Modeling Heterogeneous Seasonality With Recurrent Neural Networks Using IoT Time Series Data for Defrost Detection and Anomaly Analysis

Modeling Heterogeneous Seasonality with Recurrent Neural Networks

Using IoT Time Series Data for Defrost Detection and Anomaly Analysis

Suraj Khetarpal

A Thesis in the Field of Mathematics and Computation

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

June 2020

Abstract

Detecting anomalies and predicting failure of industrial refrigeration equipment are paramount to guarantee a reliable supply chain and consumer safety in a variety of industries such as pharmaceutical and grocery. Failure can be predicted by performing an anomaly analysis on internal temperature data that has been collected by Internet of Things (IoT) sensors. Such an analysis involves monitoring a refrigeration unit's defrost cycle, which is the seasonal (i.e. periodic) component of its temperature time series.

In many industries, Recurrent Neural Networks (RNNs) are used to analyze and forecast time series data, and Long Short Term Memory (LSTM) cells are used to remember long term dependencies. When using deep learning tools to analyze time series data, a major challenge is modeling datasets with heterogeneous seasonal components.

This thesis investigates the ability of RNNs built with LSTM cells to detect defrost events from within temperature time series that were recorded using IoT sensors. Because defrost events are seasonal and heterogeneous across time series, the successful detection of defrosts is dependent on an RNN's ability to build a model of heterogeneous seasonality.

We conducted our research in two phases. During the first phase, we generated datasets of simulated refrigeration temperature time series and used them to train and test RNNs, resulting in a 95% classification accuracy rate. During the second phase, we

analyzed the challenges inherent in modeling heterogeneous seasonality in our time series datasets. We designed new experiments and modeled heterogeneous seasonality using binary datasets consisting of 1s and 0s. Using a binary dataset whose seasonal patterns were heterogeneous in both frequency and phase, RNNs achieved 100% forecasting accuracy. However, when we added confounding features to the dataset, forecasting accuracy dropped to 71% as confounding features could easily be confused with the seasonal patterns of time series data. Our results revealed that RNNs are best suited to model heterogeneous seasonality in datasets that do not contain confounding features.

Acknowledgments

I would like to express my deepest appreciation to my thesis director, Dr. Sylvain Jaume, for guiding me through every stage of the thesis process and for his valuable advice and support. I also wish to thank, from the bottom of my heart, my thesis advisor, Dr. Jason Sroka from Digi SmartSense, who made this project possible. Jason went above and beyond to support me, providing technical expertise, advice, and insights. Lastly, I would like to thank my loving wife and daughter.

Table of Contents

Chapter I.

Introduction

The temperature fluctuations inside a refrigeration unit (i.e. a refrigerator or freezer) are predominantly driven by the unit's compressor cycle and defrost cycle. A compressor cycle is the intermittent operation of the unit's cooling system. As the compressor cycles, a thick layer of frost can accumulate around the cooling system, creating a thermal barrier that greatly decreases its efficiency and impedes its ability to regulate its temperature. To mitigate this problem, many refrigeration units periodically defrost. During a defrost event, a heater drives up the internal temperature in order to melt the layer of frost. A defrost event lasts for only a short time so that the unit's contents are not impacted.

Most defrost events occur at specific, regular time intervals. Compared to a compressor cycle, a defrost cycle usually has a much larger thermal amplitude and a much lower frequency (i.e. a larger period). When a refrigeration unit's temperature time series is plotted, the defrost events usually take the shape of a precipitous spike followed by a return to normal temperatures (see Figure 1).

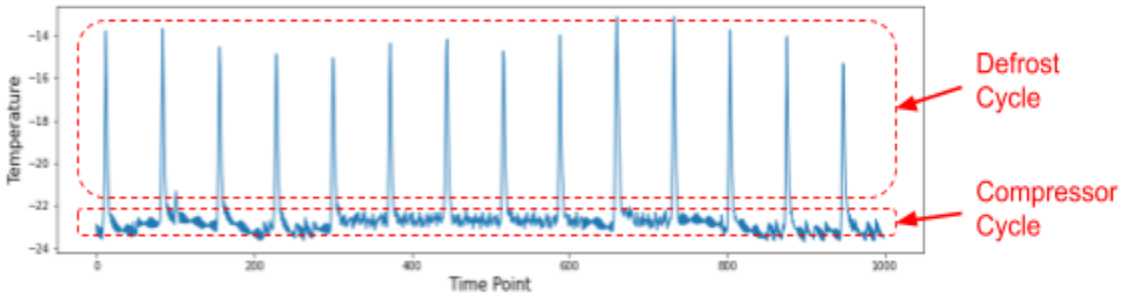


Figure 1. A typical time series of refrigeration temperature readings. The defrost cycle has a far greater amplitude than the compressor cycle, and repeats at specific, regular intervals. Note that the time series is not actually continuous; it is an ordered set of discrete time point values that have been connected for the sake of visualization.

In the context of time series analysis, a pattern that repeats at specific, regular intervals within a time series is referred to as the "seasonal" component of the time series. A seasonal component is characterized by its frequency, the shape of the repeating pattern, and its phase (i.e. the locations in time of the repeating pattern).

When a dataset is composed of many time series, it can be useful to describe the relationship that exists between the time series' seasonal components. A dataset shall be called seasonally "homogeneous" when the seasonal component of every time series has the same frequency, the same phase, and the same repeating shape. Conversely, a dataset shall be called seasonally "heterogeneous" if any of these criteria are not met. In other words, a heterogeneous dataset is one whose seasonal components have either heterogeneous frequencies, heterogeneous phases, or heterogeneous shapes.

Defrost cycles are the seasonal component of a refrigeration unit's temperature time series. Their characteristics are dependent on the refrigeration unit's design, use case, environment, age, and condition. As a result, defrost cycles vary greatly between units. For example, their shapes can vary in amplitude, width, smoothness, upward

trajectory, and downward trajectory. Therefore, refrigeration temperature datasets are almost guaranteed to be seasonally heterogeneous, in that their seasonal components (the defrost cycles) will have varying seasonal frequencies, phases, and shapes.

1.2 Problem Statement

When a refrigeration unit is unable to keep cool, its temperature sensitive contents can quickly spoil. In commercial settings, refrigeration problems can lead to the loss of large amounts of food or pharmaceuticals, and can endanger the health and safety of consumers. That is why it is crucial that refrigeration problems be detected as quickly as possible, preferably as they occur. Unfortunately, refrigeration units are not always designed to notify users when a problem is occurring, or when a catastrophic malfunction is impending. Consequently, users must find alternative ways to detect refrigeration issues.

One such way is to place an IoT temperature sensor inside of a unit, take periodic readings, and watch for anomalies in the resulting time series. An anomaly is an indicator that the unit is starting to malfunction, or that it is being used improperly. For example, if a refrigerator's temperature suddenly rises, but it is not defrosting, this could mean that the cooling system has stopped working, or that the door has been left ajar. If the defrost cycle frequency changes, this could indicate that the unit is about to malfunction.

In order for a user to recognize an anomaly in real time, they must be able to distinguish between an anomaly and a defrost event. Because both defrosts and

anomalies oftentimes involve a sudden rise in temperature, they can easily be confused. Therefore, it is necessary for the user to identify the seasonal defrost cycle within the time series of temperature readings, so they can then judge how the last time points fit into that cycle.

Clearly, it would be useful if users could detect when defrosts are occurring. One technique that is sometimes used to identify defrosts is to plot the unit's up-to-date temperature readings. The human eye can then easily observe the seasonal defrost pattern and judge where the latest readings fall in the defrost cycle. However, this technique is tedious and oftentimes infeasible. An automated approach to defrost detection using temperature readings would be preferable.

This thesis describes an investigation into the ability of recurrent neural networks (RNNs) to automatically determine whether or not a refrigeration unit is defrosting, using only a time series of internal temperature readings. The first phase of the thesis project tested whether or not RNNs could serve as defrost detectors. RNNs were trained to classify the defrost status (defrost vs. no defrost) of the last time points in temperature time series, as if they were making real time, up to date classifications.

Simulated data had to be used to train and test the RNNs. Refrigeration temperature datasets exist, but none come with time point labels indicating whether or not the unit was defrosting. Therefore, they cannot be used for supervised learning. Consequently, the thesis project began with the fabrication of a labeled dataset that simulated real refrigeration temperature time series. For the duration of this paper, this dataset will be referred to as the "simulation dataset". The simulation dataset captures

much of the complexity of real refrigerator temperature time series, which are not only seasonally heterogenous, but are also sometimes chaotic and erratic for no apparent reason. For example, entire defrost cycles are sometimes missing, and at other times, temperatures will misleadingly rise and fall as if a defrost is occurring.

As described later, experiments with the simulation dataset had mixed results. It is possible that the RNN classification accuracy suffered because the RNNs had difficulty modeling the heterogeneous seasonal components contained in the dataset. However, because we do not fully understand the limitations of RNNs, it is difficult to know for sure whether the gaps in performance were caused by an inability to model the dataset's heterogeneous seasonal components or some other intrinsic characteristic of the dataset. With this in mind, the thesis project included a second phase to more definitively test the ability of RNNs to model datasets with heterogeneous seasonal components. During this second phase, the RNNs were trained and tested on fabricated binary datasets consisting of sparse, binary time series. Within each time series, 1s appear in a seasonal fashion, at specific, regular intervals. The RNNs were tasked with forecasting whether the next time point value in the series' seasonal component would be a 0 or a 1. Because the time series are binary, the RNNs were forced to predict values based on nothing but the number of time points that had elapsed since the previous seasonally occurring 1.

Multiple binary datasets were created and tested, each with different heterogeneity characteristics. They were designed to tease apart an RNN's ability to handle different types of complexities, including heterogeneous phases, heterogeneous

frequencies, and the presence of confounding features. See the Methods section for a more detailed description of the binary datasets.

This thesis has far reaching implications. If RNNs could be trained to provide real time classifications for seasonally heterogeneous time series, they would be useful in a wide array of time series applications. For example, they could be used to perform anomaly detection in applications involving home appliances, industrial machinery, vehicles, or even human organs.

1.3 Requirements

Before an RNN can classify a time series' terminal defrost status, it must first learn to model refrigeration temperature patterns, and most importantly, seasonal defrost cycles. However, before it can model defrost cycles, the RNN must be able to recognize the individual defrost events that appear throughout each time series in the training data. Defrost events can be identified by their typical shape and their repeating, evenly spaced seasonal pattern. Shape alone is insufficient, since many time series contain confounding features, which are features that in some way resemble a defrost and that might fool a defrost detector into outputting a false positive.

Modeling refrigeration temperature patterns is further complicated by the fact that defrost cycles are heterogeneous across refrigeration units. The defrost pattern for one unit is independent of the defrost pattern of any other unit, unless they happen to be identical models and are operating in similar conditions. Defrost patterns vary in not

only frequency and shape, but also phase. A defrost cycle's phase is defined by the locations at which the repeating pattern starts and stops relative to the time series indices.

Figure 2 illustrates some of the differences that can exist in the temperature patterns of different refrigeration units. It shows real data that was sampled by temperature sensors in various refrigerators and freezers. It highlights the many ways in which real data can differ, and some of the complications that can make the data difficult to model. Note that the time series seasonal components vary in frequency, phase, and shape.

All of the variations and all of the complex features present in Figure 2 needed to be replicated in the simulation dataset. The different characteristics of the real data were each simulated as a separate time series and then added together to create a final simulated time series for the dataset. See the Methods section for a detailed description of how the different characteristics were constructed and then combined.

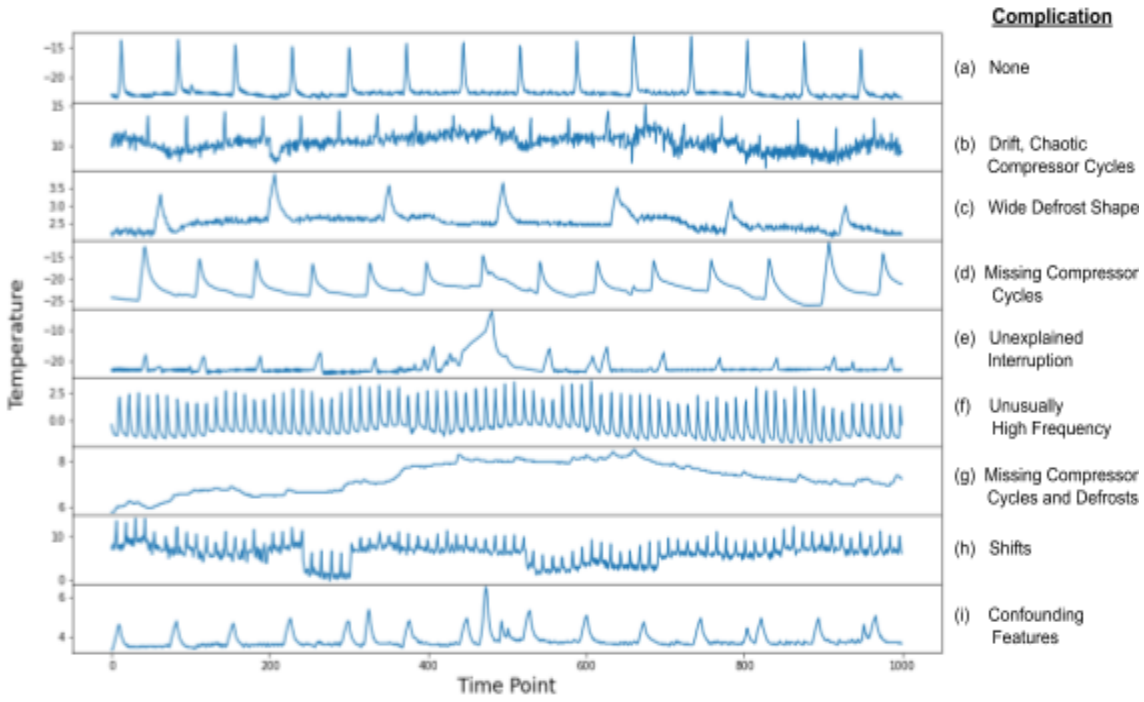


Figure 2. Examples of real refrigerator temperature time series. They have been chosen to showcase common ways in which the data varies. To the right of each time series plot is a description of a complication that makes the time series different from a standard refrigerator temperature time series. Note that the time series are not actually continuous; they are ordered sets of discrete time point values that have been connected in the plots. (a) A standard refrigerator temperature time series. (b) The time series drifts up and down over time, and the compressor cycles are tall and chaotic. (c) The defrosts are wide and prolonged, in contrast to the spike contour that typically characterizes defrosts. (d) The compressor cycles are not visible. (e) An unexplained event occurs, causing the temperature to change dramatically. It could indicate that a problem occurred, such as a door was left open or that the refrigerator malfunctioned. (f) The defrosts have an unusually high frequency, demonstrating that defrost frequency can vary greatly between refrigerators. (g) An example with no apparent defrost or compressor cycles. The RNN classifier must learn to avoid false positives when no defrosts are present. (h) The time series shifts up and down at various points in time. (i) The time series includes temperature spikes that resemble defrosts. These spikes are confounding features because they might fool a defrost detector into outputting false positives.

Much of the variation shown in Figure 2 is caused by differences in the design of the refrigeration units. However, some of the variation is caused by the placement of its contents, its contents' thermal properties, or the placement of the temperature sensor. For example, the range of temperature values can be reduced if the heating or cooling elements are covered, or if the temperature sensor is obstructed.

Some of the complications mentioned in Figure 2 might be explained by factors that are external to the refrigeration unit. For example, temperature shifts, temperature spikes, and cooling interruptions could be caused by a refrigerator door being left open, or by an interruption to the unit's power.

Note that the goal of the thesis project was not to create an RNN that can only detect defrosts for just one time series. Instead, the RNN was intended to be a generic tool that detects defrosts within any refrigeration unit's time series. Therefore, it needed to perform well on large datasets with heterogeneous seasonal components. However, it is worth noting that if real refrigeration temperature data came with defrost labels, it might be possible to custom train an RNN to predict defrost events in just one refrigeration unit and avoid all challenges associated with heterogeneity.

In order to classify the defrost status of the final time points in a time series, an RNN may have to perform three subtasks. The first subtask is perhaps the most difficult; the RNN must detect the time series' unique defrost pattern, including its frequency, shape, and location. Because the dataset is heterogeneous, the RNN cannot only learn to model one specific defrost pattern. Rather, it needs to build a generic seasonal model that adapts to the defrost pattern of each test example in real time. Once the network has detected a time series' defrost pattern, it must then determine where in that pattern the time series terminates. In other words, it must determine the location of the final time points with respect to the defrost cycle. This relative location provides the network with information about how likely it is that the last time points are part of a defrost event. Finally, the network must compare the shape of the last portion of the time series to the

typical shape of a defrost event. In order to classify the defrost status of the final time points in the time series, the RNN must decide how likely it is that a defrost is occurring based upon its terminal location in time relative to its seasonal defrost cycle, and on the shape of the last portion of the time series. This can be a difficult task, especially when the defrost is just beginning and has not yet caused the temperature to rise.

A defrost detector is a classifier that behaves like both a classifier and a forecaster. Like a traditional machine learning classifier, the defrost detector must detect features contained in data (in this case, the seasonal patterns) and use them to make an abstract judgement (defrost vs. no defrost). However, classifiers typically make judgements about an entire set of inputs, whereas a defrost detector needs to make judgements about individual data points. The ability to evaluate individual time points is a quality typically found in forecasting tools. Like a forecaster, the defrost detector must know the temporal locations of the data's features. For example, it must determine where the last time points in the time series fall within the seasonal defrost cycle. Despite the fact that defrost detection has a forecasting-like quality, it differs from previous time series forecasting applications in that it forecasts time point classes rather than time point values.

Before an RNN defrost detector can classify seasonally heterogeneous data, it must learn a general model of seasonality, so that it can work with a wide range of seasonal profiles. Most previous research on forecasting or classifying seasonal data has revealed that RNNs are able to model seasonally homogeneous datasets, but that they struggle with seasonally heterogeneous datasets (as discussed in Chapter 2)

[11,13,17,18,19]. This suggests that an RNN may only be able to learn to model a single seasonal pattern. However, this limitation has not been firmly established by previous research, and so it was worthwhile to test whether it manifests itself in the application of defrost detection.

Chapter II.

Prior Work

No previous research specifically deals with the real time detection of defrosts using temperature time series data. Therefore, the following is a general overview of the research that currently exists on modeling and predicting time series with seasonal components. The major topics covered are Seasonal-Trend Decomposition (STL Decomposition) [1], Exponential Smoothing [2], AutoRegressive Integrated Moving Average (ARIMA) [3], Machine Learning Models [4-8], and Deep Learning Neural Networks [9-20].

2.1 Traditional Univariate Time Series Modeling Techniques

STL Decomposition is a forecasting method used to decompose a time series into three components: a trend time series, a seasonal component, and a residual (or error) time series [1]. STL decomposition requires that the time series' seasonal frequency is known in advance so that it can be passed in as an input parameter. It does not provide a way to determine an unknown seasonal frequency. Therefore, it cannot be used to detect defrosts, since defrost detection requires a tool that can infer unknown seasonal frequencies.

Exponential smoothing (ES) is another forecasting method that generates error, trend, and seasonal components [2]. These components are built using weighted averages

of past data. ES has the same drawback as STL, which is that the seasonal frequency must be known in advance. Therefore, ES cannot be used to analyze multiple time series with varying, unknown seasonal frequencies.

Autoregressive integrated moving average (ARIMA) models are a popular tool for modeling univariate time series. ARIMA is a generalization of older models, including autoregressive models, moving average models, and autoregressive moving average models [3]. Despite its flexibility, ARIMA has a major drawback in that it requires the user to select and tune parameters for each time series. As a result, each time series analysis requires its own ARIMA model. A single ARIMA cannot autonomously classify many different time series with varying seasonal characteristics. This lack of flexibility and automation makes ARIMA unsuitable for modeling heterogeneous seasonal time series with unknown seasonal frequencies.

2.2 Machine Learning Models

Hundreds of machine learning models have been applied to time series classification problems [4]. For example, Lines et al. [5] were able to classify time series using a nearest neighbors algorithm with dynamic warping. Baydogan et al. [6] used random forest classifiers, and Bostrom et al. [7] used an ensemble of discriminant classifiers. In all of these applications, classification was done on entire time series rather than on individual time points, and so their methodologies cannot be used to detect defrost.

Kulkarni et al. [8] used machine learning tools to conduct some of the only research that has been done on refrigerator temperature time series classification. Specifically, they explored the topic of refrigerator temperature time series anomaly detection. In other words, they created a tool that could look at temperature data and detect if a refrigerator is malfunctioning. They used an approach that combined seasonality-trend decomposition and random forest machine learning classifiers (a type of classification tree model). Their dataset consisted of temperature time series examples from supermarket refrigerators, and they labeled each time series as either anomalous or non-anomalous depending on whether or not a repair work order had been opened on the associated refrigerator within some maximum span of time from when the data was collected.

Their work differs from this thesis project in that they generated classifications for entire time series, while this thesis project involved classifying individual time points. Furthermore, Kulkarni et al. were interested in classifying time series anomalies rather than identifying defrosts. Lastly, their work differs in that they used real, labeled data, whereas this thesis worked with fabricated datasets.

2.3 Deep Learning Models

Recently, deep learning neural networks have been used to model time series data with great success [9]. Recurrent neural networks are typically used for time series forecasting, since they are excellent at modeling temporal relationships, while convolutional neural networks are more popular for time series classification tasks [9,10].

A major challenge in modeling time series datasets with recurrent neural networks is that RNNs struggle to learn long term dependencies. For example, they have difficulty encoding long term seasonal patterns. Fortunately, this shortcoming can sometimes be alleviated by building the RNN out of Long Short Term Memory (LSTM) cells [11]. LSTM cells are well suited to capture the relationships that exist between data separated by large numbers of time points [12]. As a result, LSTMs have been used in many time series applications that involve long term dependencies, including forecasting [13] and anomaly detection [14].

Both Nelson et al. [15] and Zhang et al. [16] tested the ability of RNNs to model seasonal time series datasets, and they concluded that performance improves when the time series are deseasonalized during preprocessing. Deseasonalization is the process of removing the seasonal component from a time series before it is modeled.

More recent research has revealed that RNNs are capable of modeling seasonality when they are trained on large datasets of seasonally homogeneous time series [11,13,17,18]. Again, homogenous datasets are those that consist of time series whose seasonal components share the same frequency, shape, and phase. These studies suggested that when an RNN is trained on a homogeneous dataset, it is able to learn the underlying, shared seasonal pattern, which improves forecast accuracy [11]. Consequently, RNNs are increasingly used in big-data time series applications, often resulting in state of the art performance that surpasses traditional univariate statistical techniques.

Hewamalage et al. [11] studied the ability of recurrent neural networks to forecast seasonal time series datasets. They too concluded that RNNs can directly model seasonality when trained on seasonally homogeneous datasets. However, they also demonstrated that performance degrades when an RNN attempts to model datasets with heterogeneous seasonal components, suggesting that RNNs struggle to model heterogeneous seasonality. They recommended that heterogeneous datasets be deseasonalized during preprocessing.

Smyl [18] used a dynamic computational graph neural network to win the M4 time series forecasting competition. He used a novel hybrid approach that combined exponential smoothing with LSTM networks. The model was able to learn from large datasets whose seasonal components had homogenous seasonal frequencies but heterogeneous phases. The seasonal frequencies were known upfront and used during the exponential smoothing step to extract the seasonal component of each time series.

Bandara et al. [17] achieved state of the art time series forecasting results using a process that included a decomposition stage followed by an LSTM network stage. They created the "Long Short-Term Memory Multi-Seasonal Net (LSTM-MSNet), a decomposition based, unified prediction framework to forecast time series with multiple seasonal patterns" [17]. In one version of their framework, the time series was deseasonalized prior to entering the recurrent neural network. Traditional univariate decomposition techniques were used, requiring that the seasonal frequency had to be known in advance. In a second version, the seasonality was left intact as the data entered the recurrent neural network. In testing this later version, Bandara et al. discovered that

the RNN was good at learning to forecast seasonal data so long as all of the examples in the dataset were seasonally homogenous. Conversely, the same RNN had trouble learning from data whose seasonal components had heterogeneous shapes and phases. When the seasonally heterogeneous data was deseasonalized during preprocessing, forecast accuracy improved. Bandara did not test datasets whose seasonality varied in frequency, leaving open the question of whether or not seasonal frequency heterogeneity would further confound the recurrent neural network.

Bandara et al. [13] presented a novel approach to modeling heterogeneous time series datasets. They used a time series clustering technique to create groups of similar time series, and then trained LSTM networks to learn customized models for each group. However, prior to training the LSTM networks, they used STL decomposition to deseasonalize the data. Therefore, this technique only works for datasets with known seasonal frequencies.

Lai et al. [19] were able to effectively model seasonal data using a hybrid convolutional and recurrent neural network. The convolutional layer captured short term patterns while the recurrent layers captured long term patterns. In order to model the longest term seasonal patterns in the data, the recurrent layers featured temporal-skip connections which passed recurrent cell state data directly to temporally distant iterations of the same recurrent layers, extending the temporal reach of the information. The skip connections were tuned to match the periods of the seasonal patterns so that the network could look back at the previous cycle when predicting the current cycle. This method was very successful as long as the dataset's seasonal components all had the same

frequency. When heterogeneous datasets were tested, the neural network's prediction performance dropped below that of traditional univariate statistical techniques.

Laptev et al. [20] proposed a method for improving forecast accuracy on heterogeneous time series data. They used two separate modules in their process, an LSTM based autoencoder and an LSTM forecaster. They trained the LSTM autoencoder to generate extra features that would assist the LSTM forecaster in making predictions. The extra features were concatenated onto the original time series before being fed to the LSTM forecaster. This resulted in a 14% accuracy increase over an LSTM forecaster trained without the extra features.

Much of the aforementioned deep learning research has demonstrated that RNNs are good at modeling deseasonalized datasets and datasets with homogeneous seasonal components [11,13,17,18]. Multiple papers have discussed the difficulty that RNNs have in modeling heterogeneous seasonal datasets [11,17,19]. This thesis examines the extent of this difficulty, and explores whether RNNs can model heterogeneous seasonality in the context of defrost detection.

This research is different from most previous research on time series classification because it is concerned with classifying individual data points rather than entire time series [9]. It is also different from previous research on forecasting time series because it requires feature detection, classification, and the creation of labels.

Chapter III.

Methods

The following is a description of the methods used to generate datasets, build neural network architectures, and train and test the networks. Sections 3.1 and 3.2 describe the methods used during the defrost detection experiments, and Section 3.3 describes the methods used during the binary dataset forecasting experiments.

3.1 Simulation Dataset Generation

The simulation dataset (X, Y) is made up of a collection of simulated time series, $X = [X^1, X^2, ..., X^M]$, and a collection of corresponding time series labels, $Y = [Y^1, Y^2, ..., Y^M]$. Multiple dataset sizes were tried, including $M = 1000$, $M = 5000$, and $M = 20000$. Ultimately, the largest dataset gave the best results, and so all of the results that will be described later in this thesis were achieved using the $M = 20000$ dataset. Each time series is an ordered set of $\mathbb{N}$ real values, such that $X^m = [x_1, x_2, ..., x_N]$, with $x_n \in R$. Multiple time series lengths were tried, including $N = 150$, $N = 500$, and $N = 1000$. For each time series, there is a corresponding ordered set of N defrost labels, $Y^m = [y_1, y_2, ..., y_N]$, with $y_n \in \{True, False\}$. Each label represents the defrost status of one time point. A label of 'True' means that the time point was part of a defrost event, and a label of 'False' means that it was not.

The simulation dataset was designed to resemble real refrigeration unit temperature data. Figure 3 provides an example of a simulated time series and its labels. Each simulated time series was built using a combination of the following simulated features: compressor cycles, defrost cycles, cooling interruptions, shifts, drift, random noise, and random temperature spikes.

Each simulated time series was given a 50% probability of including a compressor cycle, drift, Gaussian noise (centered at zero), cooling interruptions, random temperature spikes, and sudden shifts. Note that in some cases, Gaussian noise can be used to simulate an especially chaotic compressor cycle, and so the effective probability of a simulated compressor cycle was 75%.

Each simulated compressor cycle (the standard, non Gaussian noise version) moved up and down linearly and with a consistent amplitude. The baseline width of the compressor cycles was sampled using $w = gamma(k = 6,\ \theta = 1)$, but it was made to drift slightly over time, and to rise and fall sinusoidally so as to mimic the variations in temperature gradients that might occur in a real refrigeration unit during the span of a day. The compressor cycle heights were randomly sampled using $h = gamma(1, 2)$.

Some real refrigeration units have no perceptible defrost cycle. Therefore, each simulated time series was only given a 90% probability of featuring a defrost cycle. This meant that when the RNNs were trained to detect defrosts, they had to attempt to avoid outputting false positives whenever they were presented with a time series that had no defrost cycle. This was difficult, especially because some of the compressor cycles have relatively large amplitudes, which makes them easily confused for defrosts.

Each simulated defrost cycle was randomly parameterized with $period = gamma(k = 3,\ \theta = 22)$ and $height = 1.5 + gamma(k = 1.5,\ \theta = 3)$. The widths (measured in time points) of the upward and downward portions of the defrost events were determined separately, using $width = gamma(1, 5)$.

Regardless of whether or not a time series featured a defrost cycle, it was still given a 50% probability of featuring random temperature spikes. These temperature spikes were designed to resemble defrosts, and so they will be referred to as "counterfeit defrosts". A counterfeit defrost is an example of a confounding feature in a time series, since it can easily fool a defrost detector into mistakenly outputting a false positive. The point of the counterfeit defrosts was to force the RNNs to learn to differentiate between defrosts that are part of a seasonal pattern and confounding features that are not. The resulting defrost detector should be more robust since it would make decisions based upon long term temporal relationships, rather than relying exclusively upon short term temperature patterns.

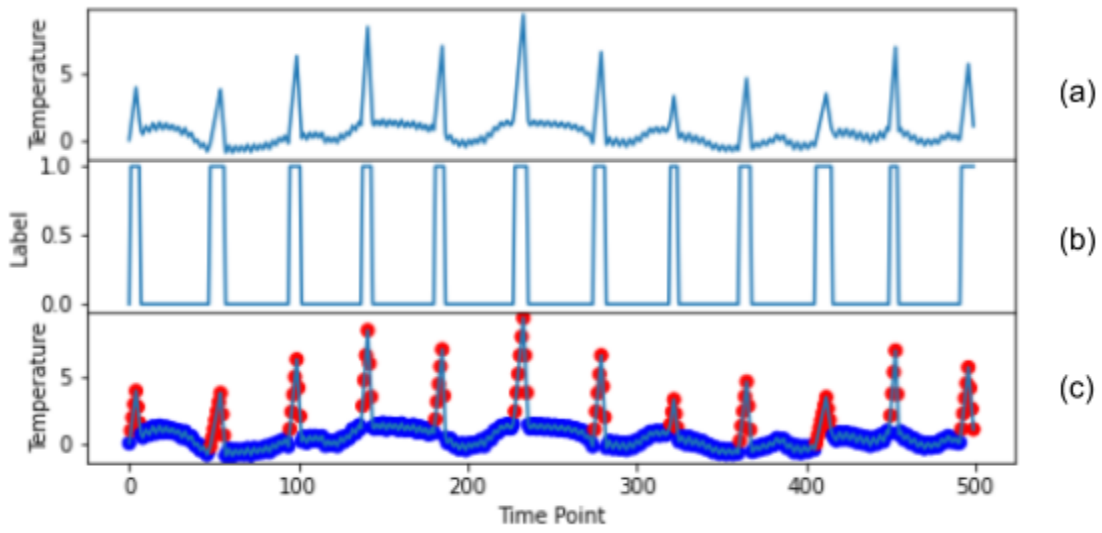


Figure 3. (a) A time series of simulated temperature readings. (b) A continuous plot of the corresponding defrost labels, where the label of True has been converted to 1 and a label of False has been converted to 0. (c) The same time series, but this time color coded according to the time point labels, with red indicating a label of True and blue indicating a label of False. Note that both the time series and the associated labels are discrete data points that have been connected in the plots for the sake of visualization.

3.2 Simulation Dataset Experiments

3.2.1 RNN Architectures

Many different RNN architectures were built and tested. Each RNN featured either 1 or 2 layers of long-short-term-memory (LSTM) cells [21], followed by a single cell dense layer with a sigmoid activation function. The number of LSTM cells per recurrent layer were either 1, 2, or 10. A binary cross entropy loss function and an Adam optimizer [22] were used to update weights.

The RNN was built using Keras on top of Tensorflow [23]. The LSTM layers were built with CuDNNLSTM, a fast LSTM implementation that is designed to work on GPUs and backed by the NVIDIA CUDA Deep Neural Network Library [24].

3.2.2 Training

Before being fed to the RNN, every time series was standardized to have a mean of 0 and a standard deviation of 1. The dataset was then split such that 1K training examples were reserved for validation, and another 1,000 training examples were reserved for testing.

Training was done using batches of size 1, 8, 16, and 64. For each training dataset / RNN architecture combination, a loss function value was calculated for each epoch using the validation data. Training was cut off when the loss stopped decreasing.

3.2.3 Classification Task

For each training example, the entire time series of timepoints was fed into the RNN, and then the RNN outputted a single defrost classification and performed backpropagation. In theory, an RNN could be trained to classify any time point in the sequence, but since this thesis project was concerned with real time defrost detection, the RNNs were trained to classify the last time point (with the exception that some were instead trained to classify the second or third from last). The dataset was balanced such that half of the time series ended with a time point that had a defrost status of 'True' and the other half ended in 'False'.

3.2.4 Performance Measures

After being trained, the RNNs were tested using simulated test data. Their classification performances were evaluated using four metrics: overall accuracy, sensitivity, specificity, and precision. The latter three metrics can be understood in terms of True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN).

$$Sensitivity = \frac{TP}{TP+FN}$$

Sensitivity is the portion of defrost time points that are correctly classified.

$$Specificity = \frac{TN}{TN+FP}$$

Specificity is the portion of non-defrost time points that are correctly classified.

$$Precision = \frac{TP}{TP+FP}$$

Precision is the portion of time points that the RNN has classified as 'True' that are actually part of a defrost event.

3.2.5 Plotting Results

Classification performance can be observed visually if a time series is plotted and then color coded according to its time point classifications. Before such a plot can be color coded, classification must be generated for the time points in the time series. Because the RNNs are trained to only provide a classification for the last time point in a time series, an iterative "sliding window" method was used to generate classification for

consecutive time points. As shown in Figure 4, a window of length *N* was slid across the subject time series, where *N* is the number of time points that are given to the RNN as input before it makes a classification. At each position, the windowed segment of *N* time points was fed to the RNN and the last time point was classified. Once the desired range of time points had been classified, they could be plotted and color coded using the classifications. This process did not generate classifications for the first *N-1* time points, and so the x-axis of the resulting classification plots had to start at time point *N*.

Classification plots were created using both simulated data and real refrigeration unit temperature data. One of their major benefits was that they provided a way to visually assess RNN classification performance on real world data.

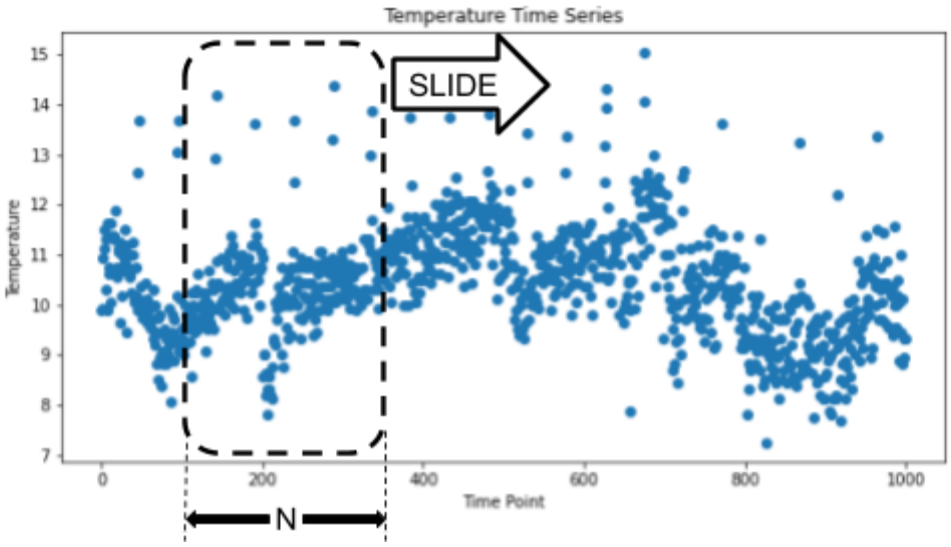


Figure 4. A sliding window is used to collect classifier input segments. Each window is used to classify the last time point in the window.

3.3 Binary Dataset Experiments

3.3.1 Binary Datasets Overview

In order to more directly test the ability of RNNs to model seasonality, a second exploration was done using four fabricated binary datasets. Each binary dataset (X, Y) is composed of a two equally sized sets of time series, $X = [X^1, X^2, ..., X^M]$ and $Y = [Y^1, Y^2, ..., Y^M]$. X provided the input data that was fed into the RNNs, while Y provided the target outputs that the RNNs were trained to forecast. The time series in Y are the seasonal components of the corresponding time series in X, meaning that the RNN was trained to predict the seasonal component of X. Dataset sizes were $M = 15000$ and $M = 25000$, and time series lengths were $N = 501$ and $N = 1001$.

Each input time series array in X is an ordered set of N binary values, such that $X^m = [x_1, x_2, ..., x_N]$, where $x_n \in \{0, 1\}$. Contained in each input time series is a seasonal component in which 1s appear at specific, fixed intervals. The seasonal frequency remains constant over the duration of each time series, but can change between time series.

In two of the four binary datasets, all time point values are set to 0 except for those that are part of the seasonal pattern of 1s. In the other two datasets, 1s appear randomly throughout the time series. Because these randomly placed 1s are not a part of any seasonal pattern, they shall be referred to as noise. To create noise within a binary datasets, each time point value was given a $1/p$ probability of being turned into a 1. The noise in the binary datasets are a type of confounding feature, since the randomly placed 1s are indistinguishable from the seasonally placed 1s. A randomly placed 1 can be thought of as being analogous to the "counterfeit defrosts" in the refrigeration simulation datasets, since both of them can be easily mistaken as being a part of a seasonal pattern.

For each input time series, there is a corresponding ordered set of N binary values, $Y^m = [y_1, y_2, ..., y_N]$, where $y_n \in \{0, 1\}$. Y^m is the seasonal component of X^m, which means that Y^m is the noiseless version of X^m. In noiseless datasets, $Y^m = X^m$.

All binary datasets were evaluated using the same performance measures and plotting techniques that were used to evaluate the refrigeration simulation datasets. See the previous sections entitled "Performance Measures" and "Plotting Results". Note that when the sliding window method was used on a binary dataset time series, it resulted in a

plot that was color coded using forecasts rather than classifications. These plots were referred to as forecast plots.

3.2.2. Binary Dataset 1: Homogeneous Frequencies, Heterogeneous Phases

Binary Dataset 1 was designed to have seasonal components with homogeneous seasonal frequencies and heterogeneous phases. The seasonal components all have a period of 100 time points, but their phases vary randomly. The dataset has no noise, and so the input time series are identical to their seasonal components. See Figure 5 for examples of the time series in Binary Dataset 1.

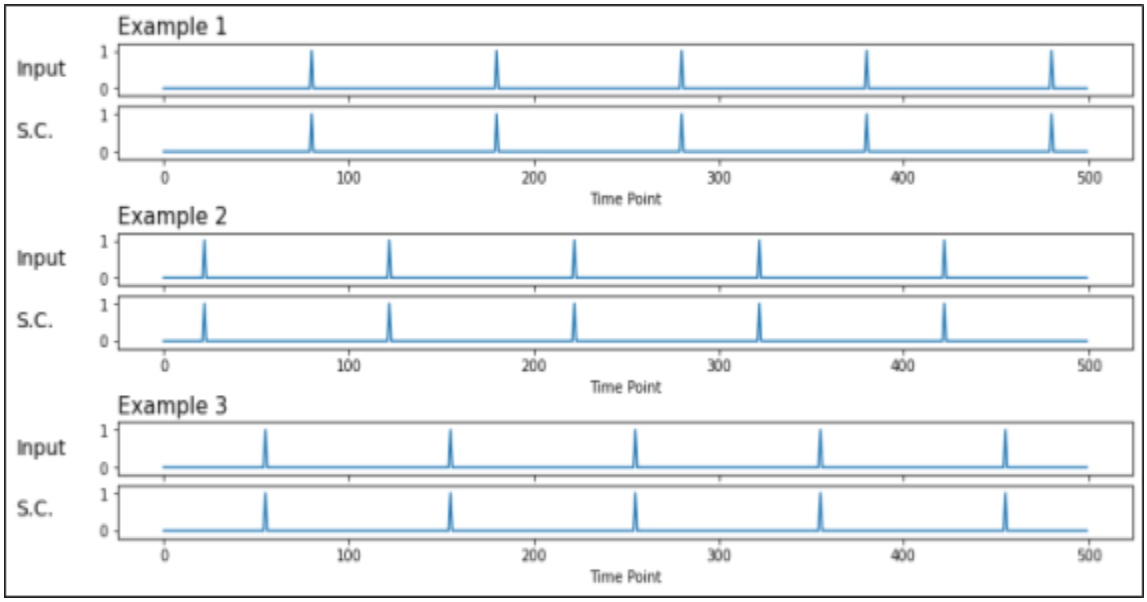


Figure 5. Three examples taken from Binary Dataset 1. For each example, the input is shown in the upper plot, and its seasonal component (S.C.) is shown in the lower plot. The time series share a common seasonal period of 100 time points, but their seasonal patterns have different phases. There is no noise in this dataset.

3.3.3 Binary Dataset 2: Homogeneous Frequencies, Heterogeneous Phases, Confounding Features

Binary Dataset 2 is identical to Binary Dataset 1 except for that noise has been added. The seasonal components have homogeneous frequencies and heterogeneous phases. 1s appear not only as part of the expected seasonal pattern, but also at random time points that are not a part of the seasonal pattern. These randomly placed 1s are confounding features. See Figure 6 for examples of the time series in Binary Dataset 2.

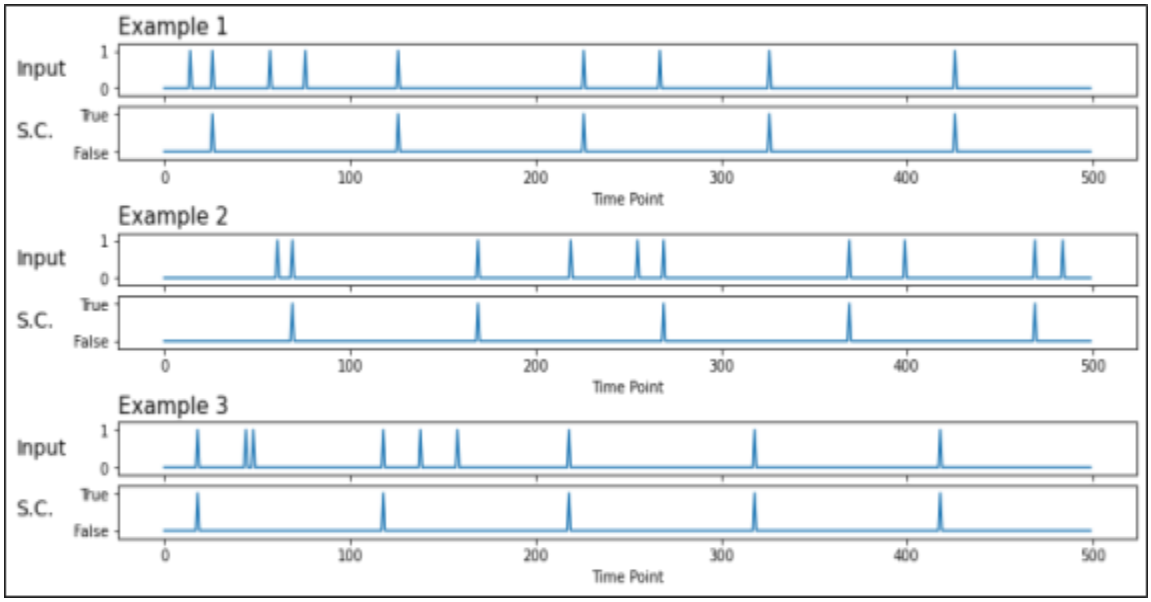


Figure 6. Three examples taken from Binary Dataset 2. For each example, the input is shown in the upper plot, and its seasonal component (S.C.) is shown in the lower plot. The time series share a common seasonal period of 100 time points, but their seasonal patterns have different phases. There is noise in this dataset.

3.3.4 Binary Dataset 3: Heterogeneous Frequencies, Heterogeneous Phases

Binary Dataset 3 is composed of time series whose seasonal components have both heterogeneous periods and heterogeneous phases. The seasonal components have periods that were randomly sampled from a uniform distribution ranging from 2 time points to 100 time points. Their phases vary randomly. The dataset has no noise, and so the input time series are identical to their seasonal components. See Figure 7 for examples of the time series in Binary Dataset 3.

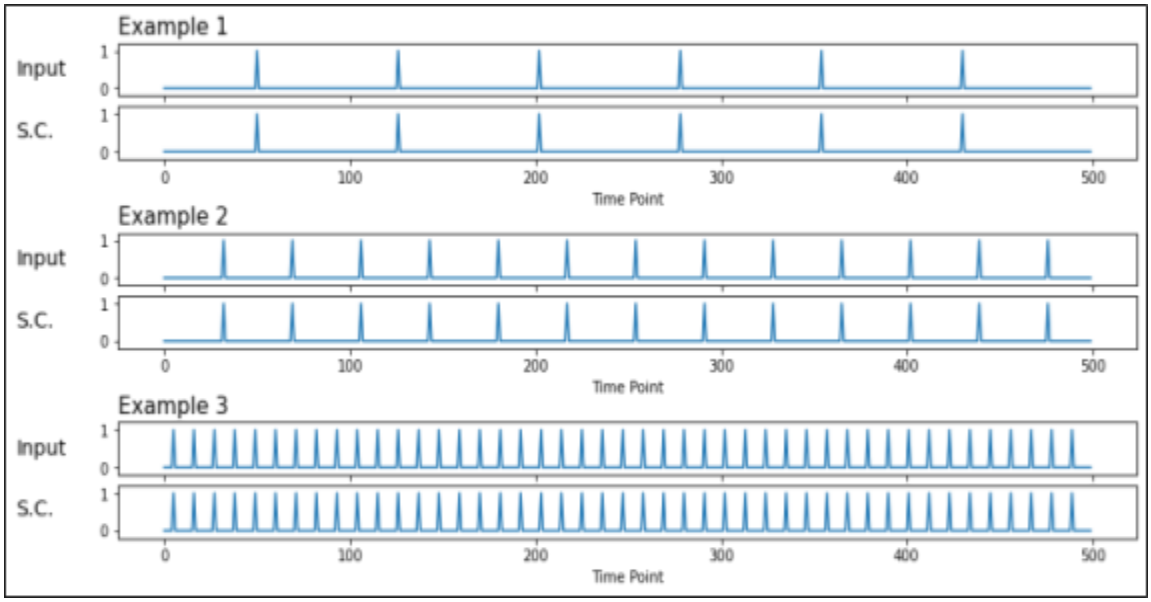


Figure 7. Three examples taken from Binary Dataset 3. For each example, the input is shown in the upper plot, and its seasonal component (S.C.) is shown in the lower plot. The seasonal components in the dataset have heterogeneous periods and phases. There is no noise in this dataset.

3.3.5 Binary Dataset 4: Heterogeneous Frequencies, Heterogeneous Phases, Confounding Features

Binary Dataset 4 is identical to Binary Dataset 3 except for that noise has been added. The seasonal components have both heterogeneous periods and heterogeneous phases. 1s appear not only as part of the expected seasonal pattern, but also at random time points that are not a part of the seasonal pattern. These randomly placed 1s are confounding features. See Figure 8 for examples of the time series in Binary Dataset 4.

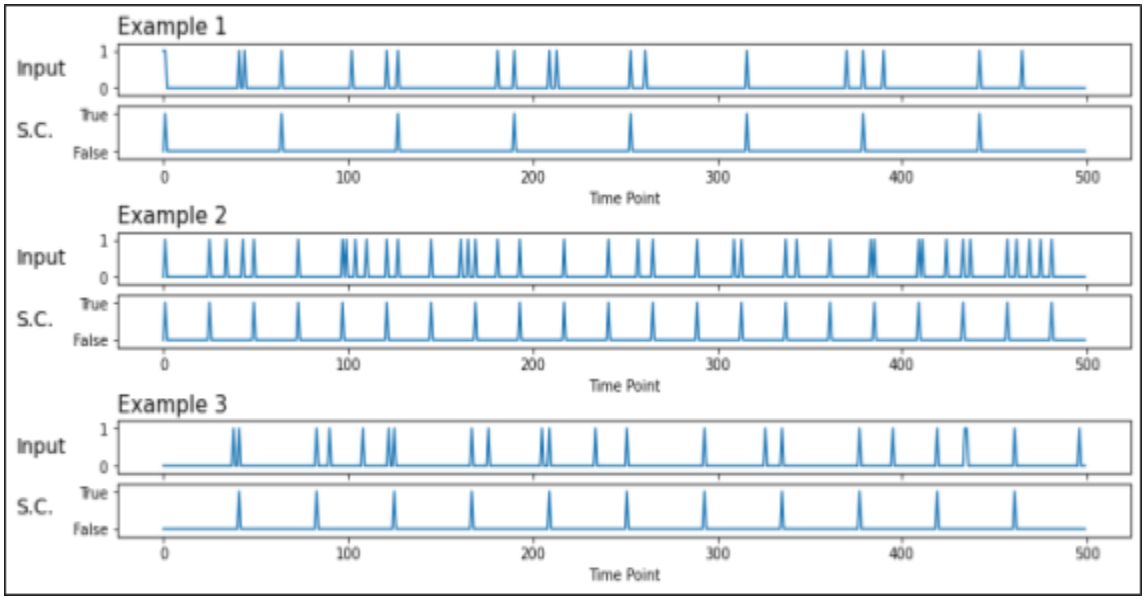


Figure 8. Three examples taken from Binary Dataset 4. For each example, the input is shown in the upper plot, and its seasonal component (S.C.) is shown in the lower plot. The seasonal components in the dataset have heterogeneous periods and phases. There is noise in this dataset.

3.3.6 Forecasting Task

In the previous experiments, RNNs were trained to classify the final time point in a time series. In the binary dataset experiments, RNNs were instead trained to forecast the next time point after the final time point in a time series. But instead of forecasting the next value in an input time series X^m, the RNN was tasked with forecasting the next value in the corresponding seasonal component time series Y^m. Given the binary nature of the time series, the RNN was forced to make predictions based entirely on the number of input time points that have elapsed since the last time point at which it saw a "1" in the input time series. If the RNN failed to model the time series' seasonal component, then it had no way to predict seasonally occurring 1s, and its true-positive prediction rate would have suffered.

Many different RNN architectures were tested. Each architecture had either 1 or 2 layers of LSTM cells followed by a single cell dense layer with a sigmoid activation function. The number of LSTM cells per layer were either 1, 2, or 10. A binary cross entropy loss function and an adam optimizer were used for training.

For the training examples of length 501, the first 500 time points were used as the input time series, and the final time point of the seasonal component, $Y^m{}_{501}$, was used as the target value that the RNN was taught to predict. For training examples of length 1001, the first 1000 time points of X^m were the input, and the target was $Y^m{}_{1001}$. For each training example, the RNN made a single prediction and performed backpropagation

once. The training datasets were designed so that the forecasted values were balanced, meaning that 50% of time series end in 0s, and 50% in 1s.

Chapter IV.

Results and Discussion

The following is a discussion of the results from the defrost detection experiments, which are covered in Sections 4.1 and 4.2, and the binary dataset forecasting experiments, covered in Section 4.3.

4.1 Simulation Dataset Results

The first set of experiments involved training RNNs to classify a refrigeration temperature simulation dataset. The classes were balanced, meaning that half of the examples ended in a defrost, and the other half did not. Various training configurations were tested, including the following:

- The time series lengths were 150, 500, and 1000
- Batch sizes of 1, 8, 16, and 64
- 1 and 2 layers of LSTM cells
- 1, 2, and 10 LSTM cells per layer

In all of these experiments, the RNNs were trained to classify either the last, second to last, or third to last time point. Table 1 provides a selection of the best results.

Experiment Configuration				Classification Performance			
LSTM Layers	Cells Per Layer	Target Time Point		Overall Accuracy	Sensitivity	Specificity	Precision
1	1	-1		0.84	0.79	0.87	0.84
1	2	-1		0.90	0.88	0.92	0.90
1	10	-1		0.94	0.91	0.97	0.96
1	20	-1		0.94	0.91	0.97	0.96
1	10	-2		0.94	0.88	0.97	0.95
1	10	-3		0.94	0.86	0.97	0.92
2	1, 1	-1		0.84	0.80	0.86	0.82
2	2, 2	-1		0.91	0.86	0.94	0.93
2	10,10	-1		0.95	0.94	0.96	0.95
2	10,10	-3		0.94	0.86	0.97	0.93

Table 1. A selection of the best result from experiments on a simulated temperature dataset in which the defrost and non-defrost classes are balanced. For each configuration, the best classification performance is shown. Various parameters were tried, including batch sizes of 1, 8, 16, and 64, and time series lengths of N=150, N=500, and N=1000. The target time point indicates which time point was classified, with -1 indicating the last time point, -2 indicating the second to last time point, and -3 indicating the third to last time point.

The highest accuracy is 95%, which was achieved by the RNN with 2 LSTM layers, with 10 LSTM cells per layer. Besides having the highest metrics, this network also had the lowest false positive rate, at 2%, and lowest false negative rate, at 3%.

For every configuration, performance always improved as the size of the simulated training dataset was increased. The best scores were achieved using the largest

dataset, which had 20,000 examples. The optimal batch size was either 1 or 8 in all cases. Surprisingly, it did not make a major difference whether the last, the second from last, or the third from last time point was the target for classification. One might expect that classifying the third from last time point would have the best results, since the RNN would be able to use the two subsequent time points (the last and second from last) as context in making the classification decision. On the other hand, one might expect that RNNs would more naturally classify the last time point, since an RNN's cell activation states continue to change all the way up to the last time point. However, neither hypothesis is supported by the results.

For every configuration, specificity was higher than precision, which was higher than sensitivity, indicating that the RNNs had more success classifying non-defrost time points correctly than defrost time points. This makes sense, since non-defrost time points generally fall within a relatively narrow temperature range. Defrost time points are sometimes more challenging to classify, since many defrost cycles begin and end with time points whose temperature values fall within the typical temperature range of non-defrost time points.

All four metrics indicate that the RNNs had some success at detecting defrosts, but they were far from perfect. It would be useful to know what kind of mistakes were made. More specifically, it would be useful to know whether the RNNs had succeeded at modeling heterogeneous seasonality and were actually using seasonal periods in their classification decisions, or if they were just classifying time point values based on temperature trajectories and variances.

The following classification plot analysis attempts to answer this question. Figure 9 shows a classification plot for a real temperature time series whose temperature readings were taken from inside an actual refrigerator. The defrosts in this time series are relatively easy to identify, and they all have roughly the same shape. The classifications were generated using the sliding window method and the best performing RNN, which was the 2 layer, 10 LSTM cells/layer RNN. As can be seen in the plot, the classifications were very accurate, with only one misclassification. Because a non-defrost time point was classified as a defrost time point, this mistake is a false positive. Note that some of the time points that were classified as non-defrost have higher temperatures than some of the time points that were classified as defrosts, indicating that the RNN was not just basing its decisions on the temperature value alone.

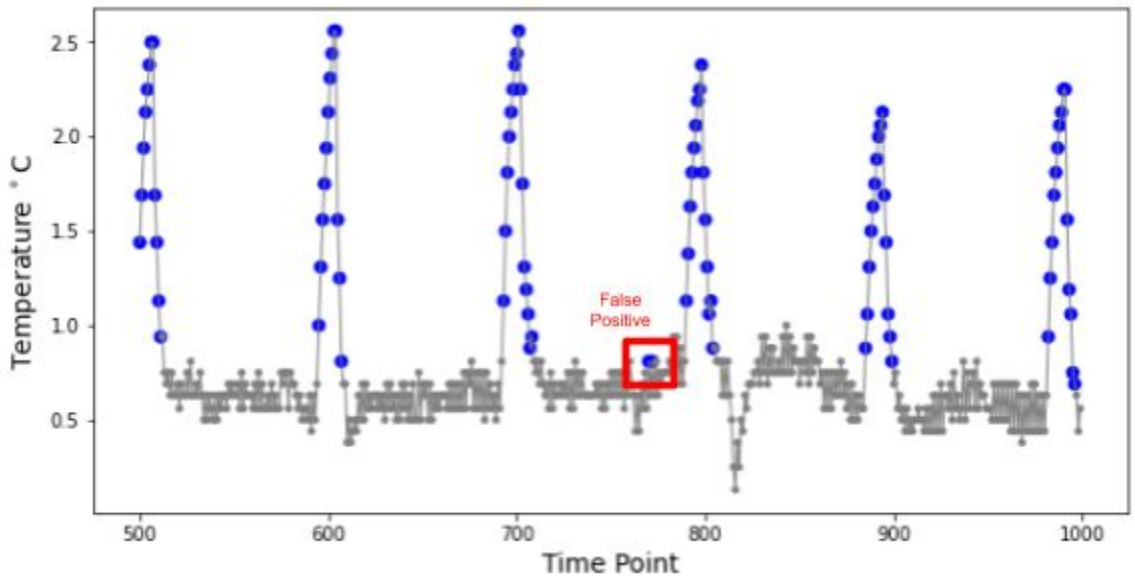


Figure 9. Classification plot for a simple, orderly temperature time series. The classifications were generated using the sliding window method with a window size of 500 time points, and they were made by the 2 Layer, 10 LSTMs/layer RNN configuration. Time points that were classified as defrost are shown in blue, and non-defrosts are shown in gray. The plot contains only one obvious misclassification. It has been highlighted with a red box.

Figure 10 shows a classification plot for a more chaotic and complex time series, again using the same 2 layer, 10 LSTM cells/layer RNN. The time series is not simulated; the temperature readings were taken from inside an actual refrigerator. Three obvious false positives appear in the plot, indicating that performance has diminished. This suggests that the trained RNN was not robust enough to model the more complicated time series.

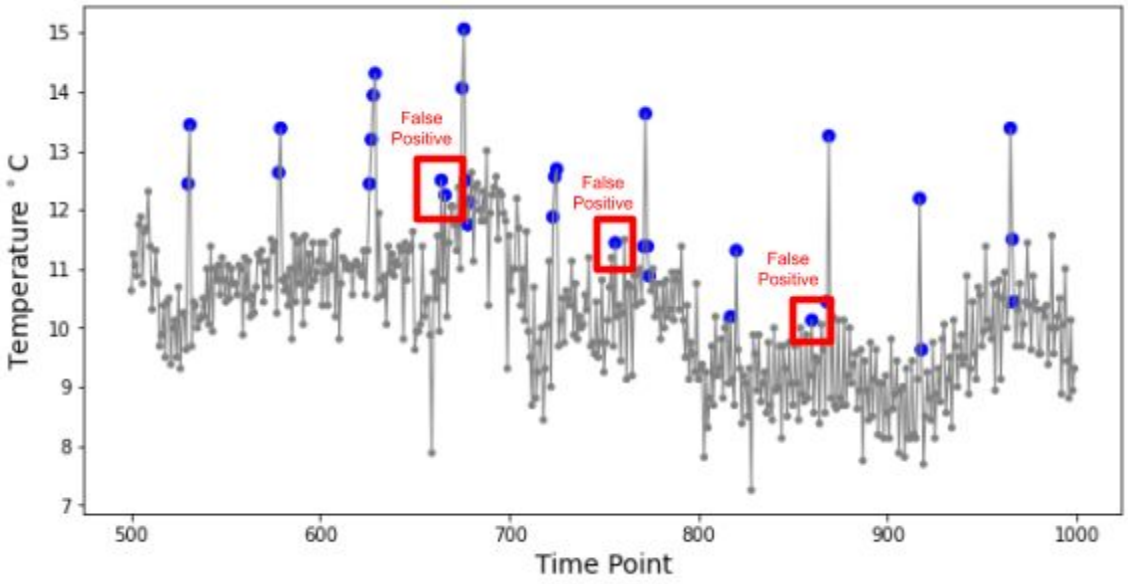


Figure 10. Classification plot for a chaotic, complex temperature time series. The classifications were generated using the sliding window method with a window size of 500 time points, and they were made by the 2 Layer, 10 LSTMs/layer RNN. Time points that were classified as defrost are shown in blue, and non-defrosts are shown in gray. Red boxes are used to highlight obvious misclassifications.

In both Figures 9 and 10, at least some portion of every defrost was detected, suggesting that the RNN is not prone to missing entire defrosts. This provides important context for the 6% error in sensitivity that was measured during testing. The error was probably caused by the RNN's failure to correctly classify the time points at the

beginnings or the ends of individual defrost events. As long as the RNN detects some portion of each defrost, the 6% sensitivity error is not cause for concern.

On the other hand, the 2% false positive rate becomes more alarming when viewed in the context of the false positives shown in Figures 9 and 10. These false positives are not connected to any defrost, and are in fact separated from any defrost by many time points. Therefore, the RNN did not merely misjudge the exact start or end point of a defrost; rather, it completely misjudged the location at which a defrost can occur. Upon inspection, it seems likely the 2% false positive rate is predominantly caused by false positives of this sort.

These mistakes demonstrate that the RNN was making two types of mistakes, the second of which is a major problem. The first type of mistake was that it was confusing non-defrosts temperature patterns for defrost patterns. In other words, the RNN was being fooled by confounding features. For example, the RNN might have seen a random temperature spike and mistook it for a defrost. This mistake is understandable, since there is no one "correct" defrost shape. In fact, the shape of a defrost varies not only between time series in the dataset, but also between defrosts within a single time series.

The second type of mistake, however, was that the RNN failed to utilize defrost seasonality (or periodicity) in its predictions. It failed to measure and enforce a consistent time interval (or period) between defrosts, implying that it has either failed to recognize that defrosts are seasonal, or is unable to learn the basic nature of seasonality in a heterogeneous dataset. Enforcement of periodicity is critical because it is the most reliable way to detect defrosts. Even though a defrost's shape can vary within a time

series, its frequency is usually consistent, and therefore, reliable (unless the refrigeration unit is failing in some way).

Figure 11 makes it clear that the RNN was in fact failing to use seasonality in its predictions. The figure shows a simulated time series that has been classified and plotted using the sliding window method. The plot is filled with misclassifications, and more importantly, there appears to be no utilization of seasonal time intervals in the classification decisions. Many of the positively classified time points are right next to each other, while others are spaced out by a great deal of time. Rather than being clustered around the correct defrost locations, the false positives are randomly placed. There are three possible explanations for this behavior: either the RNN lacks a model of seasonal time intervals, it has a weak and inconsistent model, or it has an accurate model but isn't using it well.

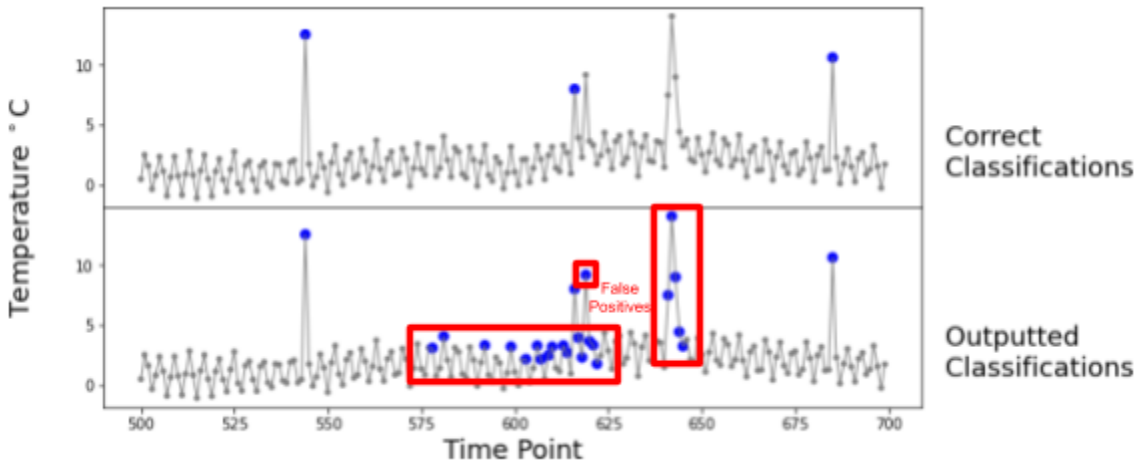


Figure 11. Classification plot for a simulated temperature time series with "counterfeit defrosts". The upper plot shows the correct classifications, with defrost time points in blue. The lower plot shows the classifications that were outputted by the 2 Layer, 10 LSTMs/layer RNN. Classifications were generated using the sliding window method with a window size of 500 time points. Time points that were classified as defrost are shown in blue, and non-defrosts are shown in gray. Misclassifications are surrounded by red boxes.

4.2 Attempt to Remediate Poor Modeling of Seasonality

Up to this point, the RNN failed to effectively model seasonality, as revealed by its tendency to output randomly placed false positives. These false positives were typically out of sync with the temporal spacing of the time series' seasonal defrost patterns. One way to compel the RNN to learn to use seasonality in its predictions is to infuse the training data with time series that ended in counterfeit defrosts. When a time series ends in a counterfeit defrost, the RNN must attempt to learn to classify the time series as non-defrost, or "False". However, the only way to recognize a counterfeit is to learn the seasonal spacing that characterizes the true defrosts. If the RNN failed to model seasonality, then it would misclassify any time series that ends in a counterfeit defrost, and the false positive rate would rise.

Two additional simulated datasets were generated in order to test whether the RNN could be compelled to learn seasonality. The first new dataset featured balanced classes, meaning that half of its time series ended in defrosts and the other half did not. Of the time series that ended in non-defrosts, half ended in counterfeit defrosts. This dataset shall be referred to as the "50/25/25 dataset". The second new dataset was split into thirds, so that one third ended in defrosts, one third in counterfeit defrosts, and the last third in neither defrosts nor counterfeit defrosts. This dataset shall be referred to as the "33/33/33 dataset". The original dataset (from Section 4.1) shall be referred to as the "50/50 dataset". Note that the 33/33/33 dataset is unbalanced, since the labels are 33% True and 67% False.

These new datasets were used to train an RNN with 2 layers and 10 LSTM cells per layer. The resulting performance metrics are shown in Table 2, and classification plots for a simulated time series are shown in Figures 12.

Ratio of Training Data Time Series				Classification Performance			
Defrost	Counterfeit Defrost	Neither Defrost Nor Counterfeit Defrost		Overall Accuracy	Sensitivity	Specificity	Precision
50%	0%	50%		0.95	0.94	0.96	0.95
50%	25%	25%		0.87	0.86	0.87	0.86
33%	33%	33%		0.87	0.83	0.89	0.87

Table 2. Classification results after training the 2 layer, 10 LSTM cells per layer RNN on datasets with different ratios of time series types. The first time series type is a time series whose last time point is part of a defrost event and that has a label of True. The second type is a time series whose last time point is part of a counterfeit defrost and has a label of False. The third type is a time series whose last time point is not part of a defrost or a counterfeit defrost, and has a label of False.

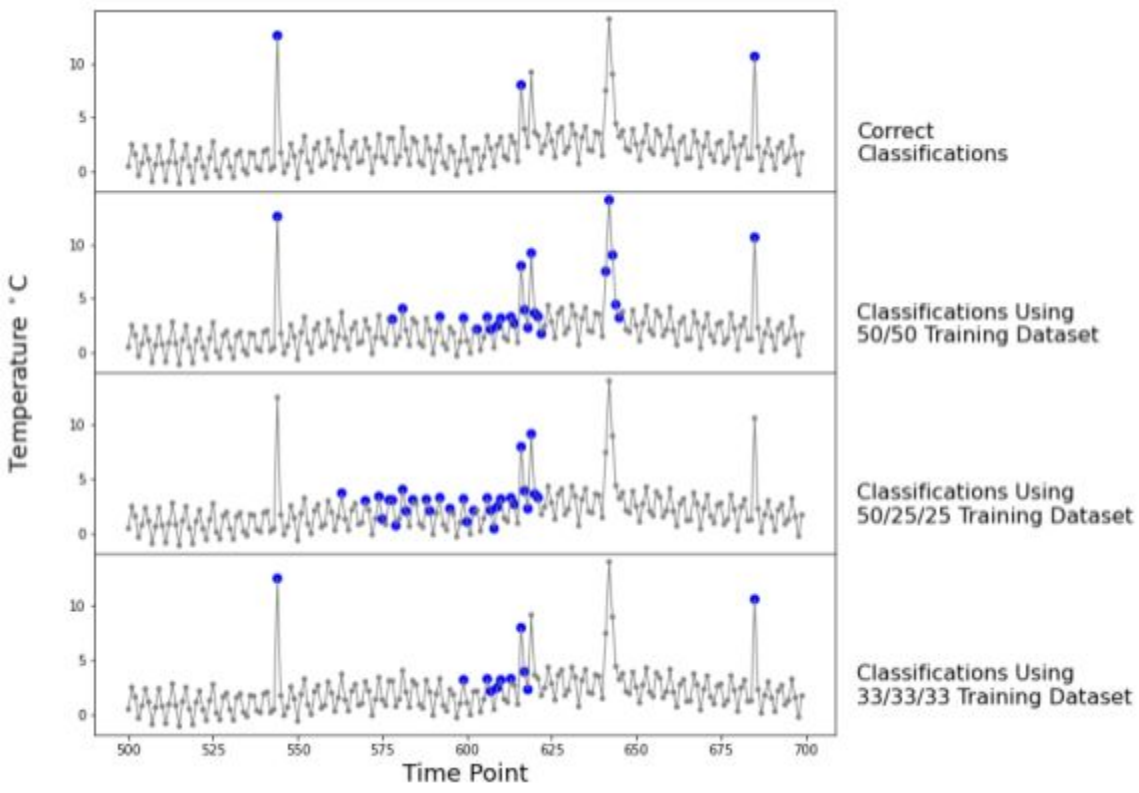


Figure 12. The top plot shows a simulated time series whose defrost time points are color coded blue. The subsequent three plots are classification plots that were generated using the sliding window method and the RNN with 2 layer, 10 LSTM cells per layer. Each classification plot was made after a different dataset was used to train the RNN. A description of each dataset is provided to the right of the plot.

All of the additional datasets resulted in lower performance metrics than the original dataset. Surprisingly, the addition of time series that ended in counterfeit defrosts did not improve the precision scores, suggesting that the RNNs were still either unable to model the datasets' heterogeneous seasonal components, or that they were still not relying heavily enough upon their internal models of seasonality when making their classification decisions.

The last plot in Figure 12 suggests that the 33/33/33 model might have learned to pay some attention to seasonal spacing, since the false positives are spaced closer to the

actual defrost. An alternative explanation is that the model was simply less prone to classifying any time point as 'True' because it was trained with an unbalanced dataset. The model's precision score was 0.87, well below that of the 50/50 models. Furthermore, its sensitivity score plummeted to 0.83, indicating that it also outputted a large number of false negatives.

Thus far, it has been unclear whether or not the trained RNNs have succeeded at modeling the heterogeneous seasonality in the simulated dataset. While the accuracy metrics have been somewhat impressive, the classification plots have revealed false positives that seem incompatible with a good model of seasonality. The next section, which focuses on the binary dataset experiments, will attempt to provide a clearer understanding of what RNNs are capable of doing with heterogeneous datasets. This will hopefully provide some much needed context that sheds light on the results from the simulated dataset.

4.3 Binary Dataset Results

As stated earlier, previous research has shown that RNNs can successfully predict datasets with fully homogeneous seasonal components, but that they struggle to predict datasets with heterogeneous seasonal components. This calls into question whether RNNs can be taught to model more than one seasonal pattern, let alone an entire heterogeneous dataset.

The next set of experiments was designed to provide a better understanding of whether RNNs are able to model heterogeneous seasonality. RNNs were trained and

tested on multiple binary datasets which had varying degrees of heterogeneity. The RNNs were tasked with forecasting the next value in the seasonal component of the input time series. The heading for each section describes the heterogeneity condition that is being tested in the experiments. The binary dataset experiments attempt to answer three questions:

1. Can RNNs model datasets whose seasonal components have heterogeneous phases?
2. Can RNNs model datasets whose seasonal components have heterogeneous frequencies?
3. Is the ability of RNNs to model heterogeneous seasonal datasets compromised by the presence of confounding features in the time series that resemble the shape of the seasonal features? Note that in the context of the simulated refrigeration unit data, the seasonal features are the defrosts, and the confounds are the "fake defrosts". In the context of the binary datasets, the seasonal features are the seasonally occurring "1"s, and the confounds are the randomly positioned "1"s (also referred to as "noise").

Note that the binary dataset experiments do not delve into the question of whether RNN can model datasets whose seasonal components have varying shapes.

4.3.1 Homogeneous Frequencies, Heterogeneous Phases

The first set of binary dataset experiments test whether RNNs can learn models of seasonality that are flexible enough to predict seasonal patterns with heterogeneous phases, but that are otherwise homogeneous. In order to answer this question, RNNs were trained to forecast Binary Dataset 1, which has homogeneous seasonal frequencies and heterogeneous phases (see Figure 5 for examples). The forecasted values are balanced, meaning that for half of the examples in the dataset, the next value in the seasonal component time series is a "1", and for the other half, it is a "0". The dataset was used to train an RNN with 1 LSTM layer with 10 cells.

After 30 epochs of training, the RNN had learned to perfectly forecast the test data. The accuracy, sensitivity, specificity, and precision were all 1. The sliding window method was used to generate forecast plots for three test examples. The forecast plots, which are shown in Figure 13, are flawless, providing visual evidence that the RNN has successfully modeled the dataset's 100 time point seasonal period. This demonstrates that RNNs are capable of modeling seasonal datasets with constant frequencies, even if the phases of the seasonal components vary across examples.

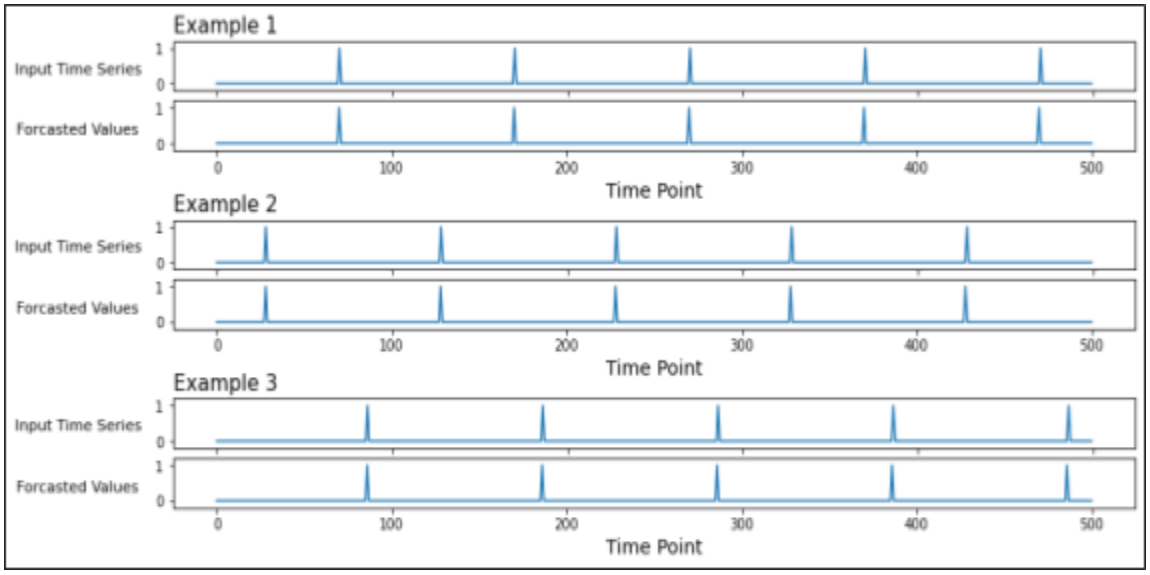


Figure 13. Forecast plots for three test examples from Binary Dataset 1. They all have seasonal periods of 100 time points, but their seasonal patterns have heterogeneous phases. The sliding window method was used to compile the RNN's forecasts into forecast plots. For each example, the forecast plot is shown below the original time series. Note that the forecast plots perfectly match the original time series, indicating that the RNN has successfully modeled seasonal patterns with a period of 100 time points.

4.3.2 Homogeneous Frequencies, Heterogeneous Phases, Confounding Features

During the second set of binary dataset experiments, the same RNN architecture was trained and tested on Binary Dataset 2 (see Figure 6 for examples), which is similar to Binary Dataset 1 except for that it includes noise, which acts as a collection of confounding features in the input time series. After training, the RNN had an accuracy, sensitivity, specificity, and precision of 0.99. Figure 14 shows forecast plots for three example time series. The seasonally occurring "1"s are highlighted in red to draw attention to the fact that the RNN has successfully forecasted every one. This indicates that the RNN has successfully encoded the 100 time point period in the seasonal pattern, despite the presence of confounding features in the time series. However, the RNN also

outputted a number of false positives, or "1"s that are not a part of the seasonal pattern, suggesting that the presence of confounding features in the input has caused the RNN to encode some sort of additional, useless prediction scheme.

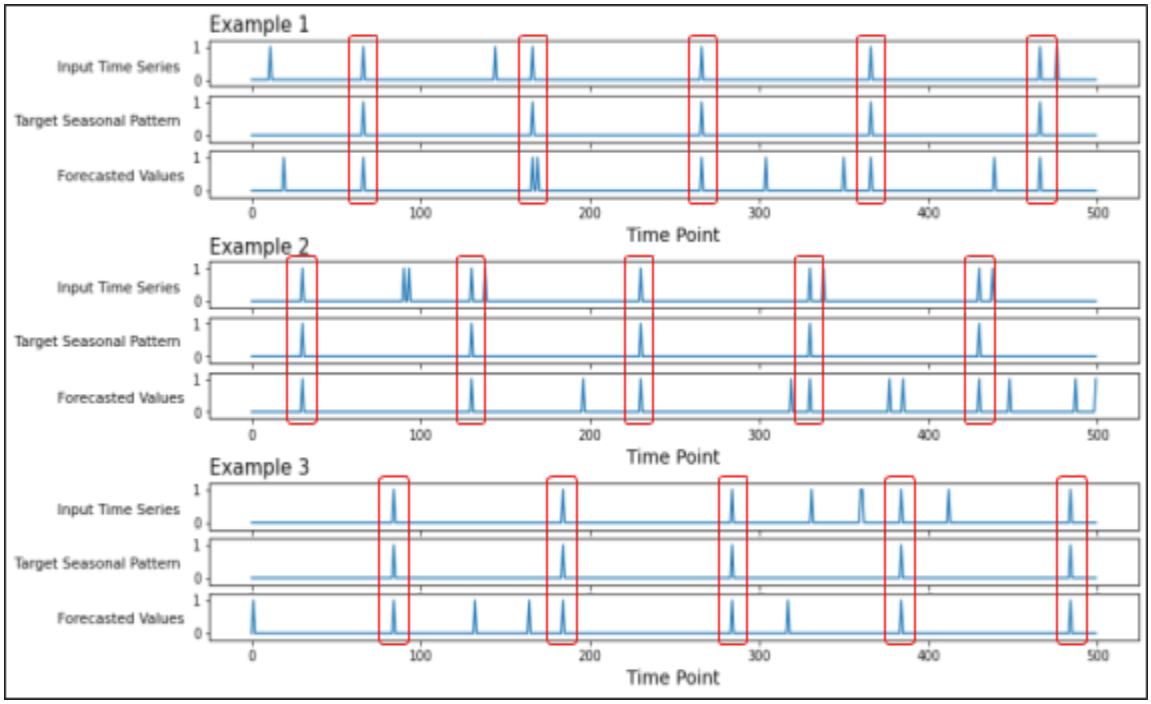


Figure 14. Forecast plots for three test examples taken from Binary Dataset 2, which is identical to Binary Dataset 1 except for that it includes noise. The sliding window method was used to compile the RNN's forecasts into forecast plots. For each example, the original time series is shown in the top plot, the target seasonal pattern (the pattern that the RNN is trying to predict) is shown in the middle plot, and the actual forecasts outputted by the RNN are shown in the bottom plot. Highlighted in red are the seasonal patterns of "1"s. Note that the forecast plots perfectly predict the seasonal pattern, indicating that the RNN has successfully modeled a seasonal pattern with a period of 100 time points. However, there are also many false positives in the forecast plots.

4.3.3 Heterogeneous Frequencies, Heterogeneous Phases

During the third set of binary dataset experiments, RNNs were trained to forecast the time series examples in Binary Dataset 3 (see Figure 7 for examples), whose seasonal

components have heterogeneous frequencies and phases. The seasonal frequencies were randomly sampled from a discrete uniform distribution, $period = unif(2, 50)$. The RNN had 1 LSTM layer with 10 cells.

After many attempts at training, a model was generated with a perfect accuracy score. This model was used to generate forecast plots from time series with varying seasonal frequencies and phases. Figure 15 shows forecast plots for seven example time series with seasonal periods of 10, 30, 50, 70, 100, 150, and 200. The figure shows in red the number of time points that elapse between when the RNN saw a "1" in the input and when it predicted that the subsequent "1" would occur. This quantity of elapsed time points is equivalent to the seasonal period that was predicted by the RNN. This predicted period stayed consistent over the duration of each forecast plot.

The forecast plots are perfect over a wide range of periods, indicating that the model adapted to the seasonal period of the input time series. Notably, the RNN even successfully forecasted time series with seasonal periods that exceeded the range of periods in the training data. For example, when the RNN was given an input time series with a seasonal period of 90 time points, it successfully forecasted a period of 90 time points, even though the training data only contained time series with a maximum seasonal period of 50 time points. This means that the RNN has encoded a model for seasonality that generalizes beyond the scope of the training data.

However, the model's flexibility did have its limits. When the input's seasonal period was 100 time points, the predicted period fell slightly short with 99 time points. As the input's period was increased further, the predicted period could not keep pace and

eventually reached an upper limit of 232 time points. Comparing the forecast plots for the input time series with periods of 300 and 400, it can be seen that the predicted period remains at 231 time points even though the actual period has increased by 100 time points.

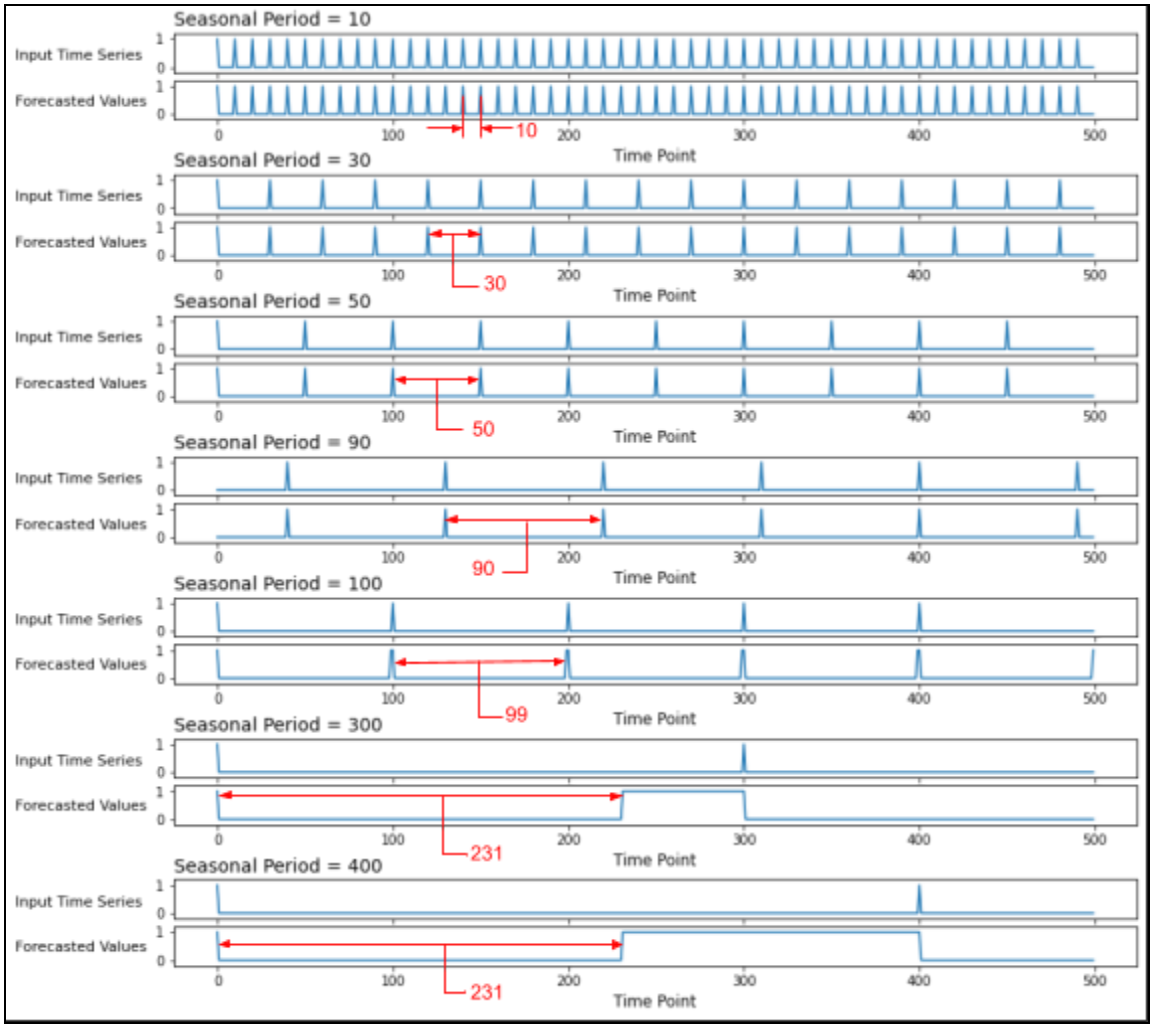


Figure 15. Forecast plots for seven input time series with heterogeneous phases and with varying seasonal frequencies of 10, 30, 50, 90, 100, 300, and 400. The sliding window method was used to compile the RNN's forecasts into forecast plots. For each example, the original time series is shown in the upper plot, and the RNN's forecasts are shown in the lower plot. Highlighted in red is the "predicted period", or the seasonal period as predicted by the RNN.

4.3.4 Heterogeneous Frequencies, Heterogeneous Phases, Confounding Features

The last set of binary dataset experiments again tested the ability of RNNs to model heterogeneous seasonal frequencies and phases, but this time with the addition of noise, which acts as a collection of confounding features in the input time series. Binary Dataset 4 (see Figure 8 for examples) was used to train and test RNNs with 1 and 2 layers of LSTM cells, with 10 cells per layer.

Despite many attempts to train a model to make accurate predictions, the best performing model only had an accuracy of 71%. The model's sensitivity score was 62% and its specificity score was 78%, meaning that it was outputting both false positives and false negatives. Figure 16 shows forecast plots that were generated using the best model. As can be seen, the predicted periods are erratic. Clearly, the addition of confounding noise to a seasonally heterogeneous dataset makes it far more difficult for an RNN to model and forecast the dataset's seasonal components.

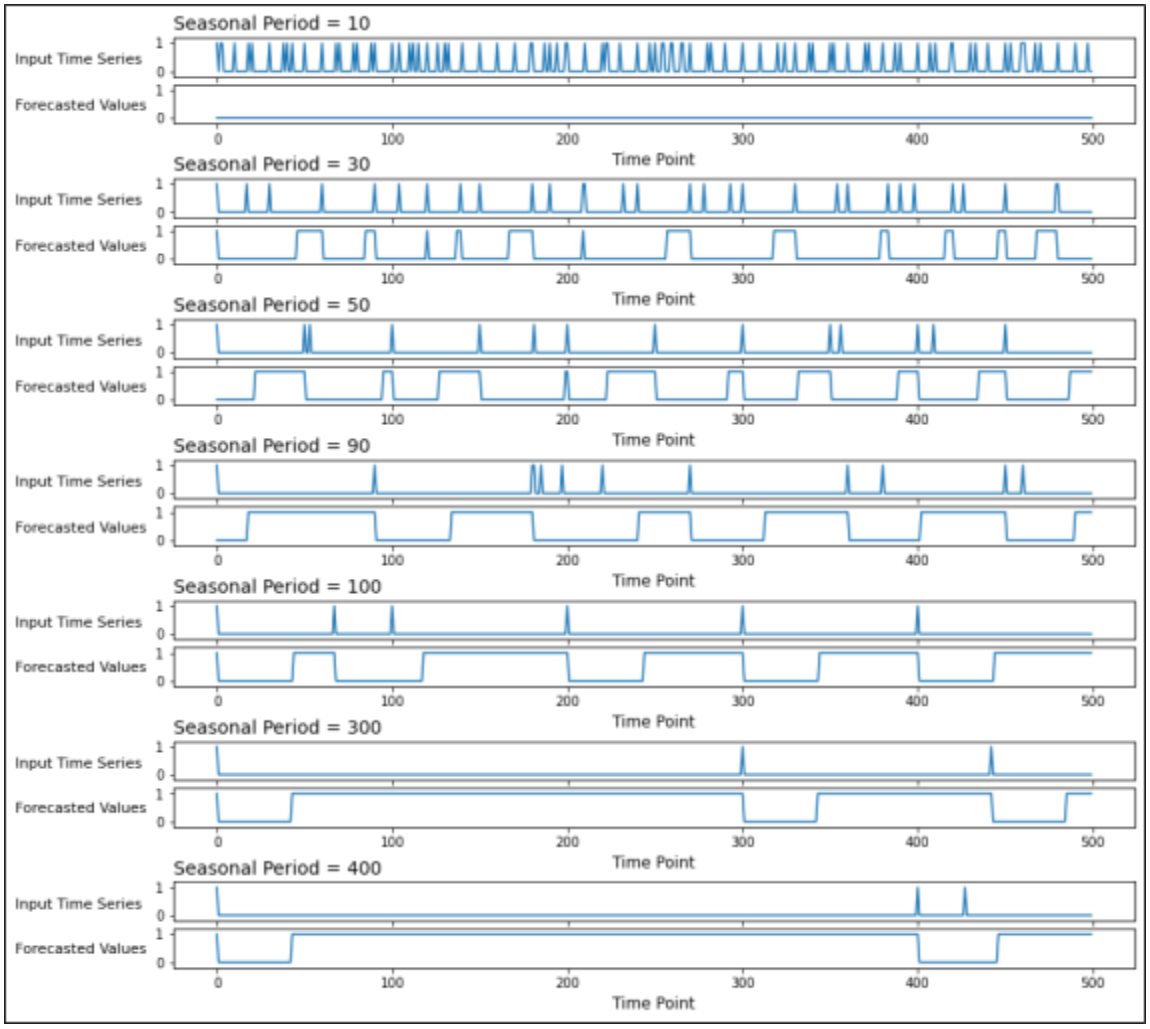


Figure 16. Forecast plots for seven input time series with heterogeneous phases and with varying seasonal frequencies of 10, 30, 50, 90, 100, 300, and 400. The time series contain noise. The sliding window method was used to compile the RNN's forecasts into forecast plots. For each example, the original time series is shown in the top plot, and the RNN's forecasts are shown in the bottom plot.

Chapter V.

Summary and Conclusion

After being trained to classify the defrost status of time series in the refrigeration temperature simulation dataset, an RNN with 2 layers of 10 LSTM cells was able to achieve 95% accuracy on the simulated test set. Unfortunately, it was impossible to measure the RNN's accuracy on real refrigeration units' temperature time series, since class labels are not available. However, classification plots made it possible to visually inspect the RNN's performance on real data. Multiple plots showed false positives at time points that did not make sense, given the seasonality of the defrost cycles. These mistakes called into question the ability of RNNs to model heterogeneous seasonality. This question was explored in the binary dataset experiments.

During the Binary Dataset 1 experiment, the trained RNNs had a 100% forecasting accuracy rate when modeling a dataset with homogeneous seasonal periods and heterogeneous seasonal phases. However, the accuracy dropped to 99% during the Binary Dataset 2 experiments, in which the input time series includes confounding features that are indistinguishable from the seasonal features. The 1% error rate was entirely caused by false positives. The empirical nature of this thesis project makes it difficult to know for sure that these false positives are unavoidable. This question can be revisited in future research.

During the Binary Dataset 3 experiments, the trained RNN had a 100% forecasting accuracy rate when modeling a dataset whose seasonal components have both heterogeneous frequencies and heterogeneous phases. This result suggests that RNNs are capable of modeling datasets with heterogeneous seasonal components, and contradicts the suggestions made in previous research that RNNs can only reliably model homogeneous datasets [11,17,19]. However, during the Binary Dataset 4 experiments, the accuracy dropped to 71%, suggesting that when confounding features are present in a dataset, it cannot model heterogeneous seasonal components. Therefore, an RNN can only be relied upon to model a heterogeneous dataset if it does not contain confounding features.

The results from the binary dataset experiments provide insights that can be used to explain the errors that were made during the experiments involving the refrigeration temperature simulation dataset. Since the simulation dataset is seasonally heterogeneous and includes confounding features (counterfeit defrosts), it is very unlikely that an RNN could learn a model to predict the seasonal components in the individual time series. This is why the RNN was prone to outputting false positives at inappropriate time points, since it never had the ability to fully discern the proper seasonal intervals.

Because refrigeration temperature datasets are seasonally heterogeneous and contain a variety of confounding features, RNNs are not the correct tool to use to predict the presence of defrosts, unless a high false positive rate is acceptable. As an alternative, a future research project might explore the possibility of using convolutional neural networks to detect defrosts.

References

1. Cleveland, R. B., Cleveland, W. S., McRae, J. E., & Terpenning, I. (1990). STL: A seasonal-trend decomposition. *Journal of official statistics*, *6*(1), 3-73.
2. Hyndman, R., Koehler, A. B., Ord, J. K., & Snyder, R. D. (2008). *Forecasting with exponential smoothing: the state space approach*. Springer Science & Business Media.
3. Box, G. E., Jenkins, G. M., Reinsel, G. C., & Ljung, G. M. (2015). *Time series analysis: forecasting and control*. John Wiley & Sons.
4. Bagnall, A., Lines, J., Bostrom, A., Large, J., & Keogh, E. (2017). The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data Mining and Knowledge Discovery*, *31*(3), 606-660.
5. Lines, J., & Bagnall, A. (2015). Time series classification with ensembles of elastic distance measures. *Data Mining and Knowledge Discovery*, *29*(3), 565-592.
6. Baydogan, M. G., Runger, G., & Tuv, E. (2013). A bag-of-features framework to classify time series. *IEEE transactions on pattern analysis and machine intelligence*, *35*(11), 2796-2802.
7. Bostrom, A., & Bagnall, A. (2017). Binary shapelet transform for multiclass time series classification. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXXII* (pp. 24-46). Springer, Berlin, Heidelberg.

8. Kulkarni, K., Devi, U., Sirighee, A., Hazra, J., & Rao, P. (2018, June). Predictive maintenance for supermarket refrigeration systems using only case temperature data. In *2018 Annual American Control Conference (ACC)* (pp. 4640-4645). IEEE.
9. Fawaz, H. I., Forestier, G., Weber, J., Idoumghar, L., & Muller, P. A. (2019). Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*, *33*(4), 917-963.
10. Borovykh, A., Bohte, S., & Oosterlee, C. W. (2017). Conditional time series forecasting with convolutional neural networks. *arXiv preprint arXiv:1703.04691*.
11. Hewamalage, H., Bergmeir, C., & Bandara, K. (2019). Recurrent neural networks for time series forecasting: Current status and future directions. *arXiv preprint arXiv:1909.00590*.
12. Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735-1780.
13. Bandara, K., Bergmeir, C., & Smyl, S. (2020). Forecasting across time series databases using recurrent neural networks on groups of similar series: A clustering approach. *Expert Systems with Applications*, *140*, 112896.
14. Malhotra, P., Vig, L., Shroff, G., & Agarwal, P. (2015, April). Long short term memory networks for anomaly detection in time series. In *Proceedings* (Vol. 89). Presses universitaires de Louvain.

15. Nelson, M., Hill, T., Remus, W., & O'Connor, M. (1999). Time series forecasting using neural networks: Should the data be deseasonalized first?. *Journal of forecasting*, *18*(5), 359-367.
16. Zhang, G. P., & Qi, M. (2005). Neural network forecasting for seasonal and trend time series. *European journal of operational research*, *160*(2), 501-514.
17. Bandara, K., Bergmeir, C., & Hewamalage, H. (2019). LSTM-MSNet: Leveraging Forecasts on Sets of Related Time Series with Multiple Seasonal Patterns. *arXiv preprint arXiv:1909.04293*.
18. Smyl, S. (2020). A hybrid method of exponential smoothing and recurrent neural networks for time series forecasting. *International Journal of Forecasting*, *36*(1), 75-85.
19. Lai, G., Chang, W. C., Yang, Y., & Liu, H. (2018, June). Modeling long-and short-term temporal patterns with deep neural networks. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval* (pp. 95-104).
20. Laptev, N., Yosinski, J., Li, L. E., & Smyl, S. (2017, August). Time-series extreme event forecasting with neural networks at uber. In *International Conference on Machine Learning* (Vol. 34, pp. 1-5).
21. Gers, F. A., Schmidhuber, J., & Cummins, F. (1999). Learning to forget: Continual prediction with LSTM.
22. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

23. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... & Ghemawat, S. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467.

24. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., & Shelhamer, E. (2014). cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759.

Appendix:

Project Code

Appendix: Project Code

May 12, 2020

```
In [0]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import pickle
        import random
        %tensorflow_version 1.x
        from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import Dense, CuDNNLSTM
        from tensorflow.keras.models import load_model
        from tensorflow.keras.optimizers import Adam
        from tensorflow.keras.callbacks import EarlyStopping

        # Import all functions that are needed to create the simulated refrigeration
        # temperature dataset and the toy datasets.
        from Refrigeration_Unit_Simulation import *
        from Generate_Binary_Datasets import *
```

1 Phase 1: Defrost Detection

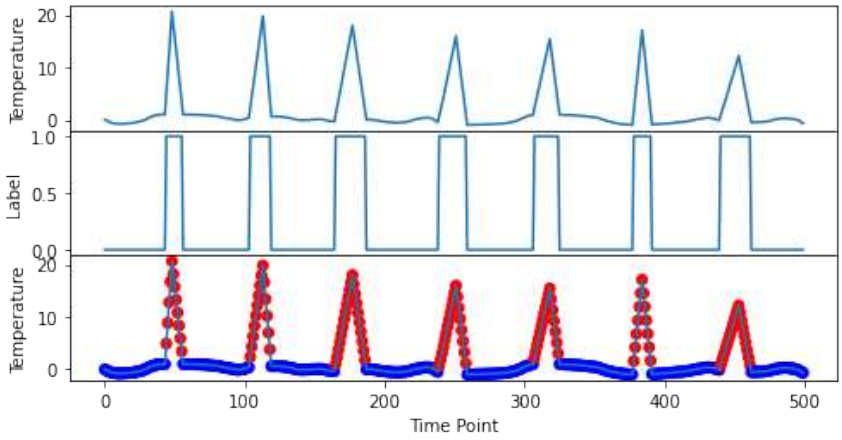
We start by building a dataset of simulated refrigeration temperature data.

```
In [0]: # X is the temperature data, Y is the defrost label data.
        X,Y,_ = simulate_refrigerator_data(num_simulations=22000, time_series_len=500, \
                                           x0y0_ratio=0.50, x1y1_ratio=0.50, \
                                           x1y0_ratio=0)
```

Let us plot the first example in the dataset to see how the temperature data and defrost labels are related. This is the code for Figure 3.

```
In [9]: fig, axs = plt.subplots(3, sharex=True, sharey=False, \
                                 gridspec_kw={'hspace': 0}, figsize=(8,4))
        fig.add_subplot(111, frameon=False)
        plt.tick_params(labelcolor='none', top=False, bottom=False, left=False, \
                        right=False)
        plt.xlabel('Time Point', fontsize=10)
        axs[0].plot(X[0])
        axs[0].set(ylabel='Temperature')
```

```
axs[1].plot(Y[0])
axs[1].set(ylabel='Label')
colormap = np.array(['b','r'])
axs[2].scatter(np.arange(len(X[0])), X[0], c=colormap[Y[0].astype(int)])
axs[2].plot(X[0])
axs[2].set(ylabel='Temperature')
for ax in axs:
    ax.label_outer()
plt.show()
```



Next we preprocess the data, which included splitting the dataset into training, validation, and test sets. Then we build the recurrent neural network and train it with the training data. Finally, we print out the confusion matrix and the performance metrics. Note that the results shown here do not match any of the results described in the thesis document, since this is not one of the training attempts that made it into the selection.

```
In [21]: # Reduce Y to its last time points, since these are what the network will be
         # trained to predict.
         Y = Y[:,-1]

         # Create training, validation, and test sets
         N = 20000 # Size of training set
         x_train, x_val, x_test = X[0:N,:], X[N:(N+1000),:], X[(N+1000):,:]
         y_train, y_val, y_test = Y[0:N], Y[N:(N+1000)], Y[(N+1000):]

         x_train = np.reshape(x_train, (x_train.shape[0],x_train.shape[1],1))
         x_val = np.reshape(x_val, (x_val.shape[0],x_val.shape[1],1))
         x_test = np.reshape(x_test, (x_test.shape[0],x_test.shape[1],1))
```

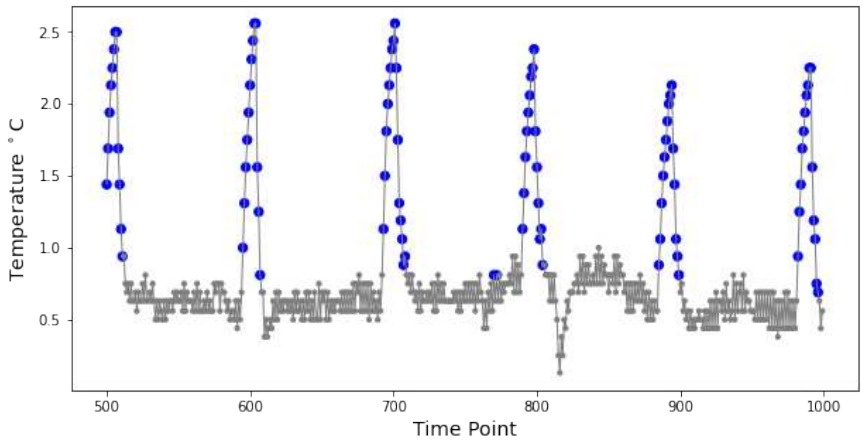
```
        # Build Recurrent Neural Network
        model = Sequential()
        model.add(CuDNNLSTM(10, return_sequences=True))
        model.add(CuDNNLSTM(10))
        model.add(Dense(1, activation='sigmoid'))
        model.compile(loss='binary_crossentropy', optimizer='adam', \
                      metrics=['accuracy'])

        # Train the model
        history = model.fit(x_train, y_train, batch_size=8, epochs=30, verbose=0, \
                            validation_data=(x_val, y_val), shuffle=True)

        model = load_model('Models/model_RNN_CuDNN_2Layer_10Neuron_SimData_50_50_\
        NoNoise_15epoch_20K_TrainData.h5')

        # Print the Performance Metrics
        from sklearn.metrics import confusion_matrix
        predictions = model.predict_classes(x_test)
        cf = confusion_matrix(y_test, np.reshape(predictions, (len(predictions),)))
        print(cf)
        print('sensitivity: ', cf[1][1]/(cf[1][1]+cf[1][0]))
        print('specificity: ', cf[0][0]/(cf[0][0]+cf[0][1]))
        print('precision:   ', cf[1][1]/(cf[1][1]+cf[0][1]))
```

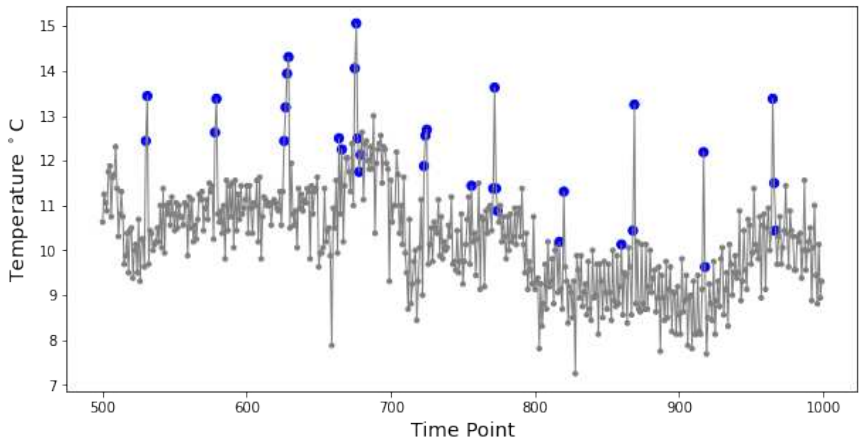
```
[[386 177]
 [ 29 408]]
sensitivity:  0.9336384439359268
specificity:  0.6856127886323268
precision:    0.6974358974358974
```

Let us examine the performance of the trained model using classification plots. We will use real refrigeration temperature data. This is code for Figures 9 and 10.

```
In [8]: file = open('train_data.pickle', 'rb')
        data = pickle.load(file)
        file.close()

        model = load_model('Models/model_RNN_CuDNN_2Layer_10Neuron_SimData_50_50_\
        NoNoise_15epoch_20K_TrainData.h5')

        t = data[52][3000:4000]
        test_predictions = predict_consecutive_readings(readings=t, look_back=500, \
                                                        model=model)
        test_predictions = np.reshape(test_predictions, (500))
        plt.figure(figsize = (10,5))
        plt.plot(np.arange(500,1000), t[499:999], c='tab:gray', linewidth=1)
        colormap = np.array(['tab:gray','b'])
```

```
sizemap = np.array([10,40])
plt.scatter(np.arange(500,1000), t[499:999], \
            c=colormap[test_predictions.astype(int)], \
            s=sizemap[test_predictions.astype(int)])
plt.xlabel('Time Point', fontsize=14)
plt.ylabel('Temperature $^\circ$C', fontsize=14)
plt.show()

t = data[63][3000:4000]
test_predictions = predict_consecutive_readings(readings=t, look_back=500, \
                                                model=model)
test_predictions = np.reshape(test_predictions, (500))
plt.figure(figsize = (10,5))
plt.plot(np.arange(500,1000), t[499:999], c='tab:gray', linewidth=1)
colormap = np.array(['tab:gray','b'])
sizemap = np.array([10,40])
plt.scatter(np.arange(500,1000), t[499:999], \
            c=colormap[test_predictions.astype(int)], \
            s=sizemap[test_predictions.astype(int)])
plt.xlabel('Time Point', fontsize=14)
plt.ylabel('Temperature $^\circ$C', fontsize=14)
plt.show()
```





1.1 Attempt to Remediate Poor Modeling of Seasonality.

So far, all trained models have failed to demonstrate that they are effectively using seasonality in their classification decisions. Therefore, we now build two additional datasets, each of which feature "counterfeit defrosts". The presence of these confounding features will hopefully force the RNN to learn to leverage seasonality in its classification decisions.

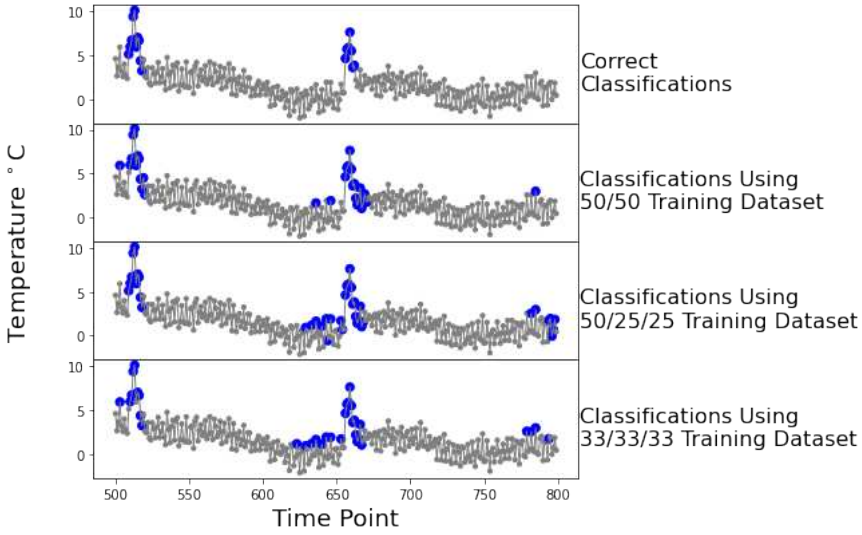
```
In [0]: X2,Y2,_ = simulate_refrigerator_data(num_simulations=22000, \
                    time_series_len=500, x0y0_ratio=0.50, x1y1_ratio=0.50, x1y0_ratio=0)
        X3,Y3,_ = simulate_refrigerator_data(num_simulations=22000, \
                    time_series_len=500, x0y0_ratio=0.50, x1y1_ratio=0.50, x1y0_ratio=0)
```

We now use the previous code to train two additional models. After the two models are trained, we load all three models and compare their classification performances on a simulated time series. This is code for Figure 12.

```
In [22]: model1 = load_model('Models/model_RNN_CuDNN_2Layer_10Neuron_SimData_50_50_\
         NoNoise_15epoch_20K_TrainData.h5')
         model2 = load_model('Models/model_RNN_CuDNN_2Layer_10Neuron_SimData_50_25_25_\
         NoNoise_15epoch_20K_TrainData.h5')
         model3 = load_model('Models/model_RNN_CuDNN_2Layer_10Neuron_SimData_33_33_33_\
         NoNoise_15epoch_20K_TrainData.h5')

         np.random.seed(886); random.seed(886)
         n=1000
         t, y, i = generate_simulation(n=n, end_in_defrost=False, \
                                       end_in_fake_defrost=False)
         plot_len = 300
```

```
look_back = 500
plot_start = look_back-1
plot_end = plot_start + plot_len
test_predictions1 = np.reshape(predict_consecutive_readings(readings=t, \
                  look_back=look_back, model=model1), (n-look_back))[:plot_len]
test_predictions2 = np.reshape(predict_consecutive_readings(readings=t, \
                  look_back=look_back, model=model2), (n-look_back))[:plot_len]
test_predictions3 = np.reshape(predict_consecutive_readings(readings=t, \
                  look_back=look_back, model=model3), (n-look_back))[:plot_len]
colormap = np.array(['tab:gray','b'])
sizemap = np.array([10,40])

fig, axs = plt.subplots(4, sharex=True, sharey=False, \
                        gridspec_kw={'hspace': 0}, figsize=(6.5,6.5))
fig.add_subplot(111, frameon=False)
plt.tick_params(labelcolor='none', top=False, bottom=False, \
                left=False, right=False)
plt.xlabel('Time Point', fontsize=18)
plt.ylabel('Temperature $^\circ$C', labelpad=20, fontsize=18)
axs[0].plot(np.arange(look_back,look_back+plot_len), t[plot_start:plot_end], \
            c='tab:gray', linewidth=1)
axs[0].scatter(np.arange(look_back,look_back+plot_len), t[plot_start:plot_end], \
               c=colormap[y[plot_start:plot_end].astype(int)], \
               s=sizemap[y[plot_start:plot_end].astype(int)])
axs[0].text(x=815, y=1, s='Correct\nClassifications', fontsize=16)
axs[1].plot(np.arange(look_back,look_back+plot_len), t[plot_start:plot_end], \
            c='tab:gray', linewidth=1)
axs[1].scatter(np.arange(look_back,look_back+plot_len), t[plot_start:plot_end], \
               c=colormap[test_predictions1.astype(int)], \
               s=sizemap[test_predictions1.astype(int)])
axs[1].text(x=815, y=1, s='Classifications Using\n50/50 Training Dataset', \
            fontsize=16)
axs[2].plot(np.arange(look_back,look_back+plot_len), t[plot_start:plot_end], \
            c='tab:gray', linewidth=1)
axs[2].scatter(np.arange(look_back,look_back+plot_len), t[plot_start:plot_end], \
               c=colormap[test_predictions2.astype(int)], \
               s=sizemap[test_predictions2.astype(int)])
axs[2].text(x=815, y=1, s='Classifications Using\n50/25/25 Training Dataset', \
            fontsize=16)
axs[3].plot(np.arange(look_back,look_back+plot_len), t[plot_start:plot_end], \
            c='tab:gray', linewidth=1)
axs[3].scatter(np.arange(look_back,look_back+plot_len), t[plot_start:plot_end], \
               c=colormap[test_predictions3.astype(int)], \
               s=sizemap[test_predictions3.astype(int)])
axs[3].text(x=815, y=1, s='Classifications Using\n33/33/33 Training Dataset', \
            fontsize=16)
for ax in axs:
    ax.label_outer()
```

```
plt.show()
```



2 Phase 2: Toy Dataset Experiments

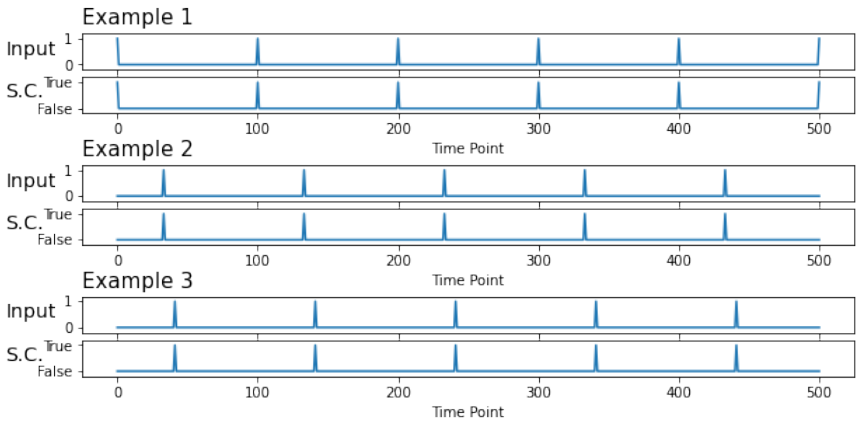
2.1 Toy Dataset 1

We start by creating Toy Dataset 1, whose seasonal components have homogenous periods and heterogeneous temporal positions.

```
In [0]: num_runs = 16000
        N = 15000 # Size of training set
        len_run=501

        num_x0y0 = int(num_runs/2)
        num_x1y1 = int(num_runs/2)
        num_x1y0 = int(num_runs*0)

        X,Y,P = generate_binary_training_data(len_run=len_run, noise=False, period=100, \
                                    period_range=None, start=None, num_x0y0=num_x0y0, \
                                            num_x1y1=num_x1y1, num_x1y0=num_x1y0)
```

In order to better understand the dataset, we plot a few examples from the dataset and their seasonal components. This is code for Figure 5.

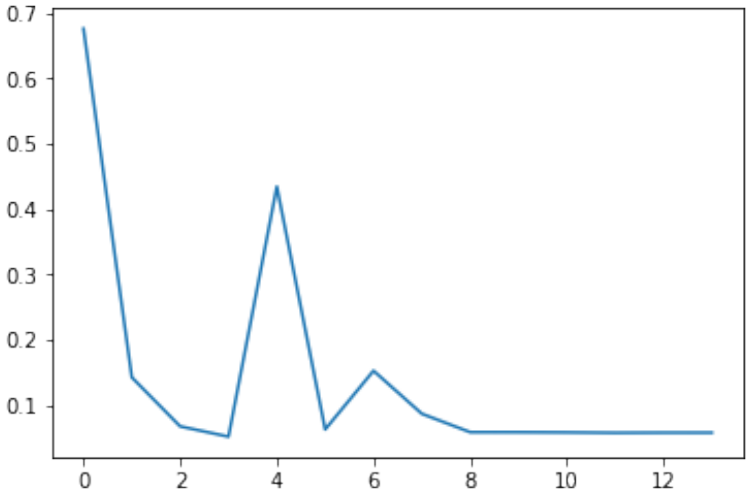
```
In [11]: fig = plt.figure(figsize = (4.5,4.5), frameon=False, linewidth=10, edgecolor='b')
         for i in range(3):
           #X,Y,p = generate_binary_run(len_run=500, noise=True, period=100, start=None)
           subplot_loc = 3*i+1
           axs = fig.add_subplot(8, 1, subplot_loc)
           axs.plot(X[i])
           axs.set_yticks(ticks=np.array([0,1]))
           axs.set_ylim([-.2,1.2])
           axs.text(x=-80, y=0.35, s='Input', fontsize=14)
           axs.set_title('Example {i}'.format(i=(i+1)), loc='left', fontsize=15)
           axs = fig.add_subplot(8, 1, subplot_loc+1)
           axs.plot(Y[i])
           axs.set_yticks(ticks=np.array([0,1]))
           axs.set_ylim([-.2,1.2])
           axs.set_yticklabels(np.array(['False', 'True']))
           axs.text(x=-80, y=0.4, s='S.C.', fontsize=14)
           axs.set_xlabel('Time Point')
         fig.subplots_adjust(left=0.3, right=2)
         plt.show()
```



We split Toy Dataset 1 into training and test sets. We use the training set to train an RNN, and then use the test set to generate performance metrics. Note that these metrics are slightly different from the results shown in the body of the Thesis, since those results were acheived after many attempts at training.

```
In [24]: X = np.reshape(X, (X.shape[0],X.shape[1],1))
         x_train, x_test = X[:N,:-1], X[N:,:-1]
         y_train, y_test = X[:N,-1,0], X[N:,-1,0]

         model_RNN = Sequential()
```

```
model_RNN.add(CuDNNLSTM(10))
model_RNN.add(Dense(1, activation='sigmoid'))
optimizer = Adam(learning_rate=0.01)
model_RNN.compile(loss='binary_crossentropy', optimizer=optimizer, \
                  metrics=['accuracy'])
callback = EarlyStopping(monitor='val_acc', restore_best_weights=True, \
                         patience=10)
history = model_RNN.fit(x_train, y_train, batch_size=64, epochs=30, verbose=0, \
                        validation_data=(x_test, y_test), shuffle=True,\
                        callbacks = [callback])

plt.plot(history.history["val_loss"]); plt.show()

from sklearn.metrics import confusion_matrix
tt = model_RNN.predict_classes(x_test)
cf = confusion_matrix(y_test, np.reshape(tt, (len(tt),)))
print(cf)
print('sensitivity: ', cf[1][1]/(cf[1][1]+cf[1][0]))
print('specificity: ', cf[0][0]/(cf[0][0]+cf[0][1]))
print('precision:   ', cf[1][1]/(cf[1][1]+cf[0][1]))
```

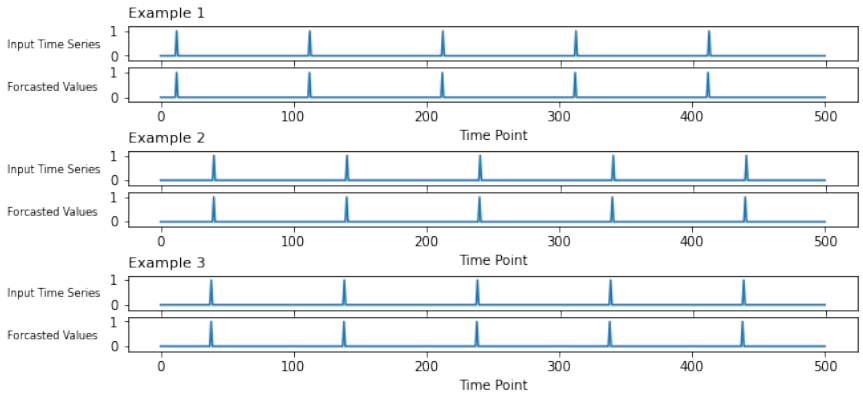



```
[[493  12]
 [  0 495]]
sensitivity:  1.0
specificity:  0.9762376237623762
```

```
precision:    0.9763313609467456
```

We visualize the RNN's forecasting performance by generating three new time series and creating forecasting plots. The plots demonstrate that the forecasts are accurate. This is code for Figure 13.

```
In [16]: fig = plt.figure(figsize = (4.5,4.5), frameon=False, linewidth=10, \
                          edgecolor='b')
         for i in range(3):
           x, y, p = generate_binary_training_data(len_run=(1000), period=100, \
                                                   start=None, num_x0y0=1, noise=False)
           x = np.ravel(x)
           y_hat = predict_consecutive_binary_readings(readings=x, look_back=499, \
                                                       model=model)
           subplot_loc = 3*i+1
           axs = fig.add_subplot(8, 1, subplot_loc)
           axs.plot(x[499:999])
           axs.set_yticks(ticks=np.array([0,1]))
           axs.set_ylim([-.2,1.2])
           axs.text(x=-115, y=0.3, s='Input Time Series', fontsize=8)
           axs.set_title('Example {i}'.format(i=(i+1)), loc='left', fontsize=11)
           axs = fig.add_subplot(8, 1, subplot_loc+1)
           axs.plot(np.arange(0,len(y_hat)),y_hat)
           axs.set_yticks(ticks=np.array([0,1]))
           axs.set_ylim([-.2,1.2])
           axs.text(x=-115, y=0.25, s='Forcasted Values', fontsize=8)
           axs.set_xlabel('Time Point', fontsize=10)
         fig.subplots_adjust(left=0.3, right=2)
         plt.show()
```



We can now tweek the above cells to create, train with, and test Toy Datasets 2, 3, and 4. When building Toy Datasets 2 and 4, we set 'noise' to True. When building Toy Datasets 3 and 4, we set 'period' to None and we set 'period_range' to (2,100).

In [0]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import pickle
import random


def add_gaussian_noise(n, scale=1, start=0, bottom=float("-inf"),
    top=float("inf")):
    """Generate a numpy array populated with Gaussian noise values.

    Parameters:
    n (int): Number of time points
    scale (float): The standard deviation of the Gaussian distribustion from
        which the noise is sampled.
    start (int): The time point at which the noise begins.  Previous time
        points are set to 0.
    bottom (float): The lower limit of the noise values.
    top (float): The upper limit of the noise values.
    """

    t = np.zeros(n)
    for i in range(start,n):
        t[i] = min(max(np.random.normal(loc=0, scale=scale),bottom),top)
    return np.array(t)

def add_sine_wave(n, period):
    """Generate a numpy array of length 'n', populated with values from a sine
        wave."""

    start = 2*np.pi*random.random()
    return np.array([np.sin(start + 2*np.pi*i/(period-1)) for i in range(n)])

def add_chaotic_drift(n, scale=0.1):
    """Generate a numpy array of length 'n' whose values drift.

    The amount that each value drifts from the previous value is sampled from a
        continuous uniform distribution
    with a min of -0.5*scale and a max of 0.5*scale.
    """

    t = [(random.random()-0.5)*scale for i in range(n)]
    for i in range(1,len(t)):
        t[i] = t[i-1] + t[i]
    return np.array(t)

def add_anchored_drift(n, top=1, bottom=-1):
    """Generate a numpy array of length 'n' whose values drift but stay within
        the range 'top' to 'bottom'."""

    t = np.zeros(n)
    delta = np.random.normal(loc=0, scale=0.35)
    for i in range(1,n):
        sign_ = 1 if t[i-1]>=0 else -1
        move_away_probability = 0.5 - sign_*t[i-1]/2
        scale = max(1/50, 0.7*move_away_probability)
```

```
            distance = abs(np.random.normal(loc=0, scale=scale))
            move_away = 1 if np.random.binomial(n=1,p=move_away_probability) else
                -1
            delta = 0.9 * delta + (1-0.9) * sign_ * distance * move_away
            t[i] = t[i-1] + delta
            if t[i] > top or t[i] < bottom:
                t[i] = t[i-1]
        return t

    def add_random_shifts(n):
        """Generate a numpy array of length 'n' whose values shift up and down
            randomly."""

        t = np.zeros(n)
        i=1
        while i < n:
            if np.random.binomial(n=1, p=0.005):
                duration = min((n-i), int(np.random.gamma(1.5,4)))
                direction = (1 if (t[i-1]<=0) else -1) * (1 if np.random.binomial(n
                    =1, p=0.8) else -1)
                height = np.random.gamma(3,0.7)
                for j in range(duration):
                    t[i+j] = t[i-1] + height * j/duration * direction
                i+=duration
            else:
                t[i] = t[i-1]
                i+=1
        return t

    def add_compressor(n, height):
        """Generate a numpy array of length 'n' whose values simulate a compressor
            cycle.

        The array values move up and down cyclically.  The period of each cycle can
            change slightly over time,
        but the 'height' remains constant.
        """

        baseline = max(np.random.gamma(6,1), 0.5)
        num_cycles = round(n/baseline*1.5)
        drift = add_anchored_drift(num_cycles) * baseline/10
        period = np.random.normal(288 / random.randint(1,3))
        cycle = add_sine_wave(n=num_cycles, period=period) * baseline/40
        intervals = [baseline+drift[i]+cycle[i] for i in range(num_cycles)]
        t = np.zeros(n)
        end_of_current_cycle = 0
        end_of_prev_cycle = 0
        next_cycle = 0
        for i in range(1, n):
            while i > end_of_current_cycle:
                end_of_prev_cycle = end_of_current_cycle
                end_of_current_cycle = end_of_current_cycle + intervals[next_cycle]
                next_cycle += 1
            halfway = end_of_prev_cycle + (end_of_current_cycle-end_of_prev_cycle)/
                2
            if i <= halfway:
```

```
                t[i] = height * (i - end_of_prev_cycle) / (halfway -
                    end_of_prev_cycle)
            else: t[i] = height * (end_of_current_cycle - i) /
                (end_of_current_cycle - halfway)
        return t

    def spawn_defrost(height, num_readings_up_average, num_readings_down_average):
        """Generate a numpy array whose values simulate the temperatures of a
            single defrost event."""

        num_readings_up = int(max(round(np.random.normal(loc=
            num_readings_up_average, scale=num_readings_up_average/5)),1))
        num_readings_down = int(max(round(np.random.normal(loc=
            num_readings_down_average, scale=num_readings_down_average/5)),1))
        t_up = [height/num_readings_up*i for i in range(1,num_readings_up+1)]
        t_down = [(height-height/num_readings_down*i) for i in range(1,
            num_readings_down)]
        t = t_up+t_down
        return np.array(t)



    def add_defrosts(n, height_baseline, anomalous=False, all_anomalous=True,
        end_in_defrost=False):
        """Generate a numpy array of length 'n' whose values simulate a defrost
            cycle.

        Each defrost event is simulated separately, which creates a small amount of
            variation between defrost events
        in the same array.
        Each defrost event's height and each interval between two defrost is
            determined by applying a small amount
        of random variation to a baseline value that remains constant throughout
            the array.
        A defrost event is called 'anomalous' when gaussian noise is added to its
            height and to the interval between
        it and the next defrost event.
        If anomalous is set to True, then setting all_anomalous to True makes all
            defrosts anomalous.
        If anomalous is set to True and all_anomalous is set to False, then only
            the second half of the defrosts
        will be anomalous.

        Parameters:
        n (int): Number of time points
        anomalous (bool): If True, make at least some of the defrosts are anomalous
            (inconsistent in height and period).
        all_anomalous (bool): If True, make all defrosts anomalous, rather than
            just the second half.
        end_in_defrost (bool): If True, a defrost will be occurring during the last
            time point in the array.
        """

        t = np.empty(shape=(0))
        labels = []
        interval_baseline = max(np.random.gamma(3,22), 12)
```

```
        anomaly_range = (0 if (not anomalous) else (n if all_anomalous else (int(n/
            2))))
        num_readings_up_average = round(min(max(np.random.gamma(1,5),1),
            interval_baseline/4))
        num_readings_down_average = round(min(max(np.random.gamma(1,5),1),
            interval_baseline/4))
        if end_in_defrost:
            height = height_baseline + add_gaussian_noise(1, bottom=-
                height_baseline/2, top=height_baseline*2, \
                                                          scale=(height_baseline/2
                                                          if anomalous else
                                                          height_baseline/5))
            defrost = spawn_defrost(height, num_readings_up_average,
                num_readings_down_average)
            cutoff_defrost = defrost[random.randint(0,(len(defrost)-1)):]
            t = np.append(t, cutoff_defrost)
            labels = labels + ([True]*len(cutoff_defrost))
            interval = int(interval_baseline + \
                           add_gaussian_noise(1, bottom=-interval_baseline, \
                                              scale=(interval_baseline/2 if
                                                  anomalous else interval_baseline/
                                                  20)))
            t = np.append(t, np.zeros(interval))
            labels = labels + ([False]*interval)
        else:
            cutoff_interval = random.randint(0,round(interval_baseline))
            t = np.append(t, np.zeros(cutoff_interval))
            labels = labels + ([False]*cutoff_interval)
        while len(t) < n:
            height_noise_scale = (height_baseline/2 if (len(t)<anomaly_range) else
                height_baseline/5)
            height = height_baseline + add_gaussian_noise(1, bottom=-
                height_baseline/2, top=height_baseline*2, \
                                                          scale=height_noise_scale)
            defrost = spawn_defrost(height, num_readings_up_average,
                num_readings_down_average)
            t = np.append(t, defrost)
            labels = labels + ([True]*len(defrost))
            interval_noise_scale = (interval_baseline/2 if anomalous else
                interval_baseline/20)
            interval = int(interval_baseline + add_gaussian_noise(1, bottom=-
                interval_baseline, scale=interval_noise_scale))
            t = np.append(t, np.zeros(interval))
            labels = labels + ([False]*interval)
        t = np.flip(t[0:n])
        labels = np.flip(labels[0:n])
        return t, labels, interval_baseline

    def add_random_events(n):
        """Generate a numpy array of length 'n' whose values simulate random spikes
            in a time series.

        The random spikes can go either up or down, and their heights fluctuate
            more than defrost heights.
        """
```

```
        num_events = int(abs(np.random.normal(loc=int(n/50), scale=int(n/50))))
        t = np.zeros(n)
        for i in range(num_events):
            height = np.random.gamma(1,1)
            num_readings_up = round(max(np.random.gamma(1.5,3),1))
            num_readings_down = round(max(np.random.gamma(1.5,3),1))
            defrost = spawn_defrost(height=height, num_readings_up=num_readings_up,
                num_readings_down=num_readings_down)
            event = [0]
            for up in range(num_readings_up):
                event = event + [event[up-1] + np.random.normal(loc=height/
                    num_readings_up, scale = height/num_readings_up)]
            for down in range(num_readings_down):
                event = event + [event[num_readings_up+down-1] + np.random.normal
                    (loc=height/num_readings_up, \

                                                        scale = height/
                                                        num_readings_up)]
            direction = (1 if np.random.binomial(n=1,p=0.5) else -1)
            event = event*direction
            location = random.randint(0,n-1-len(event))
            t[location:(location+len(event))] = event
        return np.array(t)

    def add_fake_defrosts(n, height_baseline):
        """Generate a numpy array of length 'n' whose values simulate counterfeit
            defrosts in a time series.

        Each counterfeit defrost's height is determined by applying a small amount
            of random variation to a
        baseline value, the 'height_baseline', which remains constant throughout
            the array.
        """

        num_defrosts = int(abs(np.random.normal(loc=int(n/200), scale=int(n/200))))
        t = np.zeros(n)
        for i in range(num_defrosts):
            height = np.random.normal(loc=height_baseline, scale=(height_baseline/5
                ))
            num_readings_up_average = round(max(np.random.gamma(1.5,2.5),1))
            num_readings_down_average = round(max(np.random.gamma(1.5,2.5),1))
            defrost = spawn_defrost(height, num_readings_up_average,
                num_readings_down_average)
            location = random.randint(0,n-1-len(defrost))
            t[location:(location+len(defrost))] = defrost
        return np.array(t)

    def add_fake_defrost_to_end(n, height_baseline):
        """Generate a numpy array of length 'n' that ends in a defrost.  All other
            values are zero"""

        t = np.zeros(n)
        height = np.random.normal(loc=height_baseline, scale=(height_baseline/5))
        num_readings_up_average = round(max(np.random.gamma(1.5,2.5),1))
        num_readings_down_average = round(max(np.random.gamma(1.5,2.5),1))
        defrost = spawn_defrost(height, num_readings_up_average,
```

```
            num_readings_down_average)
        cutoff = random.randint(1,len(defrost))
        t[-cutoff:] = defrost[:cutoff]
        return t

    def generate_simulation(n, defrosts=None, compressor=None, chaotic_drift=None,
        gaussian_noise=None, \
                            random_events=None, fake_defrosts=None, random_shifts=
                                None, anomalous=False, all_anomalous=True, \
                            end_in_defrost=False, end_in_fake_defrost=False,
                                chaotic_drift_scale=0.2, \
                            noise_scale=0.5, verbose=False):
        """Simulate a time series of refrigeration temperature readings.

        Simulate a refrigeration temperature time series by built by adding
            together component time series, each of
        which are simulated separately.  Component time series include a compressor
            cycle, a defrost cycle, drift,
        gaussian noise, random events, counterfeit defrosts, and random shifts.
            Each of these components is
        associated with a boolean parameter that controls whether or not the
            component is added to the time series.
        If any of these parameters are left as None (their default), then that
            component will have some random chance
        of being included.

        Parameters:
        n (int): Length of the time series.
        defrosts (bool or int): If True or 1, add a defrost cycle to the time
            series.
        compressor (bool or int): If True or 1, add a compressor cycle to the time
            series.
        chaotic_drift (bool or int): If True or 1, add drift to the time series.
        gaussian_noise (bool or int): If True or 1, add gaussian noise to the time
            series.
        random_events (bool or int): If True or 1, add random spikes up and down to
            the time series.
        fake_defrosts (bool or int): If True or 1, add counterfeit defrosts to the
            time series.
        random_shifts (bool or int): If True or 1, add random shifts up and down to
            the time series.
        anomalous (bool): If True, make at least some of the defrosts are anomalous
            (inconsistent in height and period).
        all_anomalous (bool): If True, make all defrosts anomalous, rather than
            just the second half.
        end_in_defrost (bool): If True, a defrost will be occurring during the last
            time point in the array.
        end_in_fake_defrost (bool): If True, a counterfeit defrost will be
            occurring during the last time point in the array.
        chaotic_drift_scale (float): The variance of the distribution from which
            the drift step sizes are sampled.
        noise_scale (float): The variance of the Gaussian distribution from which
            the noise is sampled.
        verbose (bool): If True, print the components that were added to the time
            series.
```

```
        Returns:
        simulated_run (numpy array): Simulated time series of temperature readings.
        labels (numpy array): Simulated time series of defrost labels.
        interval_baseline (float): Defrost periods' baseline value before random
            variance is added.
        """

        if (end_in_defrost and end_in_fake_defrost):
            raise ValueError('Cannot end in both a fake and real defrost')
        if defrosts == None:
            defrosts = np.random.binomial(n=1,p=0.9)
        if compressor == None:
            compressor = np.random.binomial(n=1,p=0.5)
        if chaotic_drift == None:
            chaotic_drift = np.random.binomial(n=1,p=0.5)
        if gaussian_noise == None:
            gaussian_noise = np.random.binomial(n=1,p=0.5)
        if random_events == None:
            gaussian_noise = np.random.binomial(n=1,p=0.5)
        if ((fake_defrosts == None) and defrosts):
            fake_defrosts = np.random.binomial(n=1,p=0.5)
        if random_shifts == None:
            random_shifts = np.random.binomial(n=1,p=0.5)

        simulated_run = add_anchored_drift(n)
        if chaotic_drift:
            simulated_run = simulated_run + add_chaotic_drift(n, scale=
                chaotic_drift_scale)
        if compressor:
            comp_cycle_height = max(np.random.gamma(1,2), 0.25)
        if gaussian_noise:
            random_noise_scale = abs(np.random.normal(loc=noise_scale, scale=
                noise_scale/2))
            if verbose: print('random_noise_scale is ', random_noise_scale)
            simulated_run = simulated_run + add_gaussian_noise(n, scale=
                random_noise_scale)
        if random_events:
            simulated_run = simulated_run + add_random_events(n)
        if random_shifts:
            simulated_run = simulated_run + add_random_shifts(n)

        height_baseline = 1.5 + np.random.gamma(1.5,3)
        if compressor: height_baseline = max(height_baseline, 2*comp_cycle_height)
        if gaussian_noise: height_baseline = max(height_baseline, 5*
            random_noise_scale)
        interval_baseline = 0
        if defrosts:
            d, labels, interval_baseline = add_defrosts(n, height_baseline=
                height_baseline, anomalous=anomalous, \
                                                        all_anomalous=all_anomalous
                                                          , end_in_defrost=
                                                          end_in_defrost)
            simulated_run = simulated_run + d
            if compressor:
                if np.random.binomial(n=1,p=0.5):
                    simulated_run = simulated_run + add_compressor(n, height=
```

```
                    comp_cycle_height)*np.invert(labels)
            else: simulated_run = simulated_run + add_compressor(n, height=
                comp_cycle_height)
    else:
        labels = np.array([0]*n)
        if compressor: simulated_run = simulated_run + add_compressor(n, height
            =comp_cycle_height)
    if fake_defrosts:
        fd = add_fake_defrosts(n=n, height_baseline=height_baseline)
        simulated_run = simulated_run + fd
    if end_in_fake_defrost:
        fd = add_fake_defrost_to_end(n=n, height_baseline=height_baseline)
        simulated_run = simulated_run + fd
    if verbose:
        print('chaotic_drift={}, compressor={}, gaussian_noise={},
            fake_defrosts={},\
        random_shifts={}'.format(chaotic_drift, compressor, gaussian_noise,
            fake_defrosts, random_shifts))
    return simulated_run, labels, interval_baseline

def simulate_refrigerator_data(num_simulations, time_series_len, x0y0_ratio=0.5
    , x1y1_ratio=0.4, x1y0_ratio=0.1):
  """Generate an entire dataset of simulated refrigeration temperature time
      series.

  Parameters:
  num_simulations (int): The number of simulated time series in the dataset.
  time_series_len (int): The length of each simulated time series.
  x0y0_ratio (float): The portion of simulated time series that will not end in
      a defrost or a counterfeit defrost.
  x1y1_ratio (float): The portion of simulated time series that will end in a
      defrost.
  x1y0_ratio (float): The portion of simulated time series that will end in a
      counterfeit defrost.

  Returns:
  X (numpy array): An array of simulated refrigeration temperature time series.
  Y (numpy array): An array of defrost label time series.
  P (numpy array): An array of the baseline defrost interval values used to
      build each time series.
  """

  num_x0y0 = int(num_simulations*x0y0_ratio)
  num_x1y1 = int(num_simulations*x1y1_ratio)
  num_x1y0 = int(num_simulations*x1y0_ratio)

  X=np.zeros(((num_x0y0+num_x1y1+num_x1y0), time_series_len))
  Y=np.zeros(((num_x0y0+num_x1y1+num_x1y0), time_series_len))
  p=np.zeros((num_x0y0+num_x1y1+num_x1y0))

  for i in range(num_x0y0):
    X[i,:], Y[i,:], p[i] = generate_simulation(n=time_series_len,
        end_in_defrost=False, end_in_fake_defrost=False)
  for i in range(num_x1y1):
    ind = num_x0y0+i
    X[(ind),:],Y[(ind),:],p[ind] = generate_simulation(n=time_series_len,
```

```
            end_in_defrost=True, end_in_fake_defrost=False)
        for i in range(num_x1y0):
          ind = num_x0y0+num_x1y1+i
          X[(ind),:],Y[(ind),:],p[ind] = generate_simulation(n=time_series_len,
            end_in_defrost=False, end_in_fake_defrost=True)
        P = np.reshape(np.array(p), (len(p),1))
        Z = np.concatenate((X,Y,P), axis=1)
        np.random.shuffle(Z)
        X = Z[:,0:X.shape[1]]
        Y = Z[:,X.shape[1]:-1]
        P = Z[:,-1]
        del Z
        return (X,Y,P)

    def predict_consecutive_readings(readings, look_back, model):
        """Classify consecutive time series values using the sliding window method.

        Parameters:
        readings (numpy array): The input time series.
        look_back(int): The size of the sliding window
        model (tensorflow.keras.Model): The model that will make the
            classifications.

        Returns:
        A numpy array of classification values.
        """

        input = []
        num_predictions = len(readings) - look_back
        test_predictions = [-1]*num_predictions
        for i in range(num_predictions):
            x = readings[i:i+look_back]
            x = (x-np.mean(x))/np.std(x)
            input.append(x)
        input = np.array(input)
        input = np.reshape(input, (input.shape[0],input.shape[1],1))
        return model.predict_classes(input)
```

```
import numpy as np


def generate_binary_run(len_run, noise=False, period=None, mean_period=None,
    period_range=None, start=None,\
                        force_end_in_1=False, force_end_in_0=False,
                            force_end_in_fake_1=False):
  """Generate a binary time series with a seasonal component for a toy dataset.

  If 'period' is 'None', then the seasonal component's period will be set to
      'period'.
  If 'period' is 'None' and 'mean_period' is not 'None', then the seasonal
      component period will be sampled
  from a Gaussian distribution centered at 'mean_period'.
  If 'period' is 'None', 'mean_period' is 'None', and 'period_range' is not
      'None', then the seasonal
  component period will be sampled from a uniform distribution with a min of
      'period_range[0]' and a max of
  'period_range[1]'.
  If 'period', 'mean_period', and 'period_range' are all 'None', then the
      seasonal component period will be
  sampled from a Gaussian distribution centered at 'len_run'/10.

  'force_end_in_1', 'force_end_in_0', and 'force_end_in_fake_1' are used to
      control the characteristics of
  the last time point in the time series.  See parameter descriptions below.
  Between 'force_end_in_1', 'force_end_in_0', and 'force_end_in_fake_1', only
      one can be set to True.
  If all three parameters are set to 'False', then the placement of the
      seasonal component will be random.

  Parameters:
  len_run (positive int): The desired length of the time series.
  noise (bool): Whether to add noide to the time series.
  period (positive int): The desired period of the seasonal component.
  mean_period (int): Center of Guassian distribution from which the period is
      sampled.
  period_range (tuple or list): Min and max values of a uniform distribution
      from which the period is sampled.
  start (int): The time point at which the first 1 of the seasonal component
      occurs.
  force_end_in_1 (bool): If True, force the last of the seasonally occurring 1s
      to fall on the last time point.
  force_end_in_0 (bool): If True, prevent the last of the seasonally occurring
      1s from falling on the last time point.
  force_end_in_fake_1 (bool): If True, prevent the last of the seasonally
      occurring 1s from falling on the last time
                              point, but then set the last time to the value 1
                                  anyways.

  Returns:
  X (numpy array): Time series values
  Y (numpy array): Seasonal component values
  period (int): Time series period
  """
  if start and (force_end_in_1 or force_end_in_0):
```

```
      raise ValueError('Cannot assign a start value if either force_end_in_1 or
          force_end_in_0 are True.')
    if (force_end_in_1 and force_end_in_0):
      raise ValueError('force_end_in_1 and force_end_in_0 cannot both be True.')
    if (force_end_in_fake_1 and force_end_in_0) or (force_end_in_fake_1 and
        force_end_in_1):
      raise ValueError('force_end_in_fake_1 and force_end_in_0 or force_end_in_1
          cannot both be True.')
    if (force_end_in_fake_1 and start==0):
      raise ValueError('force_end_in_fake_1 and start=0 cannot both be True.')
    X = np.zeros(len_run)
    Y = np.zeros(len_run).astype(int)
    if period == None:
      if mean_period:
        period = int(max(6,(np.random.normal(loc=mean_period, scale=mean_period/3
            ))))
      elif period_range:
        period = int(np.random.uniform(period_range[0], period_range[1]))
      else:
        period = int(max(6,(np.random.normal(loc=int(len_run/10), scale=int
            (len_run/10/3)))))
    #If start==None, assign a value to start
    if force_end_in_1:
      start = 0
    elif (force_end_in_0 or force_end_in_fake_1) and (start == None):
      start = int(np.random.uniform(1,period))
    elif start == None:
      start = int(np.random.uniform(0,period))

    if force_end_in_fake_1:
      noise=True
      X[0] = 1

    if noise:
      for i in range(len_run):
        if np.random.binomial(n=1,p=(1/period)):
          X[i]=1

    current = start
    while current < len_run:
      X[current] = 1
      Y[current] = 1
      current += period

    if (force_end_in_1 or force_end_in_0 or force_end_in_fake_1):
      X = np.flip(X)
      Y = np.flip(Y)

    return X,Y,period


  def generate_binary_training_data(len_run, noise=True, period=None, mean_period
      =None, period_range=None, start=None, num_x0y0=0, num_x1y1=0, num_x1y0=0,
      num_unspecified_end=0):
    """Generate a toy dataset containing binary time series with seasonal
        components.
```

```
    If 'period' is 'None', then all seasonal component periods will be set to
        'period'.
    If 'period' is 'None' and 'mean_period' is not 'None', then the seasonal
        component periods will be sampled
    from a Gaussian distribution centered at 'mean_period'.
    If 'period' is 'None', 'mean_period' is 'None', and 'period_range' is not
        'None', then the seasonal
    component periods will be sampled from a uniform distribution with a min of
        'period_range[0]' and a max of
    'period_range[1]'.
    If 'period', 'mean_period', and 'period_range' are all 'None', then the
        seasonal component periods will be
    sampled from a Gaussian distribution centered at 'len_run'/10.

    Parameters:
    len_run (positive int): The desired length of each time series.
    noise (bool): Whether to add noide to each time series.
    period (positive int): The desired period of every seasonal component.
    mean_period (int): Center of Guassian distribution from which the periods are
        sampled.
    period_range (tuple or list): Min and max values of a uniform distribution
        from which the periods are sampled.
    start (int): The time point at which the first 1 of the seasonal components
        occurs within each time series.
    num_x0y0 (int): The number of time series whose last seasonally occurring 1
        falls on the last time point.
    num_x1y1 (int): The number of time series whose last seasonally occurring 1
        doesn't fall on the last time point.
    num_x1y0 (int): The number of time series whose last seasonally occurring 1
        doesn't fall on the last time point,
                    but whose last time is guaranteed to be set to the value 1
                        anyways.
    num_unspecified_end (int): The number of time series whose seasonal
        components can fall anywhere.

    Returns:
    X (numpy array): 2D array of time series values
    Y (numpy array): 2D array of seasonal component values
    P (numpy array): 1D array of time series periods
    """

    X=np.zeros(((num_x0y0+num_x1y1+num_x1y0+num_unspecified_end), len_run))
    Y=np.zeros(((num_x0y0+num_x1y1+num_x1y0+num_unspecified_end), len_run))
    p=np.zeros((num_x0y0+num_x1y1+num_x1y0+num_unspecified_end))
    for i in range(num_x0y0):
      X[i,:], Y[i,:], p[i] = generate_binary_run(len_run=len_run, noise=noise,
          period=period, mean_period=mean_period, period_range=None, start=start,
          force_end_in_1=False, force_end_in_0=True, force_end_in_fake_1=False)
    for i in range(num_x1y1):
      ind = num_x0y0+i
      X[(ind),:],Y[(ind),:],p[ind] = generate_binary_run(len_run=len_run, noise=
          noise, period=period, mean_period=mean_period, period_range=None, start
          =start, force_end_in_1=True, force_end_in_0=False, force_end_in_fake_1=
          False)
    for i in range(num_x1y0):
```

```
        ind = num_x0y0+num_x1y1+i
        X[(ind),:],Y[(ind),:],p[ind] = generate_binary_run(len_run=len_run, noise=
            noise, period=period, mean_period=mean_period, period_range=None, start
            =start, force_end_in_1=False, force_end_in_0=False, force_end_in_fake_1
            =True)
      for i in range(num_unspecified_end):
        ind = num_x0y0+num_x1y1+num_x1y0+i
        X[(ind),:],Y[(ind),:],p[ind] = generate_binary_run(len_run=len_run, noise=
            noise, period=period, mean_period=mean_period, period_range=None, start
            =start, force_end_in_1=False, force_end_in_0=False, force_end_in_fake_1
            =False)
      P = np.reshape(np.array(p), (len(p),1))
      Z = np.concatenate((X,Y,P), axis=1)
      np.random.shuffle(Z)
      X = Z[:,0:X.shape[1]]
      Y = Z[:,X.shape[1]:-1]
      P = Z[:,-1]
      del Z
      return (X,Y,P)

  def predict_consecutive_binary_readings(readings, look_back, model):
      """Generate an array of forecasted values using the sliding window method

      Parameters:
      readings (array): The input time series
      look_back(int): The size of the sliding window
      model (tensorflow.keras.Model): The model that will make forecasts
      """

      input = []
      for i in range(len(readings)-look_back):
          x = readings[i:i+look_back]
          input.append(x)
      input = np.array(input)
      input = np.reshape(input, (input.shape[0],input.shape[1],1))
      return model.predict_classes(input)
```