Programme Area: Smart Systems and Heat

Project: WP1 Appliance Disaggregation

Title: High Frequency Appliance Disaggregation Analysis Handover

Abstract:

The ETI collected utility meter and other data (e.g. room temperatures, humidity, and HEMS control data) from five dwellings over a period of six months. Using the collected data, work was conducted to evaluate different machine learning algorithms, research appropriate data features and calibrations thereof, and test the 'art of the possible'. The work sought not only to understand historical human activity within the building, but also to estimate probabilities of future hot water usage, occupancy and heating needs. The described work resulted in Baringa and ASI jointly developing several Python scripts and notebooks for ETI. This document sets out to explain the purpose of the various scripts and notebooks required to rerun the analysis, the relationship between the various files and explain how to run the code. The document starts with an overview of the workflow and code, before diving into a detailed description of each section of the workflow and the corresponding code.

Context:

The High Frequency Appliance Disaggregation Analysis (HFADA) project builds upon work undertaken in the Smart Systems and Heat (SSH) programme delivered by the Energy Systems Catapult for the ETI, to refine intelligence and gain detailed smart home energy data. The project analysed in depth data from five homes that trialed the SSH programme's Home Energy Management System (HEMS) to identify which appliances are present within a building and when they are in operation. The main goal of the HFADA project was to detect human behaviour patterns in order to forecast the home energy needs of people in the future. In particular the project delivered a detailed set of data mining algorithms to help identify patterns of building occupancy and energy use within domestic homes from water, gas and electricity data.

High Frequency Appliance Disaggregation Analysis Handover

CLIENT: Energy Technologies Institute

DATE: 14/05/2018

Version History

Version	Date	Description	Prepared by	Approved by
1.0	14/05/2018	Final handover documentation	Alberto Favaro, Cris Lowery, Zhihan Xu	Oliver Rix

Contact

Name Oliver.Rix@baringa.com +44 7790 017 576

Confidentiality and Limitation Statement

Contents

Figures

Tables

1 Introduction

At the time of writing, the ETI is investigating the development of a Home Energy Management System (HEMS) capable of optimising the comfort of a dwelling's residents while managing the necessary energy expenditure. As part of this initiative, they are investigating a system that can learn future patterns of occupancy and needs of its residents using non-intrusive monitoring equipment from two or more utilities. Three key differentiators of the work undertaken, as compared to prior "Non-Intrusive Appliance Load Monitoring" research, are:

1. Monitoring multiple utilities to provide more information and contextual knowledge;
2. Recording high frequency electricity data to provide additional information on current property state;
3. Potential use of priors to more effectively identify behavioural patterns of property states.

To facilitate the research, the ETI collected utility meter consumption data and other data (e.g. humidity, and HEMS control data) from five dwellings over a period of six months. Using a subset of the collected data, work has been conducted to evaluate different machine learning algorithm constructs, research potential data features and calibrations thereof, and test the "art of the possible". Crucially, the work conducted does not only seek to understand historical human activity within the building, but also to predict future needs.

The described work resulted in Baringa and ASI jointly developing several Python scripts and notebooks for ETI. The data pipeline and analysis has not been fully automated, as the focus was not on building a production system, but instead on understanding the art of the possible. Future effort could be directed at automating the workflow, but there will be challenge in automating certain areas of the workflow. This document sets out to explain the purpose of the various scripts and notebooks required to rerun the analysis, the relationship between the various files and explain how to run the code. The document starts with an overview of the workflow and code, before diving into a detailed description of each section of the workflow and the corresponding code.

2 Overview

This chapter provides various overviews relating to the work before the report dives into the detail of each section of the code and how to run it. The elements covered in this chapter are a high level view of the methodology, an overview of the workflow, an overview of the code and finally an overview of the data. Subsequent chapters are broken down as is displayed in Figure 2.

2.1 High-level methodology

ETI are investigating the development of a Home Energy Management System (HEMS) capable of optimising the comfort of a dwelling's residents while managing the necessary energy expenditure. As part of this initiative, they are investigating a system that can learn future patterns of occupancy and needs of its residents using non-intrusive monitoring equipment from two or more utilities. To more accurately test how well a system can learn future needs, it was necessary to define the problem more precisely. The problem was therefore framed as several predictive models, 5 for cumulative hot water usage and 5 for occupancy, one for each of five different time horizons. Figure 1 lays out the high-level methodology followed to go from the raw data to the two predictive model categories.

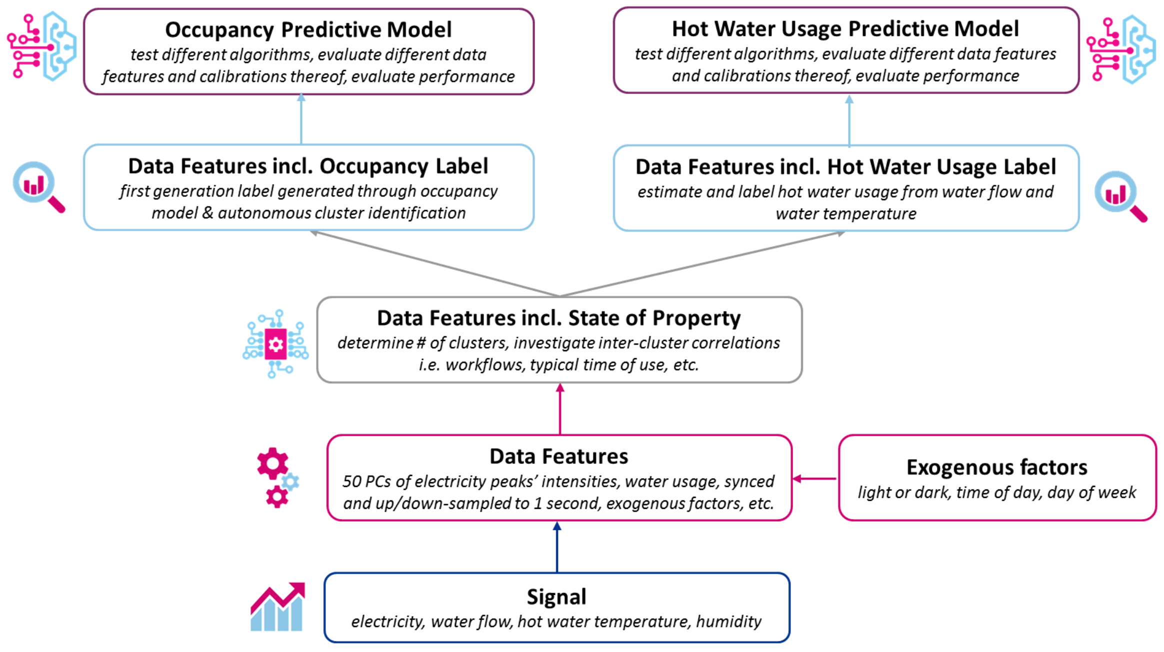
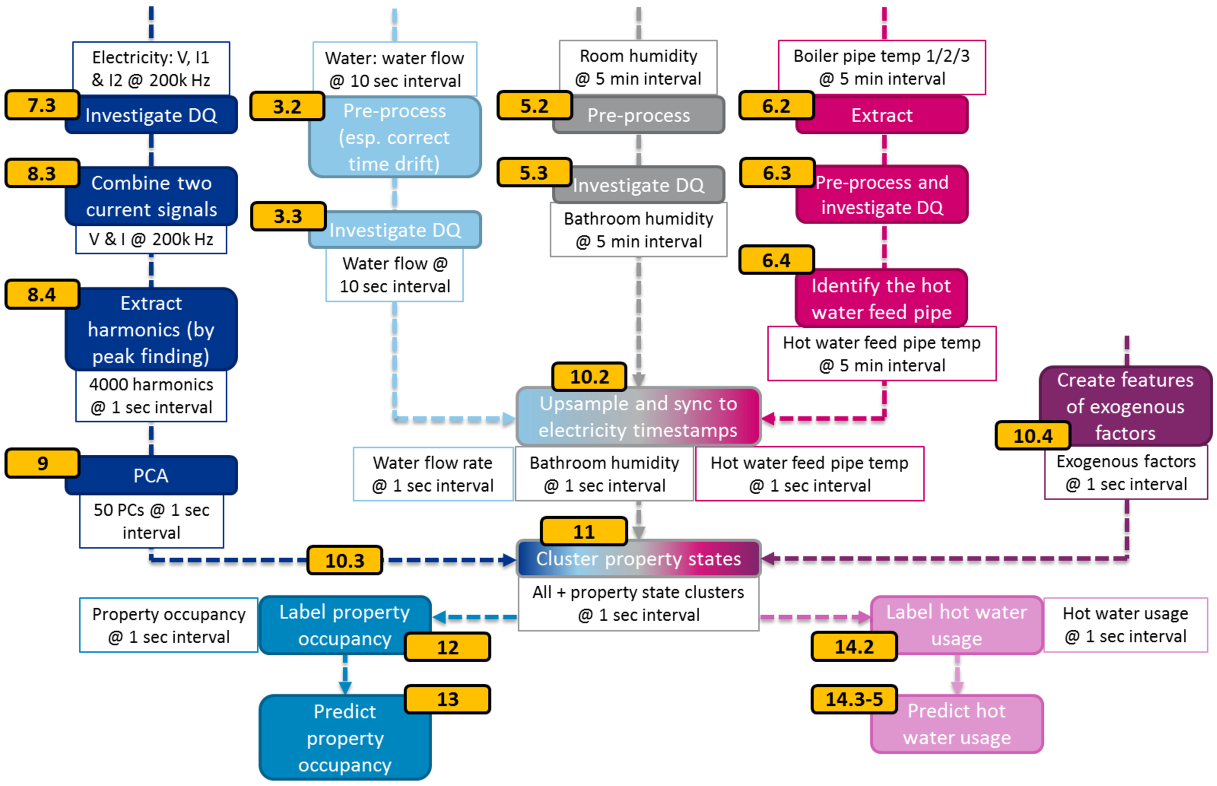

Occupancy Predictive Model
test different algorithms, evaluate different data features and calibrations thereof, evaluate performance
Hot Water Usage Predictive Model
test different algorithms, evaluate different data features and calibrations thereof, evaluate performance
Data Features incl. Occupancy Label
first generation label generated through occupancy model & autonomous cluster identification
Data Features incl. Hot Water Usage Label
estimate and label hot water usage from water flow and water temperature
Data Features incl. State of Property
determine # of clusters, investigate inter-cluster correlations i.e. workflows, typical time of use, etc.
Data Features
50 PCs of electricity peaks' intensities, water usage, synced and up/down-sampled to 1 second, exogenous factors, etc.
Exogenous factors
light or dark, time of day, day of week
Signal
electricity, water flow, hot water temperature, humidity

Figure 1: High-level methodology to produce predictive models

2.2 Workflow

In order to rerun the code, it is important to understand how the various notebooks and scripts link up. For this reason, Figure 2 illustrates the overall workflow of the project. In order to understand the details of any component of the workflow, please refer to the corresponding section of the report as labelled by the yellow boxes.

The project also leveraged SherlockML, a development environment for data science developed by ASI Data Science, which had the benefit of accelerating several of the computational tasks through its interface into Amazon Web Services. As such, some of the code has SherlockML dependencies as detailed in Table 1 i.e. the scripts relating to processing the electricity data including data quality investigations, extracting harmonics and clustering the electricity principal components. If these scripts are to be reused, they will require some tweaks to circumvent SherlockML and directly plug into the cloud processing platform of choice.



Electricity: V, I1 & I2 @ 200k Hz
7.3
Investigate DQ
8.3
Combine two current signals
V & I @ 200k Hz
8.4
Extract harmonics (by peak finding)
4000 harmonics @ 1 sec interval
9
PCA
50 PCs @ 1 sec interval
Water: water flow @ 10 sec interval
3.2
Pre-process (esp. correct time drift)
3.3
Investigate DQ
Water flow @ 10 sec interval
Room humidity @ 5 min interval
5.2
Pre-process
5.3
Investigate DQ
Bathroom humidity @ 5 min interval
Boiler pipe temp 1/2/3 @ 5 min interval
6.2
Extract
6.3
Pre-process and investigate DQ
6.4
Identify the hot water feed pipe
Hot water feed pipe temp @ 5 min interval
10.2
Upsample and sync to electricity timestamps
Water flow rate @ 1 sec interval
Bathroom humidity @ 1 sec interval
Hot water feed pipe temp @ 1 sec interval
Create features of exogenous factors
10.4
Exogenous factors @ 1 sec interval
10.3
11
Cluster property states
All + property state clusters @ 1 sec interval
Property occupancy @ 1 sec interval
Label property occupancy
12
13
Predict property occupancy
Label hot water usage
Hot water usage @ 1 sec interval
14.2
14.3-5
Predict hot water usage

Figure 2: Overview of workflow to build predictive models from raw data

2.3 Code overview

In order to facilitate the reading of the various chapters describing how to run the code, various details on running the code are captured in this section, including the RAM requirements, file locations and library dependencies. Please refer to Table 1 for the full details.

Category	File ID	Location in GIT	Filename	Final or exploratory	RAM	SherlockML required
Water	W1	notebooks/	clean_water_data_and_correct_time_drift.ipynb	Final	8 GB	N
Water	W2	notebooks/	report_water_data_quality_and_statistics.ipynb	Final	8 GB	N
Gas	G1	bin/gas_humidity_data_cleaning/	gas_hems_cleaning_script.py	Final	8 GB	N
Gas	G2	notebooks/cleaned_hems_data_exploration/	gas_data_exploration.ipynb	Final	8 GB	N
Room humidity	RH1	bin/gas_humidity_data_cleaning/	hygro_hems_cleaning_script.py	Final	8 GB	N
Room humidity	RH2	notebooks/cleaned_hems_data_exploration/	humidity_data_exploration.ipynb	Final	8 GB	N
Boiler pipe	BP1	notebooks/	extract_other_temperatures_hems.ipynb	Final	8 GB	N
Boiler pipe	BP2	notebooks/	clean_boiler_pipe_temperature_data_and_report_data_quality.ipynb	Final	8 GB	N
Boiler pipe	BP3	notebooks/	plot_water_and_HEMS_time_series_for_identifying_boiler_pipes.ipynb	Final	8 GB	N
Electricity health checks	E1	bin/	launch_elec_job_master.py launch_elec_job_worker.py	Final	20 x 8GB	Y
Electricity harmonics	EH1	bin/	launch_peak_finding_master.py launch_peak_finding_worker.py	Final	10 x 32GB	Y
Electricity PCA	PCA1	bin/	train_ipca.py	Final	64GB	N
Electricity PCA	PCA2	bin/	apply_ipca.py	Final	64GB	N

Table 1a: Overview of the code we have produced to implement the processing steps.

Category	File ID	Location in GIT	Filename	Final or exploratory	RAM	SherlockML required
Sync	SYNC1	notebooks/	upsample_water_and_HEMS_data_and_sync_and_merge_to_electricity_timestamps.ipynb	Final	32 GB	N
Sync	SYNC2	notebooks/	output_epc+w+HEMS_dataset.ipynb	Final	32 GB	N
Sync	SYNC3	notebooks/	label_hot_water_usage_and_create_priors_and_output_epc+w+HEMS+exo_dataset.ipynb	Final	32 GB	N
Electricity Clusters	EC1	bin/	launch_train_hdbscan_master.py launch_train_hdbscan_worker.py	Final	4 x 64GB	Y
Electricity Clusters	EC2	bin/	apply_hdbscan.py	Final	64GB	N
Occupancy model	OM1	notebooks/	prepare_occupancy_forecasting.ipynb	Final	32GB	N
Occupancy model	OM2	notebooks/	run_occupancy_forecasting.ipynb	Final	64GB	N
Hot water model	HWM 1	notebooks/	predict_hot_water_usage_with_cross_validation_10-minute_version.ipynb	Final	64 GB	N
Hot water model	HWM 2	notebooks/	predict_hot_water_usage_with_cross_validation_1-hour_version.ipynb	Final	64 GB	N
Hot water model	HWM 3	notebooks/	predict_hot_water_usage_with_cross_validation_4-hour_version.ipynb	Final	64 GB	N
Hot water model	HWM 4	notebooks/	predict_hot_water_usage_with_cross_validation_24-hour_version.ipynb	Final	64 GB	N
Hot water model	HWM 5	notebooks/	predict_hot_water_usage_with_cross_validation_72-hour_version.ipynb	Final	64 GB	N

Table 1b: Overview of the code we have produced to implement the processing steps.

Library	3	4	5	6	7	8	9	10	11	13	14
datetime	✓	✓	✓	✓				✓		✓	✓
distributed											
ephem								✓			
Eti_load_monitoring.etl							✓		✓		
eti_load_monitoring.hems_and_water.gas_humidity_cleaning		✓	✓								
eti_load_monitoring.utils						✓	✓		✓	✓	✓
Gc											✓
glob										✓	
H5py											
hdbscan									✓		
itertools									✓		✓
json					✓	✓	✓		✓		
logging					✓	✓	✓		✓		
math			✓								
matplotlib	✓	✓	✓	✓						✓	✓
numpy	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Os		✓	✓		✓	✓	✓		✓	✓	
pandas	✓	✓	✓	✓			✓	✓		✓	✓
pickle										✓	
Pytz		✓	✓								
requests					✓						
scipy			✓								✓
seaborn	✓	✓	✓	✓							
shutil									✓		
sklearn							✓		✓	✓	✓
sml					✓						
Sys	✓										
Time		✓	✓		✓	✓	✓		✓		

Table 2: Overview of the required libraries within each section of the code

2.4 Data overview

A key element of rerunning the work is the data. The work to date has looked at ten hard-drives, for at least part of the analysis, and in this section we provide a view of those hard drives. More specifically the below tables provide an overview of the data availability for the five properties, the folder structures, and the final dataset that was used for the predictive models. It is important to note that future work will focus on fully processing these ten hard drives, as well as extending the data preparation to an additional twenty hard drives.

	H20	H25	H45	H71	H73
No. of hard drives	1	3	3	2	1
Electricity data timespan	31 May – 3 Jul	21 Mar – 20 Jul	31 Mar – 4 Jul	28 Apr – 3 Jul	5 Jun – 3 Jul
Electricity data quality	Solar panels present	Frequent, repeated long gaps	~15 days missing; well-defined gaps	~2 days missing; well-defined gaps	~5 days missing; well-defined gaps
Water	31 May – 3 Jul	21 Mar – 20 Jul (2 water meters)	30 Mar – 4 Jul. Good data quality	28 Apr – 3 Jul. Good data quality	5 Jun – 3 Jul. Good data quality
HEMS database 1	NA	20 Mar – 8 May	20 Mar – 8 May	25 Apr – 8 May	NA
HEMS database 2	Unaudited	Unaudited	Unaudited	Unaudited	Unaudited
Home survey & floor plans	Available	Available	Available	Available	Available

Table 3: Overview of data available for each property

ETI property ID	HEMS property ID	Elec. period [mm/dd]	Hard drive [ETI ID]	Subfolder	Log file	Notes
H20	31	05/31-07/03	1138 / 1143	T05	051, 053	
H25	10	03/21-04/21	1134 / 1139	T01	011	
H25	10	03/21-04/21	1134 / 1139	T01A	011-013	
H25	10	04/21-06/02	1135 / 1140	T01	011-013, 015	
H25	10	04/21-06/02	1135 / 1140	T02	022-024	
H25	10	04/21-06/02	1135 / 1140	T03	031-034	
H25	10	06/02-07/20	1136 / 1141	T03	031-0331	T01 is a slice within T03, unlike the sequential order of subfolders in other HD.
H25	10	06/15-06/15	1136 / 1141	T01	011	T01 is a slice within T03, unlike the sequential order of subfolders in other HD.
H45	17	03/31-04/27	1121 / 1123 / 1128	T01	011-012	
H45	17	04/27-06/02	1125 / 1130	T02	021-026 & a "read me first" file	Other data than electricity cover 04/27-06/22; rest on 1122 / 1131.
H45	17	06/02-07/04	1122 / 1131	T02	021-024	Other data than electricity cover 06/22-07/04; rest on 1125 / 1130.
H71	24	04/28-06/02	1126 / 1132	T04	041-042	
H71	24	06/02-07/03	1127 / 1133	T04	041-043	
H73	38	06/05-07/03	1137 / 1142	T01	011-019	

Table 4: Electricity data folder structure of the first ten hard-drives

H45	Mar	April																													May							
Data series	31	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	1	2	3	4	5	6	7	8
Overview (excl. gas)	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	▨	🟩	🟩	🟧	🟥	🟥	🟥	🟥	🟧	🟩	🟩	🟩	▨	🟩	🟩	🟩	🟩	▨	🟩	🟩	🟩	🟩	🟩	▨	🟩	🟩	▨	🟧	🟧
Electricity	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟧	🟥	🟥	🟥	🟥	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	▨	🟩	🟩	🟩	🟩	🟩	▨	🟩	🟩	▨	🟧	▨
Water	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	▨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
Gas	🟧	▨	▨	▨	▨	▨	▨	🟧	🟥	🟥	🟥	🟧	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	🟧	▨	▨	▨	▨	▨	▨	🟧
Room humidity	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	▨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟧
Boiler pipe temp	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	▨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟧

	Electricity	Water	Gas	Room humidity, hot water temp
🟩	Complete: 50 / 51 files per day	Complete, ~ 8640 data points per day	Complete, ~ 96 data points per day	Complete or almost complete, ~ 288 data points per day
▨	Close to complete: >= 45 files per day	Undefined	Data gaps exist that can be interpolated i.e. >3 out of 4 data points available for each hour	Data gaps that can be interpolated
🟧	Missing partly: < 45 files per day	Missing partly	Missing partly: <72 data points per day	Missing partly
🟥	Missing completely: 0 files per day	Undefined	Missing completely, <= 5 data points per day	Missing completely, <= 15 data points per day

Table 5: Overview of data quality for house H45 for time period that was used in the full analysis. 30 of the 39 days displayed are utilised, as days with orange or red data quality at the overview level are removed

3 Preparing Water Consumption Data

3.1 Overview

This Chapter focuses on the initial data cleanse of the water data, prior to combining it with other datasets, which is described in detail in Chapter 9. The initial cleanse is described in two sections, the first covers a Python Notebook that accelerates the process of applying the data corrections, and the second section covers a Python Notebook that accelerates the data quality investigation. The reason the Notebooks were not converted into scripts to fully automate the processes, is that new data issues may arise in future datasets, making it risky to full automate the process at the stage of writing. It is also worth noting that both notebooks should take no longer than a few minutes to run.

To date, a few details of the data provided and that we have analysed are:

- Files are provided in pairs, a consumption data file and a data description file, which correspond to circa one month of data. There is one exception to this, which is for a property that has a dual water meter, in this case there is a second consumption data file.
- Both files are in *.txt format and are under 1 GB, making them processable with a typical laptop.
- The key data in the consumption data files are the consumption readings and their associated time stamps.
- The key data in the water description files are the information relating to time drift, which are required for syncing the corresponding water data.

3.2 <W1> Pre-processing water data files

The first step in the two-stage process, is correcting the data for known issues that are felt to be key. These were applied at a file level, rather than aggregating files at this stage and the key steps are:

- Consumption readings are meant to be 10 seconds apart and where a data point exists between two readings, it will be removed. These issues have not been observed yet, but the code will correct for them if they were to occur.
- The code checks the consumption readings for incomplete entries, identical duplicate entries (two or more entries having identical readings for the same time stamp) and conflicting duplicate entries (different readings for the same time stamp). If these are identified they will require further investigation and removal as appropriate. It is worth noting that whilst the first entry of each file appears incomplete, it is important to keep this entry as it stores the starting time of the recording, which is required for synchronisation.
- Correct for time drift (see Chapter 6 in Appendix B for full details on the source of the water time drift):
    - Firstly, determine how much time drift has occurred, using the start and end time stamps of the water consumption file, together with the end time stamp of the data description file.
    - Secondly, stretch or shrink the whole series of time stamps by applying the identified shift monotonically and linearly.

- Output the consumption data corrected using the above steps into a *.h5 file. H5 does not store the time stamps associated to each data point, but instead the actual sampling interval derived from the time drift correction (which slightly differs to the nominal sampling interval of 10 seconds) is stored in the file using the key "actual sampling interval".

It is advisable to run the Notebook iteratively and top to bottom, checking each step as you proceed. Prior to running the file, it will be necessary to adjust the location of the input files, to link to the water consumption file and the water description file that you want to process, as well as to ensure the output file is re-named so that it does not replace any previous work. This then needs to be repeated for each pair of files that are required. Two aspects of the Notebook that have not been automated and care should be taken with are:

1. The time stamps within the data description file are inconsistently formatted across different files. For example, 27/04/2017 10:38:00 vs. 27/04/2017 10:38.
2. Where the start and end of a file falls across two time zones, the time correction will need to be executed manually.

3.3 <W2> Investigating data quality

Following on from Section 3.2, which produces a *.h5 file, a Notebook is used to explore the data and investigate it for further data quality issues. This section is purely informative, as opposed to corrective, so should an issue be identified at this stage, it is necessary to go back to the previous stage and introduce a new data quality correction.

The current data quality checks that have been codified are:

- Whether the time drift is within the expected range stated in Appendix B (c. ±60 seconds per month).
- Whether the water flow histogram appears reasonable, including:
  - Whether one or more bell-shaped distributions with a couple of spikes (related to e.g. toilet flush) are present
  - No negative consumption is observed
  - Consumption has a relatively long tail at the high consumption end.
- Check for underflow and overflow instances. According to ESC's Data Specification document, the water flow sensor only captures flow between 0 and 100 litres per minute. Values below this range are considered underflow or above this range are considered overflow.
- Missing data within the file, not between files, i.e. a gap of 1 hour in the recording period.
- Daily load profile plots to check if there are any systematic recording issues.
- Plots of daily and hourly demand, to visually inspect the ranges and volatility of usage.

In order to run the Notebook, it will be necessary to re-point the input file to the file output from Section 3.2. It is worth noting that this script does not produce any output files, but simply data descriptions and plots within the Notebook for a human to review. This is because the purpose of the script is to inform the user of the data quality and any potential issues to correct for, as opposed to the improvement of data quality.

4 Preparing Gas Data

4.1 Overview

This Chapter focuses on the initial data cleanse of the gas data, which was carried out, but ultimately not used in the final model. As such, it is not necessary to follow this section to replicate the project results, but the section is still provided for informative purposes.

The initial cleanse is described in two sections, the first covers a Python Script to conduct the necessary cleansing, and the second section covers a Python Notebook that accelerates the data quality investigation. The script for data cleansing can run automatically after file names and parameters have been changed as appropriate. In the final solution provided as the HFADA project, gas data were not synchronised or merged with other data sources, mainly due to their relatively poor data quality, but the code had been generated and is therefore shared to accelerate future analysis.

To date, a few details of the gas data that we have analysed are:

- All the data is in the MongoDB database of HEMS measurements, and the format appears consistent between properties.
- Raw gas consumption data are exported into HDF5 format, prior to the below data processing steps. A Python module (mongo_util.py) has been produced for extracting the data from MongoDB to tabular format. The module is built around a few standard MongoDB queries, which one can apply on the HEMS database.
- The data is under 1 GB making it processable with a typical laptop.
- The key data are the cumulative consumption readings and their associated time stamps.

4.2 <G1> Pre-processing gas data from HEMS

The first step in the two-stage process is to apply the data cleansing. In order to run the script, the following parameters should be set at the start of the script:

1. House Number / device_id should be set to the corresponding property
2. Units should be set to meters cubed or litres. It appears that each property records in one of the two units consistently, and to identify which of the two is used for a given property the file can be opened and inspected.
3. Link to the appropriate input dataset (HDF5 file exported from HEMS database)

The script produces two output files in HDF5 format, one is a time series of cumulative gas consumption, the second file is an error log.

device_id	device_state	timestamp	Units	Value
17	normal	1494231277	m3	23.42
17	warning	1490031923	m3	0.07

Table 6: Illustrative data extract from HEMS database

The script ran the following data cleanse activities on the gas data:

- Remove entries that are labelled as ‘warning’ in the *device_state* field, which should only leave ‘normal’ entries.
- Retain only the entries for the specified *device_id* / property
- Remove any entries within the *units* column that are in the wrong units
- Remove any incomplete entries
- Remove identical duplicate entries (once removed, only singular entries should remain).
- Data appeared ordered in time, as such, any rows where the timestamp of a gas reading is prior to the previous entry’s timestamp are removed.
- Remove records where the meter is of a lower reading (value) as compared to the previous reading. This will imply a decrease in cumulative volume, which is a false outcome.
- Flag up the identified issues via an error log.

Some additional functionality that was produced and not used, which has been kept in case it accelerates future work are:

- Convert the cumulative gas flow measurements to consumption readings
- Up-sample consumption data to a 1 second frequency
- Adjust for time zones

4.3 <G2> Investigating data quality

Following the gas data cleanse activities, data quality checks should be run to investigate the dataset for further issues:

- Visual inspection of cumulative gas consumption over time paying particular attention to significant data gaps.
- The minimum, maximum and median time intervals between readings for a given period is calculated with the view of identifying significant data gaps. Data is expected once every 15 minutes. If a significant data gap is identified the user can then decide on a case by case basis how to handle the data quality issue
- For each hour, the number of data points recorded is checked, helping to identify data gaps. In any given hour, 4 data points are expected, but there is a known variance around this number.
- Heat map displaying gas consumption by hour and day.
- A flag is applied to the data that has questionable data quality, which is used for reporting, but could be extended for use in filtering.

Some additional functionality that was produced and not used, which has been kept in case it accelerates future work are to adjust for time zones and the ability to up-sample consumption data to a 1 second frequency.

In order to run the notebook it is necessary to change the filename of the input file, which is the output file generated in section 4.2. It is worth noting that this script does not produce any output files, but simply data descriptions and plots within the Notebook for a human to review. This is because the purpose of the script is to inform the user of the data quality and any potential issues to correct for, as opposed to directly improve the data quality.

5 Preparing Room Humidity Data

5.1 Overview

Raw humidity data are stored in the HEMS database. In order to cleanse and analyse the data, the humidity data is first extracted into a tabular format, after which the data is processed. Unlike the poor data quality for gas, humidity data is of a reasonably high quality, meaning it is included in the final model for the HFADA project.

Two data processing steps have been followed for the humidity data in this section, with 1 script produced to conduct the necessary cleansing, and 1 notebook produced to review the data quality. The script for data cleansing can run automatically after file names and parameters have been changed as appropriate, and that are located and configured at the beginning of the script. In contrast to the script, which is fully automated, the notebook to investigate the data quality is suggested to be run on an interactive basis. Both the script and the notebook should take no longer than a few minutes to run.

To date, a few details of the data provided that we have analysed are:

- All the humidity data is in the MongoDB database of HEMS measurements, and the format appears consistent between properties. It is worth noting that there are multiple time series per property, as there are multiple humidity sensors in a house, typically one per key room.
- Humidity data are exported into HDF5 format, prior to the below data-processing steps. A Python module (*mongo_util*.py) has been produced for extracting the data from MongoDB to tabular format. The module is built around few standard MongoDB queries, which one can apply on the HEMS database.
- The data is under 1 GB making it processable with a typical laptop. For the HFADA work 39 days of data for property H45 was used.
- The key data are the humidity readings and their associated time stamps.

5.2 <RH1> Pre-processing room humidity data from HEMS

In order to cleanse the humidity data it is necessary to first adjust the input parameters, which in this case is just the house number / device_id. It is also necessary to link to the appropriate input dataset, which in this case is the HDF5 file exported from the HEMS database. The script produces two output files in HDF5 format, one is a time series of humidity that includes a room label, the second file is an error log.

device_id	device_location	device_state	type	Units	value
9	LOUNGE	normal	Internal	%	47.26
5	BATHROOM	warning	Internal	%	-327.68

Table 7: Illustrative humidity data extract from HEMS database

The script ran the following data cleanse activities on the humidity data:

- Remove entries that are labelled as 'warning' in the *device_state* field, which should only leave 'normal' entries.
- Retain only the entries for the specified *device_id* / property
- Remove any entries within the *units* column that are in the wrong units
- Remove any incomplete entries.
- Remove identical duplicate entries (once removed, only singular entries should remain).
- Data appeared ordered in time, as such, any rows where the timestamp of a humidity reading is prior to the previous entry's timestamp are removed.
- Remove values outside the expected range of operation, which was defined as 0-95%.
- Flag up the identified issues via an error log.

Some additional functionality that was produced and not used, which has been kept in case it accelerates future work are:

- Adjust for time zones
- Up-sample consumption data to a 1 second frequency

5.3 <RH2> Investigating data quality

Following the humidity data cleanse activities, data quality checks should be run on a room-by-room basis, to investigate the dataset for further issues:

- The minimum, maximum and median time intervals between readings for a given period is calculated with the view of identifying significant data gaps. Data is expected once every 5 minutes. If a significant data gap is identified the user can then decide on a case by case basis how to handle the data quality issue
- Heatmap displaying average humidity values by hour and day.
- For each hour, the number of data points recorded is checked, helping to identify data gaps. In any given hour, 12 data points are expected, but there is a known variance around this number.
- A correlation matrix of humidity across rooms is run and visually inspected for any peculiarities i.e. all rooms are 100% correlated. To date, this has not flagged any issues.
- A flag is applied to the data that has questionable data quality, which is used for reporting, but could be extended for use in filtering.
- A function to convert between time zones was built, but is not required. The code has been included for reference, but should not be run.

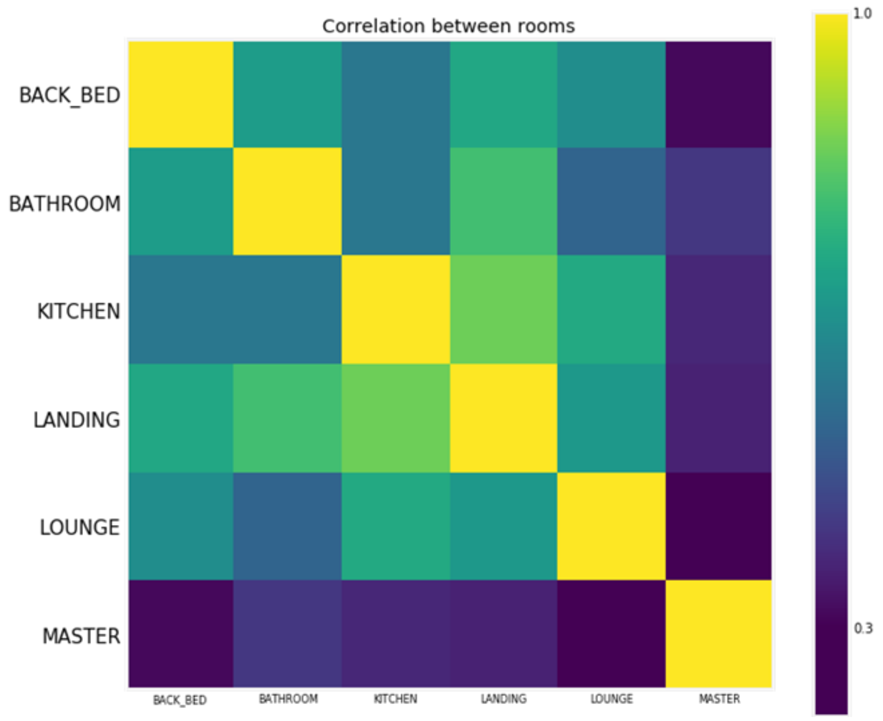


Figure 3: Correlation matrix of room-by-room humidity

6 Preparing Boiler Pipe Temperature Data

6.1 Overview

Similarly to gas and humidity data, the boiler pipe temperature data is stored in the HEMS database. A key difference is that the property analysed had three sensors on the boiler that are all labelled as "external", meaning that if the readings are extracted as usual then the three time series are pulled into one extract. In order to differentiate between the three time series, the location of the three data points of each entry is used, and in order to know which of the three locations corresponds to the right sensor it was necessary to plot the 3 time series and use human discron. This methodology was produced based on the data analysed for H45, but may not extrapolate well to future datasets, which may have different data quality issues.

Three data processing steps are applied in this section. The first focuses on extracting the boiler pipe temperature and retaining the order of the three series to allow for their distinction later on. The second focuses on general data quality across the three time series. The third section identifies which of the 3 time series is thought to be the hot water feed pipe temperature from the three pipes (hot water feed pipe temperature, heating feed pipe temperature and heating return pipe temperature).

6.2 <BP1> Extracting boiler pipe temperature data from HEMS

The temperatures of pipes within the boiler are stored in the MongoDB database of HEMS measurements. It is therefore necessary to convert the data from MongoDB into tables. To this end, we have developed a notebook (extract_other_temperatures_hems.ipynb) that queries the MongoDB, and transforms the data into tabular format. One output file is produced, with 4 separate datasets, under 4 unique keys:

- 'temperature_pipe_0': 'internal'
- 'temperature_pipe_1': pipe no. 1
- 'temperature_pipe_2': pipe no. 2
- 'temperature_pipe_3': pipe no. 3

The output is an HDF5 file located in the folder *"/project/data/hems_extracts/raw/"* and called *"house_H45_hot_water_extended.h5".*

6.3 <BP2> Pre-processing and investigating data quality

To clean boiler pipe temperature data and investigate issues relating to data quality a python notebook and the data file output in section 0 are required. The notebook processes one pipe at a time, so to process the data for all three pipes, it is necessary to change the key twice (note: temperature_pipe_0 is not required as part of this analysis, as this is already known to refer to the temperature of the sensor hub).

Timestamp	type	units	value
1490178116	External	C	-327.68
1490181685	External	C	48.18

Table 8: Illustrative boiler pipe temperature data extract from HEMS database

The notebook is comprised of two sections: cleaning boiler pipe temperature data and investigating the data quality. The notebook should be run interactively, and in chronological order. It is worth noting that there are additional checks that could be built or issues that may be encountered in future datasets, which is worth considering in future iterations.

The first section runs the below data cleanse activities on the data, and outputs one data file for each of the three boiler pipe sensors. It is necessary to run the code once per sensor.

- Removal of unreasonable temperature values, more specifically we expect to see values between 0 °C and 85 °C only with these sensors, as advised by ESC.
- Removal of incomplete entries.
- Removal of identical duplicate entries.
- Flagging up of conflicting duplicate entries. In such an event, manual investigation would be required, but no such examples have been found in the analysis conducted to date.

Secondly, once the data has been cleansed, the below data quality checks are conducted. The output of the code are data descriptions and plots within the notebook for a human to review.

- Identify any missing data or data gaps.
- Confirm whether the temperature histogram looks reasonable. A reasonable distribution is likely to be bell shaped - skewed to the right, with a peak between 18-25°C.
- Visual inspection of the hourly average of the boiler pipe temperature.

6.4 <BP3> Identifying the hot water feed pipe

Following the data cleanse activities it is necessary to identify the hot water feed pipe, which is the sensor that is required to identify hot water usage. The identification of which of the three time series corresponds to the hot water feed pipe is achieved by visualising the three time series amongst other data features for a few days. A human can then manually label these.

Figure 4 to Figure 6 illustrate the plots that can be generated with the script for one property on one day. Each figure represents one of the three pipes, and based on our understanding of boilers we concluded that pipe 1 is the hot water feed pipe, pipe 2 the heating feed, and pipe 3 the heating return. This method has only been tested for 1 month of data on 1 property, so it may be that the solution is not functional in different cases i.e. another property may have 5 sensors.

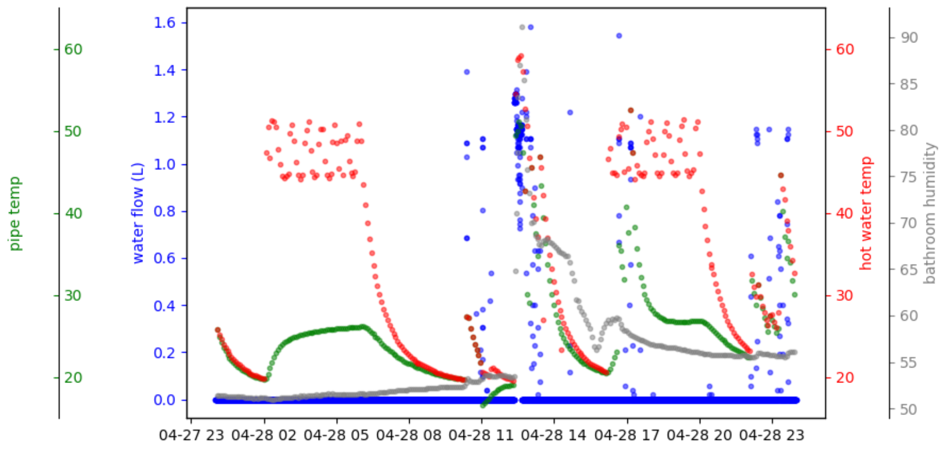


Figure 4: Boiler pipe 1 for House H45 on 28 April 2017

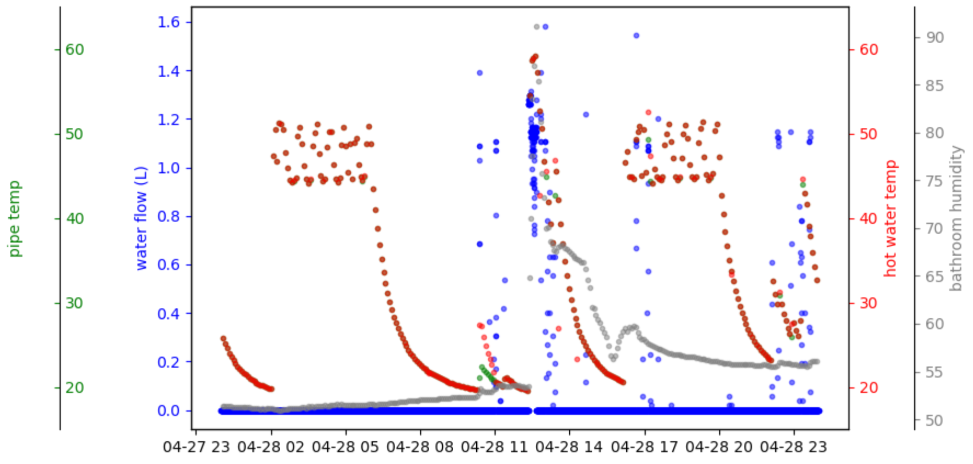


Figure 5: Boiler pipe 2 for House H45 on 28 April 2017

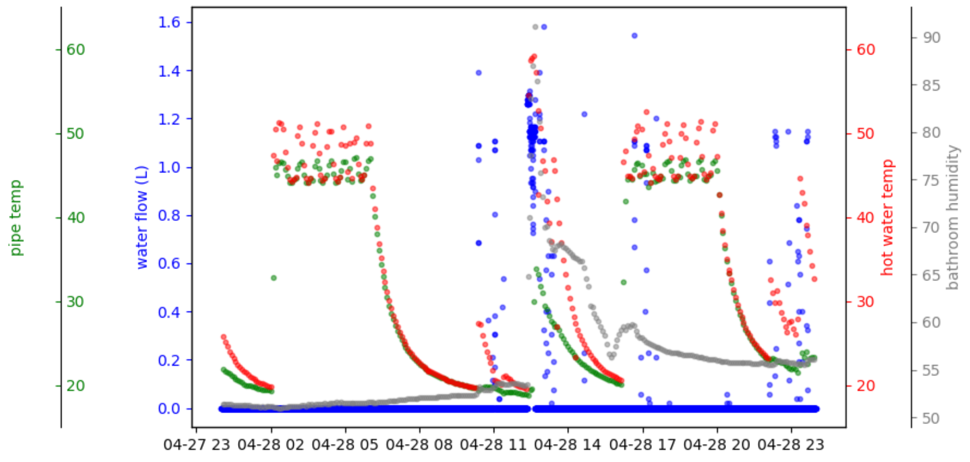


Figure 6: Boiler pipe 3 for House H45 on 28 April 2017

7 Preparing Electricity Data I: Parallel Architecture

7.1 Parallel architecture

Because of the large size of the datasets involved in this project, it was necessary to develop an architecture for parallel computing. Although it would have been possible to use a standardised framework for parallel computing - such as Apache Spark - we decided to build infrastructure customised for this project. The choice was motivated by the fact that setting up a cluster with Apache Spark requires significant work. This effort would not have been worthwhile, since all computing tasks to be performed were easily parallelizable, as they require no communication between the 'workers'. The wide functionality of Apache Spark would have been underutilised, so that it would not have been productive to invest time on this tool. Instead, we decided to develop a simple architecture for parallel computing using SherlockML, the platform that was used for the project. As this platform allows one to easily start and stop multiple servers, setting up the cluster was relatively straightforward. Over the course of the project, a customised framework was employed to (i) run quality checks on the electricity data, (ii) compress the electricity data through a peak-finding algorithm, (iii) tune the meta-parameters of the clustering algorithm.

To demonstrate the architecture, the scenario where it is necessary to process numerous independent input files and then generate multiple output files is considered (Figure 7). Most computations are carried out in parallel by servers called *workers*, while the overall cluster is managed by an additional server called *master*. The architecture assumes that each worker performs the task independently of other servers, reading and writing one file at a time. In technical terms, this constrains the architecture to easily parallelizable problems, involving no communication between the workers.

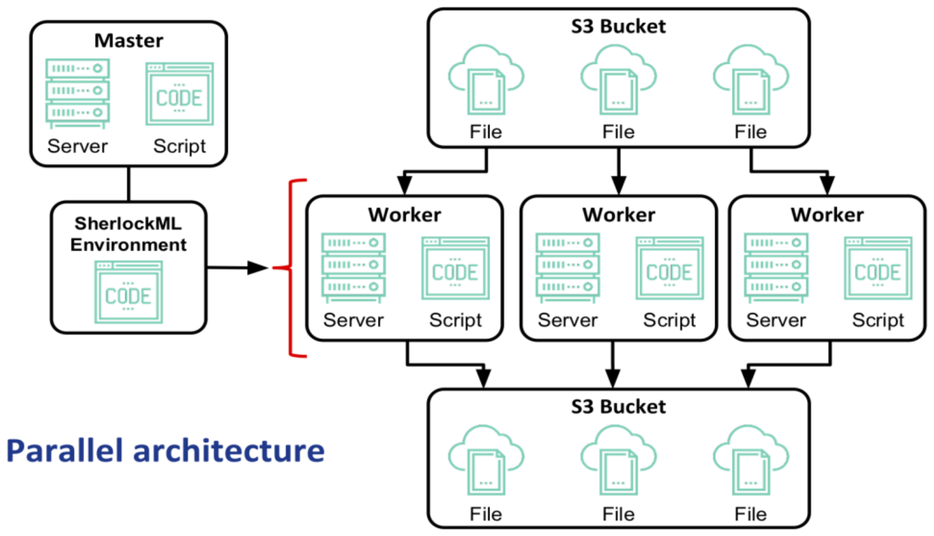


Figure 7: Illustration of parallel computing architecture for a typical data processing task

The architecture features two Python programs, namely the *master script* and the *worker script*, operating the two server types. In theory, it would have been possible to write a generic master script, valid independently of the computation being addressed. As this abstraction is fraught with difficulties, it was decided that the user would write a new master script for each problem. For example, running quality checks on the electricity data involves not only a specialised worker script but also a slightly adapted master script. In this particular case, the master script collects and polishes up the results generated by the workers.

The operation of the infrastructure for parallel computing can be summarised as follows. The master script starts a number of workers using commands from the module *parallel.py*, and it applies a SherlockML environment to each of them. In essence, the environment is a bash program that points servers to the relevant worker script, so that the main computation can begin. The workers then process files independently of each other, and upon completion of the task terminate themselves. Meanwhile, the master inquires every 60 seconds as to whether the workers are still active, keeping its other operations entirely on hold. The signal that all workers have terminated themselves indicates that the main computation is over, and triggers the master to proceed with 'wrapping-up' duties.

The tools necessary to 'manage' the cluster are provided in the module *parallel.py*, with most functionality grouped together in the class *sherlockml_cluster*. This is used by the master to create multiple worker servers, to apply a SherlockML environment on each of them, and to monitor the completion of the task. The module *parallel.py* also contains the function *group_paths*, which divides the paths of input files into *groups*, and saves these into individual *text records*. Each group determines which input files will be processed by a given worker, and each text record is named after the corresponding worker. As a result, the worker can identify which files to process by linking its name to that of a text record, and simply reading the latter into memory.

It is worthwhile to comment on the fact that the figure above displays the files as stored in an AWS S3 bucket. While this was effectively the case in the project, the architecture for parallel computing does not depend on the storage solution. Alternatives to AWS S3, such as Elastic File System, can indeed be used.

7.2 Overview of the electricity data

In this section, we offer a brief overview of the electricity dataset. A complete description is available in Appendix B: ESC's Data Specifications.

The raw electricity measurements are stored in WAV files that each contain 350 million data-points. Voltage and current signals are recorded at a sampling rate of 204,918 Hz, so that each file corresponds to around 1708 seconds, that is, approximately 30 minutes worth of data. Each measurement is expressed as a 16-bit integer in arbitrary units.

Two readings of current are provided as separate channels in the WAV files. The channels, "current 1" and "current 2" are collected from sensors that have a range (-10A, +10A) and (-90A, 90A), respectively. This setup yields measurements of higher precision whenever the current has amplitude below 10A, since one can refer to the sensor with narrower range and better resolution, whilst obtaining a range of coverage of ±90A with lower resolution. Voltage readings are found under the "voltage" channel of the WAV file.

7.3 <E1> Health checks for the electricity data

Leveraging the parallel computing architecture, including the scripts *launch_elec_job_master.py* and *launch_elec_job_worker.py*, it is possible to run various *health-checks* on the electricity data.

The electricity health checks are run at an electricity file level (a WAV file of c. 30 minutes of electricity data). For each file the following health-checks are run:

- Maximum and minimum values of current and voltage;
- Mean-square values of current and voltage;
- Mean-square values of current and voltage across each 60-second interval;
- Histograms of current and voltage;
- Percentage of readings that are "stuck", in that the sensor records the same measurement multiple times.

The outcomes of the health-checks are output into a Pickle file for a human to review for issues. Overall, the electricity data is found to be of high quality and the only data removed are a few of the half-hour electricity files, which are incomplete. The incomplete files can be identified by checking for the full amount of data points or by checking the log files, as incomplete files do not appear in the logs. A more detailed discussion of these results is provided in the *Data Quality Report*, in Appendix A. Given that no serious data quality issues were identified, if future version of the data are comparable, these scripts should not result in a change to the final output, but are worth running to validate that data is of comparable quality.

8 Preparing Electricity Data II: Downsizing by Extracting Harmonics

8.1 Overview

The original batch of 10 hard drives include approximately 30 TBs of electrical data, which is too large to be used in its raw form by machine-learning algorithms. As such, data compression is necessary and it is important to balance reducing the data size too aggressively and consequently discarding useful signals, and conversely retaining too much information resulting in a too large dataset. Selecting an algorithm and calibration thereof that strikes a good balance in this trade-off is crucial, as a poor choice can affect subsequent analysis, for example by compromising the predictive power of the dataset.

A key motivation for compressing the electricity dataset is to *lower memory requirements* in later parts of the project. The implementations of many machine-learning algorithms, particularly those available through the popular library *Scikit-Learn*, assume that the dataset fits into the memory (RAM) of the server. It follows that reducing the data size is a prerequisite to using standard implementations of machine-learning algorithms, and well-developed modelling libraries.

One may of course decide to compress less aggressively, and to use specialised tools such as *Apache Spark* to handle an increased data size that exceeds the memory of a single machine. While Spark and similar libraries contain implementations of several models, resorting to these tools still reduces the selection of algorithms, especially in the context of unsupervised machine learning. It is worthwhile to recall that in the work stream of our project concerned with predicting occupancy, the ground truth is not available, so that having a wide choice of unsupervised algorithms is vital.

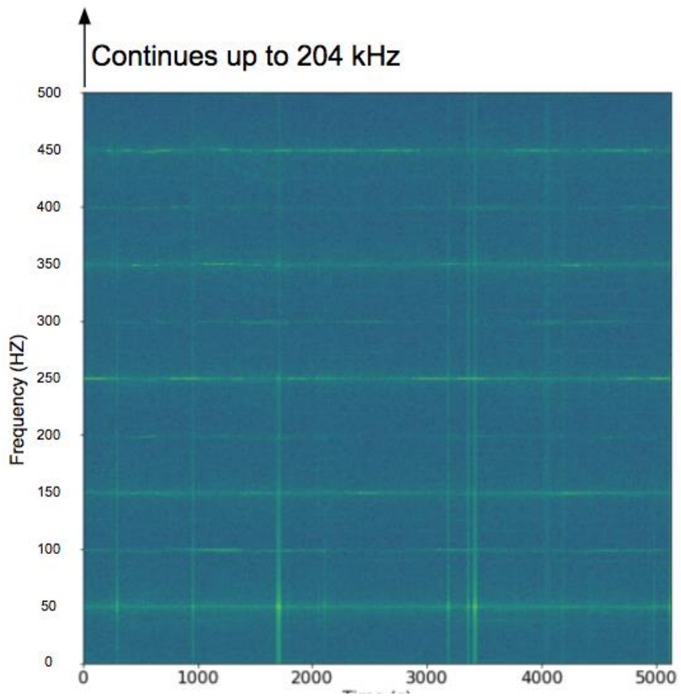


Figure 8: **Current as a function of time and frequency**

Regardless of the above considerations, even if one resolved the problems ensuing from the fact that the electricity dataset is much larger than the RAM of a server, to train machine-learning algorithms on tens of Terabytes would demand extreme computational resources (Number of CPUs). Advanced machine-learning algorithms do not scale well with the size of the dataset, and in technical terms their time-complexity is almost always worse than *O(N)*. This drastically increases the number of CPUs required so as to train a machine-learning algorithm on the full electricity dataset. In addition to being necessary from a technical perspective, reducing the data size is important from a monetary viewpoint. Because processing a smaller dataset involves less computational resources, data compression results in lower computational costs.

Here, we describe the first step in compressing the electricity data, namely the application of a peak-finding algorithm. This is based on the idea that the representations of voltage and current in frequency space display maxima called *harmonics* about every *50Hz*. As an aside, these peaks are caused by the fact that the electrical response of some circuit elements in household appliances, such as diodes, is not linear. Moreover, the periodicity in the maxima is due to the nominal frequency of alternating current in the UK being *50Hz*. Because the harmonics summarise, to a great extent, the information contained in electrical signals, we compress the voltage and current data by evaluating specific metrics at these peaks. Specifically, the amplitudes of voltage and current located between the harmonics in frequency space are discarded, thus reducing the overall size of the dataset. In Figure 8, which shows a measurement of current as a function of time and frequency, the bright horizontal lines correspond to the first 10 harmonics. Data compression is achieved by recording information about the maxima, and rejecting the values that appear in the spaces between horizontal lines.

The algorithm considered in this Chapter are implemented in the Python scripts:

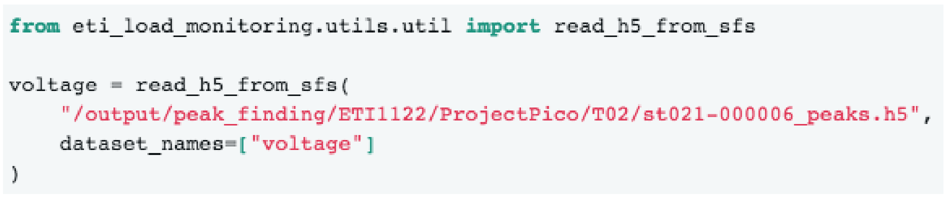
- *launch_peak_finding_master.py;*
- *launch_peak_finding_worker.py*.

As the above file names suggest, the computation relies on the parallel architecture described in Figure 7 in order to process large quantities of electricity data. The *input* of our program is the ~30-minute-long files containing raw measurements of voltage and current, and the *output* is the following metrics:

1) *Locations of the harmonics* in frequency space.
2) *Time* since the beginning of the 30-minute file.
3) *Complex amplitude of voltage* at each harmonic peak.
4) *Complex amplitude of current* at each harmonic peak.
5) *Apparent power* at each harmonic peak. This quantity is defined as the absolute value of the complex power obtained by using the voltage and current in bullet points 3) and 4).
6) *Phase angle* at each harmonic peak. This quantity is defined as the argument of the complex power obtained by using the voltage and current in bullet points 3) and 4).
7) *Active power* at each harmonic peak. This quantity is defined as the real part of the complex power obtained by using the voltage and current in bullet points 3) and 4).
8) *Reactive power* at each harmonic peak. This quantity is defined as the imaginary part of the complex power obtained by using the voltage and current in bullet points 3) and 4).

Each input 30-minute-long *WAV* file is mapped into a separate *HDF5* file containing the metrics 1) - 8) as distinct datasets. For instance, the raw electricity data from the following file, which is located in the S3 bucket: */sherlockml-snowball/ETI1122/ProjectPico/T02/st021-000006.wav*, generates this file: */output/peak_finding/ETI1122/ProjectPico/T02/st021-000006_peaks.h5,* which is located in SherlockML Filesystem (SFS).

It is worthwhile to note that the output metrics contain redundancies. For example, apparent power and phase angle combined are equivalent to active and reactive power. As a matter of fact, the two pairs are polar and Cartesian representations of the same quantity, the complex power. Even though storing redundant data consumes memory, it was deemed helpful to provide future users with a selection of pre-calculated metrics. Within each HDF5 file, the output quantities 1) - 8) correspond to the keys *"frequency"*, *"time"*, *"voltage"*, *"current"*, *"apparent_power"*, *"phase_angle"*, *"active_power"* and *"reactive_power"*. For instance, to load voltage data into RAM from the file *st021-000006_peaks.h5*, one can use the snippet of code in Figure 9.



```
from eti_load_monitoring.utils.util import read_h5_from_sfs

voltage = read_h5_from_sfs(
    "/output/peak_finding/ETI1122/ProjectPico/T02/st021-000006_peaks.h5",
    dataset_names=["voltage"]
)
```

Figure 9: Example on how to laod volatage data into RAM

8.2 <EH1> Physical units

At the start of the calculation, the input electricity data is in raw format. Because measuring voltage and current involves a conversion of signals from analog to digital, and because the data is collected directly from the sensors, the electricity readings are expressed as *16-bit integers* with *arbitrary units*. In other words, they are represented as whole numbers that range from −32767 and 32767 with the lower and upper ends of this range indicating respectively the smallest and largest numbers that the sensor is designed to measure. For example, as the voltmeters used in this project are known to operate between -500V and +500V, a reading of 32767 for the voltage in raw format corresponds to +500V.

Performing calculations with 16-bit integers creates a risk of arithmetic-overflow. For example, the product of two 16-bit integers can easily exceed the available range for numbers in this format (between −32767 and 32767). In order to reliably handle large positive and negative quantities, and thus avoid overflow errors such as the one just described, we convert the input voltage and current data to 32-bit floating-point format.

It is also worthwhile to mention that, because one is accustomed to think in terms of physical units, arbitrary units are not practical. With the purpose of making the electrical data easily interpretable, we convert voltage and current measurements to SI units.

In the script that applies the peak-finding algorithm (*launch_peak_finding_worker.py*), the translation from 16-bit integers with arbitrary units to 32-bit floating-point values with SI units is carried out by the function *raw_to_physical_multi*. This function multiplies electricity data stored in different channels of the WAV files as follows:

- for the *'voltage'* channel multiply by *float32(500.0 / 2**15)*
- for the *'current 1'* channel multiply by *float32(10.0 / 2**15)*
- for the *'current 2'* channel multiply by *float32(90.0 / 2**15)*

The function *raw_to_physical_multi* is available in the *utils.util* module.

8.3 <EH1> Combining measurements of current

As discussed previously, each WAV file stores two current signals as distinct channels. By combining these into a single data-source, it is possible to generate a reading of current that displays both high precision and a wide range. As a matter of fact, the channel *"current 1"* stores the measurements of a sensor with narrow range ($\pm$10A) and high precision, while the channel *"current 2"* stores the measurements of a sensor with wide range ($\pm$90A) and lower precision. One can select the former or the latter depending on whether the magnitude of the current is small or large, and thus generate a combined signal with superior characteristics.

In theory, it would be possible to merge the two channels by switching between the signals exactly when the current's amplitude exceeds a given threshold. This strategy, however, is prone to introducing artefacts in the combined signal. As a matter of fact, because the readings in the two channels are slightly mismatched, transitions between them can produce 'jumps' in the dataset. If the amplitude of current is close to the threshold, the above approach generates frequent channel switches, and thus numerous artefacts.

In order to avoid this problem, we make use of an alternative strategy. By default, the combined signal is set to the channel with range $\pm$10A, so as to take advantage of its higher precision. When the current's amplitude exceeds the threshold, the merged signal is set to the channel with range $\pm$90A over a time-window that stretches across the higher readings. This prevents frequent channel transitions, as the interval between switches is always greater than the span of the time-window.

Our strategy for merging the two current channels is implemented as the function *combine_currents* from the module *combining_currents.py*. This function accepts the following input:

- *current_10a* (ndarray): Measurements of current with range (-10A, +10A).
- *current_90a* (ndarray): Measurements of current with range (-90A, +90A).
- *threshold* (int or float): The value of current that determines which channel to pick.
- *sample_rate* (int or float): The sampling rate of both *current_10a* and *current_90a*.
- *window_seconds* (float): The duration in seconds of the time-window.
- *min_periods* (int): The minimum number of data points that the time-window must contain in order for the output value of current to be non-null. This argument determines the behaviour of the function close to the beginning and the end of a signal, where edge-effects may appear.
- *center* (bool): Select whether the time-window is centred at the data point where the amplitude of current is larger than *threshold*.

It is worthwhile to comment on the values that we used for some of the above parameters.

- The *threshold* for switching between channels was set to *8.5A*.
- The span of the time-window, *window_seconds*, was chosen to be *4s*.
- No requirement was placed on the minimum number of valid data-points per instance of the time-window (*min_periods=1*).
- Instances of the time-window were centred about measurements of high current (*center=True*).

8.4 <EH1> Extracting the Harmonics

As explained previously, we compress the electrical data by storing the information contained in the harmonics, and discarding the rest. Qualitatively, this technique is based on the *Short-Time Fourier Transform* (STFT) and a *peak-finding algorithm*.

The STFT describes how the frequencies that make up a signal change as a function of time. In order to generate the STFT, a rolling window is considered that divides the signal into blocks. The frequency content of the blocks is then calculated by applying a Fourier transform on each instance of the time-window. The result of this operation, and consequently of the STFT, can be expressed as a matrix whose rows and columns represent 'frequency' and 'time'. The rows correspond to the Fourier transforms, and the columns to instances of the rolling window.

We use a peak-finding algorithm to determine the frequency of the harmonics at each step in time. This is equivalent to finding the entries in the STFT matrix that have the largest absolute value. One can compute the locations of the harmonics either from the current signal or from the electrical power. We decided to rely on measurements of current, and to build the peak-finding algorithm around these steps:

- Determine the frequency f_0 of the 1st harmonic, called the *fundamental frequency*, by seeking the maximum of the STFT between *25Hz* and *75Hz.* This exploits knowledge that the 1st harmonic is located close to the nominal frequency of alternating current in the UK (*50Hz)*.
- Define bins in frequency space that 'surround' the harmonics. For example, the first bin ranges from $f_0/2$ and $3f_0/2$, the second bin ranges from $3f_0/2$ and $5f_0/2$, and the third bin ranges from $5f_0/2$ and $7f_0/2$.
- Calculate the frequencies of the harmonics by seeking the maximum of the STFT in each bin.

The use of bins makes efficient use of the fact that harmonics are spaced regularly. It converts the difficult task of finding several relative maxima into the simple task of locating the global maximum within each bin. It is useful to note that computing the relative maxima of a function involves curve-fitting, a moderately slow operation. On the other hand, identifying the global maximum requires a single scan through the values taken by the function, an extremely fast *O(N)* operation.

In the script *launch_peak_finding_worker.py* the compression of electrical data via the frequency harmonics is implemented as follows:

1) Derive the *frequencies*, *times*, and *indices* in the STFT matrix that correspond to the harmonics. This step relies on the function *get_harmonic_loc*s from the module *peak_finding.py*.
2) Derive the *complex voltage* at the harmonics. We filter the STFT of the voltage signal by means of the indices calculated in 1) so as to obtain the Fourier components of voltage at the harmonic peaks. This step relies on the function *get_harmonic_values* from the module *peak_finding.py*.
3) Derive the *complex current* at the harmonics. This step is equivalent to 2) with the voltage signal replaced by the current signal.

Lastly, the complex voltage and the complex current are employed so as to find the *apparent power*, the *phase angle*, the *active power*, and the *reactive power*. As discussed earlier, these results are saved in HDF5 files that each correspond to ~30 minutes' worth of current data.

Key parameters in the computation are the *stride* and the *window size*. The former determines how quickly the window in the STFT proceeds through the signal, and ultimately specifies the time resolution of the output data. The latter determines the span of the rolling window, where large values of this quantity correspond to an accurate sampling of the frequency space. Choosing the stride and the window size demands one to resolve a number of trade-offs, as explained in our *“insights document”*. We set the stride to be *1 second*, and the window span to be *2 seconds*. An important consequence is that the time resolution of subsequent datasets will be *1 second* too.

9 Preparing Electricity Data III: Downsizing by Principal Component Analysis

9.1 Overview

There is a limited availability of machine learning algorithms that work on datasets that do not fit into the memory of a single machine. This is particularly true for dimensionality reduction techniques. Scikit-Learn offers an incremental implementation of Principal Component Analysis (PCA) that gradually learns as manageable chunks of data are loaded into memory. This is one of the very few viable options for reducing the number of data features of a very large dataset. It was therefore decided to compress our data through this implementation of PCA.

We projected the initial ~8000 features into 50 principal components. This was observed to explain most of the variance, that is to say, to retain the vast majority of the information. At the same time, the reduction to 50 principal components produced a dataset that fits into the memory of a single server.

This chapter discusses two scripts, namely train_ipca.py and apply_ipca.py. The former trains an Incremental PCA (IPCA) model by scanning through the dataset in chunks. The latter applies the IPCA on the chunks, and concatenates them so as to form a unique data source. We train and apply the IPCA algorithm on the entire dataset for property H45.

9.2 <PCA1> Training the IPCA model

The IPCA model was trained on the results of the data-compression based on extracting the harmonics. Accordingly, the input of train_ipca.py was a large number of HDF files, each containing the equivalent of 30-minutes raw electricity measurements. The names of these files had the same format as this example: /output/peak_finding/ETI1122/ProjectPico/T02/st021-000006_peaks.h5

Once trained, the model was stored on the SherlockML Filesystem in the Pickle file: /output/pca/principal_components_H45.npz.

We used a standard implementation of IPCA from the popular library Scikit-Learn. The code was rather standard, with the exception of the following detail. Each input HDF file contained a matrix with more columns than rows. As a matter of fact, each column represented an electrical harmonic, and each rows described 1 second of electricity readings from a 30-minute series. This implied that the input HDF5 files contained matrices with dimensions (1800, 4000) approximately. This implied that the IPCA model could not be trained directly on these matrices, as the algorithm is ill defined whenever the input table is wider than tall. As such, the function accumulate_h5_from_sfs provided in the module util.py so as to vertically stack matrices. This produced tables that contain enough rows, and made it possible to train the IPCA algorithm.

9.3 <PCA2> Applying the IPCA model

The IPCA algorithm trained in section 9.2 was applied on each HDF5 data file summarising the electrical harmonics with 50 data points. The resulting chunks of processed data were concatenated into a single time-series. A reduction from around 8000 features to 50 principal components ensured that the merged dataset would fit into the memory of a single server.

Results were saved on the SherlockML File-system in the NPZ file /output/pca/principal_components_H45.npz. More precisely, the top 50 principal components were stored alongside the corresponding UTC date and time. The NPZ file then contained two keys:

- "principal_components": the dimensionally reduced electricity data
- "utc_datetime": the UTC date and time identifying each row in the dataset.

Our code was quite standard, as it relied on the familiar library Scikit-Learn. The only slight difficulty concerned the generation of a unique date-time index for the entire dataset of the house H45. The HDF5 files describing electrical harmonics specified the number of seconds since the beginning of a 30-minute measurement. In order to create a longer time index, information was required about the start date and time of each HDF5 file. We found this information in the file /output/data_summary/df_logs_09102017.pkl, previously assembled from the data collector's own logs. The continuous time index utc_datetime was thus produced.

10 Preparing a Unified Working Dataset

10.1 Overview

Following the data cleanse and data quality investigations of the various data categories, it is necessary to generate a single view of the property over time by bringing the various datasets together. The electricity timestamps is used as the master timestamps on which to merge the other data, as it is recorded at the highest frequency and is the only dataset that has data every second. The steps used to merge the data can be broken into three categories:

1. Water, bathroom humidity and boiler pipe temperature data are up sampled and synchronised to electricity timestamps, producing a few unified dataset (see Section 10.2).
2. Electricity data are merged onto the working dataset (see Section 10.3).
3. Exogenous factors are added in, producing the full version of the unified dataset, as shown in Table 9 (see Section 0).

H45	Utilities and HEMS								Exogenous factors			
	Electricity				Water flow	Hot water temp	Bathroom humidity	Hot water usage	Light or dark	Time of day	Day of week	Time of year
	PC1	PC2	…	PC50								
t0												
t0 + 1 sec												
t0 + 2 sec												
…												

Table 9: Illustrative view of the unified dataset produced by following steps in section 10

10.2 <SYNC1> Upsampling and synchronising water, bathroom humidity and boiler pipe temperature data to electricity timestamps

This section produces a first iteration of a few unified datasets by systematically stepping through the notebook. A unified dataset is produced for the time period range appearing in each water file, and these multiple files are merged into one master file in section 10.3. The steps can be broken down in 4, as per the below text.

H45	Water flow	Hot water temp	Bathroom humidity
t0			
t0 + 1 sec			
t0 + 2 sec			
…			

Table 10: Illustrative view of the unified dataset produced by following the steps in section 10.2

Step 1 - prepare the water data by processing each water data file individually:

- Ensure the time zone is UTC for each file. This requires a manual correction to be applied for BST months and no correction for GMT months.
- Integrate the water flow readings to generate a cumulative time series of water usage.
- In future runs, if data gaps are encountered, it will be necessary to insert null values of water flow to ensure the time series is continuous. This will make it easier to up sample the water data via a readily available implementation of linear interpolation e.g. numpy.interp.

Step 2 - merge water usage, bathroom humidity and boiler pipe temperature data:

- Outer join of three time series to the electricity timestamps. Although, we want the data at a 1 second time interval and an outer join means we get lots of additional time stamps, this is needed as we can only remove the additional time stamps after we have up sampled and interpolated the data.
- The three time series are up sampled to c. 1 second frequency using linear interpolation

Step 3 - differentiate cumulative water usage to get water flow rate:

- Differentiate the cumulative water usage values to obtain the water flow rate.

Step 4 - remove incomplete electricity entries:

- The outer merge used in step 2, results in data points occurring between two electricity timestamps. These are identified and removed by removing any incomplete entry for electricity.

10.3 <SYNC2> Merging in electricity data

Section 10.2 merges 4 time series onto the electricity time stamps, but the electricity data is not included in this single property view. In this section, the 50 electrical principal components are added in, by running the notebook iteratively which will run the below steps:

- Concatenate the multiple unified datasets from section 10.2 into a master data file
- Merge in electricity PCs

H45	Utilities and HEMS							
	Electricity				Water flow	Hot water temp	Bathroom humidity	Hot water usage
	PC1	PC2	…	PC50				
t0								
t0 + 1 sec								
t0 + 2 sec								
…								

Table 11: Illustrative view of the unified dataset produced by following steps in section 10.2 & 10.3

10.4 <SYNC3> Include exogenous factors

This section adds four exogenous factors onto the unified working dataset as per Table 12. To run the notebook, it is necessary to configure the input files.

H45	Utilities and HEMS							Exogenous factors			
	Electricity				Water flow	Hot water temp	Bathroom humidity	Light or dark	Time of day	Day of week	Time of year
	PC1	PC2	…	PC50							
t0											
t0 + 1 sec											
t0 + 2 sec											
…											

Table 12: Illustrative view of the unified dataset produced by following steps in section 10

11 Clustering Property States

11.1 Overview

The clustering analysis was only applied to a single house, H45, which found the property was in 1 of 25 states 89% of the time, and in an undefined state the other 11% of the time. The clustering is run on the 50 electricity principal components, and as such, the states represent electricity signatures which probably correspond to property states consisting of specific combinations of electrical appliances. The clustering was run with the purpose of simplifying the property electricity states to aid the predictive algorithms and for future analysis into human workflow patterns.

The input of the clustering algorithm is the top 50 electricity principal components that summarise the electricity data as described in Chapter 9. The output are the arrays *cluster_labels* and *cluster_probabilities*, indicating the state of the house and the corresponding probability, respectively. Further details on the output generated by the code are provided later.

The clustering algorithm chosen is HDBSCAN (Hierarchical Density-Based Clustering of Applications with Noise) for the following reasons:

- As the clusters are defined on the basis of the density of data points, the algorithm is capable of recognising clusters that are not sphere-like. This is important as visualising the principal components reveals that the clusters are irregularly shaped. Other algorithms such as k-Means are not suitable as they assume that the clusters are isotropic.
- HDDSCAN is a relatively fast algorithm that can be trained in parallel using up to 4 cores. This is vital in view of the data size that we need to process.

We train the HDBSCAN algorithm on 65% of the data, as training on 100% of the data results in much higher memory utilisation, and therefore in memory running out. It is worthwhile to note that the cluster labels are saved upon training the model, so that the state of the house for 65% of the data is readily available. We predict on the remaining 35% of the data, and combine this with the known labels so as to label 100% of the data.

One additional complexity is that the meta-parameters of HDSBCAN must be tuned. This means that, rather than training a single model, a variety of models are trained and initialised to have 500 different choices of meta-parameters. To decide which model produces the best labels, scatter plots are generated that display the clusters and data for the top two electrical principal components. An interactive application is then used, that allows one to quickly compare the figures, and thus decide on the best choice of meta-parameters. The optimal values are then employed in the prediction of house states, as described above. As the search for the best meta-parameters is computationally intensive, we make use of our custom-made infrastructure for parallel processing. This framework is then combined with other utilities for high-performance computing, namely JobLib and Dask.

This chapter expands on these topics, and in particular on the functionality of these scripts:

- launch_train_hdbscan_worker.py: Instruct an individual server (a 'worker') to train HDBSCAN for a selection of meta-parameter values.
- launch_train_hdbscan_master.py: initialise worker servers, and assign to each of them a selection of meta-parameter values for processing.

- apply_hdbscan.py: Apply an HDBSCAN model that gets pre-trained on 65% of the data on the remaining 35% of the data.

11.2 <EC1> Running different meta-parameters

Launch_train_hdbscan_worker.py clusters the data using the HDBSCAN algorithm. In order to tune HDBSCAN, it is important to vary two meta-parameters, *min_cluster_size* and *min_samples*, and it is important to note that the algorithm assigns the label '-1' to data points that do not belong to a cluster. The detailed mechanics of the algorithm can be found here: http://hdbscan.readthedocs.io/en/latest/how_hdbscan_works.html.

The *min_cluster_size* indicates the minimum number of points that a cluster must contain in order for the algorithm to recognise it as a cluster. Low values of this meta-parameter tend to result in arguably valid clusters being fragmented into smaller units. High values, on the other hand, can force valid clusters to merge into larger units. Both scenarios are undesirable, thus, a careful optimisation is needed. Because each data point represents one-second worth of electricity measurements, the minimum number of points per cluster (min_cluster_size) is actually an extent of time, i.e. a cluster of 1,000 points is 1000 seconds. This provides an intuitive figure of how many minutes over the total duration of the recording get attributed to a cluster. Motivated by these considerations, we replace the parameter min_cluster_size with the proxy minutes_per_cluster throughout our code. The two are linked via the simple equation: *minutes_per_cluster = 60 seconds x min_cluster_size*.

The *min_samples* indicates the level of confidence when a point is assigned to a cluster. Low values of this meta-parameter generate a 'resolute' clustering model, so that very few ambiguous cases are labelled "-1". High values produce a 'cautious' clustering model, so that very many ambiguous cases are labelled "-1". Again, tuning is required in order to appropriately set the level of confidence.

As mentioned previously, in order to avoid running out of memory, the HDBSCAN models are trained on 65% of the samples. This part of the dataset is randomly selected, but the instances of HDBSCAN with different values of the meta-parameters are trained on the same data. To ensure that this is the case, the random seed that initialises the selection of rows is fixed, implying that that the same pick is carried out for all values of meta-parameters.

In the implementation of HDBSCAN that was used, the trained model keeps the cluster labels in memory. This allows us to apply the algorithm on the remaining 35% of data, and simply merge these new results with the 65% stored previously. While this strategy is an efficient way to cluster 100% of the dataset, one must keep careful track of which rows belong to the training dataset and which to the prediction dataset. For this purpose, we retain the indices corresponding to the 65% of samples.

For each combination of the meta-parameters, the output of *launch_train_hdbscan_worker.py* is a Pickle file that stores a dictionary with the following entries:

- "hdbscan" (instance of hdbscan.HDSCAN): a trained clustering model.
- "sample_ix" (array): indices of the rows forming 65% of the dataset.

The Pickle files are saved in the folder "/output/hdbscan/" of the SherlockML Filesystem. To clarify the naming convention used for these files, it is useful to consider the example: *hdbscan_H45_min_samples_5_minutes_per_cluster_35.pkl.* This stores a model trained on data from the house H45, whose parameters min_samples and minutes_per_cluster take the values 5 and 35, respectively.

The tuning of meta-parameters is a parallelizable process. Different combinations of the meta-parameters are passed to the workers, whose job is to train separate instances of HDBSCAN. The workers operate independently, and do not need to exchange information. We can therefore employ our custom-made infrastructure for parallel computing so as to train 500 models with distinct meta-parameters. As detailed in the script *launch_train_hdbscan_master.py*, 5 servers with 4 cores and 64GB of RAM each are used for this purpose. Given fixed values of the meta-parameters, the specific instance of HDBSCAN is trained using 4 CPU cores. This is possible because HDBSCAN is implemented using JobLib, a framework for parallel computing.

Joblib is compatible with various schedulers that allocate tasks amongst the cores. The default scheduler is multiprocessing, and other options include ipyparallel and Dask. It was observed that, when training HDBSCAN models, the multiprocessing scheduler fails. This default back-end makes use of temporary Pickle files to exchange data between the CPU cores. At the same time, the standard implementation of HDBSCAN allocates a bulky chunk of data to the cores, which results in very large intermediate files. Because Pickle is unable to store so much data, an error occurs. To fix this issue, we switched the scheduler from multiprocessing to Dask. Instructions on how to change the back-end of JobLib are available here (http://distributed.readthedocs.io/en/latest/joblib.html).

A further issue that we encountered is the build up of memory utilisation. One would expect that, after each instance of HDBSCAN is trained, the memory usage returns to a fixed base level. It turns out, however, that the training of models in our code generates a slow increase in the base consumption of memory. This results in the calculation running out of memory, and therefore stopping when a peak of usage occurs. Although we could not identify the precise causes of the issue, it is likely that the memory build up originates from a few variables not being deleted once they are out of use. To overcome the problem, a mechanism was introduced that allows one to manually restart the code, and continue the job from where it was left. The reset forces a deletion of unused variables, clearing up enough memory for the calculation to proceed. In simple terms, the procedure that allows the code to smoothly resume its work is analogous to an old-fashioned bookmark. As soon as the instance of HDBSCAN has been trained, the program creates a dummy file in the workspace, indicating that the task has been completed. After the restart, the calculation ignores any task that corresponds to a dummy file, and moves on to training new instances of HDBSCAN. This is similar to the use of a bookmark to specify which page a reader should resume from.

11.3 Visual selection of meta-parameters

Approximate optimal values of the meta-parameters were selected by visual inspection. For each pair of *min_samples* and *minutes_per_cluster*, a figure was generated of the first two principal components with the cluster labels displayed as colours. Three data scientists were assigned distinct combinations of meta-parameters, and were asked to produce shortlists of the HDBSCAN models with good performance. Finally, the data scientists compared the shortlisted figures, reaching an agreement on the optimal values of *min_samples* and *minutes_per_cluster*.

To assist the data scientists in comparing 500 instances of HDBSCAN, we created an interactive application, as displayed in Figure 10. Drop-down menus allow one to vary the meta-parameters. Scatter plots are displayed that visualise the clusters the origin (Left) and overall (Right). In other words, the figure on the left-hand side is a zoomed-in version of that on the right-hand side. As the first two principal components are related to the active and reactive power of the fundamental harmonic, the zoomed-in plot can be taken to describe the low-power appliances. A further remark is that the colour grey indicates points that are too ambiguous to be attributed to a cluster.

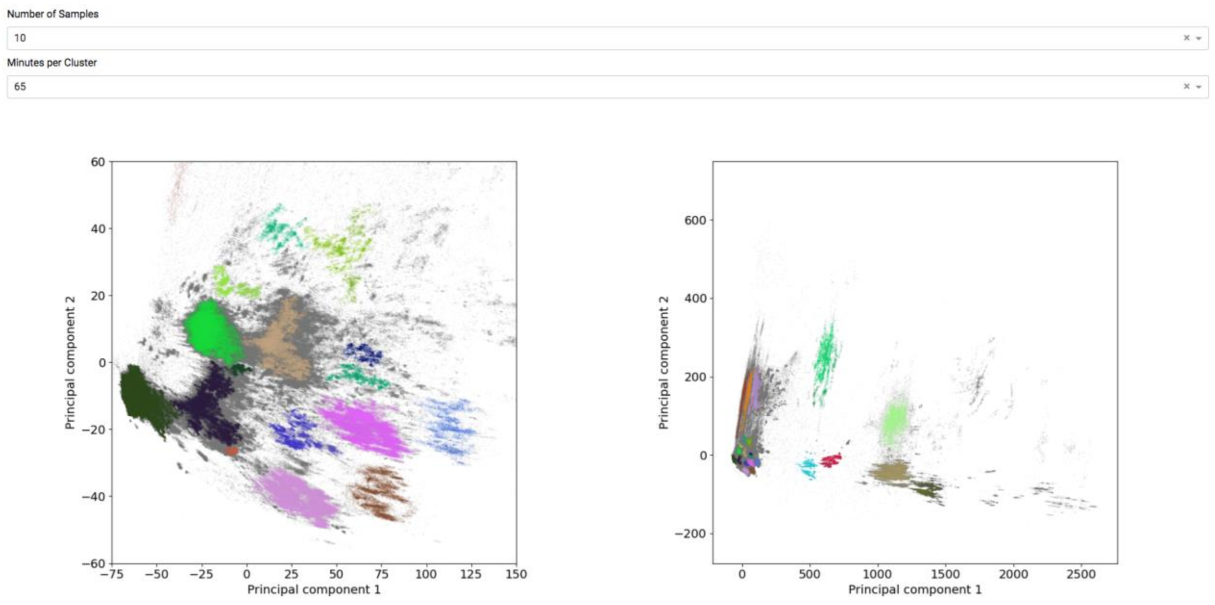

Number of Samples: 10
Minutes per Cluster: 65

Figure 10: Application to visualise results of different meta-parameters combinations, displaying the results for the final meta-parameters chosen

The criteria applied when assessing the performance of models are worth a brief discussion. We penalised the following scenarios:

1. The model is unable to decide the cluster assignment of many points. Equivalently, large areas in the figures are grey.
2. The model merges together clusters that are visually disconnected.
3. The model splits well-defined clusters into smaller units.

The optimal values of *min_samples* and *minutes_per_cluster* were found to be 10 and 65, respectively.

11.4 <EC2> Predicting with HDBSCAN

In the script *apply_hdbscan.py*, the instance of HDBSCAN with optimal parameters is applied on the 35% of data put aside earlier due to memory constraints. The resulting cluster labels and probabilities are merged with the precomputed 65%, so that one obtains predictions for 100% of the data.

The output is saved in the folder /output/hdbscan/ of SherlockML Filesystem using the NPZ file format. Because the clustering is run on data from the house H45 only, *apply_hdbscan.py* produces a single file, */output/hdbscan/labels_and_probabilities_H45.npz*. Results are stored as datasets of the NPZ file format, under the following keys:

- "cluster_labels" (array): The cluster assigned to each data point.
- "cluster_probabilities" (array): The probability that a point belongs to the assigned cluster, as opposed to any other. This describes the level of confidence in each cluster label.

- “utc_datetime” (array): The UTC date and time of each data point.

It is worth noting that the cluster labels generated when applying HDBSCAN on new data-points involve an approximation. This makes such predictions less reliable than those calculated while training the model. As an outcome, the algorithm is expected to have marginally worse performance on the additional 35% of data. Indeed, among these data-points, we observed a small increase in the percentage of examples assigned to the ‘unresolved’ cluster labelled as “-1”.

Because applying a pre-trained instance of HDBSCAN on a dataset is a relatively fast operation, we do not employ infrastructure for parallel computing. New cluster labels are computed by the function *approximate_predict* from the module hdbscan using a single CPU core.

In order to monitor the calculation, HDBSCAN is applied on sequential chunks of data, and progress is reported as soon as a chunk has been processed.

12 Labelling Property Occupancy

12.1 Overview

This chapter describes how we created a label of occupancy specifying whether someone is at home. Even though our approach relies on annotating the dataset manually, it also employs the results of advanced algorithms, such as HDBSCAN. We implemented the following strategy:

1. The HDBSCAN electricity clusters are separated into two groups, depending on whether they represent states that are *autonomous* or *non-autonomous*. We consider states consinsting only of devices that can operate without a person in the vicinity (for example, the WiFi router) as *autonomous*, and states that have a device that require human supervision (for example, the kettle) as *non-autonomous*. These definitions are useful because the occurrence of autonomous clusters may indicate that the house is free. Similarly, the appearance of non-autonomous clusters can signify that the house is occupied. As explained below, we decide on the 'nature' of each cluster by surveying the opinions of 14 data scientists.
2. Various metrics, including the probability that a cluster is not autonomous, are plotted as a function of time. By inspecting these visualisations, we determine for each 10-minute interval whether the house is occupied. This label of occupancy is based on the careful work of a single data scientist.

The section focuses on the approach followed, rather than the code used, as the exercise was conducted as a one-off exercise. Given that it took around one full day of an individual looking at the data to label one property month, it is felt that if future work were to be conducted on the dataset, that it would make sense to build a more time effective way of labelling it. As such, the code was designed as a one-off exercise and has not been shared, and the real value from this section is in the label generated.

12.2 Autonomous and non-autonomous clusters

This section considers our procedure for identifying whether a cluster is autonomous. An interactive application that presents each cluster with heat maps and histograms was developed. We then asked 14 data scientists to consider whether a cluster was autonomous (0), non-autonomous (1), or mixed (0.5). The responses were averaged so as to obtain the probability that a cluster is not autonomous.

Figure 11 presents a screenshot of our interactive application. The drop-down menu allows one to select the cluster to be examined. On the left-hand side, a heat map is shown that summarises how many times the cluster appears as a function of the hour (x-axis) and the date (y-axis). On the right-hand side, three histograms display the same data in terms of hourly profiles for working days (Mon-Fri), weekends (Sat-Sun), and all days combined.

In order to decide whether a cluster is autonomous, the 14 data scientists relied, to a good extent, on the following observation. The appearances of autonomous clusters are distributed uniformly across the day. Non-autonomous clusters, on other hand, tend to:

1. *Occur around mealtimes.* This is related to the fact that cooking appliances require human supervision.

2. *Exhibit distinct patterns over the weekend.* For example, one usually watches television in the evening on weekdays, and from the afternoon to late at night on weekends.
3. *Make no appearance between 1am and 4am.* Over this time range, we expect the occupants to be asleep, and the autonomous clusters to be predominant.

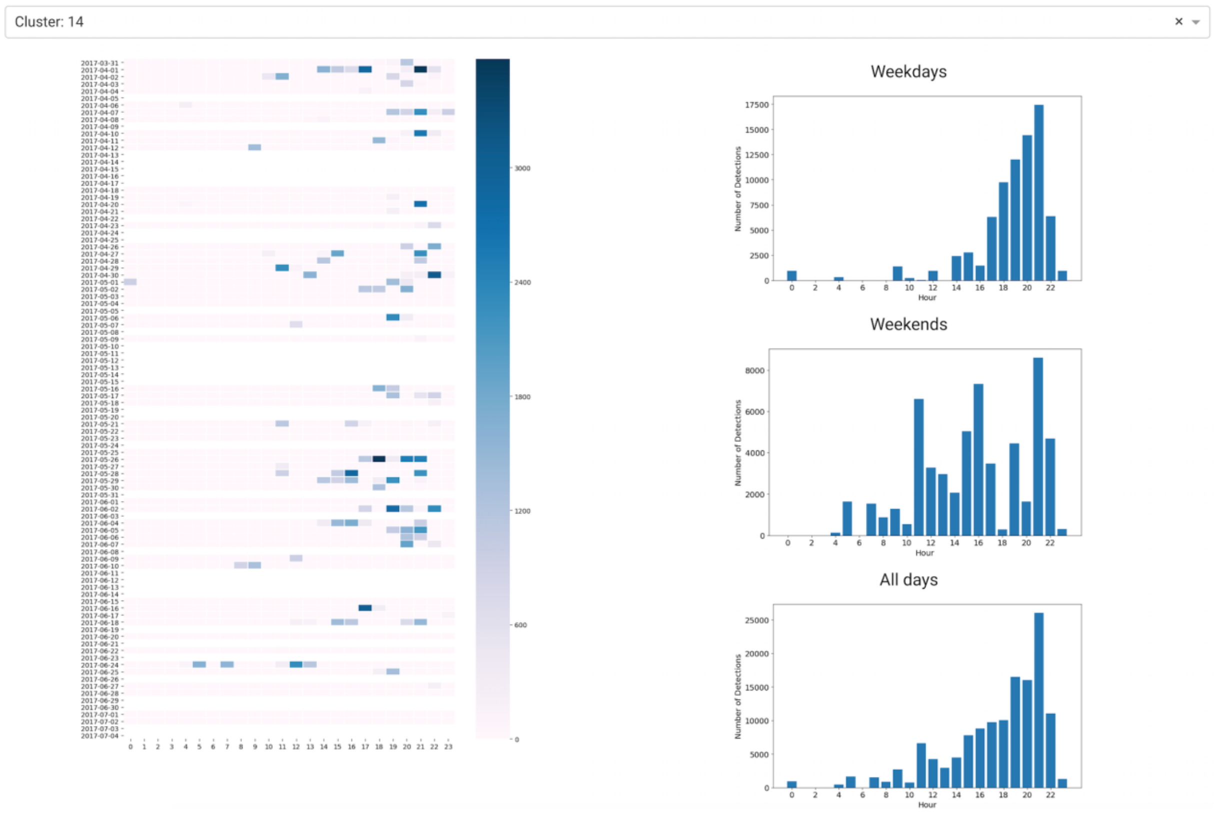


Figure 11: Interactive application displaying plots relating to cluster to help label whether it is autonomous or not

The *probability* that a cluster is not autonomous was calculated by averaging the responses (0, ½, 1) of the 14 data scientists.

12.3 Labelling property occupancy

An important step towards developing a model that forecasts occupancy is to generate an appropriate occupancy label. In this section, we explain a method for identifying whether the house is occupied over a 10-minute interval. This label of occupancy is based on the analysis of a single data scientist. We recommend that, in future projects, the opinions of multiple experts are collected so as to make the label as objective as possible.

To speed up the task of analysing the data, an interactive application was developed, see Figure 12. For each time interval, the user was asked to decide whether the house is occupied or unoccupied. The values of different quantities in the previous and subsequent 10 hours were displayed as context. More specifically, we provided time-series visualisations of:

- The probability that the HDBSCAN has detected a cluster that is not autonomous.
- The rate of water consumption in litres per second
- The temperature of the hot-water pipe in degrees Celsius

Daytime and night-time were represented by the yellow and blue bars at the top of the figure. In a

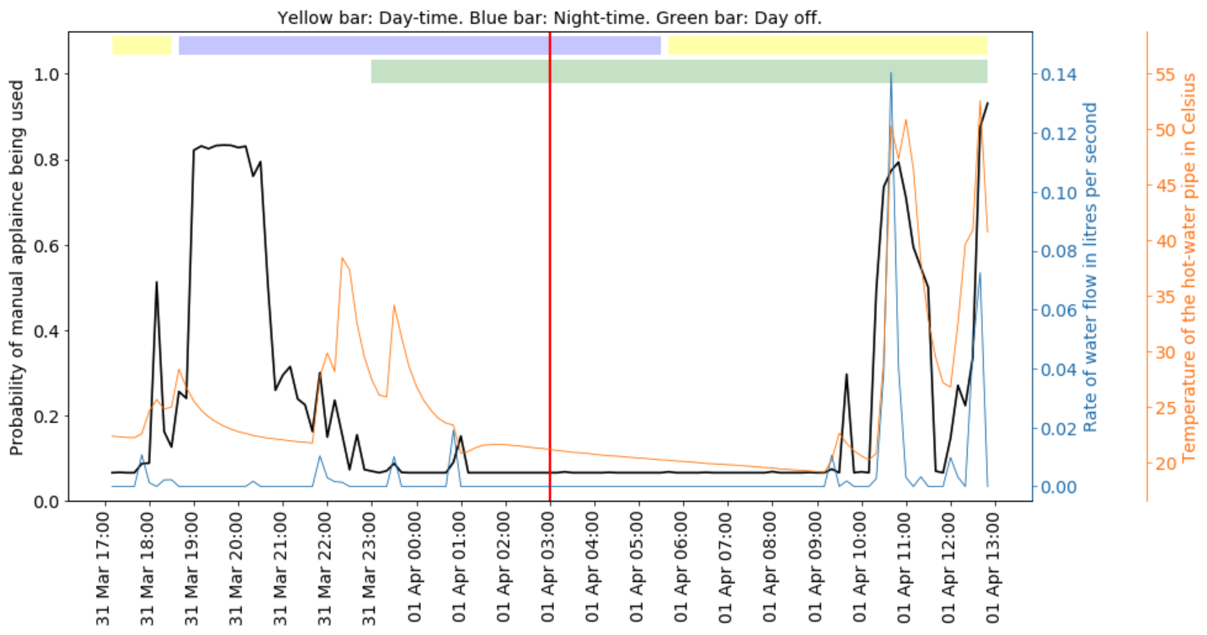


Figure 12: In the interactive visualisation, the graph was accompanied by two buttons. These allowed the user to decide whether the house is occupied or free.

similar way, we used a green bar so as to emphasize weekends and holidays.

The list below ranks the factors that the data scientist examined when generating the label of occupancy from the most to the least important:

1. High probability of the HDBSCAN cluster being 'non-autonomous' was taken to signify that the house is occupied.
2. If activity was detected around bedtime and wake-up time, we labelled the hours in-between as 'occupied', regardless of the night appearing to be quiet. As a matter of fact, one can assume that the occupants prepared for sleep in the evening, rested, and got up in the morning.
3. Mid-to-high consumption of hot water was deemed to imply that the house is occupied.
4. Breakfast, lunch and dinner were analysed with a mild expectation that occupants are at home. For example, the occurrence of autonomous clusters was interpreted as cooking. Lack of activity around mealtimes, on the other hand, was considered to indicate that the house is empty.
5. Periods of quiet that lasted more than 15 hours, especially before a weekend or a holiday, were attributed to the occupants being out of town.

13 Building Predictive Models of Property Occupancy

13.1 Overview

This Chapter focuses on building, optimising and testing the predictive models for future occupancy. This work has been broken down into two broad sections, based on the two Python Notebooks used, which consist of a data engineering phase and a running the predictive model phase. A few key points to draw the reader's attention to are:

- One Notebook is used for all predictive models, and to test different models or different time horizons, it is necessary to simply change the parameterisation in the Notebook
- The section leverages Scikit-Learn, more specifically, the Random Forest Classifier library is key to the work
- The Notebooks do not focus on the optimisation of the models, as this was done by iterating through different versions using a cross-validation dataset, and the final model is stored.

13.2 <OM1> Feature engineering

The first key step in building the occupancy model is to run the Python Notebook that prepares the dataset. As per usual, the Notebook starts by importing the key libraries and then parameterising the key variables, meaning that different configurations can easily be run by changing the parameterisation at the top of the Notebook.

The key steps the Notebook carries out are:

- Load key datasets, from previous steps, into memory;
- Merge loaded datasets and make small changes i.e. round date time to the nearest second;
- Prepare target variable by loading and processing i.e. sorting and up-sampling from 10 minute readings to 1 second readings;
- Merge target variable onto master dataset;
- Minimal feature engineering i.e. create dummy variables (month, day off, hour of day, mealtime & sleep time) and exponentially weighted moving averages for principal components, historical occupancy and water flow rates.

In order to run this Notebook, it is important to have run the earlier dependencies discussed in this report, as illustrated in Figure 2. It is then necessary to choose the right parameterisations i.e. property ID, file names, and key parameterisations of the engineered features (for instance, meal times, holiday dates, and half-life of exponentially weight moving average). The output of this section is an HDF5 that directly feeds into the next section.

13.3 <OM2> Training a predictive model

Following preparation of the master dataset as an HDF5 file in section 13.2, it is possible to run the predictive model with different configurations on different time periods. This section described the

Notebook that is used to run the predictive model and share the results. The Notebook is shared in its final format, which represents the final parameterisation for the full dataset, but prior to this state it was necessary to run various optimisations to determine the variables. For instance, the min number of leafs in the random forest was calibrated using a cross-validation dataset, which can be achieved by running the Notebook multiple times with different parameterisations. In this section, the focus is on running the Notebook, as opposed to the variations necessary and process to tweak it for cross-validation and optimising the variables or parametrisation of the random forest.

Given a choice of final model parameters and variables, the Notebook can be run to predict the target variable for the test data and report on performance i.e. ROC plot and feature importance plot, and it also saves the Random Forest object created as a Pickle file. The key input to the Notebook is the HDF5 output in section 13.2, and as per usual, the Notebook starts by importing the key libraries and then parameterising the key variables. The key steps the Notebook carries out are:

- Import libraries and parametrise as required i.e. property ID, min sample leaf size, number of trees, balance classes, and model's predictive time horizon
- Make final data preparations required i.e. shift target variable by time horizon required and separate training and testing data
- Apply random forest classifier and report key figures i.e. ROC plot, the area under the curve, and feature importance plot
- Save results as a Pickle

14 Building Predictive Models of Hot Water Usage

14.1 Overview

Starting from the unified working dataset generated in section 10.4, this section builds a hot water usage predictive model for each of the five time horizons agreed. These are 10 minutes, 1 hour, 4 hours, 24 hours & 72 hours. For each time horizon, the hot water usage value is not directly predicted, but instead it is discrsed into low, medium and high usage and the class probability is predicted.

The development of the models can be broken down into four steps:

1. Create the target variables, the hot water usage class labels;
2. Generate new features to improve predictive performance;
3. Build and optimise a random forest predictive model for each of the five time horizons.
4. Test the predictive models on a new independent dataset to get a fair measure of performance.

14.2 <SYNC3, HWM> Labelling hot water usage

Hot water usage is labelled as low, medium or high for each time horizon (72 hours was a slight exception as it did not have a medium usage, just a low and high). The labelling is done by running a Notebook end to end, which carries out 3 main steps described below.

Step 1 (generate binary hot water usage label):

- Sync3 is used to calculate the hot water usage binary label, which has been set to 1 wherever the property is using water and the pipe temperature is above 40°C (HW_TEMP_THRES=40). This construct is a proxy for hot water usage and may need to be refined for different properties.

Step 2 (calculate hot water usage for the time horizon): - hwm1…

- Using HWM1 to HWM5 (one for each time horizon), a function calculates the time period within the time range for which hot water is used i.e. for 10 minutes in the following 4 hours. This is the target variable used in the predictive model.

Step 3 (discrsed hot water usage into low, medium & high classes):

- Using HWM1 to HWM5 (one for each time horizon), the hot water usage time is discrsed into low, medium and high usage based on pre-defined time usage thresholds. The thresholds were determined through data exploration and understanding of human behaviours over the time windows, but can be easily redefined by reparametrizing the variables. For instance, if the time horizon of the predictive model is 4 hours, then the boundaries agreed are 30 seconds and 120 seconds, meaning the following variables would be set as follows:
    - HW_4HR_LM_BOUNDARY = 30
    - HW_4HR_MH_BOUNDARY = 120

14.3 <HWM> Feature engineering

Feature engineering is key to predictive performance and this section enhances the single property view with additional data features. For each of the five time horizons there is a notebook (HWM1 to HWM5), and each of these need to be run. There are five broad data feature types that are created by running the notebooks, which are the following:

1. Electricity state clusters are generated as described in Chapter 11. It is recommendable to discard the "-1" cluster, as this only represents unidentified statuses;
2. The consecutive time for which a variable is on or off leading up to the last data point. Variables used are water usage, hot water usage and electricity state clusters;
3. Increments of values over time, which was used for boiler pipe temperatures and bathroom humidity;
4. Total water usage and hot water usage over pre-defined time periods i.e. 24 hours. This captures repetitive behaviour i.e. one shower every 24 hours.
5. Memory captured through exponentially weighted moving average for electricity principal components, boiler pipe temperatures, bathroom humidity, water usage, hot water usage and electricity state cluster

14.4 <HWM> Training a predictive model with cross-validation

Starting from the output of section 14.3, a different notebook is used for each time horizon to train and validate various configurations of the predictive models. This is necessary to select between different meta-parameters and to select a good combination of features for a final model. At the top of the relevant section of the notebook, it is necessary to parameterise the run appropriately including the features to use and the meta-parameters. The steps the notebook runs are detailed below.

- It is necessary to isolate some of the data for testing, so as to not use this data in anyway for the training or validation of the models. In the HFADA project, the last 7 days of the 30 day dataset used were isolated for testing and the rest of the data was used for training and validation.
- Train and validate various random forest models with different sets of features or/and meta-parameters to select a preferred model. A few details worth noting are:
  - The number of folds in the cross validation can be specified using the variable 'k_fold', which was set to 3 for the project work
  - Up sampling is applied to the data to ensure a balanced class problem, which can improve predictive performance of the underweighted classes. The configuration used for the project was a ratio of c. 1:1:1 for the low:medium:high. To change the ratio the code found just after where the parameters are set, will require some rewriting.
  - There are many meta-parameters that can be used to optimise a random forest, in this work "*min_samples_leaf*" is used. The parameter is set at the top of the section of the notebook and a range of values are tested for each time horizon, selecting a different optimum value per time horizon.
- Report model performance with the validation set. This concept carries the view that previous hot water usage should be used as a basis to determine hot water usage during the corresponding time period in the future e.g. previous 10 minutes of hot water usage to be

used as the benchmark prediction for the forthcoming 10 minutes. In order to evaluate overall model performance, the following metrics have been included:
    - Area under the curve for a receiver operating characteristic curve;
    - Mean Absolute Errors, Root Mean Squared Errors and r-squared values;
    - Feature importance.

14.5 <HWM> Testing model performance

To get an unbiased performance metric for the final model, it is necessary to train the model on the full training and validation dataset and test it on a new dataset. To do this, the last section of the notebook for the corresponding time horizon should be run. Various performance metrics are calculated for the test set, as detailed below:

- Area under the curve for a receiver operating characteristic curve;
- Mean Absolute Errors, Root Mean Squared Errors and r-squared values;
- Feature importance.

15 Appendix A: Data Quality Report

Please see separate PDF file titled “Data Quality Report”.

16 Appendix B: ESC's Data Specifications

Please see separate document titled "Data collection and data format – ELECTRIC, WATER and HEMS-V1 MONITORING.docx".