NILM Metadata Documentation

Release 0.2.0

Jack Kelly

Apr 03, 2017

Contents

Contents:

CHAPTER 1

NILM Metadata Tutorial

Before reading this tutorial, please make sure you have read the NILM Metadata README which introduces the project. Also, if you are not familiar with YAML, please see the WikiPedia page on YAML for a quick introduction.

NILM Metadata allows us to describe many of the objects we typically find in a disaggregated energy dataset. Below is a UML Class Diagram showing all the classes and the relationships between classes:

A dark black diamond indicates a 'composition' relationship whilst a hollow diamond indicates an 'aggregation'. For example, the relationship between `Dataset` and `Building` is read as '*each Dataset contains any number of Buildings and each Building belongs to exactly one Dataset*'. We use hollow diamonds to mean that objects of one class *refer* to objects in another class. For example, each `Appliance` object refers to exactly one `ApplianceType`. Instances of the classes in the shaded area on the left are intended to be shipped with each dataset whilst objects of the classes on the right are common to all datasets and are stored within the NILM Metadata project as the 'central metadata'. Some `ApplianceTypes` contain `Appliances`, hence the box representing the `Appliance` class slightly protrudes into the 'central metadata' area on the right.

Below we will use examples to illustrate how to build a metadata schema for a dataset.

Examples

Simple example

The illustration below shows a cartoon mains wiring diagram for a domestic building. Black lines indicate mains wires. This home has a split-phase mains supply (common in North America, for example). The washing machine draws power across both splits. All other appliances draw power from a single split.

The text below shows a minimalistic description (using the NILM Metadata schema) of the wiring diagram above. The YAML below would go into the file `building1.yaml`:

```
instance: 1 # this is the first building in the dataset
elec_meters: # a dictionary where each key is a meter instance
  1:
    site_meter: true # meter 1 measures the whole-building aggregate
  2:
    site_meter: true
  3:
    submeter_of: 1 # meter 3 is directly downstream of meter 1
  4:
    submeter_of: 1
  5:
    submeter_of: 2
  6:
    submeter_of: 2
  7:
    submeter_of: 6
appliances:
- {type: kettle, instance: 1, room: kitchen, meters: [3]}
- {type: washing machine, instance: 1, meters: [4,5]}
- {type: light, instance: 1, room: kitchen, meters: [7]}
- {type: light, instance: 2, multiple: true, meters: [6]}
```

`elec_meters` holds a dictionary of dictionaries. Each key is a meter instance (a unique integer identifier within the building). We start numbering from 1 because that is common in existing datasets. Each value of the `elec_meters` dict is a dictionary recording information about that specific meter (see the documentation on the *ElecMeter* schema for full information). `site_meter` is set to `true` if this meter measures the whole-building aggregate power demand. `submeter_of` records the meter instance of the upstream meter. In this way, we can specify wiring hierarchies of arbitrary complexity.

`appliances` is a list of dictionaries. Each dictionary describes a single appliance. The appliance `type` (e.g. 'kettle' or 'washing machine') is taken from a controlled vocabulary defined in NILM Metadata. See the *Appliance* schema for more information.

For each appliance, we must also specify an `instance` (an integer which, within each building, allows us to distinguish between multiple instances of a particular appliance `type`). We must also specify a list of `meters`. Each element in this list is an integer which corresponds to a meter `instance`. In this way, we can specify which meter is directly upstream of this appliance. The vast majority of domestic appliances will only specify a single meter. We use two meters for north-American appliances which draw power from both mains legs. We use three meters for three-phase appliances.

See the documentation of the *Dataset metadata* for a full listing of all elements which can be described, or continue below for a more detailed example.

Representing REDD using NILM Metadata

The Reference Energy Disaggregation Data set (REDD) (Kolter & Johnson 2011) was the first public dataset to be released for the energy disaggregation community. It consists of six homes. Each home has its whole-home aggregate power demand measured and also has its circuits measured. REDD provides both low frequency (3 second sample period) and high frequency data. We will only specify the low frequency data in this example.

NILM Metadata can be specified as either YAML or as metadata within an HDF5 binary file. YAML is probably best for distribution with a dataset. HDF5 is used by NILMTK to store both the data itself and the metadata. The data structures are very similar no matter if the metadata is represented on disk as YAML or HDF5. The main difference is where the metadata is stored. In this example, we will only consider YAML. The YAML files are stored in a `metadata` directory included with the dataset. For details of where this information is stored within HDF5, please see the relevant sections of the *Dataset metadata* page.

First we will specify the details of the dataset, then details about each building.

Dataset

We will use the *Dataset schema* to describe the name of the dataset, authors, geographical location etc. If you want to create a minimal metadata description of a dataset then you don't need to specify anything for the `Dataset`.

This information would be stored in `dataset.yaml`.

First, let us specify the name of the dataset and the creators:

```
name: REDD
long_name: The Reference Energy Disaggregation Data set
creators:
- Kolter, Zico
- Johnson, Matthew
publication_date: 2011
institution: Massachusetts Institute of Technology (MIT)
contact: zkolter@cs.cmu.edu   # Zico moved from MIT to CMU
description: Several weeks of power data for 6 different homes.
subject: Disaggregated power demand from domestic buildings.
number_of_buildings: 6
timezone: US/Eastern   # MIT is on the east coast
geo_location:
  locality: Massachusetts   # village, town, city or state
  country: US   # standard two-letter country code defined by ISO 3166-1 alpha-2
  latitude: 42.360091 # MIT's coorindates
  longitude: -71.09416
related_documents:
- http://redd.csail.mit.edu
- >
  J. Zico Kolter and Matthew J. Johnson.
  REDD: A public data set for energy disaggregation research.
  In proceedings of the SustKDD workshop on
  Data Mining Applications in Sustainability, 2011.
  http://redd.csail.mit.edu/kolter-kddsust11.pdf
schema: https://github.com/nilmtk/nilm_metadata/tree/v0.2
```

The nominal mains voltage can be inferred from the `geo_location:country` value.

Meter Devices

Next, we describe the common characteristics of each type of meter used to record the data. See the documentation section on *MeterDevice* for full details. You can think of this as the 'specification sheet' supplied with each model of meter used to record the dataset. This information would be stored in `meter_devices.yaml`.

This data structure is one big dictionary. Each key is a model name. Each value is a dictionary describing the meter:

```
eMonitor:
  model: eMonitor
  manufacturer: Powerhouse Dynamics
  manufacturer_url: http://powerhousedynamics.com
  description: >
    Measures circuit-level power demand.  Comes with 24 CTs.
    This FAQ page suggests the eMonitor measures real (active)
    power: http://www.energycircle.com/node/14103  although the REDD
    readme.txt says all channels record apparent power.
```

```
  sample_period: 3   # the interval between samples. In seconds.
  max_sample_period: 50   # Max allowable interval between samples. Seconds.
  measurements:
  - physical_quantity: power   # power, voltage, energy, current?
    type: active   # active (real power), reactive or apparent?
    upper_limit: 5000
    lower_limit: 0
  wireless: false

REDD_whole_house:
  description: >
    REDD's DIY power meter used to measure whole-home AC waveforms
    at high frequency.  To quote from their paper: "CTs from TED
    (http://www.theenergydetective.com) to measure current in the
    power mains, a Pico TA041 oscilloscope probe
    (http://www.picotechnologies.com) to measure voltage for one of
    the two phases in the home, and a National Instruments NI-9239
    analog to digital converter to transform both these analog
    signals to digital readings. This A/D converter has 24 bit
    resolution with noise of approximately 70 µV, which determines
    the noise level of our current and voltage readings: the TED CTs
    are rated for 200 amp circuits and a maximum of 3 volts, so we
    are able to differentiate between currents of approximately
    ((200))(70 × 106)/(3) = 4.66mA, corresponding to power changes
    of about 0.5 watts. Similarly, since we use a 1:100 voltage
    stepdown in the oscilloscope probe, we can detect voltage
    differences of about 7mV."
  sample_period: 1
  max_sample_period: 30
  measurements:
  - physical_quantity: power
    type: apparent
    upper_limit: 50000
    lower_limit: 0
  wireless: false
```

Buildings, electricity meters and appliances

Finally, we need to specify metadata for each building in the dataset. Information about each electricity meter and each appliance is specified along with the building. Metadata for each building goes into `building<i>.yaml` where *i* is an integer starting from 1. e.g. `building1.yaml`

We will describe `house_1` from REDD. First, we describe the basic information about `house_1` using the *Building* schema:

```
instance: 1   # this is the first building in the dataset
original_name: house_1   # original name from REDD dataset
elec_meters:   # see below
appliances:   # see below
```

We do now know the specific geographical location of `house_1` in REDD. As such, we can assume that `house_1` will just 'inherit' `geo_location` and `timezone` from the `dataset` metadata. If we did know the geographical location of `house_1` then we could specify it in `building1.yaml`.

Next, we specify every electricity meter and the wiring between the meters using the *ElecMeter* schema. `elec_meters` is a dictionary. Each key is a meter instance. Each value is a dictionary describing that meter. To keep this short, we won't show every meter:

```
elec_meters:
  1:
    site_meter: true
    device_model: REDD_whole_house  # keys into meter_devices dictionary
    data_location: house_1/channel_1.dat
  2:
    site_meter: true
    device_model: REDD_whole_house
    data_location: house_1/channel_2.dat
  3:
    submeter_of: 0 # '0' means 'one of the site_meters'. We don't know
                   # which site meter feeds which appliance in REDD.
    device_model: eMonitor
    data_location: house_1/channel_3.dat
  4:
    submeter_of: 0
    device_model: eMonitor
    data_location: house_4/channel_4.dat
```

We could also specify attributes such as `room`, `floor`, `preprocessing_applied`, `statistics`, `upstream_meter_in_building` but none of these are relevant for REDD.

Now we can specify which appliances connect to which meters.

For reference, here is the original `labels.dat` for `house_1` in REDD:

```
1 mains
2 mains
3 oven
4 oven
5 refrigerator
6 dishwaser
7 kitchen_outlets
8 kitchen_outlets
9 lighting
10 washer_dryer
11 microwave
12 bathroom_gfi
13 electric_heat
14 stove
15 kitchen_outlets
16 kitchen_outlets
17 lighting
18 lighting
19 washer_dryer
20 washer_dryer
```

We use the *Appliance* schema to specify appliances. In REDD, all the meters measure *circuits* using CT clamps in the homes' fuse box. Some circuits deliver power to *individual* appliances. Other circuits deliver power to *groups* of appliances.

`appliances` is a list of dictionaries.

Let us start by demonstrating how we describe circuits which deliver power to an individual appliance:

```
appliances:

- type: fridge
  instance: 1
```

```
  meters: [5]
  original_name: refrigerator
```

Recall from the *Simple example* that the value of appliance `type` is taken from the NILM Metadata controlled vocabulary of appliance types. `original_name` is the name used in REDD, prior to conversion to the NILM Metadata controlled vocabulary.

Now we specify two 240-volt appliances. North American homes have split-phase mains supplies. Each split is 120 volts relative to neutral. The two splits are 240 volts relative to each other. Large appliances can connect to both splits to draw lots of power. REDD separately meters both splits to these large appliances so we specify two meters per 240-volt appliance:

```
appliances:

- type: electric oven
  instance: 1
  meters: [3, 4]   # the oven draws power from both 120 volt legs
  original_name: oven

- original_name: washer_dryer
  type: washer dryer
  instance: 1
  meters: [10, 20]
  components: # we can specify which components connect to which leg
  - type: motor
    meters: [10]
  - type: electric heating element
    meters: [20]
```

Now we specify loads which aren't single appliances but, instead, are categories of appliances:

```
appliances:

- original_name: kitchen_outlets
  room: kitchen
  type: sockets   # sockets is treated as an appliance
  instance: 1
  multiple: true   # likely to be more than 1 socket
  meters: [7]

- original_name: kitchen_outlets
  room: kitchen
  type: sockets
  instance: 2   # 2nd instance of 'sockets' in this building
  multiple: true   # likely to be more than 1 socket
  meters: [8]

- original_name: lighting
  type: light
  instance: 1
  multiple: true   # likely to be more than 1 light
  meters: [9]

- original_name: lighting
  type: light
  instance: 2   # 2nd instance of 'light' in this building
  multiple: true
  meters: [17]
```

```
- original_name: lighting
  type: light
  instance: 3   # 3rd instance of 'light' in this building
  multiple: true
  meters: [18]

- original_name: bathroom_gfi   # ground fault interrupter
  room: bathroom
  type: unknown
  instance: 1
  multiple: true
  meters: [12]
```

Note that if we have multiple distinct instances of the same type of appliance then we must use separate appliance objects for each instance and must *not* bunch these together as a single appliance object with multiple `meters`. We only specify multiple `meters` per `appliance` if there is a single appliance which draws power from more than one phase or mains leg.

In REDD, houses 3, 5 and 6 also have an `electronics` channel. How would we handle this in NILM Metadata? This is a meter which doesn't record a single appliance but records a *category* of appliances. Luckily, because NILM Metadata uses an inheritance structure for the central metadata, we already have a `CE appliance` (CE = consumer electronics). The `CE appliance` object was first built to act as an abstract superclass for all consumer electronics objects, but it comes in handy for REDD:

```
- original_name: electronics
  type: CE appliance
  instance: 1
  multiple: true
  meters: [6]
```

The full description of the REDD dataset using NILM Metadata can be found in the NILMTK project along with the metadata descriptions for many other datasets.

Summary

We have seen how to represent the REDD dataset using NILM Metadata. The example above shows the majority of the *structure* of the NILM Metadata schema for datasets. There are many more attributes that can be attached to this basic structure. Please see the *Dataset metadata* documentation for full details of all the attributes and values that can be used.

Central Metadata

A second part to the NILM Metadata project is the 'central metadata'. This 'central metadata' is stored in the NILM Metadata project itself and consists of information such as the mapping of appliance type to appliance category; and the mapping of country code to nominal voltage values. Please see the documentation page on *Central appliance metadata* for more information.

Improving NILM Metadata

The NILM Metadata schema will, of course, never be complete enough to cover every conceivable dataset! You are warmly invited to suggest changes and extensions. You can do this either using the github issue queue, or by forking the project, modifying it and issuing a pull request.

CHAPTER 2

Dataset metadata

This page describes the metadata schema for describing a dataset.

There are two file formats for the metadata: YAML and HDF5. YAML metadata files should be in a `metadata` folder. Each section of this doc starts by describing where the relevant metadata is stored in both file formats.

Dataset

This object describes aspects about the entire dataset. For example, the name of the dataset, the authors, the geographical location of the entire dataset etc.

- Location in YAML: `dataset.yaml`
- Location in HDF5: `store.root._v_attrs.metadata`

Metadata attributes (some of these attributes are adapted from the Dublin Core Metadata Initiative (DCMI)):

name (string) (required) Short name for the dataset. e.g. 'REDD' or 'UK-DALE'. Equivalent DCMI element is 'title'. If this dataset is the output of a disaggregation algorithm then *name* will be set to a short name for the algorithm; e.g. 'CO' or 'FHMM'.

long_name (string) Full name of the dataset, eg. 'Reference Energy Disaggregation Data Set'.

creators (list of strings) in the format '<Lastname>, <Firstname>'. DCMI element.

timezone (string) Please use the standard TZ name from the IANA (aka Olson) Time Zone Database e.g. 'America/New_York' or 'Europe/London'.

date (string) ISO 8601 format. e.g. '2014-06-23' Identical to the 'date' DCMI element.

contact (string) Email address

institution (string)

description (string) DCMI element. Human-readable, brief description. e.g. describe sample rate, geo location etc.

number_of_buildings (int)

identifier (string): A digital object identifier (DOI) or URI for the dataset. DCMI element.

subject (string): For example, is this dataset about domestic or commercial buildings? Does it include disaggregated appliance-by-appliance data or just whole-building data? DCMI element. Human-readable free text.

geospatial_coverage (string): Spatial coverage. e.g. 'Southern England'. Related to the 'coverage' DCMI element. Human-readable free text.

timeframe (*TimeFrame*, see below) Start and end dates for the entire dataset.

funding (list of strings) A list of all the sources of funding used to produce this dataset.

publisher (string) The entity responsible for making the resource available. Examples of a Publisher include a person, an organization, or a service. DCMI element.

geo_location (dict)

locality (string) village, town, city or state

country (string) Please use a standard two-letter country code defined by ISO 3166-1 alpha-2. e.g. 'GB' or 'US'.

latitude (number)

longitude (number)

rights_list

(list of dicts) License(s) under which this dataset is released. Related to the 'rights' DCMI element. Each element has these attributes:

uri (string) License URI

name (string) License name

description_of_subjects (string) A brief description of how subjects were recruited. Are they all PhD students, for example? Were they incentivised to reduce their energy consumption? How were they chosen?

related_documents (list of strings) References about this dataset (e.g. references to academic papers or web pages). Also briefly describe the contents of each reference (e.g. does it contain a description of the metering setup? Or an analysis of the data?) Related to the 'relation' DCMI element.

schema (string) The URL of the NILM_metadata version (tag) against which this metadata is validated. e.g. https://github.com/nilmtk/nilm_metadata/tree/v0.2

MeterDevice

Metadata describing every model of meter used in the dataset. (Please note that *ElecMeter* is used for representing individual *instances* of meters in a building whilst `MeterDevice` is used to represent information common to all instances of a specific make and model of meter). Think of this section as a catalogue of meter models used in the dataset.

- Location in YAML: `meter_devices.yaml`
- Location in HDF5: `store.root._v_attrs.metadata` in `meter_devices`

One big dict. Keys are device model names (e.g. 'EnviR'). The purpose is to record information about specific models of meter. Values are dicts with these keys:

model (string) (required) The model name for this meter device.

model_url (string) The URL with more information about this meter model.

manufacturer (string)

manufacturer_url (string)

sample_period (number) (required) The meter's nominal sample period (i.e. the length of time between consecutive samples) in seconds.

max_sample_period (number) (required) The maximum permissible length of time between consecutive samples. We assume the meter is switched off during any gap longer than `max_sample_period`. In other words, we define a 'gap' to be any two samples which are more than `max_sample_period` apart.

measurements (list) (required) The order is the order of the columns in the data table.

physical_quantity (string) (required) One of {'power', 'energy', 'cumulative energy', 'voltage', 'current', 'frequency', 'power factor', 'state', 'phase angle', 'total harmonic distortion', 'temperature'}. 'state' columns store an integer state ID where 0 is off and >0 refers to defined states. (TODO: store mapping of state ID per appliance to state name). Units: phase angle: degrees; power: watts; energy: kWh; voltage: volts; current: amps; temperature: degrees Celsius.

type (string) (required for 'power' and 'energy') Alternative Current (AC) Type. One of {'reactive', 'active', 'apparent'}.

upper_limit (number)

lower_limit (number)

description (string)

pre_pay (boolean) Is this a pre-pay meter?

wireless (boolean)

wireless_configuration (dict) All strings are human-readable free text:

base (string) Description of the base station used. Manufacturer, model, version etc.

protocol (string) e.g. 'zibgee', 'WiFi', 'custom'. If custom then add a link to documentation if available.

carrier_frequency (number) MHz

data_logger (string) Description of the data logger used

Building

- Location in YAML: `building<I>.yaml`
- Location in HDF5: `store.root.building<I>._v_attrs.metadata`

instance (int) (required) The building instance in this dataset, starting from 1

original_name (string) Original name of building from old (pre-NILM Metadata) metadata.

elec_meters (dict of dicts) (required) Each key is an integer (>= 1) representing the meter instance in this building. Each value is an `ElecMeter`. See section below on *ElecMeter*.

appliances (list of dicts) (required) See section below on *Appliance*.

water_meters (dict of dicts) Same structure as `elec_meters`.

gas_meters (dict of dicts) Same structure as `elec_meters`.

description (string)

rooms (list of dicts):

name (string) (required) one of {'lounge', 'kitchen', 'bedroom', 'utility', 'garage', 'basement', 'bathroom', 'study', 'nursery', 'hall', 'dining room', 'outdoors'}

instance (int) (optional. Starts from 1. If absent then assume to be 1.)

description (string)

floor (int) Ground floor is floor 0.

n_occupants (int) Mode number of occupants.

description_of_occupants (string) free-text describing the occupants. Number of children, teenagers, adults, pensioners? Demographics? Were all occupants away from the house during all week days?

timeframe (*TimeFrame*, see below)

periods_unoccupied (list of *TimeFrame* objects, see below) Periods when this building was empty for more than a day (e.g. holidays)

construction_year (int) Four-digit calendar year of construction.

energy_improvements (list of strings) Any post-construction modifications? Some combination of {'photovoltaics', 'solar thermal', 'cavity wall insulation', 'loft insulation', 'solid wall insulation', 'double glazing', 'secondary glazing', 'triple glazing'}

heating (ordered list of strings, with the most dominant fuel first) Some combination of {'natural gas', 'electricity', 'coal', 'wood', 'biomass', 'oil', 'LPG'}

communal_boiler boolean (set to true if heating is provided by a shared boiler for the flats)

ownership (string) one of {'rented', 'bought'}

building_type (string) one of {'bungalow', 'cottage', 'detached', 'end of terrace', 'flat', 'semi-detached', 'mid-terrace', 'student halls', 'factory', 'office', 'university'}

Building metadata which is inherited from *Dataset* but can be overridden by `Building`:

- geo_location
- timezone
- timeframe

ElecMeter

ElecMeters are the values of the `elec_meters` dict of each building (see the section on *Building* metadata above).

device_model (string) (required) `model` which keys into `meter_devices`

submeter_of (int) (required) the meter instance of the upstream meter. Or set to `0` to mean "*one of the site_meters*". In practice, `0` will be interpreted to mean "downstream of a 'MeterGroup' representing all the site meters summed together".

submeter_of_is_uncertain (boolean) Set to true if the value for *submeter_of* is uncertain.

upstream_meter_in_building (int) If the upstream meter is in a different building then specify that building instance here. If left blank then we assume the upstream meter is in the same building as this meter.

site_meter (boolean): required and set to True if this is a site meter (i.e. furthest upstream meter) otherwise not required. If there are multiple mains phases (e.g. 3-phase mains) or multiple mains 'splits' (e.g. in North America where there are two 120 volt splits) then set `site_meter=true` in every site meter. All non-site-meters directly downstream of the site meters should set `submeter_of=0`. Optionally also use `phase` to describe which phase this meter measures. What happens if there are multiple site meters in *parallel* (i.e. there are redundant meters)? For example, perhaps there is a site meter installed by the utility company which provides infrequent readings; and there is also a fancy digital site meter which measures at the same point in the wiring tree and so, in a sense, the utility meter can be considered 'redundant' but is included in the dataset for comparison). In this situation, set `site_meter=true` in every site meter. Then set `disabled=true` in all but the 'favoured' site meter (which would usually be the site meter which provides the 'best' readings). It is important to set `disabled=true` so NILMTK does not sum together parallel site meters. The disabled site meters should also set `submeter_of` to the ID of the enabled site meter. All non-site-meters directly downstream of site meters should set `submeter_of=0`.

utility_meter (boolean) required and set to True if this is meter was installed by the utility company. Otherwise not required.

timeframe (*TimeFrame* object)

name (string) (optional) e.g. 'first floor total'.

phase (int or string) (optional) Used in multiple-phase setups.

room (string) `<room name>[,<instance>]`. e.g. 'kitchen' or 'bedroom,2'. If no `instance` is specified (e.g. 'room: kitchen' then it is assumed to be 'kitchen,1' (i.e. kitchen instance 1). If the building metadata specifies set of `rooms` then the room specified here will key into the building's `rooms` (but not all datasets enumerate every room for each building).

floor (int) Not necessary if `room` is specified. Ground floor is 0.

data_location (string) (required) Path relative to root directory of dataset. e.g. `house1/channel_2.dat`. Reference tables and columns within a Hierarchical file e.g. `data.h5?table=/building1/elec/meter1` or, if this metadata is stored in the same HDF file as the sensor data itself then just use the key e.g. `/building1/elec/meter1`.

disabled (bool): Set to true if NILMTK should ignore this channel. This is useful if, for example, this channel is a redundant site_meter.

preprocessing_applied (dict): Each key is optional and is only present if that preprocessing function has been run.

clip (dict)

lower_limit

upper_limit

statistics (list of dicts): Each dict describes statistics for one set of timeframes. Each dict has:

timeframes (list of *TimeFrame* objects) (required) The timeframes over which these statistics were calculated. If the stat(s) refer to the entire timeseries then enter the start and end of the timeseries as the only TimeFrame.

good_sections (list of *TimeFrame* objects)

contiguous_sections (list of *TimeFrame* objects)

total_energy (dict) kWh

active (number)

reactive (number)

apparent (number)

Note that some of these statistics are cached by NILMTK at `building<I>/elec/cache/meter<K>/<statistic_name>`. For more details, see the docstring of `nilmtk.ElecMeter._get_stat_from_cache_or_compute()`.

WaterMeter and GasMeter

Same attributes as *ElecMeter*.

Appliance

Each appliance dict has:

type (string) (required) appliance type (e.g. 'kettle'). Use NILM Metadata controlled vocabulary. See nilm_metadata/central_metadata/appliance_types/*.yaml. Each `*.yaml` file in `nilm_metadata/central_metadata/appliance_types` is a large dictionary. Each key in these dictionaries is a legal appliance `type`.

instance (int starting from 1) (required) instance of this appliance within the building.

meters (list of ints) (required) meter instance(s) directly upstream of this appliance. This is a list to handle the case where some appliances draw power from both 120 volt legs in a north American house. Or 3-phase appliances.

dominant_appliance (boolean) (required if multiple appliances attached to one meter). Is this appliance responsible for most of the power demand on this meter?

on_power_threshold (number) watts. Not required. Default is taken from the appliance *type*. The threshold (in watts) used to decide if the appliance is *on* or *off*.

max_power (number) watts. Not required.

min_off_duration (number) (seconds) Not required.

min_on_duration (number) (seconds) Not required.

room see *ElecMeter-room*

multiple (boolean) True if there are more than one of these appliances represented by this single `appliance` object. If there is exactly one appliance then do not specify `multiple`.

count (int) If there are more than one of these appliances represented by this `appliance` object and if the exact number of appliances is known then specify that number here.

control (list of strings) Give a list of all control methods which apply. For example, a video recorder would be both 'manual' and 'timer'. The vocabulary is: {'timer', 'manual', 'motion', 'sunlight', 'thermostat', 'always on'}

efficiency_rating (dict):

certification_name (string) e.g. 'SEDBUK' or 'Energy Star 5.0'

rating (string) e.g. 'A+'

nominal_consumption (dict): Specifications reported by the manufacturer.

on_power (number) active power in watts when on.

standby_power (number) active power in watts when in standby.

energy_per_year (number) kWh per year

energy_per_cycle (number) kWh per cycle

components (list of dicts): Components within this appliance. Each dict is an Appliance dict.

model (string)

manufacturer (string)

brand (string)

original_name (string)

model_url (string) URL for this model of appliance

manufacturer_url (string) URL for the manufacturer

dates_active (list of *TimeFrame* objects, see below) Can be used to specify a change in appliance over time (for example if one appliance is replaced with another).

year_of_purchase (int) Four-digit year.

year_of_manufacture (int) Four-digit year.

subtype (string)

part_number (string)

gtin (int) http://en.wikipedia.org/wiki/Global_Trade_Item_Number

version (string)

portable (boolean)

Additional properties are specified for some Appliance Types. Please look up objects in `nilm_metadata/central_metadata/appliances/*.yaml` for details.

When an Appliance object is used as a component for an ApplianceType, then the Appliance object may have a `distributions` dict (see `ApplianceType:distributions` in *Central appliance metadata*) specified and may also use a property `do_not_merge_categories: true` which prevents the system from merging categories from the component into the container appliance.

TimeFrame

Represent an arbitrary time frame. If either start or end is absent then assume it equals the start or the end of the dataset, respectively. Please use ISO 8601 format for dates or date times (e.g. 2014-03-17 or 2014-03-17T21:00:52+00:00)

start (string)

end (string)

CHAPTER 3

Central appliance metadata

Manual

Please see the NILM Metadata README section on 'Central metadata' for a quick introduction.

Inheritance

- protypical inheritance; like JavaScript
- dicts are updated; lists are extended; other properties are overwritten
- arbitrary inheritance depth

Components

- recursive
- categories of container appliance is updated with categories from each component (unless `do_not_merge_categories: true` is set in the component)

Subtypes versus a new child object

Appliance specification objects can take a 'subtype' property. Why not use inheritance for all subtypes? The rule of thumb is that if a subtype is functionally different to its parent then it should be specified as a separate o child bject (for example, a gas hob and an electric hob clearly have radically different electricity usage profiles) but if the differences are minor (e.g. a digital radio versus an analogue radio) then the appliances should be specified as subtypes of the same object.

Naming conventions

- properties are lowercase with underscores, e.g. *subtype*
- object names (not specific makes and models) are lowercase with spaces, unless they are acronyms in which case they are uppercase (e.g. 'LED')
- category names are lowercase with spaces

Example

To demonstrate the inheritance system, let's look at specifying a boiler.

First, NILM Metadata specifies a 'heating appliance' object, which is can be considered the 'base class':

```
heating appliance:
  parent: appliance
  categories:
    traditional: heating
    size: large
```

Next, we specify a 'boiler' object, which inherits from 'heating appliance':

```
#------------- BOILERS ------------------------

boiler: # all boilers except for electric boilers

  parent: heating appliance

  synonyms: [furnace]

  # Categories of the child object are appended
  # to existing categories in the parent.
  categories:
    google_shopping:
      - climate control
      - furnaces and boilers

  # Here we specify that boilers have a component
  # which is itself an object whose parent
  # is `water pump`.
  components:
    - type: water pump

  # Boilers have a property which most other appliances
  # do not have: a fuel source.  We specify additional
  # properties using the JSON Schema syntax.
  additional_properties:
    fuel:
      enum: [natural gas, coal, wood, oil, LPG]

  subtypes:
    - combi
    - regular

  # We can specify the different mechanisms that
  # control the boiler.  This is useful, for example,
  # if we want to find all appliances which
```

```
  # must be manually controlled (e.g. toasters)
  control: [manual, timer, thermostat]

  # We can also declare prior knowledge about boilers.
  # For example, we know that boilers tend to be in
  # bathrooms, utility rooms or kitchens
  distributions:
    room:
      distribution_of_data:
        categories: [bathroom, utility, kitchen]
        values: [0.3, 0.2, 0.2]
        # If the values do not add to 1 then the assumption
        # is that the remaining probability mass is distributed equally to
        # all other rooms.
      source: subjective # These values are basically guesses!
```

Finally, in the metadata for the dataset itself, we can do:

```
type: boiler
manufacturer: Worcester
model: Greenstar 30CDi Conventional natural gas
room: bathroom
year_of_purchase: 2011
fuel: natural gas
subtype: regular
part_number: 41-311-71
efficiency_rating:
  certification_name: SEDBUK
  rating: A
nominal_consumption:
  on_power: 70
```

Schema details

Below is a UML Class Diagram showing all the classes and the relationships between classes:

(Please see the *NILM Metadata Tutorial* for more background about the NILM Metadata schema)

Below we describe all the classes and their attributes and possible values.

ApplianceType

Has many of the attributes that *Appliance* has, with the addition of:

- on_power_threshold
- min_off_duration
- min_on_duration
- control
- components

  parent (string) Name of the parent ApplianceType object from which this object inherits.

categories (dict)

traditional (enum) one of {wet, cold, consumer electronics, ICT, cooking, heating}

size (enum) one of {small, large}

electrical (list of strings) Any combination of:

- lighting, incandescent, fluorescent, compact, linear, LED
- resistive
- power electronics
- SMPS, no PFC, passive PFC, active PFC
- single-phase induction motor, capacitor start-run, constant torque

misc (enum) one of {misc, sockets}

google_shopping (list of strings) anything from the Google Shopping schema. e.g.: climate control', 'furnaces and boilers', 'renewable energy', 'solar energy', 'solar panels', 'computers', 'electronics', 'laptops', 'printers and copiers', 'print, copy, scan and fax', 'printers', 'laundry appliances', 'kitchen and dining', 'kitchen appliances', 'breadmakers'

subtypes (list of strings) A list of all the valid subtypes.

additional_properties (dict) Used for specifying additional properties which can be specified for Appliances of this ApplianceType. Each key is a property. Each value is a JSON Schema definition of the property.

do_not_inherit (list of strings) properties which should not be inherited from the parent.

synonyms (list of strings)

usual_components (list of strings) Just a list of hints for human readers.

n_ancestors (int) Filled in by `_concatenate_complete_object`.

distributions (dict) Distribution of random variables.

on_power (list of *Prior* objects) bin_edges in units of watts

on_duration (list of *Prior* objects) bin_edges in units of seconds

off_duration (list of *Prior* objects) bin_edges in units of seconds

usage_hour_per_day (list of *Prior* objects) bin_edges = [0,1,2,...,24]

usage_day_per_week (list of *Prior* objects) categories = ['mon', 'tue', ..., 'sun']

usage_month_per_year (list of *Prior* objects) bin_edges are in units of days (we need bin edges because months are not equal lengths). The first bin represents January.

rooms (list of *Prior* objects) Categorical distribution over the rooms where this appliance is likely to be used. e.g. for a fridge this might be 'kitchen:0.9, garage:0.1'. Please use the standard room names defined in room.json (category names in distributions are not automatically validated).

subtypes (list of *Prior* objects) Categorical distribution over the subtypes.

appliance_correlations (list of *Prior* objects) list of other appliances. Probability of this appliance being on given that the other appliance is on. e.g. 'tv:0.1, amp:0.4, ...' means that there is a 10% probability of this appliance being on if the TV is on. Each category name can either be just an appliance name (e.g. 'fridge') or <appliance name>,<appliance instance> e.g. 'fridge,1'

ownership_per_country (list of *Prior* objects) Probability of this appliance being owned by a household in each country (i.e. a categorical distribution where categories are standard two-letter country code defined by ISO 3166-1 alpha-2. e.g. 'GB' or 'US'. http://en.wikipedia.org/wiki/ISO_3166-1_alpha-2). If the probability refers to the entire globe then use 'GLOBAL' as the country code.

ownership_per_continent (list of *Prior* objects) Probability of this appliance being owned by a household in each country (i.e. a categorical distribution where categories are standard two-letter continent code defined at http://en.wikipedia.org/wiki/List_of_sovereign_states_and_dependent_territories_by_continent_%28data_file%29

Country

One large dict specifying country-specific information. Specified in `nilm_metadata/central_metadata/country.yaml`

Each key is a 'country' (string). Please use a standard two-letter country code defined by ISO 3166-1 alpha-2. e.g. 'GB' or 'US'.

Each value is a dict with the following attributes:

mains_voltage (dict):

nominal (number) (required) volts

upper_limit (number) volts

lower_limit (number) volts

related_documents (list of strings)

Prior

Represent prior knowledge. For continuous variables, specify either the distribution of data (i.e. the data represented in a histogram), or a density estimate (a model fitted to the data), or both. For categorical variables, specify the categorical distribution.

distribution_of_data

(dict) Distribution of the data expressed as normalised frequencies per discrete bin (for continuous variables) or per category (for categorical variables). 'categories' can be used instead of 'bin_edges' for continuous variables where it makes sense; e.g. where each bin represents a day of the week

bin_edges (list of numbers of list of strings) (required) |bin_edges| == |values| + 1

categories (list of strings) (required) |bin_edges| == |values|

values (list of numbers) (required) The normalised frequencies. For continuous variables, in integral over the range must be 1. For categorical variables, the sum of frequences can be <= 1. If < 1 then the system will assume that the remaining mass is distributed equaly across all other categories. For example, for the probability of a fridge being in a specific room, it is sufficient to just state that the probability is 0.9 for a fridge to be in the kitchen.

model

(dict) A fitted model to describe the probability density function (for continuous variables) or the probability mass function (for discrete variables). Use additional properties for the relevant parameters, written as Greek letters spelt out in lowercase English e.g. 'mu' and 'lambda' except for summary stats where we use some combination of 'min', 'max', 'mean', 'mode'.

distribution_name (enum) one of {'normal', 'inverse gaussian', 'summary stats'}

sum_of_squared_error (number)

n_datapoints (int)

date_prepared (string) ISO 8601 date format

source (enum) one of {'subjective', 'empirical from data', 'empirical from publication'}. What is the source of this prior? If from publication then use `related_documents` to provide references. If from data then provide details using the `software` and `training_data` properties.

related_documents (list of strings) If 'source==empirical from publication' then enter the reference(s) here.

software (string) the software used to generate the prior from data.

specific_to (dict):

country (string) standard two-letter country code defined by ISO 3166-1 alpha-2 e.g. 'GB' or 'US'.

continent (string) standard two-letter continent code defined on WikiPedia

distance (int) this is filled in by the `concatenate_complete_object` function and reports the distance (in numbers of generations) between this prior and the most-derived object. In other words, the larger this number, the less specific to the object this prior is. If this is not set the the prior applies to the current object.

from_appliance_type (string) this is filled in by the `concatenate_complete_object` function and reports the appliance type name from the ancestor hierarchy from which this distribution came from.

description (string)

training_data (array of dicts). Each element is a dict with these properties:

dataset (string) Short name of dataset

buildings (list of dicts):

building_id (int)

dates (list of interval-schema objects)

country (string) standard two-letter country code defined by ISO 3166-1 alpha-2 e.g. 'GB' or 'US'.

DisaggregationModel

This is not especially well defined yet. Just an initial sketch. The basic idea is that we would be able to specify models for each appliance type.

appliance_type (string) Reference to the specific *ApplianceType* that we are modelling.

model_type (enum) one of {'HMM', 'FHMM', 'gubernatorial optimisation'}

parameters (dict) Parameters specific to each model type.

DisaggregationModel re-uses several properties from *Prior* :

- training_data
- specific_to
- software
- related_documents
- date_prepared
- description

CHAPTER 4

Indices and tables

- genindex
- search