Håvard Rakbjørg Minsås

Tag Inference

Feature extraction and sensor classification from time series data

Masteroppgave i Datateknologi
Veileder: Patrick Mikalef
Medveileder: Filip Henrik Larsen & Nisha Dalal
Juni 2023

Håvard Rakbjørg Minsås

Tag Inference

Feature extraction and sensor classification from time series data

Masteroppgave i Datateknologi
Veileder: Patrick Mikalef
Medveileder: Filip Henrik Larsen & Nisha Dalal
Juni 2023

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Feature extraction and sensor classification from time series data

Håvard Rakbjørg Minsås

Abstract

The works in this thesis have been done with a goal of classifying time series data originating from a number of different building sensors, such as temperature sensors, valves, and pressure sensors. The samples are processed in such a way that each data point comes at equal intervals, before features are extracted by feature extraction libraries TSFresh and Catch22. The features are then passed into a number of different models, and the models fine tuned so to give the best accuracy for inferring the time series. The work done here show that gradient boosting methods achieve the highest accuracy, and using smaller intervals sizes is beneficial for most models. The task and data was provided by Piscada [1].

Sammendrag

Moderne bygg bruker i større grad enn tidligere ulike sensorer for å overvåke ulike deler av konstruksjonen. Slik data lagres gjerne som en tidsserie, hvor verdier assosieres med et tidsstempel. I denne oppgaven behandles denne tidsseriedataen slik at den egner seg til klassifisering basert på karaktertrekk, med et mål om å kunne identifisere typen sensor dataen har opphav ifra. Flere ulike algoritmer for klassifisering undersøkes og sammenliknes, deriblant nærmeste nabo, beslutningstre og gradientforsterkede tre. I tillegg undersøkes hvordan hyppigheten på datapunkter i en tidsserie påvirker nøyaktigheten på klassifiseringen. Resultatene viser at gradientforsterkede tre gir den høyeste nøyaktighet på 91.7 prosent. Det vises også at kortere intervaller mellom datapunktene har en positiv innvirkning på evnen til å klassifisere korrekt. Oppgaven er skrevet på vegne av Piscada.

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Background

Sensory equipment for monitoring in buildings are widely used across the world. This equipment makes it possible to monitor status in parts of buildings, as well as power consumption at all times, and are also a necessary part for automation. Within this technology a huge potential to reduce power usage, and optimize the operation of a building to suit individual needs. Currently there is no global or major regional naming convention for these types of sensors, resulting in the naming often being left to the operator of the building.

Statsbygg, the norwegian directorate of public construction and property, have defined a naming convention for sensor equipment in Norway. This ensures the same names and identification is used across buildings and industries, to help with optimal operation and communication across projects. This standard is called "Tverrfaglig merkesystem" (TFM). When TFM is used across buildings and industries transitions are made easier, mitigating the need for manual re-labeling of sensors. Without it, changing the owners can result in expensive and time consuming work to handle naming conventions. Communication across industries also becomes easier, as the same type of sensor will have the same naming convention, making misunderstanding less likely to happen. TFM is not mandated in every construction in Norway, but always used by Statsbygg and Forsvarsbygg. It is also becoming more common for larger projects to follow this standard. This is likely because the usefulness of the convention might not be as apparent in smaller buildings, but in larger buildings it makes the work to re-identify the exact type of sensor a lot easier.

The task in this project is to label sensors based on time series data, making transitions between naming conventions easier. The objective of this work is to develop a system for automatically transition between sensor naming convention. Making it easier for new users to transition to a common naming system. It is worth noting that any such model is not likely to be one hundred percent accurate, as there tend to be outliers when working with real world data. However, in generating an automatic system for such labeling, it is likely the number number of instances needing to be manually handled will be substantially reduced. The same model can also be used to identify sensors that behave abnormally, by using the same model to predict the sensor it belongs to. If the wrong prediction occur, some investigation into as of why might be necessary.

1.2 Terms

This section covers a brief description of terminology.

1.2.1 Time series

A time series is a collection of data points that are ordered sequentially based on time. While it is beneficial for the data points to have a regular spacing, it is not a strict requirement. However, having a constant interval spacing in time series analysis often simplifies the analysis process, and is a requirement for certain features to be extracted. Time series data is commonly observed in various fields, including stock market analysis, health data, and climate monitoring.

1.2.2 Feature

In this thesis, the term "features" is employed in the context of time series analysis. Features refer to specific characteristics of a time series, such as maximum value, median, or average. They provide valuable insights into the overall data set. By using features, one can represent a time series as tabular data, summarizing its properties rather than representing individual data points.

1.2.3 Inference

Inference entails deriving a conclusion from the given data. In the context of this thesis, inference is utilized to explain the process of analyzing input data, enabling the identification of the probable sensor type to which the data belongs.

1.2.4 Weak learner

A model that perform slightly better than random guessing. A typical example of a weak learner is a decision tree. For binary classification, i.e. when there is only two possible answers, the definition states that its accuracy should be slightly better than 50 percent. For multi-class classification the criteria are not so clear. The requirement of better than random guessing is said to be too weak of a description, and also more than 50 percent accurate is too strict of a requirement.

1.2.5 Strong learner

Strong learners are models that have an arbitrary good accuracy. While there is no set definition to how well it needs to perform, one can expect a strong learner to perform significantly better than random guessing.

1.3 Motivation

The goal of this project is to make a model that can, with a high percent of accuracy, determine what type of sensor a time series originated from by analysing features extracted from said time series. In turn this method can make re-labeling of sensory equipment within buildings easier, either by directly inferring the sensor, or limiting the pool of possible sensors. The idea and data used in this project have been provided by Piscada [1].

1.4 Research approach

When working with real world data as in this project the task can be divided into three main parts. First the data set needs to be prepared in the necessary ways. For this project that means handling features with invalid value extracts such as NaN and Inf, removing outlier files from the training set and standardize the time interval between data points. When the data is prepared a number of different models are selected, and trained on the training set. The models are tested with a number of different parameter-settings, as to help find the best approach. Lastly the models are evaluated by inferring time series from the test data set, and the results are compared.

Chapter 2

Related work

2.1 TFM naming convention

TFM stands for TverrFaglig-Merkesystem and is a naming convention created by Statsbygg intended for building parts and technical installations [2]. This naming convention consists of mainly three parts; location, system and component. Below you can see an example of a TFM code, used to describe a "outside air temperature sensor".

```
+215301=360.001-RT901
```

The three parts are denoted by their own character, namely the plus sign (+), the equals sign (=) and the hyphen (-). The reason for this is to easily be able to separate the individual parts, and also be able to reference a specific section of the code without having to list the entire code for the sensor.

Starting with the pluss sign that refers to main element number one. This references the location of the building the sensor is a part of, this code is set by the contractor who set up the system. In the example above building 215301 is being referenced.

Thereafter comes the equals sign that references the system the sensor is a part of, also known as main element number two. The system that this sensor is a part of is denoted by the number 360, and by looking up in Statsbyggs document for systems codes [3] one can see that this is a system for air treatment. Given that each system can also consist of multiple sensor of the same type there is also a serial number attached to main element number two. This is so that one is able to distinguish between like sensors. In this example the serial number is 001, and comes after the dot.

Lastly there is main element number three, denoted by the hyphen. Main element number three references the component. In this example the component is of the type RT. By looking up in the component code list [4] one see that RT means a temperature sensor. The remaining numbers are serial numbers like the system code in main element number two.

The TFM naming convention is also summarized in figure 2.1, showing the structure of how a sensor code is built.

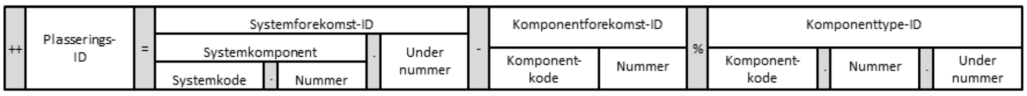


Figure 2.1: Image representation of TFM [5]

2.2 Feature extraction

Feature extraction is an important part of time series classification, as this transform the data from a set of timestamp-value pairs into a tabular data set. The latter opening the possibility to build classification models. There are several libraries for Python that focus on feature extraction, and this section will look at some of them.

TSFresh [6] is an automatic feature extraction library that builds upon the FRESH algorithm [7]. FRESH stands for FeatuRe Extraction based on Scalable Hypothesis tests. Not only does it extract a wide variety of features from each time series, but also evaluate the p-values of each feature. This way the algorithm is also able to filter out non-significant features from the data set. The algorithm was tested against, among others, all binary classification problems in the UCR time series classification archive [8] and benchmarked against other feature extraction algorithms such as linear discrimination analysis [9] and dynamic time warping [10]. The algorithms were evaluated on their ability to successfully extract relevant features and how long it took them to do so. To determine the meaningfulness of features a classification algorithm was applied to the extracted features under the assumption that the classification would perform better when the features were more meaningful. The classifier used is Random Forest, based on the findings in "Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?" by Fernandez-Delgado et al. [11]. Additionally AdaBoost [12] were selected as another classifier. The paper concludes that the FRESH algorithm is able to filter features so that AdaBoost and Random Forest classifier do not achieve worse accuracy for most data sets.

TSFEL [13], Time Series Feature Extraction Library, is another popular library to extract time series characteristics. An overview over feature can be found in TSFEL documentation at [14]. The features are divided into three categories, temporal, statistical and spectral, based on the domain of which they are calculated. It differs from libraries such as TSFresh in that it focuses more on the temporal complexity of the extracted features, making it more suited for computers with limited resources.

Catch22 [15], CAnonical Time-series CHaracteristics, is a feature extraction library that focuses solely on proven and highly informative features, removing a lot of the redundancy that other, larger feature extraction libraries often end up with. The features were selected based on test done across 93 time-series classification data sets, and the most informative features were selected out from a filtered version of hctsa feature library [16]. The reduction of features allows for the computational time to be reduced approximately 1000-fold, while only dropping accuracy when classifying by about 7 percent.

2.3 Time series classification

As the number of Internet of Things (IoT) devices continue to surge in the modern world, the amount of data continue to rise with it. A significant portion of this is time stamped data. Whether it be from devices monitoring traffic along roads, sensors monitoring temperature outside or sensors monitoring water pressure in a pipe. This data can grow in size rapidly, depending on the reporting interval, and quickly becomes too much for a human to manually interpret. However, data without insight to what it means has little value. So to gain insight we need algorithms and models to help us interpret this large set of data.

With the rise of time series data, as monitoring becomes more common, there is no surprise that there is a lot of research done on this topic. The research covers everything from binary and multi-class classification, to forecasting and prediction. For this thesis, however, the focus will be on multi-class classification, and investigate papers related to this topic.

While many innovations happen within machine learning using deep learning or neural networks, Léo Grinsztajn and his co-authors in "Why do tree-based models still outperform deep learning on tabular data?" [17] have conducted a study on medium sized tabular data sets at about 10 thousand samples where they found that tree-based models still outperforms their more complex machine learning counterparts. Both the type of data and the size of the set align very well with the ones in this thesis. They found that tree-based models are better at handling high-dimensional and correlated data than their deep learning counter parts, using less space and time while also providing better accuracy. One of the reasons listed for this is that some machine learning approaches do not handle uninformative features as well, meaning features that does not help identify the class. Tree based approaches as the decision tree have ways to determine what features are informative and not, often using the Gini index or a similar approach to calculate the gain or value for doing a split at a certain feature.

Anthony Bagnall et al. in their paper "The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances" [18] have conducted a study evaluating recent algorithm advances for time series classification. The reason for this is that many of these algorithms are only tested on a single synthetic data set, making them seem better than they might be in reality. Therefore, these recently published algorithms are tested on several different data sets, to see how well they do. Among these algorithms are Derivative transform distance [19] and Collective of Transformation-Based Ensembles [20], where the later was found to be the best algorithm by far using their benchmarks. On average more than 8 percent more accurate than their benchmarks, 1-nearest neighbour and rotational forest. In total 18 algorithms was evaluated, many of them performed worse than the benchmarks. Evaluation looked at accuracy when classifying, scalability and speed.

Hassan Ismail Fawaz et al. "Deep learning for time series classification: a review" [21] take a look at deep learning approaches for time series classification. While the field of time series classification is quite popular, and hundreds of algorithms have been proposed to deal with classification problems of this nature, very few of these approaches utilize deep learning. 8730 deep learning models was trained on 97 time series data sets and evaluated. Architectures such as Fully Convolutional Neural Networks and deep Residual Networks are able to achieve accuracy on level with the leading time series classification methods.

While XGBoost [22] is not used as a part of this thesis, it is still a widely used and recognised library for gradient boosting. During the 2015 machine learning challenge hosted by Kaggle a total of 17 out of 29 winning solutions used XGBoost in some form to achieve their result. The reason for its success in the market can be explained by its scalability, allowing it to run faster than other gradient boosting implementations, and also scale to a large number of examples.

Chapter 3

Method

3.1 Sensors

The data set and thereby the classification problem is based on being able to differentiate different types of equipment based on feature characteristics from their time series. The sensors and the number of files in both the training set and test set are listed in table 3.1.

While one ideally would like to be able to differentiate between all the 21 different types of sensor, that is not possible. The reason why it is not possible to differentiate the finer levels is that these sensors can behave differently based on the systems they are a part of. For instance a supply water temperature may be set to value ranges that equal the return water temperature of another system, making them indistinguishable from each other. These same values may also fluctuate over time in the same system. Therefore, the focus will be on trying to differentiate groups of these sensors with similar characteristics from each other. These groups can be found in table 3.3.

3.2 Raw data

The raw data used for this project is stored in .feather files, containing a data frame with two columns. The first column "timestamp" consist of a date-time referencing when the data was received. Associated with each timestamp there is a column "value", that contains the value at the time of the time stamp. The systems are set up so that they only write a new row when a change is detected in the sensor. Causing the spacing between timestamps to be uneven. The length of observation for time series are also different, varying between 6 hours up to two weeks.

The feather files are further divided into folders based on their sensor type, i.e. Chilled water return temperature sensors, and also split into folders based on the weak and year the data are from. The data is divided into two sets, one for training and one for testing. All of these files originate from the same data set, and have been divided for practical purposes by people working at Piscada. The

Sensor name	# Files Train	# Files Test
Chilled Water Return Temperature Sensor	215	58
Chilled Water Supply Temperature Sensor	138	46
Cooling Valve	40	6
Differential Pressure Sensor	1214	289
Discharge Air Static Pressure Sensor	624	171
Energy Sensor	2	1
Heat Exchanger	372	86
Hot Water Supply Temperature Sensor	313	93
Outside Air Temperature Sensor	654	146
Power Sensor	232	68
Preheat Supply Air Temperature Sensor	632	166
Pump	532	102
Reheat Valve	245	71
Return Air Temperature Sensor	810	217
Return Fan	634	145
Return Water Temperature Sensor	759	197
Supply Air Static Pressure Sensor	588	158
Supply Air Temperature Sensor	939	232
Supply Fan	624	174
Valve	582	165
Variable Frequency Drive	358	88
Total	10507	2679

Table 3.1: List over sensor types and number of files.

"train" data folder consist of 10507 files, that will be used to train the model. The "test" folder consist of 2679 files, that will be used to evaluate the accuracy of the model. This brings the total number of files to 13186.

Below, in table 3.2, you can see an extract from a time series data frame. This data is taken from a chilled water return temperature sensor. More specifically it is retrieved from the time series 2021-23_+215301=360.003-RT561_MV.feather.

Figure 3.1 show the same data as displayed in 3.2 but plotted as a graph using Matplotlib's PyPlot library [23].

Timestamp	Value
2021-06-06 21:57:54.135000+00:00	20.9
2021-06-07 02:56:22.772000+00:00	20.8
2021-06-07 03:38:34.540000+00:00	20.9
2021-06-07 03:40:04.983000+00:00	20.8
...	...
2021-06-11 19:44:47.176000+00:00	20.3
2021-06-11 20:17:56.156000+00:00	20.4
2021-06-11 20:20:56.862000+00:00	20.3
2021-06-11 20:22:27.282000+00:00	20.4

Table 3.2: Time series for Chilled Water Return Temperature Sensor.

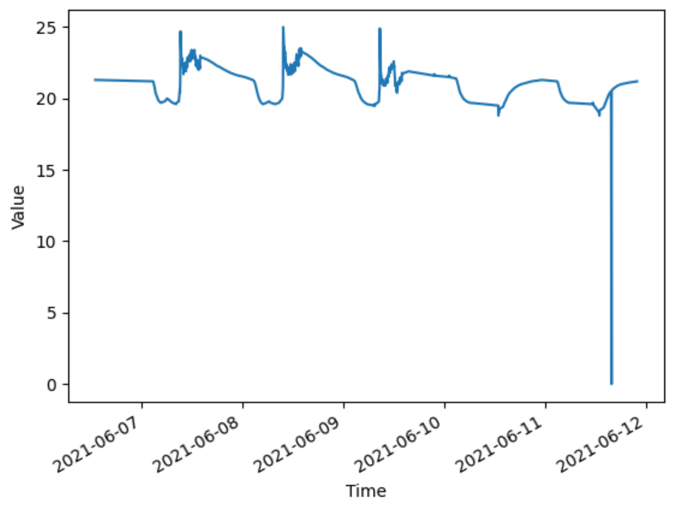


Figure 3.1: Time series for Chilled Water Return Temperature Sensor

3.3 Grouping of data

As stated in section 3.1, differentiating between the finer levels of sensors is not possible to do accurately. Instead the focus is at being able to tell groups of sensors apart. Sensors within these groups portray similar characteristics, making them difficult to differentiate. The sensors are divided into 10 groups, shown in table 3.3, and models are built around classifying a time series within the correct group. The groups are made up so that similar sensor types are grouped together, for example are most temperature sensors located in G1, and valves in G2. These groups show fairly little overlap between them, and still allow for good predictions to be made. With these groupings the distribution of files are as seen in table 3.4.

Group	Sensors
G0	Outside Air Temperature Sensor
G1	Chilled Water Return Temperature Sensor
	Chilled Water Supply Temperature Sensor
	Hot Water Supply Temperature Sensor
	Preheat Supply Air Temperature Sensor
	Return Air Temperature Sensor
	Return Water Temperature Sensor
	Supply Air Temperature Sensor
G2	Cooling Valve
	Reheat Valve
	Valve
G3	Differential Pressure Sensors
G4	Discharge Air Static Pressure Sensor
	Supply Air Static Pressure Sensor
G5	Heat Exchanger
	Variable Frequency Drive
G6	Return Fan
	Supply Fan
G7	Power Sensor
G8	Pump
G9	Energy Sensor

Table 3.3: Grouping of sensors

Sensor name	# Files Train	# Files Test
G0	654	146
G1	3806	1009
G2	867	242
G3	1214	289
G4	1212	329
G5	730	174
G6	1258	319
G7	232	68
G8	532	102
G9	2	1
Total	10507	2679

Table 3.4: File distribution by groups

3.4 Data preprocessing

As stated in the previous section, the intervals between data points are not even by default. This is due to the way sensors are set up, that they do not report values at set intervals, but rather when a change is detected. That solution is nice in of itself, so that one is not transmitting more data than necessary, however it does cause some problems when extracting features. Some of the features extracted by TSFresh only makes sense if the spacing between data points are of equal length. One might be tempted to choose an interval size that is equal to the minimum time dependent interval change given in the entire data set, so that no data points are lost, and all samples are unaltered in that sense. This however, will cause the files to become much larger, and in return feature extraction for a large number of files will be both time consuming and computationally expensive, compared to if a smaller interval size is chosen. While time and computational cost might not be an issue when one have access to cloud computing, it does impose a limit on the work done in this thesis, as it is done on an ordinary desktop. Therefore, an interval size that still capture the characteristics of time series, without causing the computational overhead to grow to large is chosen. Many time series were looked into, both the raw data and associated plots to decide on an interval that would enable affordable feature extraction while keeping as much of the original information in place. In figure 3.2 three different samples are plotted with their original interval compared to the 10 minute interval that is proposed here. From left to right the sensors plotted are heat exchanger, chilled water supply temperature sensor and variable frequency drive. As one can see from the figure, the plots still keep their characteristics, even though several data points are lost.

Using a 10 minute interval size makes it easier to do feature extraction, and reduce the time it takes for most samples to between 3 and 5 seconds. To equalize

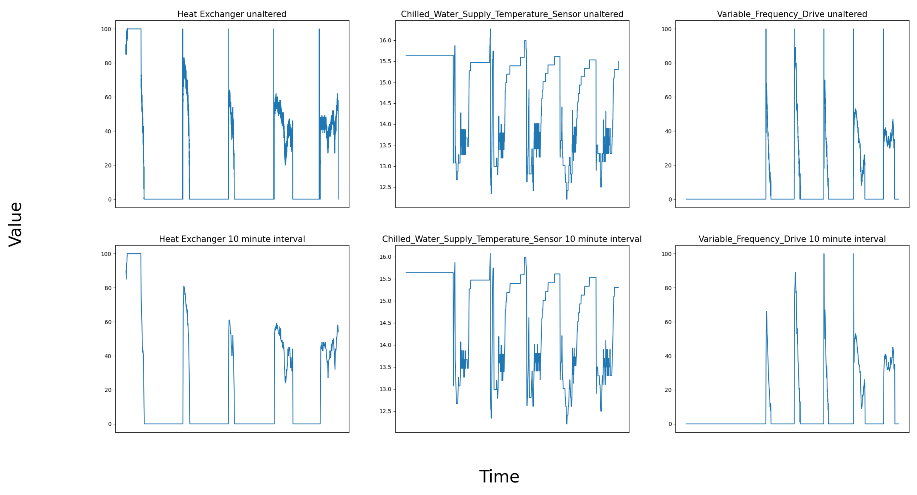


Figure 3.2: The figure shows the difference between the original time series, and their 10 minute interval copies. Top plot shows unaltered, bottom plot shows 10 minute intervals.

the interval sizes the start point and end point of each time series was extracted, and pandas date range was utilized to create 10 minutes intervals between these dates. This new date range was then filled with values from the original frame using forward fill, so that the most recent value occupies a given 10 minute block.

A possible benefit to choosing an interval size that is larger than the minimum time dependent interval is that some faulty data points might be eliminated. In figure 3.1 one can see a value that looks out of place for a temperature sensor, where it suddenly drops to zero for a short amount of time before being corrected. However, when utilizing a bigger interval this value is overridden, as seen in figure 3.3. And since such erroneous values are much less common than correct values in the time series, it is also more likely that erroneous values will be overridden using 10 minute intervals with forward fill.

In addition to the main interval size of 10 minutes, features will also be extracted at intervals of 1 minute, 1 hour and 5 hours. This is to compare the accuracy for each model, depending on the interval size. While it would also be interesting to reduce the interval size even further, it was not feasible with the available resources. Extracting features on a 1 minute interval took more than two weeks for the entirety of the data set. These features will then be tested at the different models, so that the results can be compared. This will show how much of an impact different interval sizes has when it comes to capturing characteristics of a time series. Below in figure 3.4 one can see how the interval size effects the plot of a time series. The bigger the interval, the smoother the plot becomes.

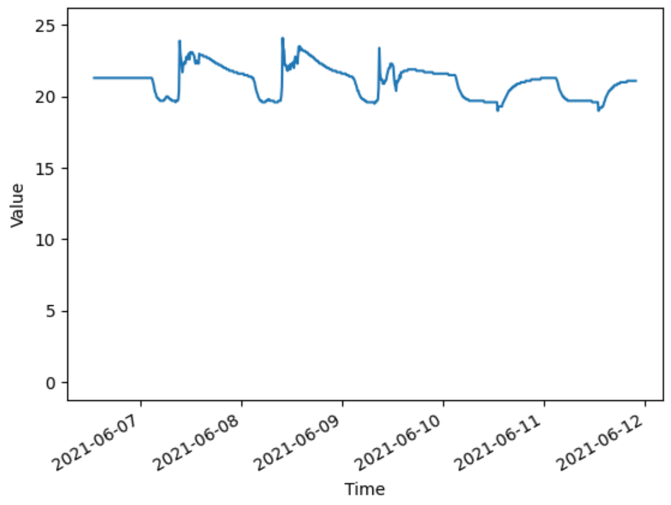


Figure 3.3: Time series for Chilled Water Return Temperature Sensor with even interval size

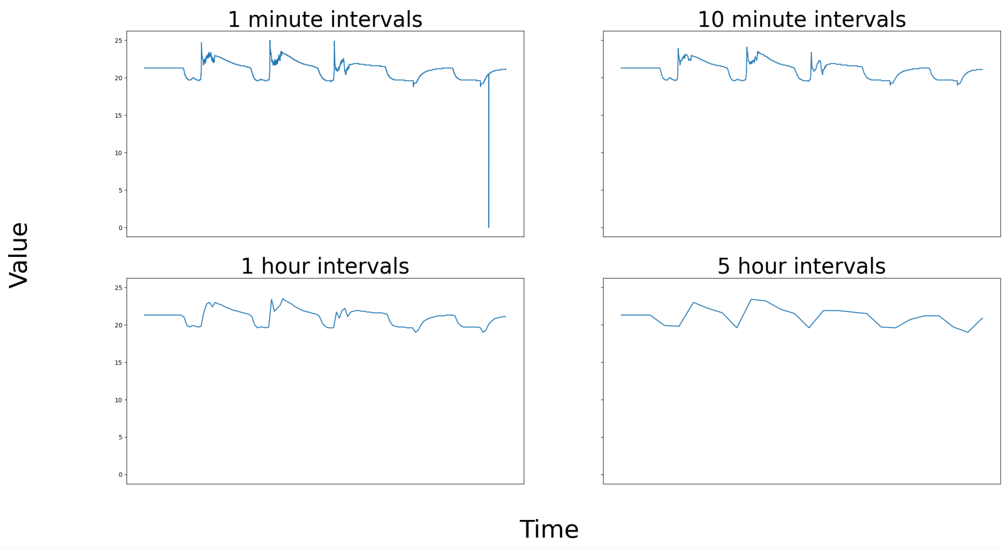


Figure 3.4: Time series plot over 4 different interval sizes.

One thing to note is that when attempting to extract features on the 1 minute interval some files become too big to be able to do so on an ordinary desktop computer. Therefore, the number of samples in both the training set and the test set have decreased some. The training set was reduced from 10507 samples down to 10466 samples, meaning that feature extraction failed for 41 samples. The test set was reduced from 2679 samples down to 2659, meaning that 20 samples failed. This lead to the calculation of an additional accuracy for approaches when utilizing the 1 minute interval. This accuracy portrays a worst case scenario, where it is assumed that every time series that could not have its features extracted were inferred wrongly.

3.5 Feature extraction

Comprehensive features from TSFresh [6] is the main feature extraction library used in this project to extract features. This is a library that allows extraction of a wast range of features from each time series, and is very useful in capturing the characteristics of each one. Using the comprehensive features parameter TS-Fresh will use 63 time series characterization methods, and compute a total of 794 features for each time series. A list over extracted features can be found in [24]. One thing to keep in mind is that a fair number of these features can not be calculated for the time series, and will return Nan-values. Many of these features can be considered redundant, but which ones is not known until a model is constructed. The decision is to keep them all, and let the different approaches figure out what features are relevant. This functionality is essential in decision trees, but methods like k-nearest neighbours do not have a way to sort features, and might suffer from this decision.

Additionally, Catch22 [15] is tested on most of the models. Catch22 is a feature extraction library that only extract a limited amount of features that have proven themselves well for a range of real world classification problems. This is a lot faster than using TSfresh's comprehensive features, but of course comes at the cost of a number of features than can prove useful for the models. There is also an additional parameter for the Catch22 feature extraction, enabling the extraction of an additional two features. Mean and standard deviation, essentially making it Catch24. The complete list of features extracted by this library can be found in [25]. The reason for also including Catch22 as a feature extraction library is comparing a set of minimally redundant feature up against a fairly redundant one, and comparing the gain or loss in accuracy from the additional features using TSFresh.

3.6 Invalid features

With the wast number of features extracted when using TSFresh and the variation in time series, one expect it to return a few features that result in NaN-values. Scikit does not take these values as input for all models, and therefore there is a need to handle NaN-values in some way. Simply removing the features that are not valid for a time series only for that time series is not an option, as scikit needs all time series to have the same dimensions. There are mainly two ways NaN-values can be handled.

Either remove all features that have NaN-values from all of the time series, this will significantly reduce the number of features in the data set. However, as TSFresh extract so many features there are still more than 200 features that are valid across all time series. This will also speed up the time it takes to train a model, as there will be less features to consider when calculating splits. This is especially noticeable with the gradient boosting tree, reducing the time it takes to construct by 1 to 2 hours.

The other option is to insert a default value whenever a NaN-value is encountered. Again there are a couple of approaches to choose from here. NaN-values can be replaced with a value that does not commonly occur within the data set, making it so NaN-values still stand out from other values in the data set. This can be a viable approach, as while NaN-values are not numerical, they can still be argued to be a feature characteristic for the time series. Alternatively a 0 or a 1 can be inserted in its place, and let the Scikit library figure out the best splits. The different approaches were tested with the decision tree, and the approach with the highest accuracy was chosen.

Some models might get a slight increase in accuracy using one of the other approaches, but for better comparability, the same approach is used in across all models. The difference in accuracy is marginal. Difference between removing all invalid features from the data set and using a default value of 1 was around 0.2 percent for the decision tree. Meaning that no models was greatly impacted by the choice made here.

3.7 Handling of outliers

Given the large number of files in the training set and test set, a modular solution for identifying and removing outliers is preferred. For this task the interquartile range [26] was chosen. This approach is easily visualized, and can be tuned to fit the need in a given data set by changing the ranges it operates on.

The original approach is based on identifying the first and third quartile. The first quartile, Q1, is defined as the value that has 25 percent of values with a lower or equal value to itself. The third quartile is defined as having 25 percent of other values greater than or equal to itself. Then you find the interquartile range that is defined as:

$$IQR = Q3 - Q1 \tag{3.1}$$

This is the value range that separate the first quartile from the third quartile. The next step is defining boundaries for outliers. The lower boundary is found by taking the first quartile and subtracting the interquartile-range multiplied by 1.5. For the upper boundary one take the third quartile and add the interquartile-range multiplied with 1.5.

$$lower\_bound = Q1 - 1.5 \times IQR \tag{3.2}$$

$$upper\_bound = Q3 + 1.5 \times IQR \tag{3.3}$$

Box plot is a built in pandas method that visualize the IQR-method, see figure 3.5. The black rectangles represent the range between Q1 and Q3, the purple lines are the lower and upper boundaries and the black circles represent outliers. Lastly the median is represented by the green line.

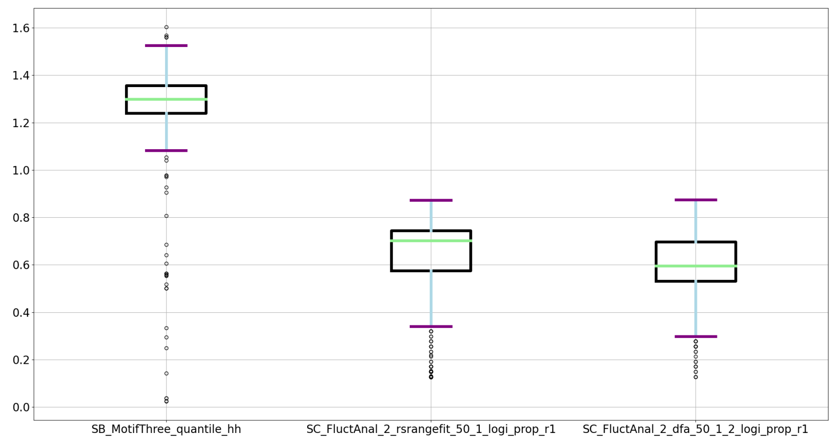


Figure 3.5: IQR-method visualized by pandas box plot. The black boxes show the first and third quartiles, while the purple lines show the upper- and lower-boundaries. The plotted dots are features outside of the bounds. Green line shows the median. Visualization is based on features extracted by Catch22 for outside air temperature sensor

Then comparing values against the lower and upper bounds. If a feature value is outside of either of these values it is identified as an outlier in the data set. Given that there are many features there is also a need to set a threshold for how many outlier features a time series can have before it is discarded. For instance if a sensor has more than 200 features outside of the IQR boundaries, it is to be discarded. Note that outliers are only calculated for each type of sensor group, not for the entire training set at once. This is to prevent discarding time series for sensors without a lot of data, for instance the energy sensor. The goal is to only discard time series that do not align with the group it is a part of. This ensures that the models are not trained on files that are obviously erroneous.

The IQR multiplier can be adjusted to allow for a wider range of values, or the quartiles can be adjusted to guarantee keeping at least 80 percent of the files. This make outlier detection stricter, ensuring that only the most noticeable outliers are removed from the data set. Ideally files that accurately portray the group it is a part of would not be discarded. This causes the outlier requirements to be stricter than the default implementation described above. The quartiles were changed from 0.25 and 0.75 to 0.10 and 0.90, making it so that at least 80 percent of features are considered within range. TSFresh extract a high number of features, where many of these can be considered uninformative. That means that they do not reflect any characteristics of the group a time series belongs to, and can be considered noise. This create the need for a high threshold for number of features outside of the range set by the IQR-approach before a time series is discarded. Several thresholds were tested. The threshold that have the highest accuracy was selected, that being a threshold of 275.

This approach is only performed on the training data, as here it is known what type of sensor it is. While the sensor for each time series in the test set is also known, this would not be the case in a real world application of this method. To eliminate files before inference is done one could use other approaches, such as setting a minimum number of data points in the time series.

When removing sensors using the parameters described above, a total of 68 files is removed from the training set. The removed files are distributed as shown in table 3.5.

When testing with different interval sizes it could be considered naturally to apply this method in each individual case. However, as this would require further fine tuning of the parameters for each individual interval size, the outliers found applying the IQR-approach on the 10 minute interval is kept regardless of interval size. With the current setting no outliers would be detected for either 1 hour intervals and 5 hour intervals. This is done as to give all models the same files to build upon.

Sensor name	# Files removed
G0	2
G1	1
G2	0
G3	59
G4	0
G5	0
G6	0
G7	6
G8	0
G9	0
Total	68

Table 3.5: Outliers from each group

It is clear that some of these outliers do not represent sensor values in a way that one would expect. In figure 3.6 the two outsiders for G0, outside air temperature sensor, are compared to two normal time series for this sensor. From the plots it is clear that the outliers do not behave in a way that would be considered normal for this type of sensor. One would expect them to fluctuate much in the same was as the two normal sensors, with temperatures rising in the morning and dropping towards the evening. The outliers are linear before abruptly reaching a maximum value.

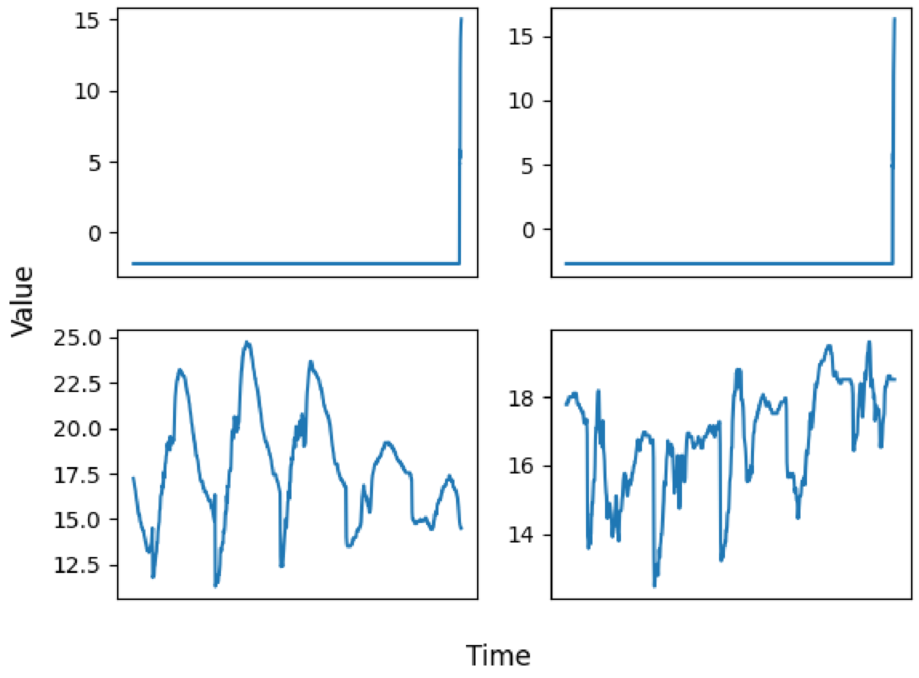


Figure 3.6: The top two plots show two outliers for outside air temperature sensor compared to two normal time series for the same sensor, seen in the bottom two plots. The interval size for all plots are 10 minutes

3.8 Scikit

Most models will be implemented by using the Scikit-learn library [27] for Python as these have a very similar implementation across models, making it possible to pass data in the same format to different models. This way one can ensure that when changes are made to data, it only has to be done in one place, before passing the new data into the models. This library also implements some handy functions on top of each model, making it easy to evaluate the accuracy of each model created, as well as easy to extract what time series were wrongly infered. Namely the score function that returns a score reflecting from 0 to 1 how many files were correctly infered, and there is also a useful method called predict, that returns an array with all the predicted values for a test set. With the similar style of implementation for each model, and the practical methods for evaluating the test set, Scikit-learn is a great way to easily compare methods. Methods where it is relevant also have a parameter called random_state that can be set to ensure that the same "random" choices are made every time, making the results replicable. For all models where this is relevant, such as the decision tree and gradient boosting tree this parameter is set to be equal to 0.

3.9 Vector search approach

To set the baseline for the results a very simple implementation was chosen. Something one would expect to achieve better results than random guessing, but not by a large margin. The approach that was settled on was a simple vector search, and was implemented from scratch in Python. Here the training data is used to calculate an average value for each feature in each sensor group. Then this average value for each group is compared against the time series that is to be inferred. The distance is calculated to each of these average vectors using Manhattan distance, and infer the sensor to be belonging to the closest average vector.

For features to have maximum impact values are also normalized, such that all features will have a similar impact on the distance measure. If one were to not do this some features overshadowing the wast majority of other features. I.e. a feature that have values in the thousands will have a much larger impact on the distance measure than a feature with values around 1. So all features are normalized by finding the maximum value in the entire training set, and dividing all feature values with this maximum value for that feature. This ensures that all values in the training set will have values between 0 and 1. The same guarantee can not be made for the test set, but it will likely be very close.

Below is the formula used to normalize feature values:

$$\sum_{i=1}^{n} \frac{\vec{F}_i}{\max F_i} = [f_1, f_2, ..., f_n] \tag{3.4}$$

Then the sensor is inferred by calculating the Manhattan distance to all average vectors and selecting the minimum.

After normalization each feature from the sensor one would want to infer is compared against each feature in an average sensor, and then summarized to find the total distance. This is done for each of the average sensors.

$$dist_i = \sum_{i=1}^{n} abs(F_i - A_i) \tag{3.5}$$

Having calculated all of these distances, dist, the shortest distance is found.

$$sensor = min(dist_1, dist_2, ..., dist_n) \tag{3.6}$$

Whatever average sensor this shortest distance was between, is the sensor that the time series will be inferred as.

3.10 K nearest neighbours

K neighbours [28] checks the time series up against all time series in the training set, then selects the k nearest neighbours by distance in the n-dimensional space. Then the sensor is infered by a soft vote. A number of different k-values were tested. The accuracy dropped off after k = 5, therefore 5 was selected as the k-value for this approach. Additionally the distance weight showed the most promise, where weights are calculated from the inverse distance from the point. Rather than a pure majority vote, distance also effect to result. This way neighbours that are closer in distance carry more influence over the inference than neighbours further away.

In figure 3.7 a black data point can be seen positioned in a two dimensional space and infered based on a soft majority vote. In this illustration the black point will be inferred as a member of the light green set.

To build this model the scikit implementation, called the KNeighborsClassifier [28], was used. The following arguments were used:

```
clf = KNeighborsClassifier(n_neighbors=5, weights="distance")
```

3.11 Decision tree

Decision tree is a natural choice when it comes to this sort of classification problem. This is because features with values are extracted, creating a problem where one is trying to identify a unique combination of feature values that identify a group. By extracting features, the data is transformed from a data set consisting of time series, into a data set of tabular data. Another reason that a decision tree is a good model for classification is that it is easily explainable. Having built the tree

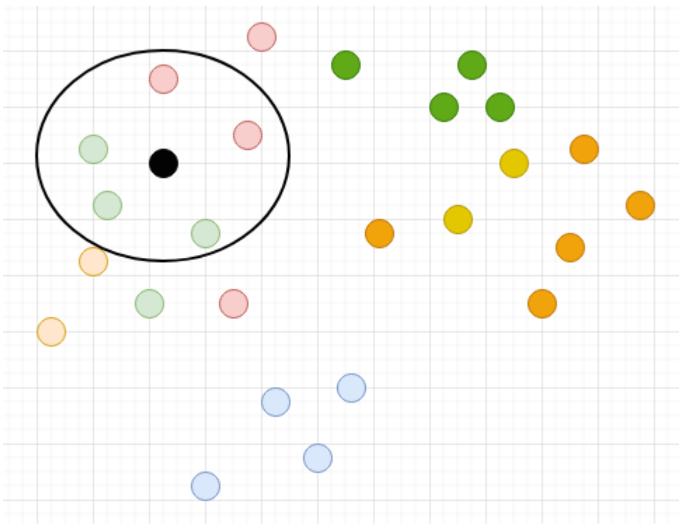


Figure 3.7: Illustration of K-neighbours method in a 2-dimensional space. The black point is infered by a soft vote over its 5 nearest neighbours, in this case light green will get the majority-vote.

it is possible to manually infer the answer by looking at the tree and navigating down to a leaf node.

The main thing to be aware of when constructing a decision tree is to limit the possibility of over fitting. If the boundaries of the tree are not limited, one might end up with a tree that fit the training data so well that it will struggle on data that differ only a little bit from the one it was trained on. Luckily there are some parameters that can be passed into the method to counter the possibility of over fitting. These are min samples leaf, that sets a lower boundary for how many samples you need to have before you can construct a leaf node. Closely related to this you have the min samples split, that sets a lower limit for how many samples a node needs to have before you can split it. This ensures that no splits are done on nodes that have very few samples. Lastly, the one that is used here, is the max depth parameter, limiting how deep the constructed tree can be.

Another small modification that must be done when dealing with the scikit library is that groups need to be renamed to numerical values, there G0 is labeled 0, G1 labeled 1 and so on.

The construction of the tree is not limited too much, only setting a max_depth = 12, for the rest the construction the tree is allowed to govern itself, as this seemed to give the best results. Other than that, the default value for invalid features is set to 1. Outlier time series are removed from the training data set, using the parameters described in section 3.7.

The Scikit implementation of DecisionTreeClassifier [29] is used to build the tree, and the following arguments were set.

```
clf = tree.DecisionTreeClassifier(max_depth=12, random_state=0)
```

3.12 Random forest

Random forest work by constructing a set number of randomly generated decision trees, then having a majority vote to decide on what sensor to infer it as. This approach should yield better results than the decision tree, without taking much longer to train. Given that it does not take a lot of time to construct a random forest tree, one can play around with a high number of estimators without worrying about growing old of age. There is however a limit where increasing the number of estimators do not have an impact on the final result. The difference between using a 1000 estimators, and using 10 000 estimators is as little as 0.001, or 0.1 percent. Making the model correctly infer an additional 1 in a 1000 time series. Even the difference between 400 estimators and 1 000 estimators is as little as 0.0004, or 0.04 percent. Therefore, a total number of 400 estimators were settled on in the random forest model.

The scikit module RandomForestClassifier [30] is used to construct the random forest model, and the following arguments were used:

```
clf = RandomForestClassifier(n_estimators=400, max_depth=16,
      random_state=0, verbose=10)
```

Verbose is set to 10 so that Scikit displays the progress during construction. This goes for all methods where this is useful.

3.13 Gradient boosting

Gradient boosting is a common and efficient technique used when construction classifier models. This thesis covers two gradient boosting approaches, namely the Gradient boosting tree and Histogram Gradient boosting. Gradient boosting combines several weak learners and combine them into a strong learner. This is done by iteratively building new learners, where each subsequent learner focuses on correcting the errors of the last one. The final estimate can be expressed as in equation (3.7) taken from [31]. Where a Function takes a input value, and the class is estimated by an initial guess and the sum of all the subsequent functions.

$$F_M(X) = F_0(X) + \sum_{m=1}^{M} F_M(x) \tag{3.7}$$

3.14 Gradient boost tree

A gradient boost tree carries some resemblance to a decision tree, but have a slightly different approach to inferring the results. While a decision tree uses a leaf node to decide on the sensor, a gradient boost tree use lots of trees to gradually come closer to the result. It starts with an initial guess (e.g. average of all the training data), then builds trees iterably continuously attempting to minimize the loss function (the average distance from the correct answer). The trees are also assigned a learning rate, that carries the weight of each subsequent tree.

Figure 3.8 shows an illustration of a gradient boost tree, and how subsequent trees help adjust the initial estimate. V_0 is the initial guess, often just the average value for what one want to infer. Then more trees are built, often of greater depth than what is visualized here. Splits are based on feature values the same way a decision tree does, prioritizing splits that have a high "gain". The leaf nodes have an adjustment value, that will be multiplied with the learning rate and added to the previous estimate. When the values from each sub-tree is added together the classification is complete.

One of the main drawbacks of this method is that it takes a lot longer to finalize this model compared to its counterparts like decision tree and random forest. This, however, is only for training, and using the model for inferring is as fast as any other approach. It does however make testing different combinations of arguments more time consuming.

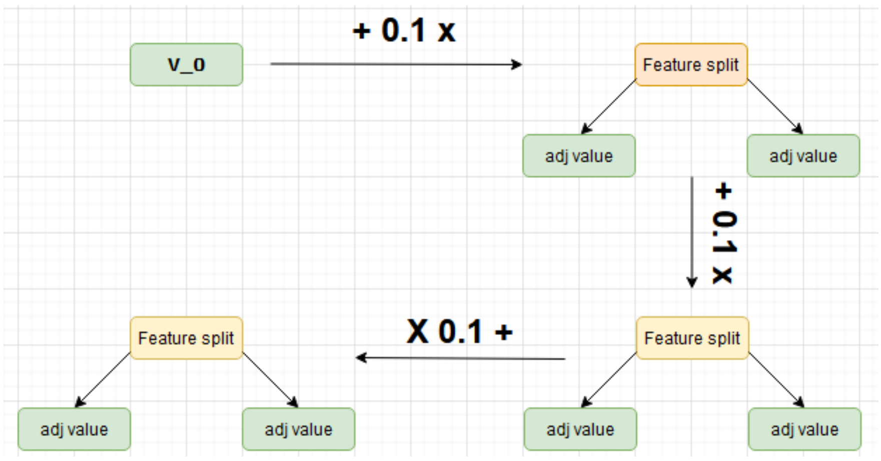


Figure 3.8: Illustration of gradient boosting tree. Inspired by [32].

Scikits GradientBoostingClassifier [33] were used to build the gradient boosted tree, and the following arguments were set when building the model:

```
clf = GradientBoostingClassifier(n_estimators=400, learning_rate=0.1,
        max_depth=6, random_state=0, verbose=10)
```

3.15 Histogram gradient boosting tree

Histogram gradient boosting is a much faster to build than for instance gradient boosting trees, when the number of samples for training grows large. The Scikit histogram gradient boosting also have support for Nan-values, but to keep the playing ground level the previous approach of replacing NaN-values with 1 is also kept for this method.

The Histogram Gradient boosting tree resembles the Gradient boosting tree, but has a different approach to figuring out splits that enables it to construct the model a lot faster when the feature space grows large. Scikits implementation is inspired by LightGBM [34], and map the continuous feature values into discrete bins. The bins are then used to construct feature histograms and are utilized to find the best splits efficiently.

Scikits HistGradientBoostingClassifier [31] builds the model and the following arguments were set.

```
clf = HistGradientBoostingClassifier(max_iter=1000, verbose=10,
      random_state=0, learning_rate=0.02, early_stopping=False)
```

3.16 Inference by majority vote

Examining the test data one can see that several time series originate from the same sensor, therefore it is of interest to see how well gradient boosting works in identifying the sensors based on all the available data. So instead of treating each sample as its own sensor, they will be grouped together by their original sensor in the test data set. Each sample is then inferred as done previously, but instead of directly inferring based on this result, it is added to a list with all other samples belonging to this particular sensor. Then the sensor will be inferred on the majority vote of the inferred samples in this list. Here three possible outcomes are allowed. Correct, when the correct sensor was infered either directly (when only one time series is available) or by a majority vote. Wrong, when the wrong sensor was infered either directly or by majority vote. Lastly inconclusive results are introduced, when the majority vote could not be settled. A majority vote could not be settled when there are two or more sensors with the same amount of votes, and have more votes than any other suggestions. This approach was tested for all methods except simple vector search.

Since each sample is no longer treated as its own sensor the number of sensors to infer is reduced. Table 3.6 show how this effect the distribution of samples. Going from a total of 2679 samples to only 302.

Note that the only difference done here is to summarize the inferred suggestions, and working out the majority vote. No changes has been done to arguments for different models.

3.17 Investigating wrongly inferred files

After having constructed a variety of models, it is interesting to study some of the time series that could not be correctly inferred. There are two types of errors that are of particular interest. The ones where a member of G_n often is inferred as a

Group	#Files
G0	16
G1	102
G2	31
G3	35
G4	29
G5	23
G6	31
G7	7
G8	27
G9	1
Total	302

Table 3.6: Grouped time series distribution

member of G_M, as this can help identify groups that resemble each other more than normal. Additionally, investigating time series where only one member of G_n was inferred as a member of G_m as this can typically be outliers.

Chapter 4

Results

4.1 Simple vector search

Table 4.1 and table 4.2 summarize the results from the simple vector search approach. Correct describes the case when the nearest average sensor was the correct one. Wrong is when the nearest average sensor was of another type.

Sensor name	Correct	Wrong	Accuracy
G0	123	23	0.842
G1	511	498	0.506
G2	0	242	0.000
G3	19	270	0.066
G4	29	300	0.088
G5	31	143	0.178
G6	256	63	0.803
G7	26	42	0.382
G8	101	1	0.990
G9	1	0	1.000
Total	1097	1582	0.409

Table 4.1: vector similarity results

Additionally, what the faulty suggestions entail have been recorded, showing the distribution of all suggestions. This is summarized in table 4.2. Reading the rows of the table show how members of a group was distributed across all groups. The columns give an overview over how many times this group was inferred, and where these suggestions came from.

Overall this approach perform better than random guessing, but the results are still very weak for what it tries to achieve. Most interestingly one can see that no sensor belonging to G2 was correctly inferred, commonly being mistaken for G1 and G0.

Sensor	G0	G1	G2	G3	G4	G5	G6	G7	G8	G9
G0	123	9	0	0	0	0	11	2	1	0
G1	430	511	0	0	0	1	51	8	8	0
G2	18	48	0	0	0	55	88	0	33	0
G3	49	79	0	19	2	5	110	6	19	0
G4	29	126	0	9	29	2	118	1	15	0
G5	10	54	0	0	1	31	62	0	16	0
G6	187	37	0	0	0	5	256	0	4	0
G7	1	21	0	0	0	1	3	26	1	15
G8	0	0	0	0	0	0	1	0	101	0
G9	0	0	0	0	0	0	0	0	0	1

Table 4.2: vector search inferred.
Reading the rows of the table you can see how the sensors belonging to a particular group was infered across all groups, while if you read the columns you'll see how many times the sensor was infered from each group.

4.2 K neighbours

For the K nearest neighbours approach the accuracy increased greatly from the simple vector search approach. The results are summarized in table 4.3 and table 4.4.

Sensor name	Correct	Wrong	Accuracy
G0	94	52	0.644
G1	977	32	0.968
G2	102	140	0.421
G3	153	136	0.529
G4	200	129	0.608
G5	75	99	0.431
G6	200	119	0.627
G7	23	45	0.338
G8	96	6	0.941
G9	0	1	0.000
Total	1920	759	0.717

Table 4.3: K-neighbours results

4.3 Decision tree

In figure 4.5 and 4.6 you can see the results of our decision tree. Most groups perform fairly well, only leaving G5 and G7 with a accuracy a fair bit lower than the other groups. A little bit surprising is the fact that a ordinary decision tree

Sensor	G0	G1	G2	G3	G4	G5	G6	G7	G8	G9
G0	94	41	4	4	1	2	0	0	0	0
G1	12	977	4	12	0	1	3	0	0	0
G2	1	8	102	24	10	36	49	1	11	0
G3	21	68	5	152	19	0	20	0	4	0
G4	0	17	20	23	200	46	18	0	5	0
G5	1	4	51	11	7	75	21	0	4	0
G6	7	0	36	26	27	21	200	0	2	0
G7	2	10	0	16	6	0	4	23	7	0
G8	0	0	6	0	0	0	0	0	96	0
G9	0	0	0	0	0	0	0	1	0	0

Table 4.4: K-neighbours inferred.
Reading the rows of the table you can see how the sensors belonging to a particular group was infered across all groups, while if you read the columns you'll see how many times the sensor was infered from each group.

succeeds in predicting the Energy Sensor, since there are only 2 such files in the training set. One could think that the gain associated with giving this group a leaf node would be so small that it would not be considered beneficial.

Sensor name	Correct	Wrong	Accuracy
G0	100	46	0.685
G1	973	36	0.964
G2	180	62	0.743
G3	214	75	0.740
G4	280	49	0.851
G5	94	80	0.540
G6	255	64	0.799
G7	38	30	0.441
G8	102	0	1.000
G9	1	0	1.000
Total	2237	442	0.835

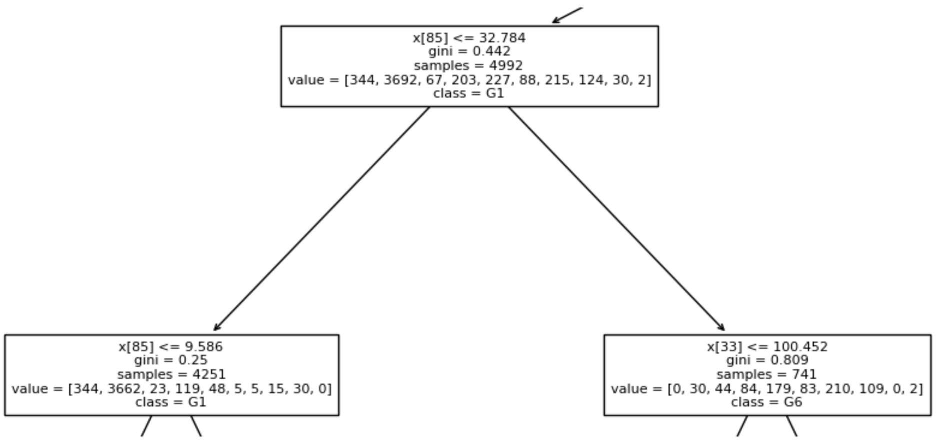
Table 4.5: Decision tree results

The tree itself grows a bit to big to be comfortably displayed here, but figure 4.1 shows what an small section of the tree looks like The first line in a node references the feature selected for a split, and the criterion for splitting. Line number two shows the Gini value, a value associated with how good the split is. This value reflects the gain from doing a split at this node and feature. A high value reflects that the split is useful for differentiating different groups. Third line shows the distribution of sensors at this node, since Scikit does not allow string values for classes during the construction, this is represented as a list, where each index is

Sensor	G0	G1	G2	G3	G4	G5	G6	G7	G8	G9
G0	100	38	1	5	0	2	0	0	0	0
G1	21	973	4	3	0	0	5	3	0	0
G2	0	0	180	16	4	20	21	0	1	0
G3	6	47	3	214	10	0	7	1	1	0
G4	0	9	1	31	280	0	3	5	0	0
G5	2	1	62	3	1	94	5	5	1	0
G6	0	4	37	13	1	9	255	0	0	0
G7	1	1	3	9	16	0	0	38	0	0
G8	0	0	0	0	0	0	0	0	102	0
G9	0	0	0	0	0	0	0	0	0	1

Table 4.6: Decision tree inferred.
Reading the rows of the table you can see how the sensors belonging to a particular group was infered across all groups, while if you read the columns you'll see how many times the sensor was infered from each group.

associated with a sensor. Lastly one can see what class this node belongs to, that is the most populous sensor at this node.


x[85] <= 32.784
gini = 0.442
samples = 4992
value = [344, 3692, 67, 203, 227, 88, 215, 124, 30, 2]
class = G1
x[85] <= 9.586
gini = 0.25
samples = 4251
value = [344, 3662, 23, 119, 48, 5, 5, 15, 30, 0]
class = G1
x[33] <= 100.452
gini = 0.809
samples = 741
value = [0, 30, 44, 84, 179, 83, 210, 109, 0, 2]
class = G6
Figure 4.1: Small sample of the generated decision tree

4.4 Random forest

Unsurprisingly the Random forest model does achieve better results than the ordinary decision tree. This can be related to the Condorcet Jury Theorem [35], that states if an independent group of deciders, all with more than 50 percent chance of having correct, then the accuracy will tend to increase with more deciders. Seeing that the decision tree has more than 80 percent accuracy total, and more than

50 percent accuracy for most sensor groups, one should also expect our random forest tree to do better than that. While it could occur to expect even higher accuracy than what is seen here, one need to remember that some time series are abnormal for the group that they belong too, and therefore might not be infered by any, or only just a few deciders.

Also here as with the decision tree it seems that the accuracy for G5 and G7 is a bit behind the other groupings. Mainly being misinterpreted as members of G2 and G4 respectively. However, the accuracy does increase a fair amount from the single decision tree. Also note that for the random forest classifier it did not succeed in inferring the Energy sensor (G9).

Sensor name	Correct	Wrong	Accuracy
G0	120	26	0.822
G1	996	13	0.987
G2	206	36	0.851
G3	216	73	0.747
G4	301	28	0.915
G5	115	59	0.661
G6	261	58	0.818
G7	43	25	0.632
G8	102	0	1.000
G9	0	1	0.000
Total	2360	319	0.881

Table 4.7: Random forest results

Sensor	G0	G1	G2	G3	G4	G5	G6	G7	G8	G9
G0	120	24	1	1	0	0	0	0	0	0
G1	6	996	0	3	0	0	4	0	0	0
G2	0	0	206	2	2	18	13	0	1	0
G3	5	52	1	216	7	0	4	0	4	0
G4	0	10	3	9	301	0	6	0	0	0
G5	0	0	49	6	0	115	3	0	1	0
G6	0	0	8	30	0	20	261	0	0	0
G7	0	1	5	2	17	0	0	43	0	0
G8	0	0	0	0	0	0	0	0	102	0
G9	0	0	0	0	0	0	0	1	0	0

Table 4.8: Random forest inferred.
Reading the rows of the table you can see how the sensors belonging to a particular group was infered across all groups, while if you read the columns you'll see how many times the sensor was infered from each group.

4.5 Gradient boosted tree

Gradient Boost tree is an extension of the ordinary decision tree where new trees are created iterably, trying their best to improve upon the results in the previous tree. For this method several parameter combinations were tested, to find the ones that yielded the best results. Several parameter combinations gave similar results, but the best one was with a learning rate of 0.1, max depth equal to 6 and 400 estimators. The results are summarized in the tables 4.11

Sensor name	Correct	Wrong	Accuracy
G0	135	11	0.925
G1	999	10	0.990
G2	203	39	0.839
G3	234	55	0.810
G4	313	16	0.951
G5	120	54	0.690
G6	279	40	0.875
G7	43	25	0.632
G8	102	0	1.000
G9	1	0	1.000
Total	2429	250	0.907

Table 4.9: Gradient boost tree results

Additionally, what the faulty suggestion entail have been recorded, meaning what other sensor group they were infered as, this is summarized in 4.10. Reading the rows of the table you can see how the sensors belonging to a particular grouped was infered across all groups, while if you read the columns you'll see how many times the sensor was infered from each group.

Gradient boosting takes considerably more time to construct than the other approaches thus far, but also yields the best results. While it takes a couple of minutes to construct a random forest classifier with 400 deciders, it takes closer to 7 hours to do the same for a gradient boosting classifier.

4.6 Histogram Gradient boosting

The histogram classifier resembles the gradient boost tree, but takes a different approach to figuring out what splits to make. This approach also results in a much faster construction of the classifier, taking around 30 minutes to complete. The methods also seems to be able to find better splits for the classifier, resulting in a slight increase in accuracy of about 1 percent.

Overall there is a high degree of accuracy for the gradient boosting methods, but it is apparent that also these are struggling the most with classifying members of G5 and G7.

Sensor	G0	G1	G2	G3	G4	G5	G6	G7	G8	G9
G0	135	11	0	0	0	0	0	0	0	0
G1	4	999	0	1	0	0	5	0	0	0
G2	1	0	203	6	0	21	11	0	0	0
G3	9	33	1	234	5	0	5	0	2	0
G4	0	8	0	4	313	1	3	0	0	0
G5	0	0	47	1	0	120	5	0	1	0
G6	0	1	15	7	2	15	279	0	0	0
G7	0	1	1	6	17	0	0	43	0	0
G8	0	0	0	0	0	0	0	0	102	0
G9	0	0	0	0	0	0	0	0	0	1

Table 4.10: Gradient boost tree inferred.
Reading the rows of the table you can see how the sensors belonging to a particular group was infered across all groups, while if you read the columns you'll see how many times the sensor was infered from each group.

Sensor name	Correct	Wrong	Accuracy
G0	132	14	0.904
G1	1000	9	0.991
G2	209	33	0.864
G3	239	50	0.827
G4	309	20	0.939
G5	125	49	0.718
G6	297	22	0.931
G7	43	25	0.632
G8	102	0	1.000
G9	1	0	1.000
Total	2457	222	0.917

Table 4.11: Histogram gradient boosting results

4.7 Majority voting

The above mentioned approaches was all, with the exception of vector similarity, tested with a majority voting as well. The results from this is briefly summarized in table 4.13.

More thorough results for the histogram gradient boost model are listed in table 4.14 and the distribution of inferred files in table 4.15, as this was the best achieving model. Keep in mind that when the accuracy is calculated the inconclusive results are treated as wrong. Therefore the accuracy is calculated using the formula seen in equation 4.1

Sensor	G0	G1	G2	G3	G4	G5	G6	G7	G8	G9
G0	132	14	0	0	0	0	0	0	0	0
G1	5	1000	0	2	0	0	2	0	0	0
G2	0	0	209	5	0	23	5	0	0	0
G3	7	32	1	239	4	0	4	0	2	0
G4	0	8	0	9	309	0	3	0	0	0
G5	0	0	39	5	0	125	4	0	1	0
G6	0	1	4	7	1	9	297	0	0	0
G7	0	1	1	6	17	0	0	43	0	0
G8	0	0	0	0	0	0	0	0	102	0
G9	0	0	0	0	0	0	0	0	0	1

Table 4.12: Histogram boosting inferred.
Reading the rows of the table you can see how the sensors belonging to a particular group was infered across all groups, while if you read the columns you'll see how many times the sensor was infered from each group.

$$accuracy = \frac{correct}{correct + wrong + unresolved} \tag{4.1}$$

Sensor name	Correct	Wrong	Unresolved	Accuracy
K-neighbours	229	57	16	0.758
Decision tree	264	31	7	0.874
Random forest	269	26	7	0.891
Gradient boost tree	277	20	5	0.917
Histogram gradient boost	281	15	6	0.927

Table 4.13: Majority voting results

Keep in mind that sensors that were unconclusive are not listed in table 4.15, so that the numbers will add up to 297 here, instead of 302.

One thing to note here is that accuracy increased for most of the groupings, but actually dropped a considerable amount for G5 and G7, the groups that have been consistently the most difficult to classify correctly. This might point towards some of the sensors behaving fairly differently from what can be observed in the training data, and that the time series from these individual sensors are faulty or different in some way.

4.8 Wrongly infered time series

Looking at some of the time series that were wrongly infered, and investigate how their plots might differ from what one would expect to see based on other time series in the test data. Expectation is that quite a few of these will be faulty in the

Sensor name	Correct	Wrong	Unresolved	Accuracy
G0	15	0	1	0.938
G1	102	0	0	1.000
G2	29	1	1	0.935
G3	31	3	1	0.886
G4	28	1	0	0.966
G5	14	6	3	0.609
G6	28	3	0	0.903
G7	4	3	0	0.571
G8	27	0	0	1.000
G9	1	0	0	1.000
Total	279	17	6	0.927

Table 4.14: Histogram boosting with majority vote results.

Sensor	G0	G1	G2	G3	G4	G5	G6	G7	G8	G9
G0	15	0	0	0	0	0	0	0	0	0
G1	0	102	0	0	0	0	0	0	0	0
G2	0	0	29	0	0	1	0	0	0	0
G3	1	2	0	31	0	0	0	0	0	0
G4	0	1	0	0	28	0	0	0	0	0
G5	0	0	5	0	0	14	1	0	0	0
G6	0	0	2	0	0	1	28	0	0	0
G7	0	1	0	1	1	0	0	4	0	0
G8	0	0	0	0	0	0	0	0	27	0
G9	0	0	0	0	0	0	0	0	0	1

Table 4.15: Histogram boosting with majority vote inferred.
Reading the rows of the table you can see how the sensors belonging to a particular group was infered across all groups, while if you read the columns you'll see how many times the sensor was infered from each group.

way that the plots will not look like one would expect based on the sensor types. That is common for real world data sets that have not been manually inspected and verified. Primarily investigating wrongly infered sensors from the Histogram Gradient classifier, as this had the highest accuracy.

Looking at G7 for the histogram classifier one can see that these sensors are often miss-inferred to be a member of G4. Looking at the different plots in figure 4.2 one can see why that might be, as a typical Power Sensor seem to be step wise, while the wrongly inferred time series seem to vary a lot up and down with more abrupt changes. Something that seems to align better with the plots seen of G4 on the right.

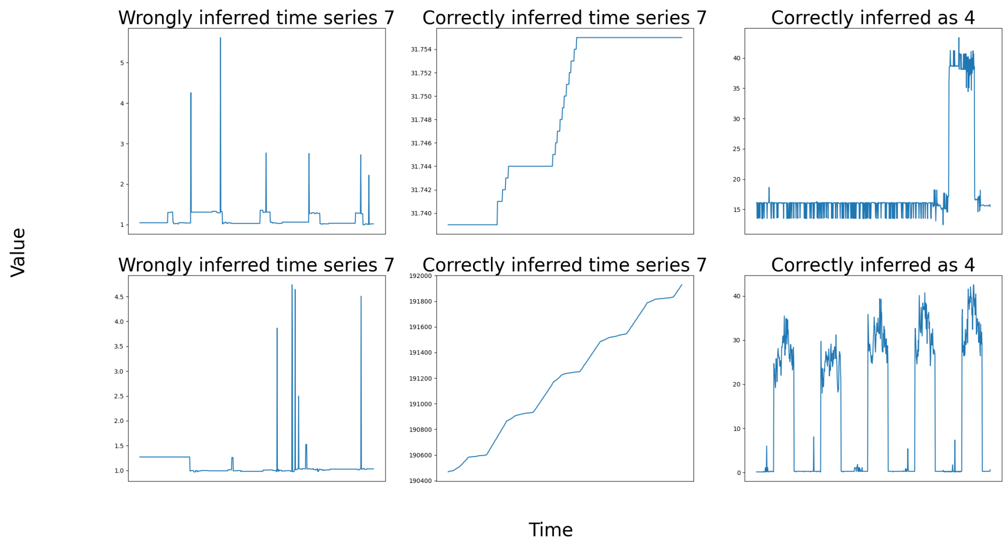


Figure 4.2: The two plots to the left represent two time series that was wrongly inferred. The two plots in the middle represent time series that were correctly inferred as members of G7. The two plots on the right shows two time series correctly inferred as members of G4.

While looking at the most common "mistake" for each grouping give a good indication in terms of groups that have the most similar feature characteristics, looking at the time series that were only inferred to be part of another group once can be a good way to locate time series where there is something wrong with the data. As seen in figure 4.3 where the member of G5 was miss-inferred as a member of G8, there seem to be something unfortunate about this time series, as it is constant.

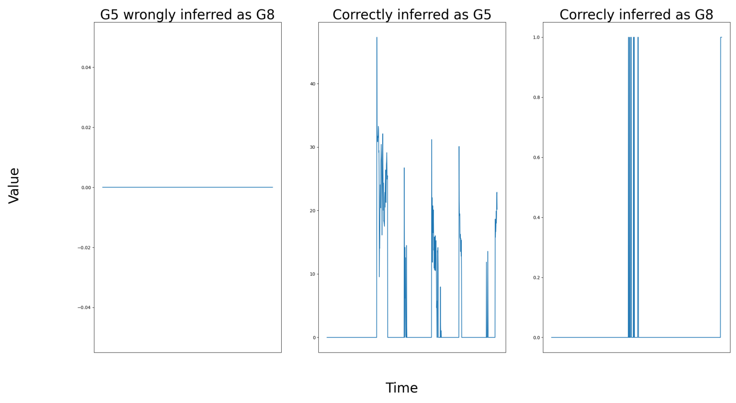


Figure 4.3: The plot to the left show the miss-inferred time series belonging to G5. The middle plot show a correctly inferred member of G5. Right plot show a correctly inferred member of G8.

Investigating further into the time series that was miss-inferred as a member of G8 and looking at the values within the feather file, one can see the reason for this unusual time series. The values are listed in table 4.16. The value is supposed to change to 100 a total of five times over the time series. Unfortunately, when using forward fill and standardizing the interval size to 10 minutes one ends up overwriting all of these values. Making a time series that appears to be constantly 0. This is an unfortunate case where the interval start align in such a way that no occurrence of the value 100.0 is the one to override the next 10 minute interval, and no occurrence of the value 100.0 last for a duration longer than 10 minutes. For sensor that frequently update their values, such as a temperature sensor, has a low probability for something like this happening. However, sensor that change values less frequently and for short amount of time are more likely to have this happen.

Timestamp	Value
2021-08-07 21:38:13.579000+00:00	0.0
2021-08-09 04:02:04.707000+00:00	100.0
2021-08-09 04:07:09.340000+00:00	0.0
2021-08-10 04:02:08.696000+00:00	100.0
2021-08-10 04:07:09.349000+00:00	0.0
2021-08-11 04:02:07.328000+00:00	100.0
2021-08-11 04:07:08.110000+00:00	0.0
2021-08-12 04:02:08.869000+00:00	100.0
2021-08-12 04:07:09.564000+00:00	0.0
2021-08-13 04:02:08.009000+00:00	100.0
2021-08-13 04:07:15.799000+00:00	0.0

Table 4.16: Timestamps and corresponding values for Heat exchanger. File: 2021-32_RC_360009LX001_C.feather'

4.9 Fixed size intervals

Up until now the focus has been on creating models using the interval size of 10 minutes. In this section the models are tested on four different interval sizes, and compared. The question that is looking to be answered is if it is worth spending much time to perform feature extraction, or can similar accuracy be achieved by using bigger interval sizes? In addition to the 10 minute interval size the models are tested for interval sizes of 1 minute, 1 hour and 5 hours. Less frequent intervals mean smaller time series, and less time and resources spent extracting characteristics. One thing to keep in mind is that not all interval sizes will be able to extract features from every time series, especially with 1 minute intervals some files require to much ram for us to be able to do so. Even smaller interval sizes would likely increase the number of time series that fails feature extraction.

In table 4.17 summarize how the interval size effected the accuracy of each model.

Interval size	1 minute	10 minutes	1 hour	5 hours
K neighbours	0.701 (0.696)	0.717	0.691	0.696
Decision tree	0.843 (0.837)	0.835	0.798	0.791
Random forest	0.895 (0.889)	0.881	0.859	0.834
Gradient boost tree	0.920 (0.913)	0.907	0.885	0.856
Histogram gradient boost	0.933 (0.926)	0.917	0.894	0.863

Table 4.17: Accuracy for different models with the different interval sizes. Parenthesis denote the accuracy if the missing files were added and treated as wrongly inferred.

As mentioned in 3.4 the 1 minute interval size leave us with fewer files both for training and testing, something that might give a skewed impression of the accuracy. Therefore, an additional accuracy have been calculated, written in parenthesis. This accuracy takes the number of correctly inferred files and divide them by the total number of files in the testing data. In this case all files that could not have their features extracted are automatically treated as wrong, giving a worst case estimate.

4.10 Catch 22 feature extraction

Given that feature extraction using TSFresh for a fine granularity use a long time, on the scale of weeks for 1 minute intervals (for the entire data set), it is interesting to examine a faster library for feature extraction, to see how it compares. Therefore, features were extracted on the same intervals as with TSFresh, using Catch 22. Each model were tested with these features to see how it compares. The results can be seen in table 4.18. Also here the same outliers that was found using TSFresh with a 10 minute interval has been utilized.

Interval size	1 minute	10 minutes	1 hour	5 hours
K neighbours	0.607	0.713	0.729	0.729
Decision tree	0.787	0.774	0.739	0.739
Random forest	0.882	0.858	0.831	0.831
Gradient boost tree	0.879	0.854	0.828	0.828
Histogram gradient boost	0.895	0.869	0.839	0.839

Table 4.18: Accuracy for different models with the different interval sizes, using the Catch22 feature extraction library

Chapter 5

Discussion

When building a model for inference many of the important choices are done on how to manipulate the data that is used for training and testing. This is rarely something the model itself is able to handle, and therefore it is up to the user to make these decisions.

5.1 Results

Unsurprisingly the simpler approaches such as vector search, decision tree and k-nearest neighbours were outperformed by approaches utilizing gradient boosting. One explanation for this can be that Gradient boosting allows for gradual approach to the correct answer, rather than deciding on the first iteration, or by a simple majority vote. When using a decision tree the leaf node has the final say in what the sensor is to be inferred as, while with gradient boosting it is the sum of the values in the leaf node that collectively decide. This also allows for some of these leaf nodes to give suggestions that might not be correct for all sensors within a group, but in a subsequent tree being able to rectify this.

K-nearest neighbours did perform surprisingly well on the TSFresh extracted features, given that a large number of these features can be considered uninformative, meaning that the could have a disruptive effect when calculating distance. It seems likely that k-nearest neighbours would perform better if feature selection was done, and only features that are descriptive of sensors were kept as part of the data set. K-nearest neighbours have no way of determining feature significance and therefore all features are weighted the same. The impact of each feature is only dependant on its value, and where it places in relation to the other data points in the feature space. The same problem can be said to be with the simple vector search, and the reason it performs so poorly is that a large number of the features are uninformative by themselves, and no approach is taken to weigh them differently. Another draw back of this simple implementation is that even though outlier time series are removed from the training set, individual outlier feature values are not removed when calculating the average vectors. The IQR-approach could have been used here as well, only using feature values between the upper and lower limit to calculate the average vectors. The other approaches that were tested all calculate some form of gain to figure out the next best approach, and is by that able to ignore features that are not descriptive of a sensor grouping.

While it is often advised to limit models such as a decision tree with max leaf size and minimum samples split to combat overfitting, this did not have a positive effect on accuracy for this data set. While the max depth was limited to 12, to keep the tree from growing very large, limiting the splits or samples in leaf seemed to reduce accuracy. The reason for this is not obviously clear, but a possible explanation for this could be that the data in the training set and test set originate from many of the same buildings. Manning that a certain level of overfitting is beneficial, since the time series in the test set have a lot in common with the training set. Utilizing the models on data from different buildings, countries and climates might make it perform significantly worse. Random forest that is a combination of multiple decision trees managed an accuracy a little less than 5 percent higher than the decision tree, and was in line with what one would expect.

Lastly there is the gradient boosting models, that also achieved the highest classification accuracy. Both approaches are comparable in terms of their successful prediction, with the histogram based approach scoring 1 percent higher, and yielding the best results for independent sample classification. Histogram gradient boosting also achieved the highest accuracy for majority voting, increasing its accuracy by 1 percent over the individual sample predictions. The fact that the increase in accuracy for majority voting was relatively small might point towards that many of the time series that were miss-inferred generally contain data that does not align well with the same sensor-type based on samples from the training set. Thereby being inherently difficult to classify correctly.

When working with real world data it is unlikely that a model is able to reach 100 percent accuracy unless one have strict criteria for what type of time series one is willing to infer. Thereby disqualifying a sample from being tested. Things to look after could be making sure that the number of data points is at a certain minimum, ensuring that enough information is in place. Additionally one could verify that the time series is not an outlier for all groups within the set of groups. If a sample is an outlier for all groups then it is likely that any inference made will be wrong.

5.2 Interval sizes

For all methods, bar the K-nearest neighbours, one can see that accuracy increase as the interval size decreases. Even the worst case scenario for the 1 minute interval, still outperform the accuracy of the 10 minute interval. This is as one would expect as a smaller interval size will be a more accurate portrayal of the actual time series. However, given the large amount of time it takes to extract features for the training set, around 10 days, it was difficult to extract for smaller interval sizes with the resources available. The reason for K-nearest neighbours jumping a bit back and forth in terms of accuracy is also not unexpected as it is reliant on information gained from its neighbours, and when a large amount of features are uninformative, it might disrupt the inference. One issue with the way features are handled, without any filtering of informative features, for the K-nearest neighbours approach is that the many uninformative features are not removed. That is features that might vary much even within the same sensor grouping, and these values will have an effect on the position in space, and therefore also the distance to the sensor one would like to infer. TSFresh do have implementation to filter out the most descriptive features, something that is likely to have increased the accuracy for both vector search and k-nearest neighbours.

5.3 Catch22 feature extraction

Table 4.18 the results for classification based on catch22 is summarized. Most of the results are unsurprising for the majority of models. However, given that these features supposedly are highly expressive, it is surprising to not see any notable increase in accuracy for the distance based K-nearest neighbours approach. Given that there should be very few non-expressive features in the data, one would expect distance based inference to achieve better results than with all the noise present in TSFresh's extracted features.

5.4 Inferred files

When looking at the inference tables 4.2, 4.4, 4.6, 4.8, 4.10, 4.12, 4.15, one can see that G9 generally is a good group in the sense that other sensors relatively rarely is inferred to be part of this group, meaning that the characteristics are generally unique enough to be filtered out. One thing to keep in mind for this group in particular though is that both the training set and testing set is very small for the energy sensor, having only two time series for training and one for testing. The, in total, three time series for the energy sensor also originate from the same building, and are part of the same system, making them very similar. G8, pump, is also fairly good at inferring the correct files, and at least for gradient boosting methods few other Groups are mistaken to be part of group 8.

Generally for all models one can see that G5 and G7 are the groups with the lowest accuracy, and looking at the inferred tables for the gradient boosting approaches one see that G5 is most commonly mislabeled as G2. Meaning that the heat exchangers and Variable frequency drives can be mistaken to be a Valve belonging to G2. G7 is most commonly mislabeled as to be part of G4, meaning that the Power sensor can be mistaken to be a Air pressure sensor.

Something interesting was also that, even when using majority voting, the groups G5 and G7 under-performed compared to the other categories, and looking into the distribution of the inferred files further one can see that some sensors have 0 time series associated with themselves that are correctly inferred. Among these are the heat exchanger with the following code 360001LX471_C. This sensor have

12 time series associated with itself, where, out of these 12, 7 were miss-inferred as members of G2, 3 as members of G6 and the last two as members of G3.

The same goes for the power sensor s434005OE001_Effekt, where all 12 were miss-inferred as members of G4. From this it becomes apparent that some of these sensors behave abnormally compared to their sensor group, and is a big reason for G5 and G7 underperforming in terms of accuracy.

5.5 Testing

While the gradient boosting approaches showed some very promising results, being able to classify more than 90 percent of the time series present in the test set, there are some considerations to make when reading the results. Primarily that the methods are all trained and tested on static data sets. In turn this can cause parameter tuning that is reliant on this exact data, and exposes the process to the possibility of overfitting. To verify the integrity of these methods one should be able to test it on more data. This is the same concern that is raised by Anthony Bagnall et al. in their paper "The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances" [18]. Where classification algorithms are tested on a single static data set that fits well with the task the given algorithm is well suited at handling. This in turn make algorithms seem better than they generally are.

5.6 Future work

For this project to expand into a complete system it is encouraged to develop methods to detect time series that are likely to not be accurately inferred. To do this one should be able to detect time series that behave abnormally. This can be time series that have a lot of values one would not expect, like dropping abruptly outside of normal range as seen in figure 3.1. Also to ensure that enough information can be obtained there could be a minimum number of data points in a time series before executing classification. There is also room to investigating further approaches for classification. Especially neural networks and deep learning approaches is interesting to compare against the mostly tree-based methods tested in this thesis.

Bibliography

[1] *Piscada*, `https://www.piscada.com/`, Accessed: 2023-06-10.

[2] A. Fylling, *Pa 0802 tverrfaglig merkesystem (tfm)*, `https://dok.statsbygg.no/wp-content/uploads/2022/10/PA-0802-Tverrfaglig-merkesystem-TFM.pdf`, Accessed: 2023-05-19, 2017.

[3] T. Mosleth, *Pa 0802 vedlegg 9.1 – systemkodeliste*, `https://dok.statsbygg.no/wp-content/uploads/2021/02/PA-0802-Vedlegg-9.1-Systemkodeliste.pdf`, Accessed: 2023-05-30, 2020.

[4] T. Mosleth, *Pa 0802 vedlegg 9.2 – komponentkodeliste*, `https://dok.statsbygg.no/wp-content/uploads/2021/02/PA-0802-Vedlegg-9.2-Komponentkodeliste.pdf`, Accessed: 2023-05-30, 2020.

[5] A. Fylling, *Pa 0805 bruk av standard norges tverrfaglig merkesystem (ns-tfm) i statsbygg*, `https://dok.statsbygg.no/wp-content/uploads/2022/10/PA-0805-Bruk-av-Standard-Norges-Tverrfaglig-Merkesystem-NS-TFM-i-Statsbygg.pdf`, Accessed: 2023-06-04, 2022.

[6] M. Christ, N. Braun, J. Neuffer and A. W. Kempa-Liehr, 'Time series feature extraction on basis of scalable hypothesis tests (tsfresh–a python package),' *Neurocomputing*, vol. 307, pp. 72–77, 2018.

[7] M. Christ, A. W. Kempa-Liehr and M. Feindt, 'Distributed and parallel time series feature extraction for industrial big data applications,' *arXiv preprint arXiv:1610.07717*, 2016.

[8] Y. Chen, E. Keogh, B. Hu, N. Begum, A. Bagnall, A. Mueen and G. Batista, *The ucr time series classification archive*, `www.cs.ucr.edu/~eamonn/time_series_data/`, Accessed: 2023-06-06, Jul. 2015.

[9] B. D. Fulcher and N. S. Jones, 'Highly comparative feature-based time-series classification,' *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 12, pp. 3026–3037, 2014.

[10] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann and E. Keogh, 'Experimental comparison of representation methods and distance measures for time series data,' *Data Mining and Knowledge Discovery*, vol. 26, pp. 275–309, 2013.

[11] M. Fernández-Delgado, E. Cernadas, S. Barro and D. Amorim, 'Do we need hundreds of classifiers to solve real world classification problems?' *The journal of machine learning research*, vol. 15, no. 1, pp. 3133–3181, 2014.

[12] Y. Freund and R. E. Schapire, 'A desicion-theoretic generalization of on-line learning and an application to boosting,' in *Computational Learning Theory: Second European Conference, EuroCOLT'95 Barcelona, Spain, March 13–15, 1995 Proceedings 2*, Springer, 1995, pp. 23–37.

[13] M. Barandas, D. Folgado, L. Fernandes, S. Santos, M. Abreu, P. Bota, H. Liu, T. Schultz and H. Gamboa, 'Tsfel: Time series feature extraction library,' *SoftwareX*, vol. 11, p. 100 456, 2020.

[14] *List of available features*, `http://www.cs.ucr.edu/~eamonn/time_series_data/`, Accessed: 2023-06-05.

[15] C. H. Lubba, S. S. Sethi, P. Knaute, S. R. Schultz, B. D. Fulcher and N. S. Jones, 'Catch22: Canonical time-series characteristics: Selected through highly comparative time-series analysis,' *Data Mining and Knowledge Discovery*, vol. 33, no. 6, pp. 1821–1852, 2019.

[16] B. D. Fulcher and N. S. Jones, 'Hctsa: A computational framework for automated time-series phenotyping using massive feature extraction,' *Cell systems*, vol. 5, no. 5, pp. 527–531, 2017.

[17] L. Grinsztajn, E. Oyallon and G. Varoquaux, 'Why do tree-based models still outperform deep learning on tabular data?' *arXiv preprint arXiv:2207.08815*, 2022.

[18] A. Bagnall, J. Lines, A. Bostrom, J. Large and E. Keogh, 'The great time series classification bake off: A review and experimental evaluation of recent algorithmic advances,' *Data mining and knowledge discovery*, vol. 31, pp. 606–660, 2017.

[19] T. Górecki and M. Łuczak, 'Non-isometric transforms in time series classification using dtw,' *Knowledge-based systems*, vol. 61, pp. 98–108, 2014.

[20] A. Bagnall, J. Lines, J. Hills and A. Bostrom, 'Time-series classification with cote: The collective of transformation-based ensembles,' *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 9, pp. 2522–2535, 2015.

[21] H. Ismail Fawaz, G. Forestier, J. Weber, L. Idoumghar and P.-A. Muller, 'Deep learning for time series classification: A review,' *Data mining and knowledge discovery*, vol. 33, no. 4, pp. 917–963, 2019.

[22] T. Chen and C. Guestrin, 'Xgboost: A scalable tree boosting system,' in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[23] *Pyplot documentation*, `https://matplotlib.org/3.5.3/api/_as_gen/matplotlib.pyplot.html`, Accessed: 2023-06-11.

[24] M. C. et al., *Overview on extracted features*, https://tsfresh.readthedocs.io/en/latest/text/list_of_features.html, Accessed: 2023-03-25, 2023.

[25] B. D. Fulcher, *Featurelist*, https://github.com/DynamicsAndNeuralSystems/catch22/blob/main/featureList.txt, Accessed: 2023-04-13, 2023.

[26] H. Vinutha, B. Poornima and B. Sagar, 'Detection of outliers using interquartile range technique from intrusion dataset,' in *Information and Decision Sciences: Proceedings of the 6th International Conference on FICTA*, Springer, 2018, pp. 511–518.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, 'Scikit-learn: Machine learning in python,' *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[28] *Sklearn.neighbors.kneighborsclassifier*, https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html, Accessed: 2023-05-26.

[29] *Sklearn.tree.decisiontreeclassifier*, https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier, Accessed: 2023-05-26.

[30] *Sklearn.ensemble.randomforestclassifier*, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html, Accessed: 2023-05-26.

[31] J. Garcia, *Histograms for efficient gradient boosting*, https://robotenique.github.io/posts/gbm-histogram/, Accessed: 2023-05-24, 2020.

[32] J. S. Andrea Perlato, *Gbm(gradient boosting model)*, https://www.andreaperlato.com/theorypost/ensemble-learning-with-gradient-boosting/, Accessed: 2023-05-10, 2020.

[33] *Sklearn.ensemble.gradientboostingclassifier*, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html, Accessed: 2023-05-26.

[34] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye and T.-Y. Liu, 'Lightgbm: A highly efficient gradient boosting decision tree,' *Advances in neural information processing systems*, vol. 30, 2017.

[35] P. J. Boland, 'Majority systems and the condorcet jury theorem,' *Journal of the Royal Statistical Society: Series D (The Statistician)*, vol. 38, no. 3, pp. 181–189, 1989.

Appendix A

Additional Material

Relevant code can be found at `https://github.com/HavardRMinsas/Tag_Inference.git`