Open Energy Efficiency Meter Documentation

Release v0.4.12-alpha

Phil Ngo

October 26, 2016

Contents

Warning: The *eemeter* package is under rapid development; we are working quickly toward a stable release. In the mean time, please proceed to use the package, but as you do so, recognize that the API is in flux and the docs might not be up-to-date. Feel free to contribute changes or open issues on github to report bugs, request features, or make suggestions.

This package holds the core methods used by the of the **Open Energy Efficiency** energy efficiency metering stack. Specifically, the `eemeter` package abstracts the process of building and evaluating models of energy consumption or generation and of using those to evaluate the effect of energy efficiency interventions at a particular site associated with a particular project.

The `eemeter` package is only one part of the larger Open Energy Efficiency technology stack. Briefly, the architecture of the stack is as follows:

- `eemeter`: Given project and energy data, the `eemeter` package is responsible for creating models of energy usage under different project conditions, and for using those models to evaluate energy efficiency projects.
- `datastore`: The `datastore` application is responsible for validating and storing project data and associated energy data, for using the `eemeter` to evaluate the effectiveness of these projects using the data it stores, and for storing and serving those results. It exposes as REST API for handling these functions.
- `etl`: The `etl` package provides tooling which helps to extract data from various formats, transform that data into the format accepted by datastore, and load that transformed data into the appropriate `datastore` instance. ETL stands for Extract, Transform, Load.

CHAPTER 1

Usage

1.1 Guides

1.1.1 Introduction

The OpenEEmeter is an open source software package that uses metered energy data to manage aggregate demand capacity across a portfolio of retail customer accounts. The software package consists of three main parts:

1. an Extract-Transform-Load (ETL) toolkit for processing project, energy, and building data (github);
2. a core calculation library (this package) that implements standardized methods (github); and
3. a datastore application for storing post-ETL inputs and computed outputs (github).

More information about this architecture can be found in *Architecture Overview*.

Core use cases

The OpenEEmeter has been designed specifically to provide weather-normalized energy savings measurements for a portfolio of projects using monthly billing data or interval smart meter data. The main outputs for this core use case are project and portfolio-level are:

- Gross Energy Savings
- Annualized Energy Savings
- Realization Rate (when savings predictions are available)

More information about these methods can be found in *Methods Overview*.

Other potential use cases

The OpenEEmeter can also be configured to manage energy resources across a portfolio of buildings, including potentially:

- Analytics of raw energy data
- Portfolio management
- Demand side resource management

Data requirements

The EEmeter requires a combination of trace data, project data, and weather data to calculate weather-normalized savings. At its most rudimentary, the EEmeter requires a *trace* of consumption data along with project data indicating the completion date and location of the project.

The completion of a *project* demarcates the shift between a *baseline modeling period* and a *reporting modeling period*. For more information on this, see *Methods Overview*.

The EEmeter is configured to manage *project* and *trace* data. Trace data can be electricity, natural gas, or solar photovoltaic data of any frequency - from monthly billing data to high-frequency sensor data (see *1) Meters and Smart Meters - where does energy data come from?*).

Where project and trace data originate from different database sources, a common key must be available to link projects with their respective traces.

Project data

Project data is typically a set of attributes that can be used for advanced savings analytics, but at minimum must contain a date to demarcate start and end of *intervention* periods.

Each project must have, at minimum:

- a unique project id
- start and end dates of known interventions
- a ZIP code (for gathering associated weather data)
- a set of associated traces

Other data can also be associated with projects, including (but not limited to):

- savings predictions
- square footage
- cost

Trace data

Each trace must have, at minimum,

- a link to a project id
- a unique id of its own
- an *interpretation*
- a set of records

Each record within a trace must have:

- a time period (start and end dates)
- a value and assiciated units of
- a boolean “estimated” flag

The EEmeter will reject traces not meeting built-in data sufficiency requirements.

Loading data

The *eemeter* python package is a calculation engine which is not desigend for data storage. Instead, project and trace data are stored in the *datastore* alongside outputs from the *eemeter*.

To load data into the datastore, EEMeter comes bundled with an *ETL Toolkit*. If you are deploying the open source software, you will need to write or customize a parser to load your data into the ETL pipeline. We rely on a python module called luigi to manage the bulk importation of data.

More on this *architecture*.

External analysis

You may decide that you want to use EEmeter results to analyze project data that does not get parsed and uploaded into the *datastore*. We have made it easy to export your EEmeter results through an API or through a web interface. Other options include a direct database connection to a BI tool like Tableau or Salesforce.

1.1.2 Background

1) Meters and Smart Meters - where does energy data come from?

Energy data is generated by hardware devices that measure electricity and natural gas flow. A device like this is generally referred to as a "meter" (though this is distinct from the software-based "EEmeter" - see *Methods Overview*). The most common and ubiquitous measuring device is a utility-owned meter used for determining billing. Some utilities have upgraded their meters to provide hourly or 15-minute interval measurements. These so-called "smart meters" use Advanced Metering Infrastructure (AMI) to transmit data back to utilities for processing in near-real time. Other devices that generate energy data include sub-meters, external sensors, and embedded sensors.

Note: The "smart" in smart meter can be a bit of a misnomer. Despite higher measurement frequency and wireless data transmission, these smart meters collect essentially the same data that electricity meters did in the 1950s. Each meter datapoint consists of a timestamp and an incremental value of consumption. We call this string of data characterized by paired sets of timestamps and meter readings a *trace*. Traces form the basis of the energy modeling in the EEmeter.

Just like the odometer in your car doesn't tell you how fast you are traveling, the meter on your house doesn't tell you how much energy you have consumed. Consumption must be calculated. In the past, energy companies simply determined your rate of consumption by taking monthly meter readings and calculating the difference. With smart meters, these datapoints can be captured more frequently and with greater precision, allowing for more sophisticated forms of billing.

2) Measuring Energy Savings and the Transition to Demand Side Management

The OpenEEmeter replaces traditional approaches to program-related energy measurement. Utilizing newly available smart meter data, the OpenEEmeter solves the problem of measuring energy savings and opens new doors for managing demand side programs.

Historically, energy *savings* have been measured in one of three ways. The first (and least costly) approach is to take laboratory measurements of different energy-consuming devices (e.g., light bulbs) and calculate the difference in consumption from one to the next, then estimate the savings over a given period of time, taking into consideration typical usage patterns. This first approach is limited by the accuracy and availability of physical models.

The second (and most costly) approach samples consumption data prior to and following an intervention of some sort (e.g., an energy efficiency retrofit), and estimates savings after controlling for building-specific factors like occupancy,

temperature, energy intensity, etc. This second approach is limited by low availability of data describing these building-specific factors (thus making it very costly).

A third (post-hoc) approach has recently emerged that takes a population-level sample of similar buildings and compares with a treatment group of buildings that have received an energy efficiency upgrade (or other intervention). This approach assumes that all buildings will be affected equally by exogenous factors, leaving only endogenous factors (i.e., the efficiency upgrade) to account for the energy consumption difference.

In the analog era of traditional meters and monthly bills, efforts to improve energy efficiency emphasized fairly static and permanent changes in consumption. A whole-home retrofit, for example, would reduce energy demand without requiring any additional behavioral or lifestyle changes. A one-time intervention would provide years of benefit, and our metering technology at the time provided a way to measure the performance of these measures.

With the introduction of smart meters, utilities have transitioned from simple efficiency programs to a suite of programs under the umbrella of demand side management (DSM). These new measures fall into three broad categories including time of day, demand, and net metering. The OpenEEmeter expands the programmatic interface of energy efficiency to engage with emergent technologies and market based demand side engagement programs.

3) How the OpenEEmeter is valuable: Baselining, Normalization, and Modeling Energy Use

Smart meter data allows for more complexity in statistical models. Rather than relying on simple regression experiments to normalize energy consumption, analysts can parse the impact of exogenous and endogenous factors independently and iteratively. The notion of baseload energy use can even be disaggregated into multiple demand states. For example, a home will use very little energy when empty, a bit more when occupied, and a large amount when appliances and heating or cooling systems are operating. These demand states can be measured against various sorts of interventions, thus enabling both traditional energy efficiency savings measurements, but also leveraging modern load balancing tools.

The OpenEEmeter calculates energy savings in real time by selecting a sample of consumption data prior to an intervention, weather-normalizing it to establish a baseline, and calculating the difference between projected energy usage and actual energy usage following the intervention. This method maintains the cost-effectiveness of the naive predicted savings approach, the real-world integrity of the building efficiency approach, without sacrificing on time as with the post hoc control group approach.

1.1.3 Architecture Overview

The complete eemeter architecture consists primarily of a datastore application (see *datastore*), which houses energy and project data, and a data pipeline toolkit (see *ETL Toolkit*) that helps get data into the datastore.

These two work in tandem to take raw energy data in whatever form it exists and compute energy savings using the eemeter package. The methods and models used within the datastore for computing energy savings are kept in a library package called eemeter, which can also be used independent of the datastore application (see *eemeter*).

Each of these components are open sourced under an MIT License and can be found on github:

- eemeter
- datastore
- etl

The core calculation engine is separated from the datastore in order to allow easier development of and evaluation of its methods, but this architecture also makes it possible to embed the calculation engine or any of its useful modules (such as the *weather module*) in other applications.

The data structures in each - the eemeter and the datastore - mirror each other. This simplifies data transfer and eases interpretation of results.

1.1.4 Methods Overview

The EEmeter provides multiple methods for calculating *energy savings*. All of these methods compare *energy demand* from a modeled counterfactual pre-*intervention baseline* to post-intervention energy demand. Some of these methods, including the most conventional, *weather normalize* energy demand.

These basic methods [1] rely on a *modeled* relationship between weather patterns and energy demand. The particular models used by the EEmeter are described more precisely in *Modeling Overview*.

Modeling periods

For any savings calculation, the period of time prior to the start of any *interventions* taking place as part of a project we term the *baseline period*. This period is used to establish models of the relationship between *energy demand* and a set of factors that represent or contribute to *end use demand* (such as weather, time of day, or day of week) for a particular building _prior_ to an intervention. The *baseline* becomes a reference point from which to make comparisions to post-intervention energy performance. The baseline period is one of two types of *modeling period* frequently occurring in the EEmeter.

The second half of the savings calculation concerns what happens after an intervention. Any post-intervention period for which energy savings is calculated is called a *reporting period* because it is the period of time over which energy savings is reported. A project generally has only one *baseline period*, but it might have multiple *reporting periods*. These are the second type of *modeling period* to frequent occur in the EEmeter.

The extent of these periods will, in most cases, be determined by the start and end dates of the interventions in a project. However, in some cases, the intervention dates are not known, or are ongoing, and must be *modeled* because they cannot be stated explicitly. We refer to models which account for the latter scenario as *structural change models*; these are covered in greater detail in *Modeling Overview*.

EEmeter structures which capture this logic can be found in the API documentation for *eemeter.structures*.

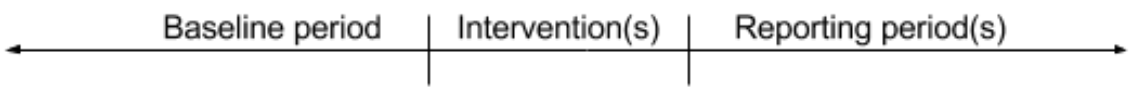

Fig. 1.1: Pre-intervention baseline period and post-intervention reporting periods on a project timeline.

Trace modeling

The relationship between energy demand and various external factors can differ drastically from building to building, and (usually!) changes after an intervention. Modeling these relationships properly with statistical confidence is a core strength of the EEmeter.

As noted in the *background*, we term a set of energy data points a *trace*, and a building or project might be associated with any number of traces. In order to calculate savings models, each of these traces must be modeled.

Before modeling, traces are segmented into components which overlap each baseline and reporting period of interest, then are modeled separately. [2] This creates up to $n * m$ models for a project with n traces and m modeling periods.

Each of these models attempts to establish the relationship between *energy demand* and external factors as it performed during the particular modeling period of interest. However, since the extent to which a model successfully describes these relationships varies significantly, these must be considered only in conjunction with model error and goodness of

[1] Additional information on *why* this method is used in preference to other methods is described in the *Introduction*.
[2] This is not quite true for *structural change models*. This is covered in more detail in *Modeling Overview*.

fit metrics *Modeling Overview*. Any estimate of energy demand given by any model fitted by the EEmeter is associated with variance and confidence bounds.

In practice the number of models fitted for any particular project might be fewer than $n * m$ due to missing or insufficient data (see *Data sufficiency*). The EEmeter takes these failures into account and considers them when building summaries of savings.

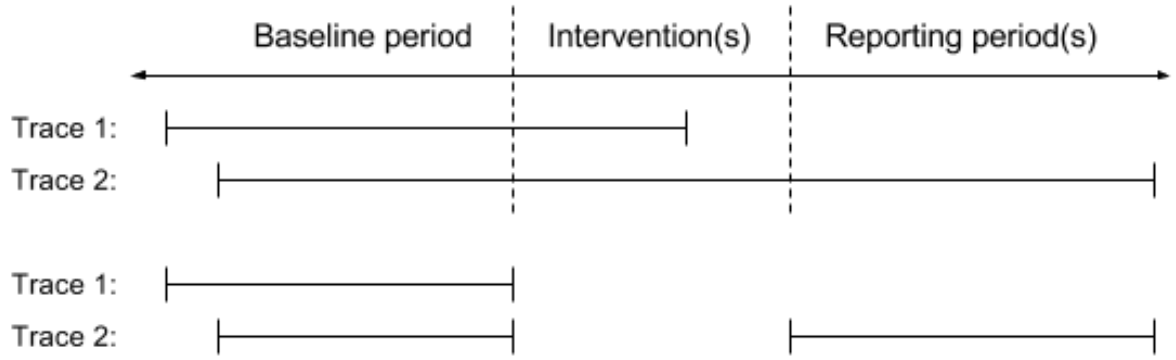

Fig. 1.2: An example of trace segmenting with two traces, one baseline period and one reporting period. **Trace 1** is segmented into just one component - the baseline component - because data for the reporting period is missing. **Trace 2** is segmented into one baseline component and one reporting component. The segments of **Trace 1** and **Trace 2** have different lengths, but models of their energy demand behavior can still be built.

Weather normalization

Once we have created a model, we can apply that model determine an estimate of of energy demand during arbitrary weather scenarios. The two most common weather scenarios for which the EEmeter will estimate demand are the "*normal*" weather year and the observed reporting period weather year. This is generally necessary because the data observed in the baseline and reporting periods occurred during different time periods with different weather – and valid comparisons between them must account for this. Estimating energy performance during the "normal" weather attempts to reduce bias in the savings estimate by accounting for the peculiarity (as compared to other years or seasons) of the relevant observed weather.

In an attempt to reduce the number of arbitrary factors influencing results, we only ever compare model estimates or data over that has occurred over the same weather scenario and time period. This helps (in the aggregate) to ensure equivalency of *end use demand* pre- and post- intervention.

Savings

If the data and models show that *energy demand* is reduced relative to equivalent *end use demand* following an intervention, we say that there have been energy savings, or equivalently, that energy performance has increased.

Energy savings is necessarily a difference; however, this difference must be taken carefully, given missing data and model error, and is only taken *after* the necessary *aggregation* steps.

The equation for savings is always:

$$S_{\text{total}} = E_{\text{b}} - E_{\text{r}}$$

or

$$S_{\text{percent}} = \frac{E_{\text{b}} - E_{\text{r}}}{E_{\text{b}}}$$

where

- S_{total} is aggregate total savings
- S_{percent} is aggregate percent savings
- E_{b} is aggregate energy demand as under baseline period conditions
- E_{r} is aggregate energy demand as under reporting period conditions

Depending on the type of energy savings desired, the values E_{b} and E_{r} may be calculated differently. The following types of savings are supported:

- *Annualized weather normal*
- *Gross predicted*
- *Gross observed*

Annualized weather normal

The *annualized weather normal* estimates savings as it may have occurred during a *"normal" weather* year. It does this by building models of both the baseline and reporting energy demand and using each to weather-normalize the energy values.

$$E_{\text{b}} = \text{M}_{\text{b}}\left(\text{X}_{\text{normal}}\right)$$

$$E_{\text{r}} = \text{M}_{\text{r}}\left(\text{X}_{\text{normal}}\right)$$

where

- M_{b} is the model of energy demand as built using trace data segmented from the baseline period.
- M_{r} is the model of energy demand as built using trace data segmented from the reporting period.
- X_{normal} are temperature and other covariate values for the weather normal year.

Gross predicted

The *gross predicted* method estimates savings that have occurred from the completion of the project interventions up to the date of the meter run.

$$E_{\text{b}} = \text{M}_{\text{b}}\left(\text{X}_{\text{r}}\right)$$

$$E_{\text{r}} = \text{M}_{\text{r}}\left(\text{X}_{\text{r}}\right)$$

where

- M_{b} is the model of energy demand as built using trace data segmented from the baseline period.
- M_{r} is the model of energy demand as built using trace data segmented from the reporting period.
- X_{r} are temperature and other covariate values for reporting period.

Gross observed

The *gross observed* method estimates savings that have occurred from the completion of the project interventions up to the date of the meter run.

$$E_{\text{b}} = \text{M}_{\text{b}}\left(\text{X}_{\text{r}}\right)$$

$$E_{\text{r}} = \text{A}_{\text{r}}$$

where

- M_b is the model of energy demand as built using trace data segmented from the baseline period.
- A_r are the actual observed energy demand values from the trace data segmented from the baseline period. If the actual data has missing values, these are interpolated using gross predicted values (i.e., $M_r(X_r)$).
- X_r are temperature and other covariate values for reporting period.

Aggregation rules

Because even an individual project may have multiple traces describing its energy demand, we must be able to aggregate trace-level results before we can obtain project-level or portfolio-level savings. Ideally, this aggregation is a simple sum of trace-level values. However, trace-level results are often littered with messy results which must be accounted for; some may be missing data, have bad model fits, or have entirely failed model builds. The EEmeter must successfully handle each of these cases, or risk invalidating results for entire portfolios.

The aggregation steps are as follows:

1. Select scope (project, portfolio) and gather all trace data available in that scope
2. Select baseline and reporting period. For portfolio level aggregations in which baseline and reporting periods may not align, select reporting period type and use the default baseline period for each project.
3. Group traces by *interpretation*
4. Compute E_b and E_r:
   (a) Compute (or retrieve) $E_{t,b}$ and $E_{t,r}$ for each trace t.
   (b) Determine, for each $E_{t,b}$ and $E_{t,r}$ whether or not it meets *criteria* for inclusion in aggregation.
   (c) Discard *both* $E_{t,b}$ and $E_{t,r}$ for any trace for which either $E_{t,b}$ or $E_{t,r}$ has been discarded.
   (d) Compute $E_b = \sum_t E_{t,b}$ and $E_r = \sum_t E_{t,r}$ for remaining traces. Errors are propgated according to the principles in *Error propogation*.
5. Compute savings from E_b and E_r as usual.

Inclusion criteria

For inclusion in aggregates, $E_{t,b}$ and $E_{t,r}$ must meet the following criteria

1. If `ELECTRICITY_ON_SITE_GENERATION_UNCONSUMED`, which represents solar generation, is available, and if solar panels were installed as one of the project interventions, blank $E_{t,b}$ should be replaced with 0.
2. Model has been successfully built.

Error propogation

Errors are propgated as if they followed χ^2 distributions.

Weather data matching

Since weather and temperature data is so central to the activity of the EEmeter, the particulars of how weather data is obtained for a project is often of interest. Weather data sources are determined automatically within the EEmeter

using an internal mapping [3] bewteen ZIP codes [4] and weather stations. The source of the weather normal data may differ from the source of the observed weather data.

There is a jupyter notebook outlining the process of constructing the weather data available here.

1.1.5 Modeling Overview

Basic modeling principles

Model error

Data sufficiency

Types of models

Weekday and Seasonal effects regression model

Hidden markov model

1.1.6 Glossary

- **annualized weather normal**: an estimate of annual energy demand under a *weather normal*.
- **baseline**: a pre-intervention reference point or starting point from which to compare post-intervention *energy demand*.
- **baseline period**: a time period before a *retrofit* of interest for which to model, observe, or estimate *energy performance*. Generally used in reference to a *reporting period* or set of reporting periods.
- **building performance**: see *energy performance*.
- **demand capacity**: the extent to which *energy-performance* increases from a baseline for a *reporting period* following an *intervention*.
- **demand response project**: a set of *interventions* designed to shift the time of day or day of week of *energy-demand*, generally toward off-peak hours.
- **end use**: an energy-consuming service such as lighting, space cooling, space heating, refrigeration, or water heating, particularly as provided by a building or set of buildings.
- **end use demand**: the extent to which an *end use* is needed. May vary by season, occupancy, time of day, day of week, or purpose of building.
- **energy demand**: the amount of energy needed to satisfy *end use demand*.
- **energy efficiency project**: a set of *interventions* designed to reduce overall *energy demand* relative to equivalent *end use demand*.
- **energy model**: a mathematical description of *energy demand*, particularly in response to *end use demand* scenarios.
- **energy savings**: an increase in *energy performance* indicating lower *energy demand* for *equivalent end use demand*.
- **energy performance**: the extent to which *end use demand* causes *energy demand*. Higher performance indicates lower energy demand for *equivalent* end use demand. Sometimes referred to as *building performance*.

[3] Available on github.
[4] The ZIP codes used in this mapping aren't strictly ZIP codes, they're actually *ZCTAs*.

- **energy trace**: see *trace*
- **gross observed**: an estimate of *energy demand* over the *reporting period* as given by *baseline* models and observed values from the reporting period.
- **gross predicted**: an estimate of *energy demand* as given by the *baseline* and reporting models evaluated over the *reporting period*.
- **intervention**: a set of upgrades or performance improvements on physical infrastructure of an existing building (see *retrofit*), or of behavior of individuals living in an existing building.
- **modeling period**: a period of time over which an *energy model* is to be created for a particular *trace*. This is a generalization of *baseline* and *reporting* periods. Modeling periods generally fall into one of those two categories.
- **projected baseline energy demand**: a counterfactual estimate of *energy demand* as it might have been under a particular *end use demand* scenario had an intervention not occurred.
- **project**: an *intervention* or *retrofit* for which there is an expected change in *energy demand*.
- **reporting period**: a time period after a *retrofit* of interest over which to model, observe, or estimate *energy performance*. Generally used in reference to a *baseline period*.
- **retrofit**: a set of *interventions* taking place at a particular building or site which modify pre-existing structures, installations or appliances.
- **structural change model**: a model which takes tries to determine the most probably extents of *baseline* and *reporting* periods for a *project* given its *trace* data.
- **trace**: a single time series of measured values associated with units at a particular (not necessarily fixed) frequency.
- **trace interpretation**: the meaning of the trace data. Possible interpretations are outlined in *eemeter.structures*
- **Typical Meteorological Year 3 (TMY3)**: A set of publicly available *weather normals* designed by the National Renewable Energy Laboratory (NREL). Used by EEMeter for *weather normalization*.
- **weather normalization**: a technique to account for differences in *end use demand* due to variations in weather patterns which uses a model of weather-dependent *energy demand* to determine a counterfactual energy demand under a weather conditions described by a *weather normal*.
- **weather normal**: a set of (not necessarily observed) weather data designed to reflect a "typical" weather scenario. Often covers a time period of 1 year. Used in *weather normalization*. See *TMY3*.
- **ZIP Code Tabulation Area (ZCTA)**: a set of geographical areas based on US Postal Service (USPS) ZIP codes, necessitated by the fact that ZIP codes do not map easily onto geographies. Built and maintained by the US Census Bureau. Contains only about three quarters of valid ZIP codes. ZIP code and ZCTA do not always match. More information.

1.1.7 Why open source?

All of our savings algorithms are free and open source. We don't believe that standard weights and measures should be the private property of any particular entity. It's much better for everyone, from contractors to program administrators, if the measurement tools are equally available to everyone.

1.2 eemeter

1.2.1 Installation

To get started with the eemeter, use pip:

```
$ pip install eemeter
```

Make sure you have the latest version:

```
>>> import eemeter; eemeter.get_version()
'0.4.12'
```

1.2.2 Topics

Basic Usage

This tutorial will cover the three basic steps for using the `eemeter` package:

1. *data prepartation*
2. *running meters*
3. *inspecting results*

This tutorial is also available as a jupyter `notebook`:

Before getting started, download some sample energy data and project data:

- energy data `CSV`
- project data `CSV`

This sample data was created using `this` jupyter notebook which you should reference if you have questions about the data.

Note: Most users of the EEmeter stack do not directly use the `eemeter` package for loading their data. Instead, they use the *datastore*, which uses the eemeter internally. To learn to use the datastore, head over to *this tutorial*.

Data preparation

The basic container for project data is the `eemeter.structures.Project` object. This object contains all of the data necessary for running a meter.

There are three items it requires:

1. An `EnergyTraceSet`, which is a collection of `EnergyTrace` s
2. An `list` of `Intervention` s
3. An `eemeter.structures.ZIPCodeSite`

Let's start by creating an `EnergyTrace`. Internally, `EnergyTrace` objects use numpy and pandas, which are nearly ubiquitous python packages for efficient numerical computation and data analysis, respectively.

Since this data is not in a format eemeter recognizes, we need to load it. Let's load this data using a parser we create to turn this data into a format that eemeter recognizes.

We will load data from formatted records using an *eemeter.io.serializer.ArbitraryStartSerializer*.

```
# library imports
from eemeter.structures import (
    EnergyTrace,
    EnergyTraceSet,
    Intervention,
    ZIPCodeSite,
    Project
)
from eemeter.io.serializers import ArbitraryStartSerializer
from eemeter.ee.meter import EnergyEfficiencyMeter
import pandas as pd
import pytz
```

First, we import the energy data from the sample CSV and transform it into records.

```
energy_data = pd.read_csv(
    'sample-energy-data_project-ABC_zipcode-50321.csv',
    parse_dates=['date'], dtype={'zipcode': str})

records = [{
    "start": pytz.UTC.localize(row.date.to_datetime()),
    "value": row.value,
    "estimated": row.estimated,
} for _, row in energy_data.iterrows()]
```

The records we just created look like this:

```
>>> records
[
    {
        'estimated': False,
        'start': datetime.datetime(2011, 1, 1, 0, 0, tzinfo=<UTC>),
        'value': 57.8
    },
    {
        'estimated': False,
        'start': datetime.datetime(2011, 1, 2, 0, 0, tzinfo=<UTC>),
        'value': 64.8
    },
    {
        'estimated': False,
        'start': datetime.datetime(2011, 1, 3, 0, 0, tzinfo=<UTC>),
        'value': 49.5
    },
    ...
]
```

Next, we load our records into an `EnergyTrace`. We give it units `"kWh"` and interpretation `"ELECTRICITY_CONSUMPTION_SUPPLIED"`, which means that this is electricity consumed by the building and supplied by a utility (rather than by solar panels or other on-site generation). We also pass in an instance of the record serializer `ArbitraryStartSerializer` to show it how to interpret the records.

```
energy_trace = EnergyTrace(
    records=records,
    unit="KWH",
    interpretation="ELECTRICITY_CONSUMPTION_SUPPLIED",
    serializer=ArbitraryStartSerializer())
```

The energy trace data looks like this:

```
>>> energy_trace.data[:3]
                           value estimated
2011-01-01 00:00:00+00:00   57.8     False
2011-01-02 00:00:00+00:00   64.8     False
2011-01-03 00:00:00+00:00   49.5     False
```

Though we only have one trace here, we will often have more than one trace. Because of that, projects expect an `EnergyTraceSet`, which is a labeled set of `EnergyTrace` objects. We give it the `trace_id` supplied in the CSV.

```
energy_trace_set = EnergyTraceSet([energy_trace], labels=["DEF"])
```

Now we load the rest of the project data from the sample project data CSV. This CSV includes the project_id (Which we don't use in this tutorial), the ZIP code of the building, and the dates retrofit work for this project started and completed.

```
project_data = pd.read_csv(
    'sample-project-data.csv',
    parse_dates=['retrofit_start_date', 'retrofit_end_date']).iloc[0]
```

We create an `Intervention` from the retrofit start and end dates and wrap it in a list:

```
retrofit_start_date = pytz.UTC.localize(project_data.retrofit_start_date)
retrofit_end_date = pytz.UTC.localize(project_data.retrofit_end_date)

interventions = [Intervention(retrofit_start_date, retrofit_end_date)]
```

Then we create a `ZIPCodeSite` for the project by passing in the zipcode:

```
site = ZIPCodeSite(project_data.zipcode)
```

Now we can create a project using the data we've loaded:

```
project = Project(energy_trace_set=energy_trace_set,
                  interventions=interventions,
                  site=site)
```

This completes the `eemeter` data loading process.

Running meters

To run the EEmeter on the project, instantiate an `EnergyEfficiencyMeter` and run the `.evaluate(project)` method, passing in the project we just created:

```
meter = EnergyEfficiencyMeter()
results = meter.evaluate(project)
```

That's it! Now we can inspect and use our results.

Inspecting Results

Let's quickly look through the results object so that we can understand what they mean. The results are embedded in a nested python `dict`:

```
>>> results
{
    'weather_normal_source': TMY3WeatherSource("725460"),
    'weather_source': ISDWeatherSource("725460"),
    'modeling_period_set': ModelingPeriodSet(),
    'modeled_energy_traces': {
        'DEF': SplitModeledEnergyTrace()
    },
    'modeled_energy_trace_derivatives': {
        'DEF': {
            ('baseline', 'reporting'): {
                'BASELINE': {
                    'annualized_weather_normal': (11051.638608992347, 142.473017350216, 156.41867795
                    'gross_predicted': (31806.370855869744, 251.56911436695583, 276.19340851303582, 1
                },
                'REPORTING': {
                    'annualized_weather_normal': (8758.2778181960675, 121.92101539941024, 137.246310
                     'gross_predicted': (25208.101373932539, 215.27979428803133, 242.34015188210202,
                }
            }
        }
    },
    'project_derivatives': {
        ('baseline', 'reporting'): {
            'ALL_FUELS_CONSUMPTION_SUPPLIED': {
                'BASELINE': {
                    'annualized_weather_normal': (11051.638608992347, 142.473017350216, 156.41867795
                    'gross_predicted': (31806.370855869744, 251.56911436695583, 276.19340851303582, 1
                },
                'REPORTING': {
                    'annualized_weather_normal': (8758.2778181960675, 121.92101539941024, 137.246310
                    'gross_predicted': (25208.101373932539, 215.27979428803133, 242.34015188210202, 1
                }
            },
            'ELECTRICITY_CONSUMPTION_SUPPLIED': {
                'BASELINE': {
                    'annualized_weather_normal': (11051.638608992347, 142.473017350216, 156.41867795
                    'gross_predicted': (31806.370855869744, 251.56911436695583, 276.19340851303582, 1
                },
                'REPORTING': {
                    'annualized_weather_normal': (8758.2778181960675, 121.92101539941024, 137.246310
                    'gross_predicted': (25208.101373932539, 215.27979428803133, 242.34015188210202, 1
                }
            },
            'ELECTRICITY_ON_SITE_GENERATION_UNCONSUMED': None,
            'NATURAL_GAS_CONSUMPTION_SUPPLIED': None
        }
    },
}
```

Note the contents of the dictionary:

- `'weather_source'`: An instance of `eemeter.weather.ISDWeatherSource`. The weather source used to gather observed weather data. The station at which this weather was recorded can be found by inspecting `weather_source.station`. (Matched by ZIP code)
- `'weather_normal_source'`: An instance of `eemeter.weather.TMY3WeatherSource`. The weather normal source used to gather weather normal data. The station at which this weather normal data was recorded can be found by inspecting `weather_normal_source.station`. (Matched by ZIP code)

- `'modeling_period_set'`: An instance of `eemeter.structures.ModelingPeriodSet`. The modeling periods determined by the intervention start and end dates; includes groupings. The default grouping for a single intervention is into two modeling periods called "baseline" and "reporting".
- `'modeled_energy_traces'`: `SplitModeledEnergyTraces` instances keyed by `trace_id` (as given in the `EnergyTraceSet`; includes models and fit statistics for each modeling period.
- `'modeled_energy_trace_derivatives'`: energy results specific to each modeled energy trace, organized by trace_id and modeling period group.
- `'project_derivatives'`: Project-level results which are aggregated up from the `'modeled_energy_trace_derivatives'`.

The project derivatives are nested quite deeply. The nesting of key-value pairs is as follows:

- 1st layer: Modeling Period Set id: a tuple of 1 baseline period id and 1 reporting period id, usually `('baseline', 'reporting')` - contains the results specific to this pair of modeling periods.
- 2nd layer: Trace interpretation: a string describing the trace interpretation; in our case `"ELECTRICITY_CONSUMPTION_SUPPLIED"`
- 3rd layer: `'BASELINE'` and `'REPORTING'` - these are fixed labels that always appear at this level; they demarcate the baseline aggregations and the reporting aggregations.
- 4th layer: *'annualized_weather_normal'* and *'gross_predicted'* - these are also fixed labels that always appear at this level to indicate the type of the savings values.

At the final layers are a 4-tuple of results `(value, lower, upper, n)`: `value`, indicating the estimated expected value of the selected result; `lower`, a number which can be subtracted from `value` to obtain the lower 95% confidence interval bound; `upper`, a number which can be added to `value` to obtain the upper 95% confidence interval bound, and `n`, the total number of records that went into calculation of this value.

To obtain savings numbers, the reporting value should be subtracted from the baseline value as described in *Methods Overview*.

Let's select the most useful results from the eemeter, the project-level derivatives. Note the modeling_period_set selector at the first level: *('baseline', 'reporting')*

```
project_derivatives = results['project_derivatives']
```

```
>>> project_derivatives.keys()
dict_keys([('baseline', 'reporting')])
```

```
modeling_period_set_results = project_derivatives[('baseline', 'reporting')]
```

Now we can select the desired interpretation; four are available.

```
>>> modeling_period_set_results.keys()
dict_keys(['NATURAL_GAS_CONSUMPTION_SUPPLIED', 'ALL_FUELS_CONSUMPTION_SUPPLIED', 'ELECTRICITY_CONSUME
```

```
electricity_consumption_supplied_results = modeling_period_set_results['ELECTRICITY_CONSUMPTION_SUPPI
```

The interpretation level results are broken into `"BASELINE"` and `"REPORTING"` in all cases in which they are available; otherwise, the value is *None*.)

```
>>> electricity_consumption_supplied_results.keys()
dict_keys(['BASELINE', 'REPORTING'])
```

```
baseline_results = electricity_consumption_supplied_results["BASELINE"]
reporting_results = electricity_consumption_supplied_results["REPORTING"]
```

These results have two components as well - the type of savings.

```
>>> baseline_results.keys()
dict_keys(['gross_predicted', 'annualized_weather_normal'])
>>> reporting_results.keys()
dict_keys(['gross_predicted', 'annualized_weather_normal'])
```

We select the results for one of them:

```
baseline_normal = baseline_results['annualized_weather_normal']
reporting_normal = reporting_results['annualized_weather_normal']
```

As described above, each energy value also includes upper and lower bounds, but can also be used directly to determine savings.

```
percent_savings = (baseline_normal[0] - reporting_normal[0]) / baseline_normal[0]
```

```
>>> percent_savings
0.20751319075256849
```

This percent savings value (~20%) is consistent with the savings created in the fake data.

Weather Data Caching

In order to avoid putting an unnecessary load on external weather sources, weather data is cached by default using json in a directory `~/.eemeter/cache`. The location of the directory can be changed by setting:

```
$ export EEMETER_WEATHER_CACHE_DIRECTORY=<full path to directory>
```

1.2.3 API

eemeter.ee

eemeter.ee.derivatives

class eemeter.ee.derivatives.**Derivative**(*label*, *value*, *lower*, *upper*, *n*, *serialized_demand_fixture*)

label
Alias for field number 0

lower
Alias for field number 2

n
Alias for field number 4

serialized_demand_fixture
Alias for field number 5

upper
Alias for field number 3

value
Alias for field number 1

class eemeter.ee.derivatives.**DerivativePair**(*label*, *derivative_interpretation*, *trace_interpretation*, *unit*, *baseline*, *reporting*)

baseline
Alias for field number 4

derivative_interpretation
Alias for field number 1

label
Alias for field number 0

reporting
Alias for field number 5

trace_interpretation
Alias for field number 2

unit
Alias for field number 3

eemeter.ee.derivatives.**annualized_weather_normal**(*formatter*, *model*, *weather_normal_source*)

Annualize energy trace values given a model and a source of 'normal' weather data, such as Typical Meteorological Year (TMY) 3 data.

Parameters

- **formatter** (*eemeter.modeling.formatter.Formatter*) – Formatter that can be used to create a demand fixure. Must supply the `.create_demand_fixture(index, weather_source)` method.
- **model** (*eemeter.modeling.models.Model*) – Model that can be used to predict out of sample energy trace values. Must supply the `.predict(demand_fixture_data)` method.
- **weather_normal_source** (*eemeter.weather.WeatherSource*) – WeatherSource providing weather normals.

Returns

out – Dictionary with the following item:

- `"annualized_weather_normal"`: 4-tuple with the values `(annualized, lower, upper, n)`, where
  - `annualized` is the total annualized (weather normalized) value predicted over the course of a 'normal' weather year.
  - `lower` is the number which should be subtracted from `annualized` to obtain the 0.025 quantile lower error bound.
  - `upper` is the number which should be added to `annualized` to obtain the 0.975 quantile upper error bound.
  - `n` is the number of samples considered in developing the bound - useful for adding other values with errors.

Return type dict

eemeter.ee.derivatives.**gross_predicted**(*formatter*, *model*, *weather_source*, *reporting_period*)

Find gross predicted energy trace values given a model and a source of observed weather data.

Parameters

- **formatter** (*eemeter.modeling.formatter.Formatter*) – Formatter that can be used to create a demand fixure. Must supply the `.create_demand_fixture(index, weather_source)` method.
- **model** (*eemeter.modeling.models.Model*) – Model that can be used to predict out of sample energy trace values. Must supply the `.predict(demand_fixture_data)` method.
- **weather_source** (*eemeter.weather.WeatherSource*) – WeatherSource providing observed weather data.
- **baseline_period** (eemeter.structures.ModelingPeriod) – Period targetted by baseline model.
- **reporting_period** (eemeter.structures.ModelingPeriod) – Period targetted by reporting model.

Returns

out – Dictionary with the following item:

- `"gross_predicted"`: 4-tuple with the values `(annualized, lower, upper, n)`, where
  - `gross_predicted` is the total gross predicted value over time period defined by the reporting period.
  - `lower` is the number which should be subtracted from `gross_predicted` to obtain the 0.025 quantile lower error bound.
  - `upper` is the number which should be added to `gross_predicted` to obtain the 0.975 quantile upper error bound.
  - `n` is the number of samples considered in developing the bound - useful for adding other values with errors.

Return type dict

eemeter.ee.meter

class `eemeter.ee.meter.`**EnergyEfficiencyMeter**(*settings=None*)

The standard way of calculating energy efficiency savings values from project data.

Parameters **settings** (*dict*) – Dictionary of settings (ignored; for now, this is a placeholder).

evaluate(*project*, *weather_source=None*, *weather_normal_source=None*)

Main entry point to the meter, taking in project data and returning results indicating energy efficiency performance.

Parameters

- **project** (eemeter.structures.Project) – Project for which energy effiencicency performance is to be evaluated.
- **weather_source** (*eemeter.weather.WeatherSource*) – Weather source to be used for this meter. Overrides weather source found using `project.site`. Useful for test mocking.
- **weather_normal_source** (*eemeter.weather.WeatherSource*) – Weather normal source to be used for this meter. Overrides weather source found using `project.site`. Useful for test mocking.

Returns

out – Results of energy efficiency evaluation, organized into the following items.

- `"modeling_period_set"`: `eemeter.structures.ModelingPeriodSet` determined from this project.
- `"modeled_energy_traces"`: dict of dispatched modeled energy traces.
- `"modeled_energy_trace_derivatives"`: derivatives for each modeled energy trace.
- `"project_derivatives"`: Project summaries for derivatives.
- `"weather_source"`: Matched weather source
- `"weather_normal_source"`: Matched weather normal source.

Return type dict

eemeter.io

eemeter.io.serializers

class `eemeter.io.serializers.`**`ArbitrarySerializer`**(*parse_dates=False*)

Arbitrary data at arbitrary non-overlapping intervals. Often used for montly billing data. Records must all have the "start" key and the "end" key. Overlaps are not allowed and gaps will be filled with NaN.

For example:

```
>>> records = [
...     {
...         "start": datetime(2013, 12, 30, tzinfo=pytz.utc),
...         "end": datetime(2014, 1, 28, tzinfo=pytz.utc),
...         "value": 1180,
...     },
...     {
...         "start": datetime(2014, 1, 28, tzinfo=pytz.utc),
...         "end": datetime(2014, 2, 27, tzinfo=pytz.utc),
...         "value": 1211,
...         "estimated": True,
...     },
...     {
...         "start": datetime(2014, 2, 28, tzinfo=pytz.utc),
...         "end": datetime(2014, 3, 30, tzinfo=pytz.utc),
...         "value": 985,
...     },
... ]
...
>>> serializer = ArbitrarySerializer()
>>> df = serializer.to_dataframe(records)
>>> df
                           value estimated
2013-12-30 00:00:00+00:00  1180.0     False
2014-01-28 00:00:00+00:00  1211.0      True
2014-02-27 00:00:00+00:00     NaN     False
2014-02-28 00:00:00+00:00   985.0     False
2014-03-30 00:00:00+00:00     NaN     False
```

class eemeter.io.serializers.**ArbitraryStartSerializer**(*parse_dates=False*)

Arbitrary start data at arbitrary non-overlapping intervals. Records must all have the "start" key. The last data point will be ignored unless an end date is provided for it. This is useful for data dated to future energy use, e.g. billing for delivered fuels.

For example:

```
>>> records = [
...     {
...         "start": datetime(2013, 12, 30, tzinfo=pytz.utc),
...         "value": 1180,
...     },
...     {
...         "start": datetime(2014, 1, 28, tzinfo=pytz.utc),
...         "value": 1211,
...         "estimated": True,
...     },
...     {
...         "start": datetime(2014, 2, 28, tzinfo=pytz.utc),
...         "value": 985,
...     },
... ]
...
>>> serializer = ArbitrarySerializer()
>>> df = serializer.to_dataframe(records)
>>> df
                           value estimated
2013-12-30 00:00:00+00:00  1180.0     False
2014-01-28 00:00:00+00:00  1211.0      True
2014-02-28 00:00:00+00:00     NaN     False
```

class eemeter.io.serializers.**ArbitraryEndSerializer**(*parse_dates=False*)

Arbitrary end data at arbitrary non-overlapping intervals. Records must all have the "end" key. The first data point will be ignored unless a start date is provided for it. This is useful for data dated to past energy use, e.g. electricity or natural gas bills.

For example:

```
>>> records = [
...     {
...         "end": datetime(2013, 12, 30, tzinfo=pytz.utc),
...         "value": 1180,
...     },
...     {
...         "end": datetime(2014, 1, 28, tzinfo=pytz.utc),
...         "value": 1211,
...         "estimated": True,
...     },
...     {
...         "end": datetime(2014, 2, 28, tzinfo=pytz.utc),
...         "value": 985,
...     },
... ]
...
>>> serializer = ArbitrarySerializer()
>>> df = serializer.to_dataframe(records)
>>> df
                           value estimated
2013-12-30 00:00:00+00:00  1211.0      True
```

```
2014-01-28 00:00:00+00:00   985.0     False
2014-02-28 00:00:00+00:00     NaN     False
```

eemeter.io.parsers

class eemeter.io.parsers.**ESPIUsageParser**(*xml*)
Parse ESPI XML files.

Basic usage:

```
>>> from eemeter.io.parsers import ESPIUsageParser
>>> with open("/path/to/example.xml") as f:
...     parser = ESPIUsageParser(f)
>>> energy_traces = list(parser.get_energy_traces())
```

Parameters **xml** (*str, filepath, file buffer*) – XML data to parse

get_energy_traces(*service_kind_default='electricity'*)
Retrieve all energy trace records stored as IntervalReading elements in the given ESPI Energy Usage XML.

Energy records are grouped by interpretation and returned in EnergyTrace objects.

Parameters **service_kind_default** (*str*) – Default fuel type to use in parser if ReadingType/commodity field is missing.

Yields **energy_trace** (*eemeter.structures.EnergyTrace*) – Energy data traces as described in the xml file.

has_solar()
Returns True if there is a "reverse" flow direction in this file, indicating presence of solar photo voltaics.

TODO: Verify that this is the correct way to determine this - are there false positives or false negatives? Is there a more straightforward flag to use somewhere else?

eemeter.modeling

eemeter.modeling.formatters

class eemeter.modeling.formatters.**ModelDataFormatter**(*freq_str*)
Formatter for model data of known or predictable frequency. Basic usage:

```
>>> formatter = ModelDataFormatter("D")
>>> formatter.create_input(energy_trace, weather_source)
                           energy tempF
2013-06-01 00:00:00+00:00    3.10  74.3
2013-06-02 00:00:00+00:00    2.42  71.0
2013-06-03 00:00:00+00:00    1.38  73.1
                                   ...
2016-05-27 00:00:00+00:00    0.11  71.1
2016-05-28 00:00:00+00:00    0.04  78.1
2016-05-29 00:00:00+00:00    0.21  69.6
>>> index = pd.date_range('2013-01-01', periods=365, freq='D')
>>> formatter.create_input(index, weather_source)
                           tempF
2013-01-01 00:00:00+00:00   28.3
2013-01-02 00:00:00+00:00   31.0
```

```
2013-01-03 00:00:00+00:00   34.1
                            ...
2013-12-29 00:00:00+00:00   12.3
2013-12-30 00:00:00+00:00   26.0
2013-12-31 00:00:00+00:00   24.1
```

create_demand_fixture(*index*, *weather_source*)

Creates a DatetimeIndex ed dataframe containing formatted demand fixture data.

Parameters

- **index** (*pandas.DatetimeIndex*) – The desired index for demand fixture data.
- **weather_source** (*eemeter.weather.WeatherSourceBase*) – The source of weather fixture data.

Returns **input_df** – Predictably formatted input data. This data should be directly usable as input to applicable model.predict() methods.

Return type pandas.DataFrame

create_input(*trace*, *weather_source*)

Creates a DatetimeIndex ed dataframe containing formatted model input data formatted as follows.

Parameters

- **trace** (eemeter.structures.EnergyTrace) – The source of energy data for inclusion in model input.
- **weather_source** (*eemeter.weather.WeatherSourceBase*) – The source of weather data.

Returns **input_df** – Predictably formatted input data. This data should be directly usable as input to applicable model.fit() methods.

Return type pandas.DataFrame

class eemeter.modeling.formatters.**ModelDataBillingFormatter**

Formatter for model data of unknown or unpredictable frequency. Basic usage:

```
>>> formatter = ModelDataBillingFormatter()
>>> energy_trace = EnergyTrace(
        "ELECTRICITY_CONSUMPTION_SUPPLIED",
        pd.DataFrame(
            {
                "value": [1, 1, 1, 1, np.nan],
                "estimated": [False, False, True, False, False]
            },
            index=[
                datetime(2011, 1, 1, tzinfo=pytz.UTC),
                datetime(2011, 2, 1, tzinfo=pytz.UTC),
                datetime(2011, 3, 2, tzinfo=pytz.UTC),
                datetime(2011, 4, 3, tzinfo=pytz.UTC),
                datetime(2011, 4, 29, tzinfo=pytz.UTC),
            ],
            columns=["value", "estimated"]
        ),
        unit="KWH")
>>> trace_data, temp_data = formatter.create_input(energy_trace, weather_source)
>>> trace_data
2011-01-01 00:00:00+00:00    1.0
2011-02-01 00:00:00+00:00    1.0
```

```
2011-03-02 00:00:00+00:00    2.0
2011-04-29 00:00:00+00:00    NaN
dtype: float64
>>> temp_data
period                    hourly
2011-01-01 00:00:00+00:00 2011-01-01 00:00:00+00:00  32.0
                          2011-01-01 01:00:00+00:00  32.0
                          2011-01-01 02:00:00+00:00  32.0
...                                                   ...
2011-03-02 00:00:00+00:00 2011-04-28 21:00:00+00:00  32.0
                          2011-04-28 22:00:00+00:00  32.0
                          2011-04-28 23:00:00+00:00  32.0
>>> index = pd.date_range('2013-01-01', periods=365, freq='D')
>>> formatter.create_input(index, weather_source)
                           tempF
2013-01-01 00:00:00+00:00   28.3
2013-01-02 00:00:00+00:00   31.0
2013-01-03 00:00:00+00:00   34.1
                            ...
2013-12-29 00:00:00+00:00   12.3
2013-12-30 00:00:00+00:00   26.0
2013-12-31 00:00:00+00:00   24.1
```

create_demand_fixture(*index*, *weather_source*)

Creates a `DatetimeIndex` ed dataframe containing formatted demand fixture data.

Parameters

- **index** (*pandas.DatetimeIndex*) – The desired index for demand fixture data.
- **weather_source** (*eemeter.weather.WeatherSourceBase*) – The source of weather fixture data.

Returns **input_df** – Predictably formatted input data. This data should be directly usable as input to applicable model.predict() methods.

Return type pandas.DataFrame

create_input(*trace*, *weather_source*)

Creates two `DatetimeIndex` ed dataframes containing formatted model input data formatted as follows.

Parameters

- **trace** (`eemeter.structures.EnergyTrace`) – The source of energy data for inclusion in model input.
- **weather_source** (*eemeter.weather.WeatherSourceBase*) – The source of weather data.

Returns

- **trace_data** (*pandas.DataFrame*) – Predictably formatted trace data with estimated data removed. This data should be directly usable as input to applicable model.fit() methods.
- **temperature_data** (*pandas.DataFrame*) – Predictably formatted temperature data with a pandas `MultiIndex`. The `MultiIndex` contains two levels - 'period', which corresponds directly to the trace_data index, and 'hourly' or 'daily', which contains, respectively, hourly or daily temperature data. This is intended for use like the following:

```
>>> temperature_data.groupby(level='period')
```

This data should be directly usable as input to applicable model.fit() methods.

eemeter.modeling.models

class eemeter.modeling.models.seasonal.**SeasonalElasticNetCVModel**(*cooling_base_temp=65*, *heating_base_temp=65*, *n_bootstrap=100*)

Linear regression using daily frequency data to build a model of formatted energy trace data that takes into account HDD, CDD, day of week, month, and holiday effects, with elastic net regularization.

Parameters

- **cooling_base_temp** (*float*) – Base temperature (degrees F) used in calculating cooling degree days.
- **heating_base_temp** (*float*) – Base temperature (degrees F) used in calculating heating degree days.
- **n_bootstrap** (*int*) – Number of points to exclude during bootstrap error estimation.

class eemeter.modeling.models.billing.**BillingElasticNetCVModel**(*cooling_base_temp=65*, *heating_base_temp=65*, *n_bootstrap=100*)

Linear regression of energy values against CDD/HDD with elastic net regularization.

Parameters

- **cooling_base_temp** (*float*) – Base temperature (degrees F) used in calculating cooling degree days.
- **heating_base_temp** (*float*) – Base temperature (degrees F) used in calculating heating degree days.
- **n_bootstrap** (*int*) – Number of points to exclude during bootstrap error estimation.

eemeter.processors

eemeter.processors.dispatchers

eemeter.processors.dispatchers.**get_energy_modeling_dispatches**(*modeling_period_set*, *trace_set*)

Dispatches a set of applicable models and formatters for each pairing of modeling period sets and trace sets given.

Parameters

- **modeling_period_set** (eemeter.structures.ModelingPeriodSet) – ModelingPeriod s to dispatch.
- **trace_set** (eemeter.structures.EnergyTraceSet) – EnergyTrace s to dispatch.

eemeter.processors.interventions

eemeter.processors.interventions.**get_modeling_period_set**(*interventions*)

Creates an applicable modeling period set given a list of interventions.

Parameters **interventions** (*list of eemeter.structures.Intervention*) – Interventions for which to build ModelingPeriodSet.

eemeter.processors.location

eemeter.processors.location.**get_weather_normal_source**(*site*)
Finds most relevant WeatherSource given project site.

Parameters **site** (eemeter.structures.ZIPCodeSite) – Site to match to weather source data.

Returns **weather_source** – Closest data-validated weather source in the same climate zone as project ZIP code, if available.

Return type *eemeter.weather.TMY3WeatherSource*

eemeter.processors.location.**get_weather_source**(*site*)
Finds most relevant WeatherSource given project site.

Parameters **site** (eemeter.structures.ZIPCodeSite) – Site to match to weather source data.

Returns **weather_source** – Closest data-validated weather source in the same climate zone as project ZIP code, if available.

Return type *eemeter.weather.ISDWeatherSource*

eemeter.structures

class eemeter.structures.**EnergyTrace**(*interpretation*, *data=None*, *records=None*, *unit=None*, *placeholder=False*, *serializer=None*)
Container for time series energy data.

Parameters

- **interpretation** (*str*) – The way this energy time series in the data attribute should be interpreted. The complete list of supported options is as follows:
  - ELECTRICITY_CONSUMPTION_SUPPLIED: Represents the amount of utility-supplied electrical energy consumed on-site, as metered at a single usage point, such as a utility-owned electricity meter. Specifically does not include consumption of electricity generated on site, such as by locally installed solar photovoltaic panels.
  - ELECTRICITY_CONSUMPTION_TOTAL: Represents the amount of electrical energy consumed on-site, including both utility-supplied and on-site generated electrical energy. Equivalent, for a single electricity meter, to ELECTRICITY_CONSUMPTION_SUPPLIED - ELECTRICITY_ON_SITE_GENERATION_CONSUMED.
  - ELECTRICITY_CONSUMPTION_NET: Represents the amount of utility-supplied electrical energy consumed on-site minus the amount of unconsumed electrical energy generated on site and fed back into the grid at a single usage point, such as a utility-owned electricity meter. Equivalent, for a single electricity meter, to ELECTRICITY_CONSUMPTION_SUPPLIED - ELECTRICITY_ON_SITE_GENERATION_UNCONSUMED.
  - ELECTRICITY_ON_SITE_GENERATION_TOTAL: Represents the amount of locally generated electrical energy consumed on-site plus the amount of locally generated electrical energy returned to the grid, as metered at a single usage point. Equivalent, for a single electricity meter, to ELECTRICITY_ON_SITE_GENERATION_CONSUMED + ELECTRICITY_ON_SITE_GENERATION_UNCONSUMED.

  - ELECTRICITY_ON_SITE_GENERATION_CONSUMED: Represents the amount of locally generated electrical energy consumed on-site, such as energy generated by solar photovoltaic panels.
  - ELECTRICITY_ON_SITE_GENERATION_UNCONSUMED: Represents the amount of excess locally generated energy, which instead of being consumed on-site, is fed back into the grid or sold back a utility.
  - NATURAL_GAS_CONSUMPTION_SUPPLIED: Represents the amount of energy supplied by a utility in the form of natural gas and used on site, as metered at a single usage point. Though under the labeling scheme used for electricity interpretetations the labels NATURAL_GAS_CONSUMPTION_TOTAL and NATURAL_GAS_CONSUMPTION_NET would be equivalent for natural gas, NATURAL_GAS_CONSUMPTION_SUPPLIED is prefered for its greater specificity.

- **data** (*pandas.DataFrame, default None*) – A pandas DataFrame with two columns and a timezone-aware DatetimeIndex. Timestamps in the index are assumed to refer to the start of each period, and the period ends are assumed to coincide with the start of the following period. Thus, the value of the last datetime should always be NaN, since is purpose is only to cap the end of the last period, and not to represent a time period over which energy was consumed. The DatetimeIndex does not need to have uniform frequency, such as those specified in pandas using the freq attribute.
  - value: Amount of energy between this index and the next.
  - estimated: Whether or not the value was estimated. Particularly relevant for monthly billing data.

  If serializer instance is provided, this should instead be records in the format expected by the serializer.
- **unit** (*str*) – The name of the unit in which the energy time series is given. These names are normalized to either 'KWH' or 'THERM' as follows:
  - 'kwh' becomes 'KWH' with no unit conversion multiplier.
  - 'kWh' becomes 'KWH' with no unit conversion multiplier.
  - 'KWH' becomes 'KWH' with no unit conversion multiplier.
  - 'therm' becomes 'THERM' with no unit conversion multiplier.
  - 'therms' becomes 'THERM' with no unit conversion multiplier.
  - 'thm' becomes 'THERM' with no unit conversion multiplier.
  - 'THERM' becomes 'THERM' with no unit conversion multiplier.
  - 'THERMS' becomes 'THERM' with no unit conversion multiplier.
  - 'THM' becomes 'THERM' with no unit conversion multiplier.
  - 'wh' becomes 'KWH' with a unit conversion multiplier of 0.001.
  - 'Wh' becomes 'KWH' with a unit conversion multiplier of 0.001.
  - 'WH' becomes 'KWH' with a unit conversion multiplier of 0.001.
- **placeholder** (*bool*) – Indicates that this instance is a placeholder - that while for some reason the data associated with it is unavailable, its existence is still important in considering a whole site.

- **serializer** (*consumption.BaseSerializer*) – Serializer instance to be used to deserialize records into a pandas dataframe. Must supply the `to_dataframe(records)` method.

class `eemeter.structures.`**EnergyTraceSet**(*traces*, *labels=None*)

A container for energy traces which ensures that each is labeled.

Parameters

- **traces** (*list or dict of eemeter.structures.EnergyTrace objects*) – EnergyTrace objects to be included in this list.
- **labels** (*list of str*) – Unique labels for traces, used only if *traces* is not a dictionary.

itertraces()

Iterates over traces, yielding *(label, trace)* pairs.

class `eemeter.structures.`**Intervention**(*start_date*, *end_date=None*)

Represents an intervention with a start date, and maybe an end date. Multiple interventions can be composed within a project.

Parameters

- **start_date** (*datetime.datetime*) – Must be timezone aware
- **end_date** (*datetime.datetime or None, default None*) – Must be timezone aware. If None, intervention is assumed to be ongoing.

class `eemeter.structures.`**ModelingPeriod**(*interpretation*, *start_date=None*, *end_date=None*)

Represents a period of time over which to select data from a Trace for contiguous modeling. Carries an "interpretation", for which there are two options, *"BASELINE"* and *"REPORTING"*. The period is defined by a single optional start date and a single optional end date. If the start date is not given, the start date is considered to be negative infinity; if the end date is not given, the end date is considered to be positive infinity.

A ModelingPeriod is a time period, defined by start and end dates, over which the process behind a trace can be expected, for modeling purposes, to have roughly the same energy response to end use demand. Note that this criterion might not be particularly well specified without reference to a particular intervention and set of modeling conditions.

Parameters

- **interpretation** (*str, {"BASELINE", "REPORTING"}*) – The way this ModelingPeriod should be interpreted.
  - *"BASELINE"* means that this modeling period represents the time *before* an intervention or set of interventions.
  - *"REPORTING"* means that this modeling period represents the time *after* an intervention or set of interventions.
- **start_date** (*datetime.datetime or None*) – The date marking the earliest date of the ModelingPeriod. *None* indicates a start_date of negative infinity. If interpretation is "REPORTING", start_date cannot be *None*.
- **end_date** (*datetime.datetime or None*) – The date marking the latest date of the ModelingPeriod. *None* indicates an end_date of positive infinity. If interpretation is "BASELINE", end_date cannot be *None*.

class `eemeter.structures.`**ModelingPeriodSet**(*modeling_periods*, *groupings*)

Represents a set of labeled modeling periods of interest, grouped into meaningful comparison sets. Labels can be arbitrary.

Basic usage:

```
>>> modeling_periods = {
...     "modeling_period_1": ModelingPeriod(
...         "BASELINE",
...         end_date=datetime(2000, 1, 1, tzinfo=pytz.UTC),
...     ),
...     "modeling_period_2": ModelingPeriod(
...         "REPORTING",
...         start_date=datetime(2000, 2, 1, tzinfo=pytz.UTC),
...     ),
...     "modeling_period_3": ModelingPeriod(
...         "REPORTING",
...         start_date=datetime(2000, 2, 1, tzinfo=pytz.UTC),
...     ),
... }
...
>>> grouping = [
...     ("modeling_period_1", "modeling_period_2"),
...     ("modeling_period_1", "modeling_period_3"),
... ]
...
>>> mps = ModelingPeriodSet(modeling_periods, grouping)
```

class eemeter.structures.**Project**(*energy_trace_set*, *interventions*, *site*)
Container for storing project data.

Parameters

- **trace_set** (*eemeter.structures.TraceSet*) – Complete set of energy traces for this project. For a project site that has, for example, two electricity meters, each with two traces (supplied electricity kWh, and solar-generated kWh) and one natural gas meter with one trace (consumed natural gas therms), the *trace_set* should contain 5 traces, regardless of the availablity of that data. Traces which are unavailable should be represented as 'placeholder' traces.
- **interventions** (*list of eemeter.structures.Intervention*) – Complete set of interventions, planned, ongoing, or completed, that have taken or will take place at this site as part of this project.
- **site** (*eemeter.structures.Site*) – The site of this project.

class eemeter.structures.**ZIPCodeSite**(*zipcode*)
ZIP-code-based site location descriptor.

Parameters **zipcode** (*str*) – A five-digit zipcode identifier.

eemeter.weather

GSODWeatherSource

class eemeter.weather.**GSODWeatherSource**(*station*, *cache_directory=None*)
The GSODWeatherSource draws weather data from the NOAA Global Summary of the Day FTP site. It stores fetched data locally by default in a SQLite database at ~/eemeter/cache/weather_cache.db, unless you use set the following environment variable to something different:

```
$ export EEMETER_WEATHER_CACHE_DIRECTORY=/path/to/custom/directory
```

Basic usage is as follows:

```
>>> from eemeter.weather import GSODWeatherSource
>>> ws = GSODWeatherSource("722880")  # or another 6-digit USAF station
```

This object can be used to fetch weather data as follows, using an daily frequency time-zone aware pandas DatetimeIndex covering any stretch of time.

```
>>> import pandas as pd
>>> import pytz
>>> index = pd.date_range('2015-01-01', periods=365,
...     freq='D', tz=pytz.UTC)
>>> ws.indexed_temperatures(index, "degF")
2015-01-01 00:00:00+00:00    43.6
2015-01-02 00:00:00+00:00    45.0
2015-01-03 00:00:00+00:00    47.3
                             ...
2015-12-29 00:00:00+00:00    48.0
2015-12-30 00:00:00+00:00    46.4
2015-12-31 00:00:00+00:00    47.6
Freq: D, dtype: float64
```

add_year (*year*, *force_fetch=False*)
Adds temperature data to internal pandas timeseries

Note: This method is called automatically internally to keep data updated in response to calls to *.indexed_temperatures()*

Parameters

- **year** (*{int, string}*) – The year for which data should be fetched, e.g. "2010".
- **force_fetch** (*bool, default=False*) – If `True`, forces the fetch; if `False`, checks to see if locally available before actually fetching.

add_year_range (*start_year*, *end_year*, *force_fetch=False*)
Adds temperature data to internal pandas timeseries across a range of years.

Note: This method is called automatically internally to keep data updated in response to calls to *.indexed_temperatures()*

Parameters

- **start_year** (*{int, string}*) – The earliest year for which data should be fetched, e.g. "2010".
- **end_year** (*{int, string}*) – The latest year for which data should be fetched, e.g. "2013".
- **force_fetch** (*bool, default=False*) – If True, forces the fetch; if false, checks to see if year has been added before actually fetching.

indexed_temperatures (*index*, *unit*, *allow_mixed_frequency=False*)
Return average temperatures over the given index.

Parameters

- **index** (*pandas.DatetimeIndex*) – Index over which to supply average temperatures. The `index` should be given as either an hourly ('H') or daily ('D') frequency.
- **unit** (*str, {"degF", "degC"}*) – Target temperature unit for returned temperature series.

Returns **temperatures** – Average temperatures over series indexed by `index`.

Return type pandas.Series with DatetimeIndex

ISDWeatherSource

class `eemeter.weather.`**ISDWeatherSource**(*station*, *cache_directory=None*)

The `ISDWeatherSource` draws weather data from the NOAA Integrated Surface Database (ISD) FTP site. It stores fetched hourly data locally by default in a SQLite database at `~/eemeter/cache/weather_cache.db`, unless you use set the following environment variable to something different:

```
$ export EEMETER_WEATHER_CACHE_DIRECTORY=/path/to/custom/directory
```

Basic usage is as follows:

```
>>> from eemeter.weather import ISDWeatherSource
>>> ws = ISDWeatherSource("722880")  # or another 6-digit USAF station
```

This object can be used to fetch weather data as follows, using an hourly or daily frequency time-zone aware pandas DatetimeIndex covering any stretch of time.

```
>>> import pandas as pd
>>> import pytz
>>> daily_index = pd.date_range('2015-01-01', periods=365,
...     freq='D', tz=pytz.UTC)
>>> ws.indexed_temperatures(daily_index, "degF")
2015-01-01 00:00:00+00:00    43.550000
2015-01-02 00:00:00+00:00    45.042500
2015-01-03 00:00:00+00:00    47.307500
                               ...
2015-12-29 00:00:00+00:00    47.982500
2015-12-30 00:00:00+00:00    46.415000
2015-12-31 00:00:00+00:00    47.645000
Freq: D, dtype: float64
>>> hourly_index = pd.date_range('2015-01-01', periods=365*24,
...     freq='H', tz=pytz.UTC)
>>> ws.indexed_temperatures(hourly_index, "degF")
2015-01-01 00:00:00+00:00    51.98
2015-01-01 01:00:00+00:00    50.00
2015-01-01 02:00:00+00:00    48.02
                             ...
2015-12-31 21:00:00+00:00    62.06
2015-12-31 22:00:00+00:00    62.06
2015-12-31 23:00:00+00:00    62.06
Freq: H, dtype: float64
```

add_year(*year*, *force_fetch=False*)

Adds temperature data to internal pandas timeseries

Note: This method is called automatically internally to keep data updated in response to calls to *.in-*

dexed_temperatures()

Parameters

- **year** (*{int, string}*) – The year for which data should be fetched, e.g. "2010".
- **force_fetch** (*bool, default=False*) – If `True`, forces the fetch; if `False`, checks to see if locally available before actually fetching.

add_year_range (*start_year*, *end_year*, *force_fetch=False*)
Adds temperature data to internal pandas timeseries across a range of years.

Note: This method is called automatically internally to keep data updated in response to calls to *.indexed_temperatures()*

Parameters

- **start_year** (*{int, string}*) – The earliest year for which data should be fetched, e.g. "2010".
- **end_year** (*{int, string}*) – The latest year for which data should be fetched, e.g. "2013".
- **force_fetch** (*bool, default=False*) – If True, forces the fetch; if false, checks to see if year has been added before actually fetching.

indexed_temperatures (*index*, *unit*, *allow_mixed_frequency=False*)
Return average temperatures over the given index.

Parameters

- **index** (*pandas.DatetimeIndex*) – Index over which to supply average temperatures. The `index` should be given as either an hourly ('H') or daily ('D') frequency.
- **unit** (*str, {"degF", "degC"}*) – Target temperature unit for returned temperature series.

Returns **temperatures** – Average temperatures over series indexed by `index`.

Return type pandas.Series with DatetimeIndex

TMY3WeatherSource

class `eemeter.weather.`**TMY3WeatherSource** (*station*, *cache_directory=None*, *preload=True*)
The `TMY3WeatherSource` draws weather data from the NREL's Typical Meteorological Year 3 database. It stores fetched data locally by default in a SQLite database at `~/eemeter/cache/weather_cache.db`, unless you use set the following environment variable to something different:

```
$ export EEMETER_WEATHER_CACHE_DIRECTORY=/path/to/custom/directory
```

Basic usage is as follows:

```
>>> from eemeter.weather import TMY3WeatherSource
>>> ws = TMY3WeatherSource("724830")  # or another 6-digit USAF station
```

This object can be used to fetch weather data as follows, using an daily frequency time-zone aware pandas DatetimeIndex covering any stretch of time.

```
>>> import pandas as pd
>>> import pytz
>>> daily_index = pd.date_range('2015-01-01', periods=365,
...     freq='D', tz=pytz.UTC)
>>> ws.indexed_temperatures(daily_index, "degF")
2015-01-01 00:00:00+00:00    38.6450
2015-01-02 00:00:00+00:00    40.4900
2015-01-03 00:00:00+00:00    43.9175
                              ...
2015-12-29 00:00:00+00:00    43.7750
2015-12-30 00:00:00+00:00    43.6250
2015-12-31 00:00:00+00:00    46.9250
Freq: D, dtype: float64
>>> hourly_index = pd.date_range('2015-01-01', periods=365*24,
...     freq='H', tz=pytz.UTC)
>>> ws.indexed_temperatures(hourly_index, "degF")
2015-01-01 00:00:00+00:00    51.80
2015-01-01 01:00:00+00:00    50.00
2015-01-01 02:00:00+00:00    50.00
                             ...
2015-12-31 21:00:00+00:00    53.60
2015-12-31 22:00:00+00:00    55.40
2015-12-31 23:00:00+00:00    55.40
Freq: H, dtype: float64
```

indexed_temperatures(*index*, *unit*)

Return average temperatures over the given index.

Parameters

- **index** (*pandas.DatetimeIndex*) – Index over which to supply average temperatures. The `index` should be given as either an hourly ('H') or daily ('D') frequency.
- **unit** (*str, {"degF", "degC"}*) – Target temperature unit for returned temperature series.

Returns **temperatures** – Average temperatures over series indexed by `index`.

Return type pandas.Series with DatetimeIndex

Location

eemeter.weather.location.**climate_zone_is_supported**(*climate_zone*)

True if given Climate Zone is supported.

Parameters **climate_zone** (*str*) – String representing a climate_zone.

Returns **supported** – *True* if supported, otherwise *False*.

Return type bool

eemeter.weather.location.**climate_zone_to_tmy3_stations**(*climate_zone*)

Return TMY3 weather stations falling within in the given climate zone.

Parameters **climate_zone** (*str*) – String representing a climate zone.

Returns **stations** – Strings representing TMY3 station ids.

Return type list of str

eemeter.weather.location.**climate_zone_to_usaf_stations**(*climate_zone*)
Return USAF weather stations falling within in the given climate zone.

Parameters **climate_zone** (*str*) – String representing a climate zone.

Returns **stations** – Strings representing USAF station ids.

Return type list of str

eemeter.weather.location.**climate_zone_to_zipcodes**(*climate_zone*)
Return ZIP codes with centroids in the given climate zone.

Parameters **climate_zone** (*str*) – String representing a climate zone.

Returns **zipcodes** – Strings representing USPS ZIP codes.

Return type list of str

eemeter.weather.location.**haversine**(*lat1*, *lng1*, *lat2*, *lng2*)
Calculate the great circle distance between two points on the earth (specified in decimal degrees)

Parameters

- **lat1** (*float*) – Latitude coordinate of first point.
- **lng1** (*float*) – Longitude coordinate of first point.
- **lat2** (*float*) – Latitude coordinate of second point.
- **lng2** (*float*) – Longitude coordinate of second point.

Returns **distance** – Kilometers between the two lat/lng coordinates.

Return type float

eemeter.weather.location.**lat_lng_to_climate_zone**(*lat*, *lng*)
Return the closest ZIP code using latitude and longitude coordinates.

Parameters

- **lat** (*float*) – Latitude coordinate.
- **lng** (*float*) – Longitude coordinate.

Returns **climate_zone** – String representing a climate zone.

Return type str, None

eemeter.weather.location.**lat_lng_to_tmy3_station**(*lat*, *lng*)
Return the closest TMY3 station ID using latitude and longitude coordinates.

Parameters

- **lat** (*float*) – Latitude coordinate.
- **lng** (*float*) – Longitude coordinate.

Returns **station** – String representing a TMY3 weather station ID or None, if none was found.

Return type str, None

eemeter.weather.location.**lat_lng_to_usaf_station**(*lat*, *lng*)
Return the closest USAF station ID using latitude and longitude coordinates.

Parameters

- **lat** (*float*) – Latitude coordinate.
- **lng** (*float*) – Longitude coordinate.

Returns **station** – String representing a USAF weather station ID or None, if none was found.

Return type str, None

eemeter.weather.location.**lat_lng_to_zipcode**(*lat*, *lng*)
Return the closest ZIP code using latitude and longitude coordinates.

Parameters

- **lat** (*float*) – Latitude coordinate.
- **lng** (*float*) – Longitude coordinate.

Returns **zipcode** – String representing a USPS ZIP code, or None, if none was found.

Return type str, None

eemeter.weather.location.**tmy3_station_is_supported**(*station*)
True if given TMY3 weather station is supported. USAF IDs.

Parameters **station** (*str*) – 6-digit string representing a weather station.

Returns **supported** – *True* if supported, otherwise *False*.

Return type bool

eemeter.weather.location.**tmy3_station_to_climate_zone**(*station*)
Return the climate zone of the station.

Parameters **station** (*str*) – String representing a USAF Weather station ID

Returns **climate_zone** – String representing a climate zone.

Return type str

eemeter.weather.location.**tmy3_station_to_lat_lng**(*station*)
Return the latitude and longitude coordinates of the given station.

Parameters **station** (*str*) – String representing a TMY3 USAF Weather station ID

Returns **lat_lng** – Latitude and longitude coordinates.

Return type tuple of float

eemeter.weather.location.**tmy3_station_to_zipcodes**(*station*)
Return the zipcodes that map to this station.

Parameters **station** (*str*) – String representing a USAF Weather station ID

Returns **zipcode** – String representing a USPS ZIP code.

Return type list of str

eemeter.weather.location.**usaf_station_is_supported**(*station*)
True if given USAF weather station is supported. USAF IDs.

Parameters **station** (*str*) – 6-digit string representing a weather station.

Returns **supported** – *True* if supported, otherwise *False*.

Return type bool

eemeter.weather.location.**usaf_station_to_climate_zone**(*station*)
Return the climate zone of the station.

Parameters **station** (*str*) – String representing a USAF Weather station ID

Returns **climate_zone** – String representing a climate zone

Return type str

eemeter.weather.location.**usaf_station_to_lat_lng**(*station*)
Return the latitude and longitude coordinates of the given USAF station.

Parameters **station** (*str*) – String representing a USAF Weather station ID

Returns **lat_lng** – Latitude and longitude coordinates.

Return type tuple of float

eemeter.weather.location.**usaf_station_to_zipcodes**(*station*)
Return the zipcodes that map to this USAF station.

Parameters **station** (*str*) – String representing a USAF Weather station ID

Returns **zipcodes** – Strings representing a USPS ZIP code mapped to from this station.

Return type list of str

eemeter.weather.location.**zipcode_is_supported**(*zipcode*)
True if given ZIP Code is supported. ZCTA only.

Parameters **zipcode** (*str*) – 5-digit string representing a zipcode.

Returns **supported** – *True* if supported, otherwise *False*.

Return type bool

eemeter.weather.location.**zipcode_to_climate_zone**(*zipcode*)
Return the climate zone of the ZIP code (by latitude and longitude centroid of ZIP code).

Parameters **zipcode** (*str*) – String representing a USPS ZIP code.

Returns **climate_zone** – String representing a climate zone

Return type str

eemeter.weather.location.**zipcode_to_lat_lng**(*zipcode*)
Return the latitude and longitude centroid of a particular ZIP code.

Parameters **zipcode** (*str*) – String representing a USPS ZIP code.

Returns **lat_lng** – Latitude and longitude coordinates.

Return type tuple of float

eemeter.weather.location.**zipcode_to_tmy3_station**(*zipcode*)
Return the nearest TMY3 station (by latitude and longitude centroid) of the ZIP code.

Parameters **zipcode** (*str*) – String representing a USPS ZIP code.

Returns **station** – String representing a TMY3 Weather station (USAF ID).

Return type str

eemeter.weather.location.**zipcode_to_usaf_station**(*zipcode*)
Return the nearest USAF station (by latitude and longitude centroid) of the ZIP code.

Parameters **zipcode** (*str*) – String representing a USPS ZIP code.

Returns **station** – String representing a USAF weather station ID

Return type str

1.2.4 Development

Testing

This library uses the py.test framework. To develop locally, clone the repo, and in a virtual environment execute the following commands:

```
$ git clone https://github.com/openeemeter/eemeter
$ cd eemeter
$ mkvirtualenv eemeter
$ pip install -r dev_requirements.txt
$ pip install -e .
$ tox
```

Building Documentation

Documentation is built using the `sphinx` package. To build documentation, make sure that dev requirements are installed:

```
$ pip install -r dev_requirements.txt
```

And run the following from the root project directory.

```
$ make -C docs html
```

To clean the build directory, run the following:

```
$ make -C docs clean
```

1.3 datastore

The datastore is an application for housing energy and project data which provides a REST API for loading data, computing energy savings, and inspecting results. Like the eemeter library, the datastore is open source and available on github under an MIT license.

The datastore uses the django web framework with a PostgreSQL database.

1.3.1 Development Setup

Clone the repo and change directories

```
git clone git@github.com:openeemeter/datastore.git
cd datastore
```

Install required python packages

We recommend using virtualenv (or virtualenvwrapper) to manage python packages

```
mkvirtualenv datastore
pip install -r requirements.txt
pip install -r dev-requirements.txt
```

Define the necessary environment variables

```
# django
export DJANGO_SETTINGS_MODULE=oeem_energy_datastore.settings
export SECRET_KEY=<django-secret-key>  # random string

# postgres
export DATABASE_URL=postgres://user:password@host:5432/dbname

# for API docs - should reflect the IP or DNS name where datastore will be deployed
export SERVER_NAME=0.0.0.0:8000
export PROTOCOL=http  # or https

# For development only
export DEBUG=true

# For celery background tasks
export CELERY_ALWAYS_EAGER=true

  or

export BROKER_TRANSPORT=redis
export BROKER_URL=redis://user:password@host:9549
```

If developing on the datastore, you might consider adding these to your virtualenv postactivate script:

```
vim /path/to/virtualenvs/datastore/bin/postactivate

# Refresh environment
workon datastore
```

Run database migrations

```
python manage.py migrate
```

Seed the database

```
python manage.py dev_seed
```

Start a development server

```
python manage.py runserver
```

1.3.2 Topics

Basic Usage

This tutorial is also available as a jupyter `notebook`:

Before getting started, download some sample energy data and project data:

- energy data `CSV`
- project data `CSV`

Running meters

This topic page covers scheduling and executing meter runs on the datastore.

Background

Running a meter means pulling trace data, matching it with relevant project data, and evaluating its energy effiency performance. This is the central task performed by the datastore, so if the specifics are unfamiliar, there is a bit more background information worthy of review in the *Methods Overview* section of the *guides*.

Note: We will use the `requests` python package for making requests, but you could just as easily use a tool like cURL or Postman.

If you have the *eemeter* package installed, you will also have the *requests* package installed, but if not, you can install it with:

```
$ pip install requests
```

A request using the requests library looks like this:

```
import requests
url = "https://example.com"
data = {
    "first_name": "John",
    "last_name": "Doe"
}
requests.post(url + "/api/users/", json=data)
```

which is equivalent to:

```
POST /api/users/ HTTP/1.1
Host: example.com
{
    "first_name": "John",
    "last_name": "Doe"
}
```

Setup

For this demonstration, we will assume that you have the following setup, although of course yours will likely differ:

1. a datastore application running at `https://example.openeemeter.org/`
2. a project with primary key 1, associated with traces 2, 3
3. a project with primary key 2, associated with trace 4
4. a trace primary key 2 (`ELECTRICITY_CONSUMPTION_SUPPLIED`)
5. a trace primary key 3 (`NATURAL_GAS_CONSUMPTION_SUPPLIED`)
6. a trace primary key 4 (`NATURAL_GAS_CONSUMPTION_SUPPLIED`)

You should run something like the following, which sets up the variables we will be using below.

```
# setup
import requests

url = "https://example.openeemeter.org"
access_token = "INSERT_TOKEN_HERE"
headers = {"Authorization": "Bearer {}".format(access_token)}
```

Scheduling a single meter run

There are a few ways to schedule a meter run. The simplest is the following, which triggers a meter run with default model and formatter for the specified trace (primary key 2):

```
data = {"trace": 2}
response = requests.post(url + "/api/v1/meter_runs/",
                         json=data, headers=headers)
```

```
>>> response.json()
{
    'id': 1,
    'project': 1,
    'trace': 2,
    'status': 'PENDING',
    'meter_input': None,
    'formatter_class': None,
    'formatter_kwargs': None,
    'model_class': None,
    'model_kwargs': None,
    'failure_message': None,
    'traceback': None,
    'added': '2016-09-28T23:57:21.454235Z',
    'updated': '2016-09-28T23:57:21.454260Z'
}
```

The response shows us the complete specification of the meter run behavior, which is as follows:

1. the project was determined implicitly from the trace,
2. the status is `"PENDING"`, which means the tasks is scheduled but not yet running or completed
3. the `meter_input` has not yet been created (this is the complete serialized input to the meter, as required by the eemeter.)
4. the model class, formatter class, and keyword arguments are left blank, indicating that default values will be used.
5. the failure message and traceback are unpopulated, indicating no errors in execution (yet)

If you wish, you can also specify many of these properties explicitly:

```
data = {
    "trace": 2,
    "project": 2,
    "model_class": "MyModel",
    "model_kwargs": {
        "parameter_1": 1.5,
        "parameter_2": [0.8, 0.2],
    },
    "formatter_class": "MyFormatter",
    "formatter_kwargs": {},
```

```
}
response = requests.post(url + "/api/v1/meter_runs/",
                         json=data, headers=headers)
```

```
>>> response.json()
{
    'id': 2,
    'project': 2,
    'trace': 2,
    'status': 'PENDING',
    'meter_input': None,
    'model_class': 'MyModel',
    'model_kwargs': {
        'parameter_1': 1.5,
        'parameter_2': [0.8, 0.2],
    },
    'formatter_class': 'MyFormatter',
    'formatter_kwargs': {},
    'failure_message': None,
    'traceback': None,
    'added': '2016-09-28T23:58:35.233478Z',
    'updated': '2016-09-28T23:58:35.233492Z'
}
```

Or, if you leave out the project and trace attributes, you can specify the exact serialized input:

```
data = {
    "meter_input": {...},
}
response = requests.post(url + "/api/v1/meter_runs/",
                         json=data, headers=headers)
```

```
>>> response.json()
{
    'id': 3,
    'project': None,
    'trace': None,
    'status': 'PENDING',
    'meter_input': 'https://example.storage.googleapis.com/media/meter_inputs/010f59ae-15e9-4c43-8431
    'formatter_class': None,
    'formatter_kwargs': None,
    'model_class': None,
    'model_kwargs': None,
    'failure_message': None,
    'traceback': None,
    'added': '2016-09-28T23:59:02.667663Z',
    'updated': '2016-09-28T23:59:02.667681Z'
}
```

Scheduling bulk meter runs

To schedule bulk meter runs, instead of specifying a project and/or trace, you specify a set of targets, which are sets of project and/or trace.:

```
data = {
    "targets": [
        {
```

```
            "project": 1,
        },
        {
            "project": 2,
        }
    ]
}
response = requests.post(url + "/api/v1/meter_runs/bulk/",  # note: different url!
                         json=data, headers=headers)
```

```
>>> response.json()
[
    [
        {
            'id': 4,
            'project': 1,
            'trace': 2,
            'status': 'PENDING',
            'meter_input': None,
            'formatter_class': None,
            'formatter_kwargs': None,
            'model_class': None,
            'model_kwargs': None,
            'failure_message': None,
            'traceback': None,
            'added': '2016-09-29T00:01:43.152522Z',
            'updated': '2016-09-29T00:01:43.152545Z'
        },
        {
            'id': 5,
            'project': 1,
            'trace': 3,
            'status': 'PENDING',
            'meter_input': None,
            'formatter_class': None,
            'formatter_kwargs': None,
            'model_class': None,
            'model_kwargs': None,
            'failure_message': None,
            'traceback': None,
            'added': '2016-09-29T00:01:43.152557Z',
            'updated': '2016-09-29T00:01:43.152576Z'
        }
    ],
    [
        {
            'id': 6,
            'project': 2,
            'trace': 4,
            'status': 'PENDING',
            'meter_input': None,
            'formatter_class': None,
            'formatter_kwargs': None,
            'model_class': None,
            'model_kwargs': None,
            'failure_message': None,
            'traceback': None,
            'added': '2016-09-29T00:01:43.152578Z',
```

```
            'updated': '2016-09-29T00:01:43.152590Z'
        }
    ]
]
```

Note how results are returned grouped by target; each of the traces associated with the specified project are triggered simultaneously.

If model or formatter class or kwarg arguments are supplied, they will be applied to all meter_runs.

Once you have completed meter runs, you can create aggreations of the results.

See how to run aggregations: *Running Aggregations*.

Running aggregations

We assume the same setup we used in *Running meters*

Scheduling a single aggregation run

Aggregations of meter results are likewise scheduled through the API. They are scheduled as unions of derivatives from 3 sets of objects: projects, traces, or derivatives. You may specify any set of projects, traces, or derivatives from which to draw derivatives for aggregation.

Since aggregations must be across like objects, trace interpretation and derivative interpretation can be supplied as filters, or left implicit (although you will get errors if there are inconsistencies).

The following will create an aggregation (sum) of derivatives from projects 1 and 2 with the interpretation annualized_weather_normal from traces matching the interpretation "E_C_S".

```
data = {
    "projects": [1, 2],
    "derivatives": [],
    "traces": [],
    "trace_interpretation": "E_C_S",
    "derivative_interpretation": "annualized_weather_normal",
    "aggregation_interpretation": "SUM",
}
response = requests.post(url + "/api/v1/aggregation_runs/",
                         json=data, headers=headers)
```

```
>>> response.json()
{
    'id': 1,
    'status': 'PENDING',
    'projects': [1, 2],
    'traces': [],
    'derivatives': [],
    'aggregation_input': 'https://example.storage.googleapis.com/media/aggregation_inputs/3cdfc090-ec
    'trace_interpretation': 'E_C_S',
    'derivative_interpretation': 'annualized_weather_normal',
    'aggregation_interpretation': 'SUM',
}
```

Scheduling bulk aggregation runs

The mechanism for scheduling bulk aggregation runs is analogous to the mechanism for scheduling bulk meter runs. If interpretation fields are left off, the implication is that all types of aggregations should be attempted. Only aggregations for which 1 or more derivative is available matching the interpretation will be included. For the bulk method, interpretations should be supplied as lists, as shown in comments.

```
data = {
    "targets": [
        {
            "projects": [1, 2],
            "derivatives": [],
            "traces": [],
            # 'trace_interpretations': ['E_C_S', 'NG_C_S'],
            # 'derivative_interpretations': ['annualized_weather_normal', 'gross_predicted'],
        }
    ]
}
response = requests.post(url + "/api/v1/aggregation_runs/bulk/",
                         json=data, headers=headers)
```

```
>>> response.json()
[
    [
        {
            'id': 2,
            'status': 'PENDING',
            'derivatives': [],
            'projects': [1, 2],
            'traces': [],
            'aggregation_input': 'https://example.storage.googleapis.com/media/meter_inputs/010f59ae-
            'trace_interpretation': 'E_C_S',
            'derivative_interpretation': 'annualized_weather_normal',
            'aggregation_interpretation': 'SUM',
        },
        {
            'id': 3,
            'status': 'PENDING',
            'derivatives': [],
            'projects': [1, 2],
            'traces': [],
            'aggregation_input': 'https://example.storage.googleapis.com/media/meter_inputs/30eca307-
            'trace_interpretation': 'E_C_S',
            'derivative_interpretation': 'gross_predicted',
            'aggregation_interpretation': 'SUM',
        },
        {
            'id': 5,
            'status': u'PENDING',
            'derivatives': [],
            'projects': [1, 2],
            'traces': [],
            'aggregation_input': 'https://example.storage.googleapis.com/media/meter_inputs/7fc34cd6-
            'trace_interpretation': 'NG_C_S',
            'derivative_interpretation': 'annualized_weather_normal',
            'aggregation_interpretation': 'SUM',
        },
        {
```

```
            'id': 5,
            'status': u'PENDING',
            'derivatives': [],
            'projects': [1, 2],
            'traces': [],
            'aggregation_input': 'https://example.storage.googleapis.com/media/meter_inputs/c2d95844-
            'trace_interpretation': 'NG_C_S',
            'derivative_interpretation': 'gross_predicted',
            'aggregation_interpretation': 'SUM',
        }
    ]
]
```

PostgreSQL tables

A data dictionary describing available datastore database tables.

Data loaded through ETL

Name of Table	Name of Row	Description of Row
datastore_project		
	id	Primary key
	project_id	Unique project identifier provided by the user
	project_owner_id	Foreign key to *datastore_projectowner* table (not currently used, but coul
	baseline_period_start	[null]
	baseline_period_end	Populated through ETL from project data
	reporting_period_start	Populated through ETL from project data
	reporting_period_end	[null]
	zipcode	Populated through ETL from project data
	added	Date Added
	updated	Date updated
datastore_consumptionmetadata		Refers to an energy trace (a time series of data from a meter)
	id	Primary key
	project_id	Foriegn key to *datastore_project* table
	interpretation	What type of Energy Data: Supplied from Grid; Unconsumed Onsite Ger
	label	Used to distinguish traces of the same interpretation and unit within a sin
	unit	Kwh, etc.
	added	Date that the data was added to the database
	updated	Timestamp for last updated
datastore_consumptionrecord		Contains each consumption interval
	id	Primary key
	metadata_id	Foreign key to *datastore_consumptionmetadata* table
	value	Consumption within the particular interval
	estimated	T/F
	start	start time of interval; end is given by next record (as ordered by start time
datastore_projectattribute		Custom attributes associated with a project.
	id	Primary key
	key_id	Foreign key to the *datastore_projectattributekey* table
	project_id	Foreign key to the *datastore_project* table
	float_value	[null] unless associated *datastore_projectattributekey* row with *datatype*

Table 1.1 – continued from previous page

Name of Table	Name of Row	Description of Row
	integer_value	[null] unless associated *datastore_projectattributekey* row with *datatype*
	boolean_value	[null] unless associated *datastore_projectattributekey* row with *datatype*
	char_value	[null] unless associated *datastore_projectattributekey* row with *datatype*
	date_value	[null] unless associated *datastore_projectattributekey* row with *datatype*
	datetime_value	[null] unless associated *datastore_projectattributekey* row with *datatype*
datastore_projectattributekey		Types of custom attributes associated with projects
	id	Primary key
	datatype	FLOAT/INTEGER/BOOLEAN/CHAR/DATE/DATETIME
	name	Unique Name of Project Attribute type
	display_name	Name of the Project Attribute type for display purposes

Meter run result data

Data here is all organized underneath the *datastore_projectresult* table.

Name of Table	Name of Row	Description of Row
datastore_projectresult		One for each meter run/result for each project
	id	Primary key
	project_id	Foreign key to the *datastore_project* table
	meter_settings	[blank] would indicate any special settings used when running
	eemeter_version	0.4.0
	meter_class	EnergyEfficiencyMeter; refers to which class of meter, i.e., DE
	weather_normal_source_station	Uses TMY3 'normal' weather for a particular region - station
	weather_source_station	Observed temperatures - station given by 6-digit USAF id
	added	Date added
	updated	Date updated
datastore_derivative		One per interpretation per modeling period per energy trace
	id	Primary key
	energy_trace_model_result_id	foreign key to *datastore_energytracemodelresult* table
	value	value of interpretation
	upper	number to be added to *value* to obtain 95% upper bound
	lower	number to be subtracted from *value* to obtain 95% lower boun
	n	number of modeled values used in determining the derivative
	interpretation	e.g., annualized weather normal; gross predicted
datastore_derivativeaggregation		One per project per derivative interpretation per trace interpret
	id	Primary key
	project_result_id	Foreign key to *datastore_projectresult* table
	modeling_period_group_id	Foreign key to *datastore_modelingperiodgroup* table
	interpretation	Type of output - e.g., annualized weather normal; gross predic
	trace_interpretation	electricity consumption supplied (e_c_s); all fuels; natural gas
	baseline_value	Sum of values that match interpretations
	baseline_upper	amount to be added to *baseline_value* to obtain 95% upper bo
	baseline_lower	amount to be subtracted from *baseline_value* to obtain 95% lo
	baseline_n	number of values that comprise the aggregation (sum of the n'
	reporting_value	Sum of values that match interpretations
	reporting_upper	number to be added to *reporting_value* to obtain 95% upper b
	reporting_lower	number to be subtracted from *reporting_value* to obtain 95% l
	reporting_n	number of values that comprise the aggregation (sum of the n'
datastore_energytracemodelresult		One per modeling period per energy trace id

Name of Table	Name of Row	Description of Row
	id	Primary key
	energy_trace_id	Foreign key to *datastore_consumptionmetadata* table
	project_result_id	Foreign key to *datastore_projectresult* table
	modeling_period_id	Foreign key to *datastore_modelingperiod* table
	cvrmse	Coefficient of Variation of Root Mean Square Error (normaliz
	lower	Amount to be subtracted from any individual modeled data po
	upper	Amount to be added to any individual modeled data point to st
	n	Number of datapoints in the section of the trace used for the m
	r2	r^2 error; extent to which the model captures the variation in th
	rmse	Error term (root mean square error)
	model_serialization	[blank] - the way that we store the model in case we want to re
	status	SUCCESS/FAILURE
	traceback	If *status* is 'FAILURE', the python traceback for the error that
	input_start_date	Start date of data used for model building
	input_end_date	End date of data used for model building
	input_n_rows	Number of data points used for model building
	records_start_date	Date of first available energy trace record, including those not
	records_end_date	Date of last available energy trace record, including those not
	records_count	Count of available energy trace records, including those not ex
datastore_modelingperiod		Project baseline end and project reporting start constitute the e
	id	Primary key
	interpretation	BASELINE/REPORTING
	project_result_id	Foreign key to *datastore_projectresult* table
	start_date	If baseline, *start_date* can be null; if reporting *start_date* requ
	end_date	If baseline, *end_date* required; if reporting *end_date* can be nu
datastore_modelingperiodgroup		Meaningful pairs of Baseline and Reporting periods.
	id	Primary key
	project_result_id	Foreign key to *datastore_projectresults* table
	baseline_period_id	Foreign key to *datastore_modelingperiod* table
	reporting_period_id	Foreign key to *datastore_modelingperiod* table

Other reference tables

Name of Table	Name of Row	Description of Row
datastore_projectrun		Table of tasks used to trigger meter runs; should correspond to project results
	id	Primary key
	project_id	Foreign key to the *datastore_project* table
	project_result_id	Foreign key to the *datastore_projectresult* table
	meter_class	EnergyEfficiencyMeter; refers to which class of meter, i.e., DR meter
	project_settings	[blank] would indicate any special settings used when running the meter
	status	Task status: one of SUCCESS/PENDING/FAILED/RUNNING
	added	Date added
	updated	Date updated
datastore_projectblock		Groupings of projects
	id	Primary key
	name	Name for block
	added	Date added
	updated	Date updated
datastore_projectblockprojects		Many-to-many through table for project block groupings
	id	Primary key
	project_id	Foreign key to the *datastore_project* table
	project_block_id	Foreign key to the *datastore_projectblock* table
datastore_projectowner		The datastore-specific user model; created in one-to-one with django User model.
	id	Primary key
	user_id	Foreign key to the django User table
	added	Date added
	updated	Date updated

Trace-centric metering tables

Name of Table	Name of Row	Description of Row
metering_aggregationderivativestatus		Many-to-many through table storing how each derivative was i
	id	Primary key
	status	ACCEPTED/REJECTED
	baseline_status	ACCEPTED/REJECTED/DEFAULT
	reporting_status	ACCEPTED/REJECTED/DEFAULT
	aggregation_result_id	Primary key of aggregation result
	derivative_id	Primary key of derivative
metering_aggregationresult		Describes results of aggregations
	id	Primary key
	trace_interpretation	Interpretation of all traces included in result
	derivative_interpretation	Interpretation of all derivatives included in result
	aggregation_interpretation	Aggregation function
	aggregation_output	Filename of JSON serialization of aggregation output
	unit	Unit of measure of all values, lower and upper bounds

Table 1.3 – continued from previous page

Name of Table	Name of Row	Description of Row
	baseline_value	Aggregated baseline value
	baseline_lower	Amount to be subtracted from baseline_value to obtain lower b
	baseline_upper	Amount to be added to baseline_value to obtain upper bound o
	baseline_n	Number of prediction points included in aggregation
	reporting_value	Aggregated reporting value
	reporting_lower	Amount to be subtracted from reporting_value to obtain lower
	reporting_upper	Amount to be added to reporting_value to obtain upper bound
	reporting_n	Number of prediction points included in aggregation
	differential_direction	BASELINE_MINUS_REPORTING/REPORTING_MINUS_B
	differential_value	Aggregated difference between baseline and reporting
	differential_lower	Amount to be subtracted from differential_value to obtain lowe
	differential_upper	Amount to be added to differential_value to obtain upper bound
	differential_n	Number of prediction points included in aggregation
	added	Date added
	updated	Date updated
	aggregation_run_id	Primary key of corresponding aggregation_run
metering_aggregationrun		Describes aggregation task to be performed
	id	Primary key
	aggregation_input	Filename of JSON serialization of aggregation input
	status	PENDING/RUNNING/SUCCESS/FAILURE
	traceback	Traceback of error, if any
	trace_interpretation	Interpretation of all traces included in result
	derivative_interpretation	Interpretation of all derivatives included in result
	aggregation_interpretation	Aggregation function
	added	Date added
	updated	Date updated
metering_aggregationrun_derivatives		Many-to-many through table describing derivatives included in
	id	Primary key
	aggregationrun_id	Primary key of aggregation run
	meterderivative_id	Primary key of derivative
metering_aggregationrun_projects		Many-to-many through table describing projects included in an
	id	Primary key
	aggregationrun_id	Primary key of aggregation run
	project_id	Primary key of project
metering_aggregationrun_traces		Many-to-many through table describing traces included in an a
	id	Primary key
	aggregationrun_id	Primary key of aggregation run
	consumptionmetadata_id	Primary key of trace
metering_meterderivative		Table of predictive and descriptive summaries of savings
	id	Primary key
	interpretation	Interpretation of derivative (e.g., gross_predicted/annualized_w
	unit	Unit of values, upper and lower bounds.
	baseline_value	Modeled counterfactual baseline value
	baseline_lower	Amount to be subtracted from baseline_value to obtain lower b
	baseline_upper	Amount to be added to baseline_value to obtain upper bound o
	baseline_n	Number of points in baseline demand fixture
	reporting_value	Modeled reporting period value
	reporting_lower	Amount to be subtracted from reporting_value to obtain lower
	reporting_upper	Amount to be added to reporting_value to obtain upper bound

Table 1.3 – continued from previous page

Name of Table	Name of Row	Description of Row
	reporting_n	Number of points in reporting demand fixture
	added	Date added
	updated	Date updated
	meter_result_id	Primary key of meter result this derivative was extracted from
	modeling_period_group_id	Primary key of modeling period group describing baseline and
	trace_id	Primary key of trace this derivative applies to
metering_meterresult		Table of meter run results
	id	Primary key
	meter_output	Filename of JSON serialization of meter output
	status	SUCCESS/FAILURE
	eemeter_version	Version of eemeter library used to calculate this result
	datastore_version	Version of datastore application used to calculate this result
	model_class	Name of model class
	model_kwargs	Keyword arguments to model class
	formatter_class	Name of formatter class
	formatter_kwargs	Keyword arguments to formatter class
	added	Date added
	updated	Date updated
	meter_run_id	Primary key of meter run
	project_id	Primary key of project data
	trace_id	Primary key of trace
metering_meterrun		Table of meter runs
	id	Primary key
	meter_input	Filename of JSON serialiation
	status	PENDING/RUNNING/SUCCESS/FAILURE
	failure_message	Failure message, if any
	traceback	Traceback text, if error occured
	model_class	Name of model class supplied, if any
	model_kwargs	Model class keyword arguments supplied, if any
	formatter_class	Name of formatter class supplied, if any
	formatter_kwargs	Formatter class keyword arguments supplied, if any
	added	Date added
	updated	Date updated
	project_id	Primary key of project data
	trace_id	Primary key of trace
metering_modelingperiod		Table describing a modeling period
	id	Primary key
	label	Label to distinguish from other baseine/reporting/periods in sa
	interpretation	BASELINE/REPORTING
	start	Date of modeling period start, if any (can be blank for baseline
	end	Date of modeling period end, if any (can be blank for reporting
	meter_result_id	Primary key of containing meter result
metering_modelingperiodgroup		Table describing a pair of modeling periods (baseline + reporti
	id	Primary key
	baseline_id	Primary key of baseline modeling period
	meter_result_id	Primary key of containing meter result
	reporting_id	Primary key of reporting modeling period
metering_modelresult		Table storing results from modeling
	id	Primary key

Table 1.3 – continued from previous page

Name of Table	Name of Row	Description of Row
	status	SUCCESS/FAILURE
	traceback	Traceback, if any
	start_date	Start date of data used in modeling
	end_date	End date of data used in modeling
	n_rows	number of rows supplied as input to modeling
	r2	R-squared model fit
	cvrmse	Coefficient of variation of root mean squared error (rmse norma
	rmse	root mean squared error
	lower	Value to be subtracted from any individual predicted point to ob
	upper	Value to be added to aby individual predicted point to obtain up
	added	Date added
	updated	Date updated
	meter_result_id	Primary key of meter result
	modeling_period_id	Primary key of modeling period
	trace_id	Primary key of trace

1.3.3 API

1.4 ETL Toolkit

The ETL toolkit is provided to assist moving data from its source into the datastore.

"ETL" stands for Extract-Transform-Load. These three steps outline the actions the ETL toolkit helps with and are as follows:

- **Extract**: obtain data from an external (non-datastore) source.
- **Transform**: convert that data into a form usable the datatore.
- **Load**: move the transformed data into the datastore.

The ETL library is not run directly. Rather, its components are used to build ETL pipelines that are specific to a datastore instance.

1.4.1 Installation

To install the ETL library, run the following:

```
$ git clone https://github.com/openeemeter/etl
$ cd etl
$ pip install -r requirements.txt
```

For more information, see github.

1.4.2 API

...

CHAPTER 2

References

- PRISM
- ANSI/BPI-2400-S-2012
- NREL's Uniform Methods

License

MIT

e