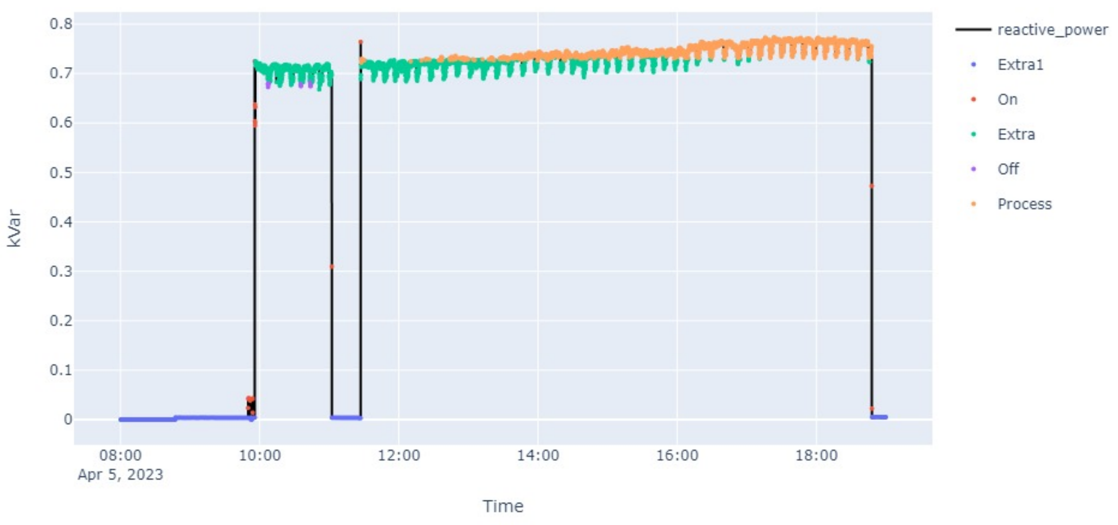

Figure 4.3: HMM clustering on data affected by fluctuations

hidden states to be the recommended ABIC minus one. Surely, after the fluctuations are removed, we can further deal with the ”purple” states. It should also be possible to flatten first the unwanted trend changes, merge back the whole series and then apply a single instance of the HMM algorithm.

Furthermore, Figure 4.4 shows the same data after the first HMM application, however with the *current* feature being exemplified as it allows to differentiate between the working modes easier. More exactly:

- blue: represents the state when the machine is turned off
- green: represents the state when the machine is turned on, but it doesn’t process anything
- orange: represents possible intermediate changes, alarming when the machine starts or stops operating

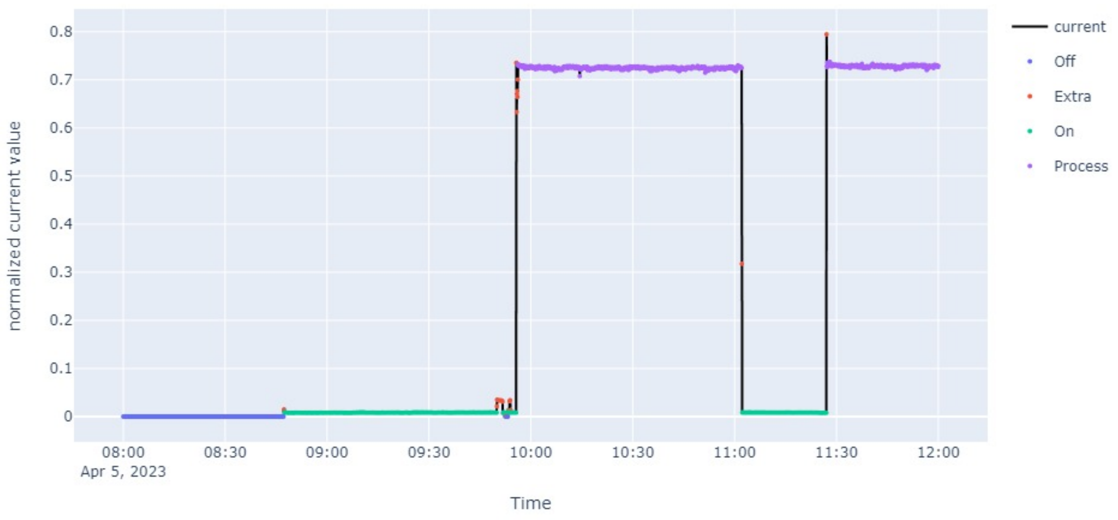

Figure 4.4: The result after the first clustering step

- purple: represents the state when the machine is processing something (perhaps heating, cooling or other PET specific operations)

4.3.5 Removing Data Fluctuations

As emphasized previously, the ABIC coefficient doesn't resolve the fluctuation issue. So in order to address this, we will employ the Time Series Decomposition algorithm in order to flatten the unwanted trend changes.

Previously, from Figure 4.4 we have observed that the cluster represented by a high-valued *current* contains the state where some processing is done (this is basically the purple sequence). Thus, furthermore we can use this information to select the specific cluster which has the highest average value for the *current* (in order to perform a further decomposition and processing on it).

The relevant code that performs the aforementioned operations, namely to find the cluster which we would further like to process, alongside the Time Series Decomposition (TSD) can be seen below. For TSD an additive model was used and the frequency was chosen to be 120 periods (visually it was observed that one mode lasts around 10 minutes - and this matches with 120 periods, as the data was already resampled from 1 to 5 seconds).

```
#finding the cluster which has the highest "current" average
process_data = scaled_data.assign(clusters = first_clusters)
clusters_average = process_data.groupby('clusters')['current'].
                                                        mean()
highest_average_cluster = clusters_average.idxmax()

process_data = process_data[process_data['clusters'] ==
                highest_average_cluster][["reactive_power"]]

#applying Time Series Decomposition
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition=seasonal_decompose(process_data['reactive_power'], model='
                                    additive', period=120)
residual = decomposition.resid
p_data['rp_decomposed'] = residual #rp = reactive power
```

After applying TST, the residual was kept, as this is the relevant component which does not include trend or seasonal changes - as described in Section 3.2.

The ”before and after” of those steps are illustrated in Figure 4.5 - where it can be seen that rp_decomposed doesn't fluctuate, as opposed to reactive_power. Within this figure, the straight lines are due to the fact that only the data corresponding to the highest average *current* value was kept. Additionally, the difference seen for *value* appears since the residuals were centered around 0 (admittedly, it also enabled a better illustration so this wasn't modified).

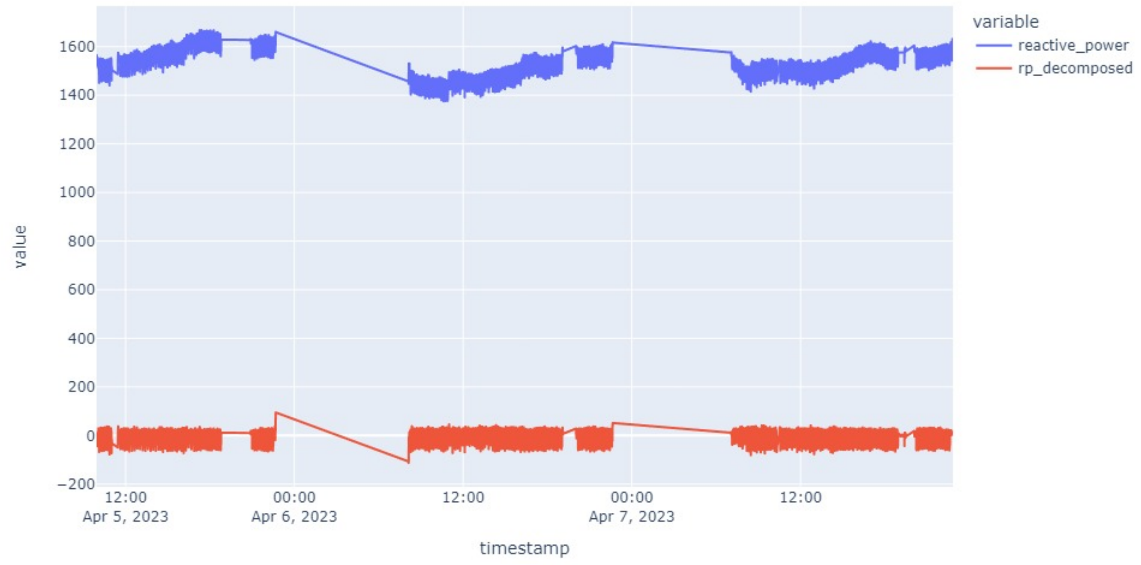

Figure 4.5: Before and after applying Time Series Decomposition

4.3.6 The final clustering part of the Hidden Markov Model

After the *reactive power* feature is decomposed, we can apply the final phase of our HHMM - taking advantage that the *reactive power* feature distinguishes two more modes easily (for the ”purple” sequence). Since the residual data is centered around 0, it is also necessary to standardize the data again. Implemented in code, all those two steps are written similarly as seen previously in this chapter with their respective counterpart.

Figure 4.6 shows the result after applying the final clustering part of the HMM. Mainly, it can be seen that in contrast to Figure 4.4, the previous purple state is now split into two more operating modes (which might represent that in one mode the PET machine is heating the PET resin, while in the other one the machine cools down the heated preforms).

For a better view of this aspect we can also take a look at Figure 4.7 - where the same portion of data is used, but the clusters are more clearly separated.

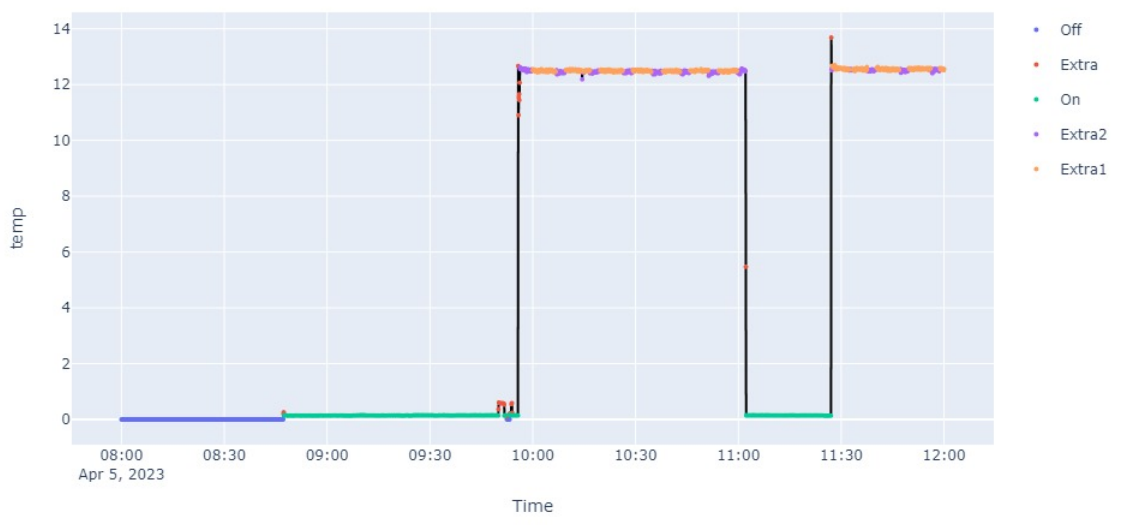

Figure 4.6: The result after the final clustering step

In addition to the previous detected operating modes (black: machine turned off, white: machine turned on and not processing, cyan: changes from the idle state to any working mode), there are now two more modes present, which we might assume that they are: yellow - which could be the one related to heating; and magenta - which could correspond to cooling.

This concludes the Time Series Clustering Part. To summarise, we first scaled the preprocessed data and then determined the optimal number of clusters by using the ABIC coefficient. Afterwards, a first instance of HMM was used to split the main states - and the state that corresponds to the part where the machine is doing some actual work is further split using another application of the HMM algorithm. However, prior to the last clustering step a Time Series Decomposition that flattens the unwanted trend fluctuations is employed.

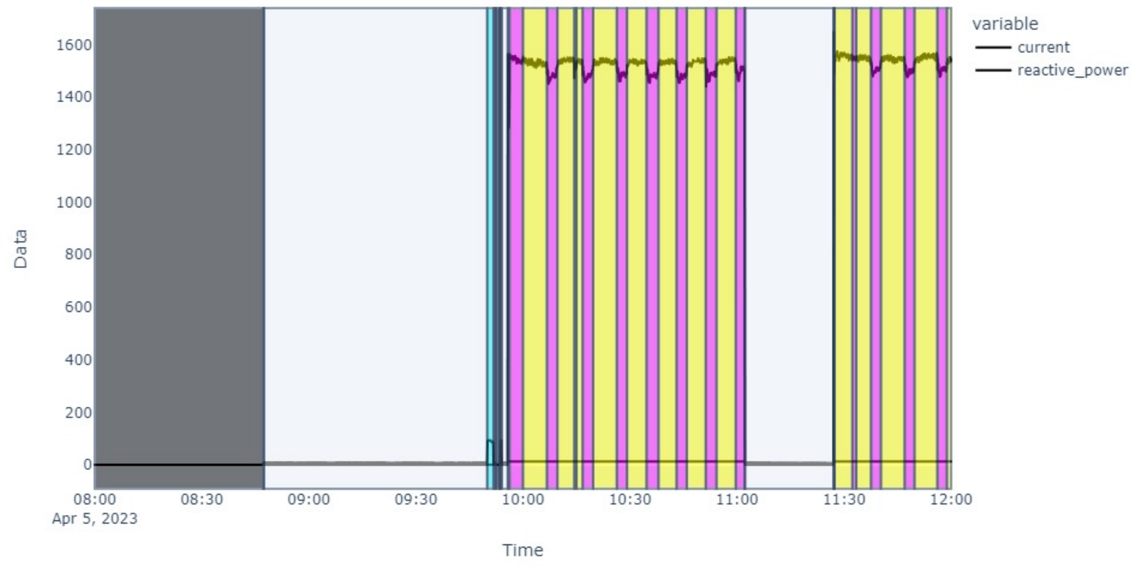

Figure 4.7: A better view of the working modes

4.4 Implementing the Anomaly Detection

4.4.1 Anomaly Detection using Hidden Markov Models

The idea of using Hidden Markov Models to detect anomalies was mentioned in Section 2.4. Essentially, the process starts by computing the likelihood of the time series - given the already trained model - and then those data points which correspond to a very low likelihood (meaning that those data points most likely don't belong to any state) are flagged.

This is done practically as seen in the code from below. First, we exponentiate the result of the compute_log_likelihood function - in order to obtain the original value from the logarithm - and then we select all data points for which their corresponding likelihood falls below the 0.1-th percentile of the entire data.

```
log_probability = model._compute_log_likelihood(data)
likelihoods = np.sum(np.exp(log_probability), axis=1)
```

```
threshold = np.percentile(likelihoods, 0.1)
anomalies = np.where(likelihoods < threshold)[0]
```

While this is a fast and simple technique of detecting anomalies, it also has a major drawback since it can only be applied to detect anomalies from the features that were used in the training part of the model. More exactly, if we trained the HMM using some environment parameters, then we can only find deviations within those features (and not from the emissions data, as an example).

4.4.2 Anomaly Detection using the LSTM Autoencoder

To address the limitation of the previous method, we can use a more efficient model, namely an Autoencoder - which was previously described in Chapter 3. More specifically, a Bidirectional Long Short-Term Memory (LSTM) Network is further used to build this network. In contrast to the HMM method, this architecture can also detect anomalies within the emissions time series.

The implementation of this model starts by splitting the data into groups (one for each cluster) and then one Autoencoder is trained for each group. The exact architecture of the network (which is partially inspired from [36]) can be seen in the followings.

```
def create_autoencoder(timesteps, n_features):
    autoencoder = Sequential([
        Bidirectional(
            LSTM(64, activation='tanh', return_sequences=True),
            input_shape=(timesteps, n_features)
        ),
        Bidirectional(
            LSTM(32, activation='tanh',
            return_sequences=False)),
        RepeatVector(timesteps),
        Bidirectional(
            LSTM(32, activation='tanh',
            return_sequences=True)),
        Bidirectional(
            LSTM(64, activation='tanh',
            return_sequences=True)),
        TimeDistributed(Dense(n_features))
    ])
    autoencoder.compile(
        optimizer=Adam(learning_rate=0.001), loss='mse')
    return autoencoder
```

This Autoencoder is mainly composed of an Encoder and a Decoder. The Encoder consists of two bidirectional LSTM layers, for which each LSTM cell has as output a vector of size 64 (and 32 respectively). The first layer outputs such a vector for each timestep (since return_sequences is set to True), while the second layer outputs a single final vector. Therefore, in order to match the dimension needed at the Decoder, a RepeatVector layer is further introduced in order to copy that final vector for each timestep.

On the other hand, the Decoder mirrors the architecture outlined above for the Encoder. Afterwards, a TimeDistributed layer is used to restore the original number of features for each timestep. Lastly, the parameters for training are set (the *MSE* loss function alongside the *Adam* optimizer).

As mentioned previously, an Autoencoder is trained for each working mode and further the model is used to predict the original data. Moreover, the difference between the original and the predicted data enables to detect anomalies - as a significant differences can represent outliers found within the data. This procedure is illustrated for one working mode in the code from below.

```
autoencoder = create_autoencoder(timesteps, n_features)
autoencoder.fit(
    cluster_data, cluster_data,
    epochs=20, batch_size=32, verbose=1)
predictions = autoencoder.predict(cluster_data)
mse = np.square(np.subtract(cluster_data, predictions))
threshold = np.percentile(mse, 99)
anomalies = np.where(mse > threshold)
```

In the case from above, those data points for which the MSE is above the 99-th percentile are considered as anomalies.

Further results related to anomalies are presented in the following chapter and this concludes the part regarding the anomaly detection implementation.

Chapter 5

Experimental Results

The following results are primarily of a visual nature, as given the versatility of the PET bottle production alongside the limitation of benchmarks, not many evaluation metrics were suitable for our implemented algorithms.

However, those visual results were supported by experts who were directly involved in obtaining the actual data from the PET bottle machines - thus, they naturally had more insight regarding the operating modes within this specific industry and also more awareness about potential anomalies happening in production.

The first metrics applied were the scores used to automatically determine the number of hidden states, namely BIC and ABIC - which were described previously in Section 4.3.2.

Figure 5.1 shows how BIC doesn't stabilize and tends to suggest the model with the highest number of states possible - a **lower** value of BIC means that the model should perform better with that number of clusters.

In contrast, the ABIC coefficient penalizes the model harder, stabilizes faster and doesn't suggest to choose the highest number possible of hidden states, as seen in Figure 5.2 - a **higher** value of ABIC means that the model should perform better with that number of clusters.

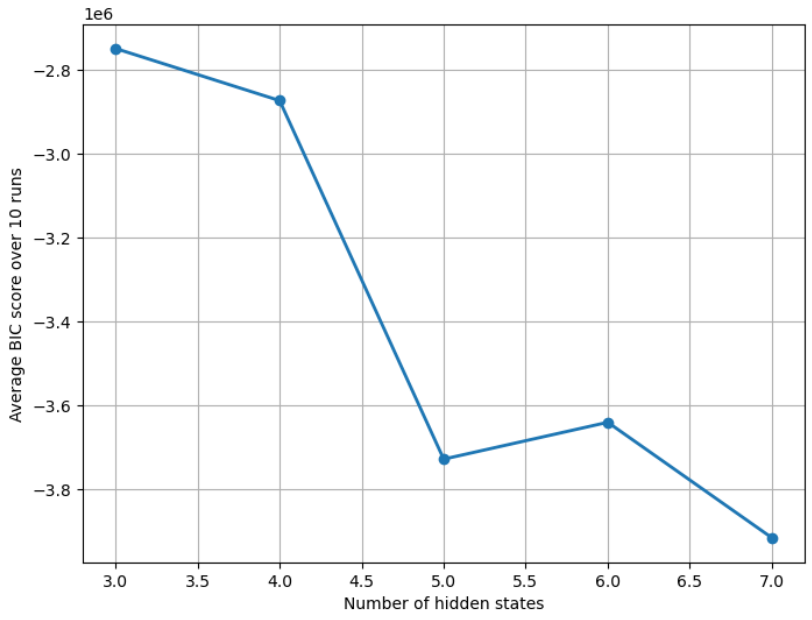

Figure 5.1: BIC scores for different number of clusters

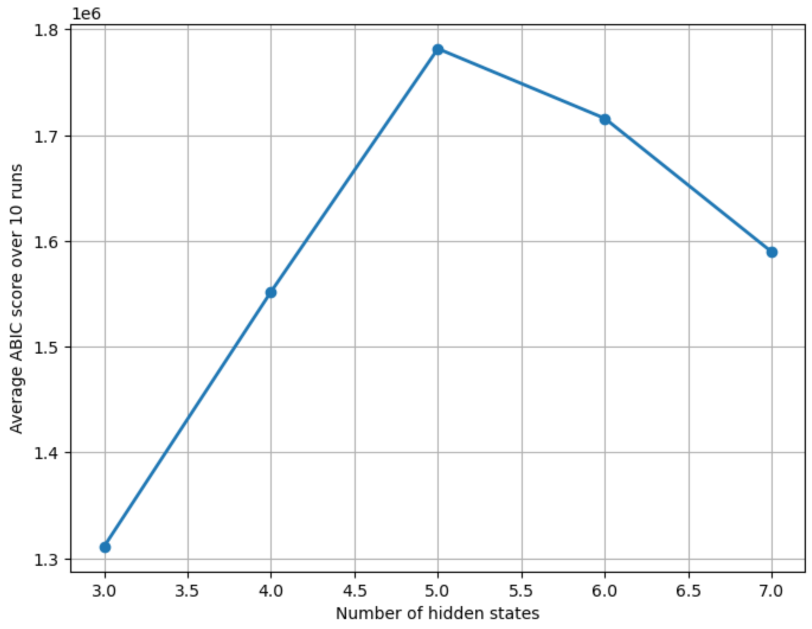

Figure 5.2: ABIC scores for different number of clusters

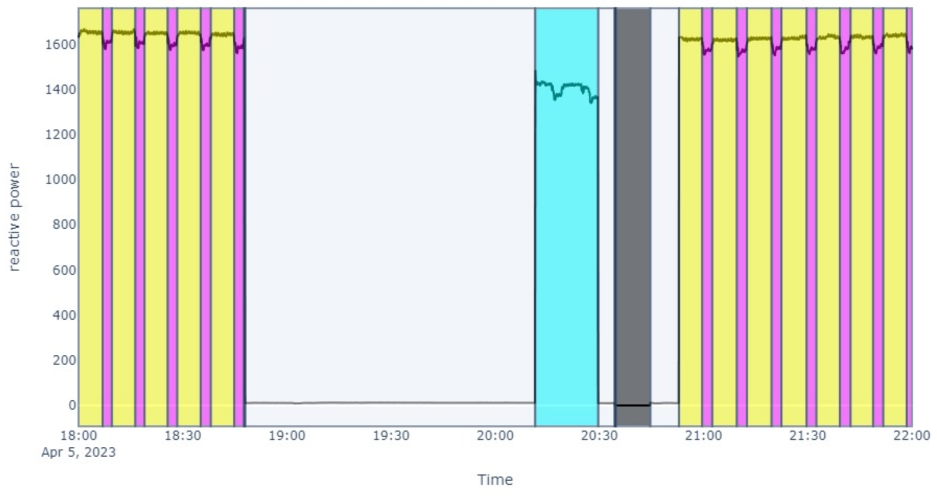

Figure 5.3: Some clusters obtained after applying HMM

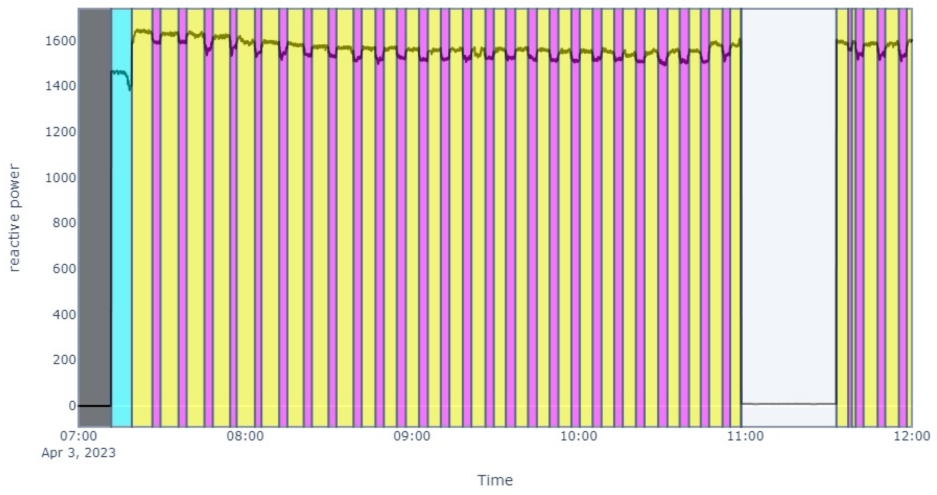

Figure 5.4: Other clusters obtained after applying HMM

Since the nature of the problem is time series clustering, visual results tend to give better insights compared to the metrics from above. Figures 5.3 and 5.4 displays some cases where the operating modes are clearly distinguished.

Most likely, black represents the case when the machine is turned off, white represents the idle case when the machine is turned on and not doing anything, cyan represents changes from the previous idle state to any working mode and yellow plus magenta represents the modes where the machine is alternatively heating or cooling. However, the important aspect is that those clusters enables an easier task for the anomaly detection process.

In order to provide an efficient LSTM-based Autoencoder, it was essential build an efficient architecture which learns as best as possible to reconstruct the given time series. Thus, the *MSE* loss function was further employed in order to measure the model's performance - as illustrated in the following table for the most relevant cases:

Architecture	Activation Function	Output Sizes	MSE
Bi-LSTM	tanh	64x32x32x64	**0.05984**
Bi-LSTM	relu	64x32x32x64	0.10334
LSTM	tanh	64x32x32x64	0.19288
Bi-LSTM	tanh	32x32x32x32	0.08896
Bi-LSTM	tanh	64x32x16x16x32x64	0.07904

Above different architectures, activation functions and output sizes (this description can be seen in Section 4.4.2) were tested. With MSE being the loss function and relu alongside tanh the possible activation functions. As it was highlighted, the best performance was obtained by a Bidirectional LSTM-based Autoencoder that utilizes a hyperbolic tangent activation across 4 layers (with their output sizes at every timesteps being 64, 32, 32 and 64).

The Figures 5.5, 5.6 and 5.7 illustrates some relevant cases where the aforementioned model detected some serious deviations within the data. Those anomalies correspond to single data points (highlighted with a red rectangle).

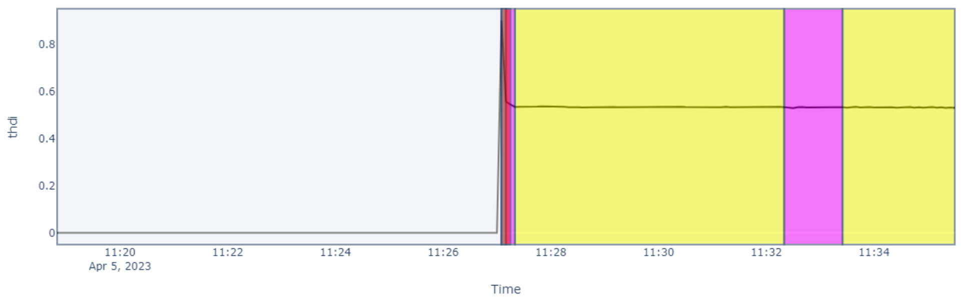

Figure 5.5: An anomaly within the THDI feature

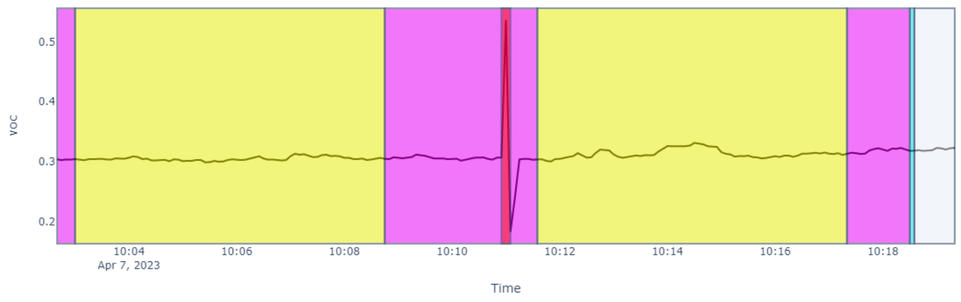

Figure 5.6: An anomaly within the *voc* feature

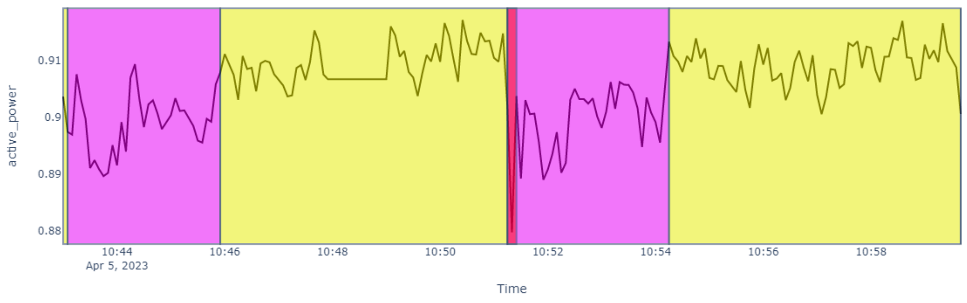

Figure 5.7: An anomaly within the Active Power feature

In addition to the Autoencoder, HMM was also used to detect anomalies - as described in Section 4.4.1. Admittedly, this had the disadvantage of being restricted to flag anomalies only within the energy time series - as those were the features which could detect the working modes. The results of HMM in this context are exemplified in Figure 5.8 and Figure 5.9.

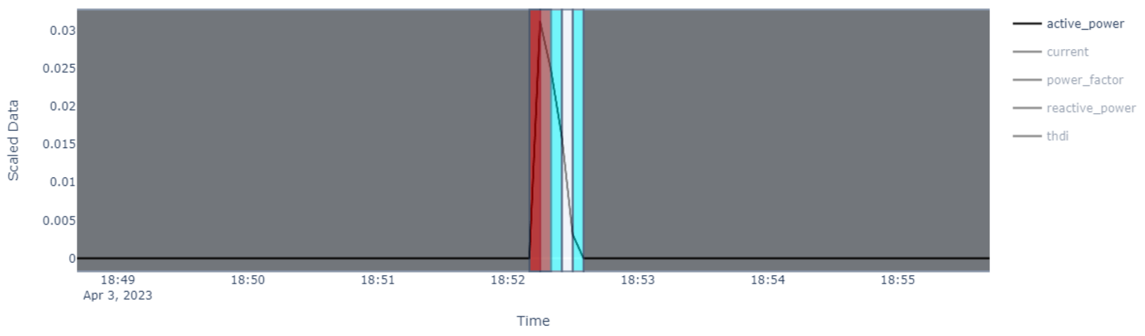

Figure 5.8: An anomaly within the Active Power feature detected by the HMM

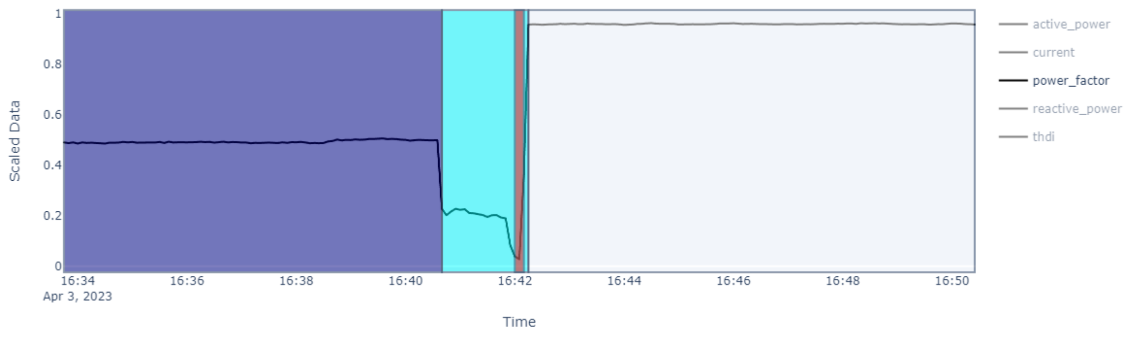

Figure 5.9: An anomaly within the Power Factor feature detected by the HMM

Chapter 6

Conclusions and Future Work

Throughout this study, a Machine Learning-based software, consisting of a pipeline with two components, was applied for the PET bottle manufacturing data. The first component made the best out of the Hidden Markov Models (HMMs) in order to perform Time Series Clustering, whereas the second component utilized both HMMs and LSTM-based Autoencoders in order to flag possible anomalies.

More exactly, to cluster the time series related to the energy consumption, a series of steps were taken. First, the optimal number of clusters was determined automatically using a novel coefficient that evaluated the model's performance. Namely, an Adjusted Bayesian Information Criterion (ABIC) was proposed, which penalizes the model harder in order to stabilize and refrain from suggesting the highest possible number of hidden states.

Afterwards, a three-phase Hierarchical Hidden Markov Model was employed. This embedded two instances of HMM with Time Series Decomposition (TSD). More exactly, the TSD algorithm was used to flatten unwanted data fluctuations, as those complicated things when it came to differentiate between the working modes of the PET bottle machine.

The anomaly detection step was performed using two different techniques: one that took advantage of the Hidden Markov Model and one that employed a modern Deep Learning architecture, namely an LSTM-based Autoencoder.

In the Hidden Markov Model approach, those data points for which the model couldn't assign a decent likelihood of belonging to any of the hidden states, were flagged as anomalies.

In contrast, the Autoencoder naturally aimed to detect anomalies through its architecture - as it learned to reconstruct the time series in an encoder-decoder manner. Thus, the anomalies were identified as those data points which, when reconstructed, presented major differences compared to the original ones.

The results were judged in a large portion through visual approaches, but additionally some metrics were applied in order to validate each model's performance.

Finally, those steps addressed the need for developing a sustainable PET bottle production, as it facilitates the analysis of the data obtained from the afferent machines. In addition, those algorithms can provide a starting point for the application of other techniques, which aim to improve the environmental impact in this industry - or in other related fields as well.

One possible extension for the existing software could be to develop a prediction mechanism in order to preemptively alarm future anomalies. Algorithms such as the ARIMA model, or even those that were already applied in this study (HMMs and Recurrent Neural Networks) can be taken into consideration.

Additionally, it would be a slight improvement to apply those techniques on some labeled benchmarks (in a classification manner), as it would allow to measure more precisely the performance of this project.

List of Figures

References

[1] Berkeley Plastics Task Force. Pollution and hazards from manufacturing. https://ecologycenter.org/plastics/ptf/report3. Online. Last time accessed on June 5, 2023.

[2] Dynaplast. Pet bottle manufacturing process. http://dynaplast.com.my/our-services/. Online. Last time accessed on June 6, 2023.

[3] Syrigos Antonios and Emmanuel Adamides. Environmental and economic assessment of the pet bottles manufacturing process: A case study. *Sustainable Design and Manufacturing*, 2022.

[4] VARIOFORM PET. From granulate to preform and bottle. http://www.varioform.at/en/produktion. Online. Last time accessed on June 5, 2023.

[5] KB DELTA. Single-stage vs. two-stage blow molding machines: A guide. https://kbdelta.com/blog/single-vs-two-stage-blow-molding-machines.html. Online. Last time accessed on June 5, 2023.

[6] Manufacturing of plastic bottles (pet). http://newengineeringpractice.blogspot.com/2011/08/

manufacturing-of-plastic-bottles.html. Online. Last time accessed on June 5, 2023.

[7] Forbes. Major innovation brings a new level of sustainability to pet bottle recycling. https://www.forbes.com/sites/businessreporter/2020/11/02/major-innovation-brings-a-new-level-of-sustainability-to-pet-bottle-recycling. Online. Last time accessed on June 6, 2023.

[8] D. A. Wahab, A. Hussain, E. Scavino, M. M. Mustafa, and H. Basri. Development of a prototype automated sorting system for plastic recycling. *American Journal of Applied Sciences*, 2006.

[9] Hellen Wanini Mwangi and Mpai Mokoena. Using deep learning to detect polyethylene terephthalate (pet) bottle status for recycling. *Global Journal of Computer Science and Technology)*, 2019.

[10] Seoyeong Lee, Yongwan Yun, Seonggwan Park, Sujeong Oh, Chaegyu Lee, and Jongpil Jeong. Two phases anomaly detection based on clustering and visualization for plastic injection molding data. *Elsevier*, 2022.

[11] T. Warren Liao. Clustering of time series data — a survey. *Pattern Recognition*, 2005.

[12] Saeed Aghabozorgi, Ali Seyed Shirkhorshidi, and Teh Ying Wah. Time-series clustering – a decade review. *Pattern Recognition*, 2005.

[13] Jeremy Reed and Chin-Hui Lee. A study on music genre classification based on universal acoustic models. *ISMIR*, 2006.

[14] Padhraic Smyth. Clustering sequences with hidden markov models. *Advances in Neural Information Processing Systems 9*, 1996.

[15] Jennifer Pohle, Roland Langrock, Floris M. van Beest, and Niels Martin Schmidt. Selecting the number of states in hidden markov models - pitfalls, practical challenges and pragmatic solutions. *Journal of Agricultural, Biological, and Environmental Statistics*, 2017.

[16] Guansong Pang, Chunhua Shen, Longbing Cao, and Anton van den Hengel. Deep learning for anomaly detection: A review. *ACM Computing Surveys*, 2021.

[17] Konduri Aditya, Hemanth Kolla, W. Philip Kegelmeyer, Timothy M. Shead, Julia Line, and Warren L. Davis IV. Anomaly detection in scientific data using joint statistical moments. *Journal of Computational Physics*, 2019.

[18] Juan Manuel Garcia, Tomás Navarrete, and Carlos Orozco. Workload hidden markov model for anomaly detection. *Proceedings of the International Conference on Security and Cryptograph*, 2006.

[19] Sucheta Chauhan and Lovekesh Vig. Anomaly detection in ecg time signals via deep long short-term memory networks. *IEEE*, 2015.

[20] Daniel Jurafsky and James H. Martin. Speech and language processing. `https://web.stanford.edu/~jurafsky/slp3/A.pdf`, Draft of January 7, 2023. Online. Last time accessed on June 3, 2023.

[21] Brilliant.org. Markov chains. `https://brilliant.org/wiki/markov-chains/`. Online. Last time accessed on June 3, 2023.

[22] Hristo Hristov. An introduction to the hidden markov model. `https://www.baeldung.com/cs/hidden-markov-model`, 2022. Online. Last time accessed on June 30, 2023.

[23] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2), 1989.

[24] Read Jonathon. Hidden markov models and dynamic programming. *University of Oslo*, 2011.

[25] Rob Hyndman and George Athanasopoulos. Forecasting: Principles and practice. https://otexts.com/fpp2/components.html. Online. Last time accessed on June 26, 2023.

[26] Rami Krispin. Decomposition of time series. https://ramikrispin.github.io/halloween-time-series-workshop/ts-decomposition.html. Online. Last time accessed on June 26, 2023.

[27] Rob Hyndman and George Athanasopoulos. Forecasting: Principles and practice. https://otexts.com/fpp2/classical-decomposition.html. Online. Last time accessed on June 26, 2023.

[28] Daniel Jurafsky and James H. Martin. Speech and language processing. https://web.stanford.edu/~jurafsky/slp3/9.pdf, Draft of January 7, 2023. Online. Last time accessed on June 28, 2023.

[29] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.

[30] Ekaansh Khosla, Dharavath Ramesh, Rashmi Priya Sharma, and Samuel Nyakotey. Rnns-rt: Flood based prediction of human and animal deaths in bihar using recurrent neural networks and regression techniques. *Procedia Computer Science*, 2018.

[31] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. *Neural computation*, 1997.

[32] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. http://karpathy.github.io/2015/05/21/rnn-effectiveness/, 2015. Online. Last time accessed on June 30, 2023.

[33] M. A. Kramer. Autoassociative neural networks. *Computers and Chemical Engineering*, 16(4), 1992.

[34] Hoang Duy Trinh, Engin Zeydan, Lorenza Giupponi, and Paolo Dini. Detecting mobile traffic anomalies through physical control channel fingerprinting: a deep semi-supervised approach. *IEEE*, 2019.

[35] Mike Clayton. Is julia really faster than python and numpy? `https://towardsdatascience.com/is-julia-really-faster-than-python-and-numpy-242e0a5fe34f`. Online. Last time accessed on June 11, 2023.

[36] Dimitre Oliveira. Time series forecasting with lstm autoencoders. `https://www.kaggle.com/code/dimitreoliveira/time-series-forecasting-with-lstm-autoencoders`, 2020. Online. Last time accessed on June 30, 2023.