pvops

Release 0.3.0

pvOps Developers

Jan 29, 2024

AVAILABLE RESOURCES:

pvops is a python package for PV operators & researchers. It consists of a set of documented functions for supporting operations research of photovoltaic (PV) energy systems. The library leverages advances in machine learning, natural language processing and visualization tools to extract and visualize actionable information from common PV data including Operations & Maintenance (O&M) text data, timeseries production data, and current-voltage (IV) curves.

Table 1: Module Overview

Module	Type of data	Highlights of functions
text	O&M records	• fill data gaps in dates and categorical records • visualize word clusters and patterns over time
timeseries	Production data	• estimate expected energy with multiple models • evaluate inverter clipping
text2time	O&M records and production data	• analyze overlaps between O&M and production (timeseries) records • visualize overlaps between O&M records and production data
iv	IV records	• simulate IV curves with physical faults • extract diode parameters from IV curves • classify faults using IV curves

CHAPTER

ONE

STATEMENT OF NEED

Continued interest in PV deployment across the world has resulted in increased awareness of needs associated with managing reliability and performance of these systems during operation. Current open-source packages for PV analysis focus on theoretical evaluations of solar power simulations (e.g., *pvlib*; [HHM18]), specific use cases of empirical evaluations (e.g., *RdTools*; [DJN+18] and *Pecos*; [KS16] for degradation analysis), or analysis of electroluminescene images (e.g., *PVimage*; [PKL+20]). However, a general package that can support data-driven, exploratory evaluations of diverse field collected information is currently lacking. To address this gap, we present *pvOps*, an open-source, Python package that can be used by researchers and industry analysts alike to evaluate different types of data routinely collected during PV field operations.

PV data collected in the field varies greatly in structure (i.e., timeseries and text records) and quality (i.e., completeness and consistency). The data available for analysis is frequently semi-structured. Furthermore, the level of detail collected between different owners/operators might vary. For example, some may capture a general start and end time for an associated event whereas others might include additional time details for different resolution activities. This diversity in data types and structures often leads to data being under-utilized due to the amount of manual processing required. To address these issues, *pvOps* provides a suite of data processing, cleaning, and visualization methods to leverage insights across a broad range of data types, including operations and maintenance records, production timeseries, and IV curves. The functions within *pvOps* enable users to better parse available data to understand patterns in outages and production losses.

1.1 User Guide

1.1.1 Installation

pvops is tested on Python versions 3.8, 3.9, 3.10, and 3.11 and depends on a variety of packages.

The latest release of pvops is accessible via PYPI using the following command line prompt:

```
$ pip install pvops
```

Alternatively, the package can be installed using github:

```
$ git clone https://github.com/sandialabs/pvOps.git
$ cd pvops
$ pip install .
```

NLTK data

Functions in the text package rely on the "punkt" dataset from the nltk package. After proper installation of pvops, run the commands:

```
>>> import nltk
>>> nltk.download('punkt')
>>> nltk.download('stopwords')
```

Those operating under a proxy may have difficulty with this installation. This stack exchange post may help.

1.1.2 Text Guide

Module Overview

This module aims to support the consistent extraction of key features in O&M data:

- timestamp information
- characteristic categorical information
- a concise synopsis of the issue for context

Implemented functions include those for filling in data gaps (text.preprocess submodule), machine learning analyses to fill in gaps in categorical information and to generate concise summary strings (text.classify submodule), functions to prepare data for natural language processing (text.nlp_utils submodule), and a visualization suite (text.visualize submodule).

An example implementation of all capabilities can be found in text_class_example.py for specifics, and tutorial_textmodule.ipynb for basics.

Text pre-processing

preprocess

These functions process the O&M data into concise, machine learning-ready documents. Additionally, there are options to extract dates from the text.

- *preprocessor()* acts as a wrapper function, utilizing the other preprocessing functions, which prepares the data for machine learning.
  - See `text_class_example.prep_data_for_ML` for an example.
- *preprocessor()* should be used with the keyword argument *extract_dates_only* = *True* if the primary interest is date extraction instead continuing to use the data for machine learning.
  - See `text_class_example.extract_dates` module for an example.

Text classification

classify

These functions process the O&M data to make an inference on the specified event descriptor.

- *classification_deployer()* is used to conduct supervised or unsupervised classification of text documents. This function conducts a grid search across the passed classifiers and hyperparameters.
  - The *supervised_classifier_defs()* and *unsupervised_classifier_defs()* functions return default values for conducting the grid search.
  - See `text_class_example.classify_supervised` or `text_class_example.classify_unsupervised` modules for an example.
- Once the model is built and selected, classification (for supervised ML) or clustering (for unsupervised ML) analysis can be conducted on the best model returned from the pipeline object.
  - See `text_class_example.predict_best_model` module for an example.

Utils

utils

These helper functions focus on performing exploratory or secondary processing activities for the O&M data.

- `pvops.text.nlp_utils.remap_attributes()` is used to reorganize an attribute column into a new set of labels.

NLP Utils

utils

These helper functions focus on processing in preparation for NLP activities.

- *summarize_text_data()* prints summarized contents of the O&M data.
- *Doc2VecModel* performs a gensim Doc2Vec transformation of the input documents to create embedded representations of the documents.
- *DataDensifier* is a data structure transformer which converts sparse data to dense data.
- *create_stopwords()* concatenates a list of stopwords using both words grabbed from nltk and user-specified words

Visualizations

These functions create visualizations to get a better understanding about your documents.

- *visualize_attribute_connectivity()* visualizes the connectivity of two attributes.
- *visualize_attribute_timeseries()* evaluates the density of an attribute over time.
- *visualize_cluster_entropy()* observes the performance of different text embeddings.
- *visualize_document_clusters()* visualizes popular words in clusters after a cluster analysis is ran.

- *visualize_word_frequency_plot()* visualizes word frequencies in the associated attribute column of O&M data.

1.1.3 Text2Time Guide

Module Overview

Aligning production data with O&M tickets is not a trivial task since intersection of dates and identification of anomalies depends on the nuances within the two datasets. This set of functions facilitate this data fusion. Key features include:

- conducting quality checks and controls on data.
- identification of overlapping periods between O&M and production data.
- generation of baseline values for production loss estimations.
- calculation of losses from production anomalies for specific time periods.

An example of usage can be found in tutorial_text2time_module.ipynb.

The text2time package can be broken down into three main components: *data pre-processing*, *utils*, and *visualizations*.

Data pre-processing

text2time.preprocess module

These functions pre-process user O&M and production data to prepare them for further analyses and visualizations.

- *om_date_convert()* and *prod_date_convert()* convert dates in string format to date-time objects in the O&M and production data respectively.
- *data_site_na()* handles missing site IDs in the user data. This function can be used for both O&M and production data.
- *om_datelogic_check()* detects and handles issues with the logic of the O&M date, specifically when the conclusion of an event occurs before it begins.
- *om_nadate_process()* and *prod_nadate_process()* detect and handle any missing time-stamps in the O&M and production data respectively.

Utils

text2time.utils module

These functions perform secondary calcuations on the O&M and production data to aid in data analyses and visualizations.

- `iec_calc()` calculates a comparison dataset for the production data based on an irradiance as calculated by IEC calculation.
- *summarize_overlaps()* summarizes the overlapping production and O&M data.
- *om_summary_stats()* summarizes statistics (e.g., event duration and month of occurrence) of O&M data.
- *overlapping_data()* trims the production and O&M data frames and only retain the data where both datasets overlap in time.
- *prod_anomalies()* detects and handles issues when the production data is input in cumulative format and unexpected dips show up in the data.

- *prod_quant()* calculates a comparison between the actual production data and a baseline (e.g. from a model from *timeseries models*).

Visualizations

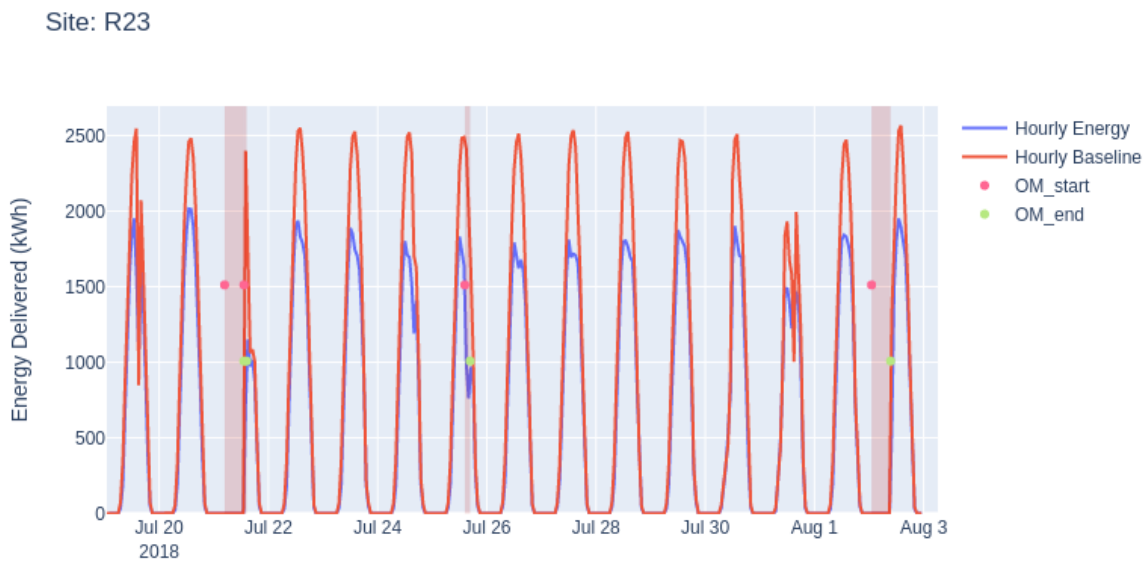
text2time.visualize module

These functions visualize the processed O&M and production data:

- *visualize_categorical_scatter()* generates categorical scatter plots of chosen variable based on specified category (e.g. site ID) for the O&M data.

- *visualize_counts()* generates a count plot of categories based on a chosen categorical variable column for the O&M data. If that variable is the user's site ID for every ticket, a plot for total count of events can be generated.

- *visualize_om_prod_overlap()* creates a visualization that overlays the O&M data on top of the coinciding production data.



Example Code

Load in OM data and convert dates to python date-time objects

```
>>> import pandas as pd
>>> import os
>>> from pvops.text2time import preprocess

>>> example_OMpath = os.path.join('example_data', 'example_om_data2.csv')
>>> om_data = pd.read_csv(example_OMpath, on_bad_lines='skip', engine='python')
>>> om_col_dict = {
... 'siteid': 'randid',
... 'datestart': 'date_start',
```

(continues on next page)

(continued from previous page)

```
... 'dateend': 'date_end',
... 'workID': 'WONumber',
... 'worktype': 'WOType',
... 'asset': 'Asset',
... 'eventdur': 'EventDur', #user's name choice for new column (Repair Duration)
... 'modatestart': 'MonthStart', #user's name choice for new column (Month when an event␣
↪begins)
... 'agedatestart': 'AgeStart'} #user's name choice for new column (Age of system when␣
↪event begins)
>>> om_data_converted = preprocess.om_date_convert(om_data, om_col_dict)
```

1.1.4 Timeseries Guide

Module Overview

These funcions provide processing and modelling capabilities for timeseries production data. Processing functions prepare data to train two types of expected energy models:

- AIT: additive interaction trained model, see Hopwood and Gunda [HG22] for more information.
- Linear: a high flexibility linear regression model.

Additionally, the ability to generate expected energy via IEC standards (iec 61724-1) is implemented in the `iec` module.

An example of usage can be found in *tutorial_timeseries_module.ipynb <https://github.com/sandialabs/pvOps/blob/master/tutorials/tutor*

Preprocess

- `pvops.timeseries.preprocess.prod_inverter_clipping_filter()` filters out production periods with inverter clipping. The core method was adopted from *pvlib/pvanalytics*.
- `pvops.timeseries.preprocess.normalize_production_by_capacity()` normalizes power by site capacity.
- `pvops.timeseries.preprocess.prod_irradiance_filter()` filters rows of production data frame according to performance and data quality. NOTE: this method is currently in development.
- `pvops.timeseries.preprocess.establish_solar_loc()` adds solar position data to production data using pvLib.

Models

- `pvops.timeseries.models.linear.modeller()` is a wrapper method used to model timeseries data using a linear model. This method gives multiple options for the learned model structure.
- `pvops.timeseries.models.AIT.AIT_calc()` Calculates expected energy using measured irradiance based on trained regression model from field data.
- `pvops.timeseries.models.iec.iec_calc()` calculates expected energy using measured irradiance based on IEC calculations.

Example Code

load in data and run some processing functions

1.1.5 IV Guide

Module Overview

These functions focus on current-voltage (IV) curve simulation and classification.

Note: To use the capabilites in this module, pvOps must be installed with the `iv` option: `pip install pvops[iv]`.

Tutorials that exemplify usage can be found at:

- tutorial_iv_classifier.ipynb.
- tutorial_iv_diode_extractor.ipynb.
- tutorial_iv_simulator.ipynb.

extractor

- *`extractor`* primarily features the *`BruteForceExtractor`* class, which extracts diode parameters from IV curves (even outdoor-collected).

physics_utils

`physics_utils` contains methods which aid the IV Simulator's physics-based calculations and the preprocessing pipeline's correction calculations.

- *`calculate_IVparams()`* calculates key parameters of an IV curve.
- *`smooth_curve()`* smooths IV curve using a polyfit.
- *`iv_cutoff()`* cuts off IV curve greater than a given voltage value.
- *`intersection()`* computes the intersection between two curves.
- *`T_to_tcell()`* calculates a cell temperature given ambient temperature via NREL weather-correction tools.
- *`bypass()`* limits voltage to above a minimum value.
- *`add_series()`* adds two IV curves in series.
- *`voltage_pts()`* provides voltage points for an IV curve.
- *`gt_correction()`* corrects IV trace using irradiance and temperature using one of three available options.

preprocess

preprocess contains the preprocessing function * *preprocess()* which corrects a set of data according to irradiance and temperature and normalizes the curves so they are comparable.

simulator

simulator holds the `IV Simulator` class which can simulate current-voltage (IV) curves under different environmental and fault conditions. There is also a utility function *create_df()* for building an IV curve dataframe from a set of parameters.

utils

utils holds the utility function *get_CEC_params()* which connects to the California Energy Commission (CEC) database hosted by pvLib for cell-level and module-level parameters.

timeseries_simulator

timeseries_simulator contains *IVTimeseriesGenerator*, a subclass of the IV Simulator, which allows users to specify time-based failure degradation patterns. The class *TimeseriesFailure* is used to define the time-based failures.

1.1.6 Abbreviations/Terminology

- AIT: Additive Interaction Model described in [HG22]
- CEC: California Energy Commission
- WS: wind speed
- Varr: Voltage array
- T: Average cell temperature
- Rsh_mult: Multiplier usually less than 1 to simulate a drop in RSH
- Rs_mult: Multiplier usually less than 1 to simulate a drop in RS
- Io_mult: Multiplier usually less than 1 to simulate a drop in IO
- Il_mult: Multiplier usually less than 1 to simulate a drop in IL
- nnsvth_mult: Multiplier usually less than 1 to simulate a drop in NNSVTH
- E: Irradiance
- Tc: Cell temp
- gt: (G - Irradiation and T - temperature)
- v_rbd: Reverse bias diode voltage
- v_oc: Open circuit voltage

1.2 pvOps Tutorials

Check out the tutorials below!

1.2.1 Text2Time Module Tutorial

This notebook demonstrates the use of pvops to analyze a combination of operations and maintenance (OM) and production data. The data will be processed and cleaned in preparation for an intersection analysis and subsequent visualizations.

Import modules

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import shutil
import sys
import os
```

[2]:
```
from pvops.text2time import visualize, utils, preprocess
from pvops.timeseries.models import linear, iec
```

1. Load and explore data

Define csv paths to OM, production, and meta data.

[3]:
```
example_OMpath = os.path.join('example_data', 'example_om_data2.csv')
example_prodpath = os.path.join('example_data', 'example_prod_data_cumE2.csv')
example_metapath = os.path.join('example_data', 'example_metadata2.csv')
```

Load in csv files as pandas DataFrames. `prod_data` contains energy production and irradiance data over time for potentially multiple sites. `om_data` contains operations and maintenance tickets. `metadata` contains information about the sites, such as location and DC size.

[4]:
```
prod_data = pd.read_csv(example_prodpath, on_bad_lines='skip', engine='python')
om_data = pd.read_csv(example_OMpath, on_bad_lines='skip', engine='python')
metadata = pd.read_csv(example_metapath, on_bad_lines='skip', engine='python')
```

Explore production data

[5]:
```
prod_data
```

[5]:
```
     randid              Date     Energy  Irradiance
0       R23               NaN     1000.0         NaN
1       R23    7/19/2018 1:00     1000.0         NaN
2       R23    7/19/2018 2:00        0.0         NaN
3       R23    7/19/2018 3:00        0.0         NaN
4       R23    7/19/2018 4:00     1000.0         NaN
...     ...                ...        ...         ...
1049    R27  10/13/2018 23:45  5289528.0         NaN
1050    R27   10/14/2018 0:00  5289528.0         NaN
1051    R27   10/14/2018 0:15  5289528.0         NaN
```

(continues on next page)

(continued from previous page)

```
1052    R27   10/14/2018 0:30  5289528.0         NaN
1053    R27   10/14/2018 0:45  5289528.0         NaN

[1054 rows x 4 columns]
```

Explore OM data

[6]:
```
om_data
```

[6]:
```
   randid     Asset        date_start          date_end  WONumber      WOType  \
0     NaN  Inverter    5/2/2018 12:00   5/17/2018 16:00       100  Corrective
1     R23  Facility   5/19/2018 15:44   5/19/2018 13:04       101  Preventive
2     R23  Facility    6/15/2018 6:46   6/15/2018 10:30       102  Corrective
3     R23  Facility   6/18/2018 11:20   6/18/2018 14:03       103  Corrective
4     R23  Facility    7/21/2018 4:45   7/21/2018 13:15       104  Vegetation
5     R23  Inverter   7/21/2018 13:16   7/21/2018 14:25       105  Corrective
6     R23  Inverter   7/25/2018 14:20   7/25/2018 16:40       106  Corrective
7     R23  Inverter    8/1/2018 11:45               NaN       107  Corrective
8     R23  Facility     8/2/2018 1:05     8/2/2018 9:28       108  Corrective
9     R27  Facility   9/14/2018 10:00   9/16/2018 16:00         1  corrective
10    R27  Facility   9/24/2018 10:00   9/16/2018 17:00         2  vegetation
11    R27     Other    9/19/2018 7:00  10/11/2018 20:00         3  corrective
12    R27  Facility  10/13/2018 12:00  10/13/2018 17:00         4  preventive
13    R27     other  10/14/2018 11:00               NaN         5  preventive

                                          GeneralDesc
0   Inverter 1.1 Contactor 7, Inverter 1.2 Contact...
1                Site offline due to grid disturbance
2                  Plant trip due to grid disturbance
3             Site trip due to cause grid disturbance
4                Site tripped due to grid disturbance
5       Inverter failed to start following plant trip
6     inverter offline due to high ambient temp fault
7                    Inverter major underperformance
8                   Site trip due to grid disturbance
9   hurricane florence outages/response. complete ...
10   Vegetation maintenance activities were performed
11  hurricane response. perform site inspection to...
12                         Monthly visual inspection
13                         Monthly visual inspection
```

Explore metadata

[7]:
```
metadata
```

[7]:
```
  randid  DC_Size_kW         COD  latitude  longitude
0    R23        2500  10/20/2013       -80        -35
1    R27         475  10/21/2017       -81        -36
```

2. Prepare data for analysis

Assigning dictionaries to connect pvOps variables with user's column names.

[8]:
```
#Format for dictionaries is {pvops variable: user-specific column names}
prod_col_dict = {'siteid': 'randid',
                 'timestamp': 'Date',
                 'energyprod': 'Energy',
                 'irradiance':'Irradiance',
                 'baseline': 'IEC_pstep', #user's name choice for new column (baseline␣
↪expected energy defined by user or calculated based on IEC)
                 'dcsize': 'dcsize', #user's name choice for new column (System DC-size,␣
↪extracted from meta-data)
                 'compared': 'Compared',#user's name choice for new column
                 'energy_pstep': 'Energy_pstep'} #user's name choice for new column

om_col_dict = {'siteid': 'randid',
               'datestart': 'date_start',
               'dateend': 'date_end',
               'workID': 'WONumber',
               'worktype': 'WOType',
               'asset': 'Asset',
               'eventdur': 'EventDur', #user's name choice for new column (Repair␣
↪Duration)
               'modatestart': 'MonthStart', #user's name choice for new column (Month␣
↪when an event begins)
               'agedatestart': 'AgeStart'} #user's name choice for new column (Age of␣
↪system when event begins)

metad_col_dict = {'siteid': 'randid',
                  'dcsize': 'DC_Size_kW',
                  'COD': 'COD'}
```

2.1 Convert date strings to date-time objects

O&M Data

[9]:
```
#Note:  NaNs are converted to NaTs
om_data_converted = preprocess.om_date_convert(om_data, om_col_dict)

print('---Original data types---')
print(om_data.dtypes)
print('\n---Post-processed data types---')
print(om_data_converted.dtypes)
print('\n---Converted data frame---')
om_data_converted.head()
```

```
---Original data types---
randid         object
Asset          object
date_start     object
date_end       object
```

(continues on next page)

(continued from previous page)

```
WONumber        int64
WOType         object
GeneralDesc    object
dtype: object

---Post-processed data types---
randid                 object
Asset                  object
date_start     datetime64[ns]
date_end       datetime64[ns]
WONumber                int64
WOType                 object
GeneralDesc            object
dtype: object

---Converted data frame---
```

[9]:
```
  randid     Asset          date_start            date_end  WONumber  \
0    NaN  Inverter 2018-05-02 12:00:00 2018-05-17 16:00:00       100
1    R23  Facility 2018-05-19 15:44:00 2018-05-19 13:04:00       101
2    R23  Facility 2018-06-15 06:46:00 2018-06-15 10:30:00       102
3    R23  Facility 2018-06-18 11:20:00 2018-06-18 14:03:00       103
4    R23  Facility 2018-07-21 04:45:00 2018-07-21 13:15:00       104

       WOType                                       GeneralDesc
0  Corrective  Inverter 1.1 Contactor 7, Inverter 1.2 Contact...
1  Preventive                Site offline due to grid disturbance
2  Corrective                  Plant trip due to grid disturbance
3  Corrective             Site trip due to cause grid disturbance
4  Vegetation                Site tripped due to grid disturbance
```

Production data

[10]:
```
prod_data_converted = preprocess.prod_date_convert(prod_data, prod_col_dict)

print('---Original data types---')
print(prod_data.dtypes)
print('\n---Post-processed data types---')
print(prod_data_converted.dtypes)
print('\n---Converted data frame---')
prod_data_converted.head()
```

```
---Original data types---
randid         object
Date           object
Energy        float64
Irradiance    float64
dtype: object

---Post-processed data types---
randid                object
```

(continues on next page)

(continued from previous page)

```
Date          datetime64[ns]
Energy               float64
Irradiance           float64
dtype: object

---Converted data frame---
```

[10]:
```
  randid                Date  Energy  Irradiance
0    R23                 NaT  1000.0         NaN
1    R23 2018-07-19 01:00:00  1000.0         NaN
2    R23 2018-07-19 02:00:00     0.0         NaN
3    R23 2018-07-19 03:00:00     0.0         NaN
4    R23 2018-07-19 04:00:00  1000.0         NaN
```

3. Handling data quality issues

3.1 Missing site-IDs

Drop rows where site-ID is NAN in OM-data (helpful when multiple sites are in O&M data frame and a NAN doesn't identify a specific site)

[11]:
```
om_data_sitena, addressed = preprocess.data_site_na(om_data_converted, om_col_dict)
addressed #printing row that was addressed
```

[11]:
```
  randid     Asset          date_start            date_end  WONumber  \
0    NaN  Inverter 2018-05-02 12:00:00 2018-05-17 16:00:00       100

       WOType                                        GeneralDesc
0  Corrective  Inverter 1.1 Contactor 7, Inverter 1.2 Contact...
```

Print post-processed data frame

[12]:
```
om_data_sitena.head()
```

[12]:
```
  randid     Asset          date_start            date_end  WONumber  \
1    R23  Facility 2018-05-19 15:44:00 2018-05-19 13:04:00       101
2    R23  Facility 2018-06-15 06:46:00 2018-06-15 10:30:00       102
3    R23  Facility 2018-06-18 11:20:00 2018-06-18 14:03:00       103
4    R23  Facility 2018-07-21 04:45:00 2018-07-21 13:15:00       104
5    R23  Inverter 2018-07-21 13:16:00 2018-07-21 14:25:00       105

       WOType                                    GeneralDesc
1  Preventive           Site offline due to grid disturbance
2  Corrective             Plant trip due to grid disturbance
3  Corrective        Site trip due to cause grid disturbance
4  Vegetation           Site tripped due to grid disturbance
5  Corrective  Inverter failed to start following plant trip
```

3.2 O&M Start-dates that occur after concluding date (inverted dates)

Addressing issue by swapping dates

[13]:
```
om_data_checked_s, addressed = preprocess.om_datelogic_check(om_data_sitena, om_col_dict,
↪ 'swap')
addressed
```

[13]:
```
   randid     Asset          date_start            date_end  WONumber  \
1     R23  Facility 2018-05-19 15:44:00 2018-05-19 13:04:00       101
10    R27  Facility 2018-09-24 10:00:00 2018-09-16 17:00:00         2

        WOType                                       GeneralDesc
1   Preventive                Site offline due to grid disturbance
10  vegetation  Vegetation maintenance activities were performed
```

Print post-processed data frame

[14]:
```
om_data_checked_s.head()
```

[14]:
```
  randid     Asset          date_start            date_end  WONumber  \
1    R23  Facility 2018-05-19 13:04:00 2018-05-19 15:44:00       101
2    R23  Facility 2018-06-15 06:46:00 2018-06-15 10:30:00       102
3    R23  Facility 2018-06-18 11:20:00 2018-06-18 14:03:00       103
4    R23  Facility 2018-07-21 04:45:00 2018-07-21 13:15:00       104
5    R23  Inverter 2018-07-21 13:16:00 2018-07-21 14:25:00       105

       WOType                                     GeneralDesc
1  Preventive           Site offline due to grid disturbance
2  Corrective             Plant trip due to grid disturbance
3  Corrective        Site trip due to cause grid disturbance
4  Vegetation           Site tripped due to grid disturbance
5  Corrective  Inverter failed to start following plant trip
```

Addressing issue by dropping rows instead of swapping dates

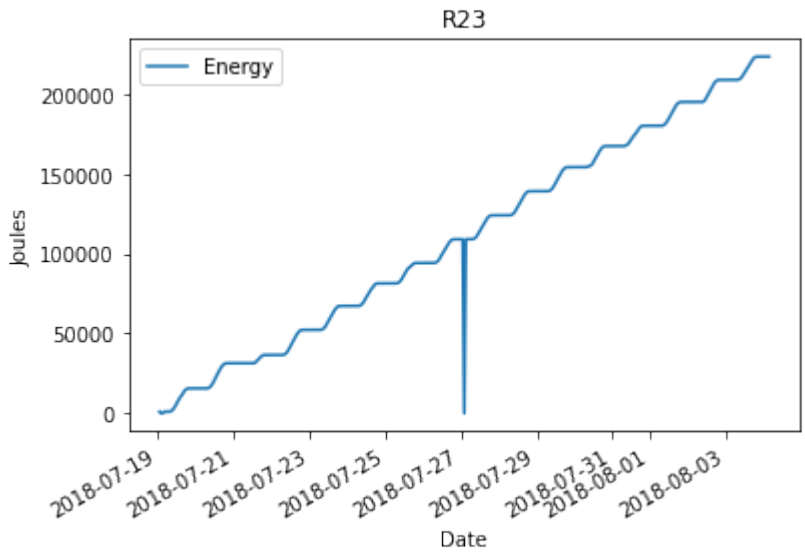
[15]:
```
om_data_checked_d, date_error = preprocess.om_datelogic_check(om_data_sitena, om_col_
↪dict, 'drop')
om_data_checked_d.head()
```

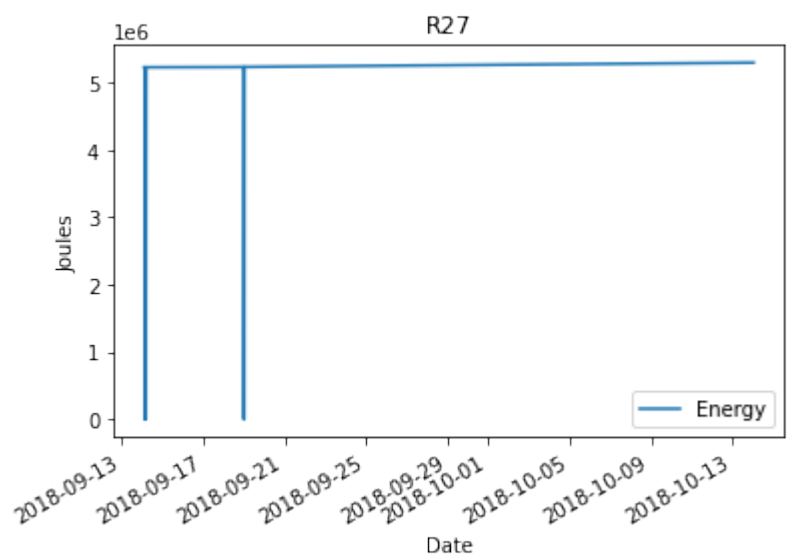
[15]:
```
  randid     Asset          date_start            date_end  WONumber  \
2    R23  Facility 2018-06-15 06:46:00 2018-06-15 10:30:00       102
3    R23  Facility 2018-06-18 11:20:00 2018-06-18 14:03:00       103
4    R23  Facility 2018-07-21 04:45:00 2018-07-21 13:15:00       104
5    R23  Inverter 2018-07-21 13:16:00 2018-07-21 14:25:00       105
6    R23  Inverter 2018-07-25 14:20:00 2018-07-25 16:40:00       106

       WOType                                      GeneralDesc
2  Corrective               Plant trip due to grid disturbance
3  Corrective          Site trip due to cause grid disturbance
4  Vegetation             Site tripped due to grid disturbance
5  Corrective    Inverter failed to start following plant trip
6  Corrective  inverter offline due to high ambient temp fault
```

3.3 Unexpected drops in energy delivered (when collected on cumulative basis)

Visualize pre-processed data

[16]:
```
plotvar = 'Energy'
for sid in prod_data_converted.loc[:, 'randid'].unique():
    mask = prod_data_converted.loc[:, 'randid']== sid
    prod_data_converted.loc[mask].plot(x='Date',y=plotvar, title=sid)
    plt.ylabel('Joules')
```





Addressing issue by forward-filling, which propagates last valid observation forward.

[17]:
```
prod_data_anom, addressed = utils.prod_anomalies(prod_data_converted, prod_col_dict, 1.0,
↪ np.nan, ffill=True)
addressed
```

[17]:
```
    randid                Date  Energy  Irradiance
```

(continues on next page)

(continued from previous page)

```
2      R23 2018-07-19 02:00:00     0.0         NaN
3      R23 2018-07-19 03:00:00     0.0         NaN
194    R23 2018-07-27 02:00:00     0.0         NaN
395    R27 2018-09-14 04:00:00     0.0    0.616897
397    R27 2018-09-14 04:30:00     0.0    0.306548
860    R27 2018-09-19 00:15:00     0.0    0.000000
```

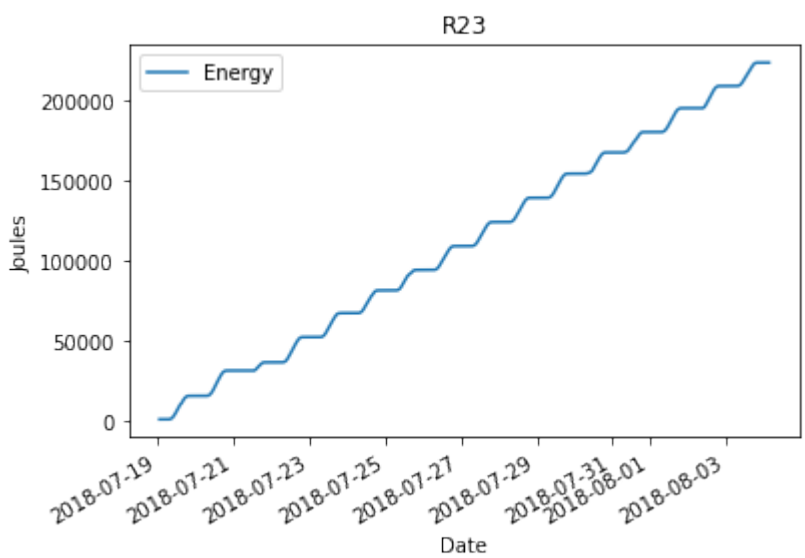
Print post-processed data frame

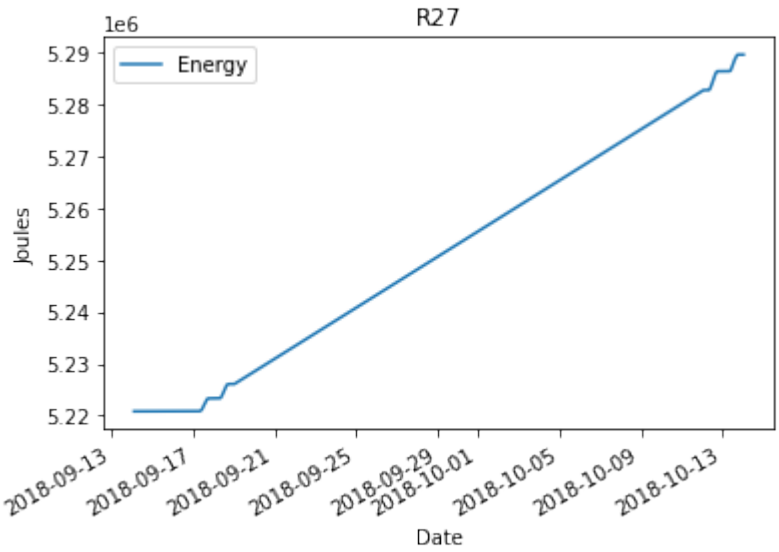
[18]:
```
prod_data_anom
```

[18]:
```
     randid                Date     Energy  Irradiance
0       R23                 NaT     1000.0         NaN
1       R23 2018-07-19 01:00:00     1000.0         NaN
2       R23 2018-07-19 02:00:00     1000.0         NaN
3       R23 2018-07-19 03:00:00     1000.0         NaN
4       R23 2018-07-19 04:00:00     1000.0         NaN
...     ...                  ...        ...         ...
1049    R27 2018-10-13 23:45:00  5289528.0         NaN
1050    R27 2018-10-14 00:00:00  5289528.0         NaN
1051    R27 2018-10-14 00:15:00  5289528.0         NaN
1052    R27 2018-10-14 00:30:00  5289528.0         NaN
1053    R27 2018-10-14 00:45:00  5289528.0         NaN

[1054 rows x 4 columns]
```

Quick visualization of post-processed data

[19]:
```
plotvar = 'Energy'
for sid in prod_data_anom.loc[:, 'randid'].unique():
    mask = prod_data_anom.loc[:, 'randid']== sid
    prod_data_anom.loc[mask].plot(x='Date',y=plotvar, title=sid)
    plt.ylabel('Joules')
```





3.4 Missing time-stamps in data

Production Data

Dropping rows with missing time-stamps

[20]:
```
prod_data_datena_d, addressed = preprocess.prod_nadate_process(prod_data_anom, prod_col_
↪dict, pnadrop=True)
addressed
```

[20]:
```
    randid Date     Energy  Irradiance
0      R23  NaT     1000.0         NaN
388    R27  NaT  5220831.0    0.095835
```

Print post-processed data frame

[21]:
```
prod_data_datena_d
```

[21]:
```
     randid                Date     Energy  Irradiance
1       R23 2018-07-19 01:00:00     1000.0         NaN
2       R23 2018-07-19 02:00:00     1000.0         NaN
3       R23 2018-07-19 03:00:00     1000.0         NaN
4       R23 2018-07-19 04:00:00     1000.0         NaN
5       R23 2018-07-19 05:00:00     1000.0         NaN
...     ...                  ...        ...         ...
1049    R27 2018-10-13 23:45:00  5289528.0         NaN
1050    R27 2018-10-14 00:00:00  5289528.0         NaN
1051    R27 2018-10-14 00:15:00  5289528.0         NaN
1052    R27 2018-10-14 00:30:00  5289528.0         NaN
1053    R27 2018-10-14 00:45:00  5289528.0         NaN

[1052 rows x 4 columns]
```

Identifying rows with missing time-stamps but not dropping them

[22]:
```
prod_data_datena_id, addressed = preprocess.prod_nadate_process(prod_data_anom, prod_col_
↪dict, pnadrop=False)
prod_data_datena_id
```

[22]:
```
     randid                Date     Energy  Irradiance
0       R23                 NaT     1000.0         NaN
1       R23 2018-07-19 01:00:00     1000.0         NaN
2       R23 2018-07-19 02:00:00     1000.0         NaN
3       R23 2018-07-19 03:00:00     1000.0         NaN
4       R23 2018-07-19 04:00:00     1000.0         NaN
...     ...                  ...        ...         ...
1049    R27 2018-10-13 23:45:00  5289528.0         NaN
1050    R27 2018-10-14 00:00:00  5289528.0         NaN
1051    R27 2018-10-14 00:15:00  5289528.0         NaN
1052    R27 2018-10-14 00:30:00  5289528.0         NaN
1053    R27 2018-10-14 00:45:00  5289528.0         NaN

[1054 rows x 4 columns]
```

O&M Data

Dropping rows when end-date of an event is missing

[23]:
```
om_data_datena_d, addressed = preprocess.om_nadate_process(om_data_checked_s, om_col_
↪dict, om_dendflag='drop')
addressed
```

[23]:
```
   randid     Asset          date_start date_end  WONumber      WOType  \
7     R23  Inverter 2018-08-01 11:45:00      NaT       107  Corrective
13    R27     other 2018-10-14 11:00:00      NaT         5  preventive

                         GeneralDesc
7   Inverter major underperformance
13         Monthly visual inspection
```

Print post-processed data frame

[24]:
```
om_data_datena_d
```

[24]:
```
   randid     Asset          date_start            date_end  WONumber  \
1     R23  Facility 2018-05-19 13:04:00 2018-05-19 15:44:00       101
2     R23  Facility 2018-06-15 06:46:00 2018-06-15 10:30:00       102
3     R23  Facility 2018-06-18 11:20:00 2018-06-18 14:03:00       103
4     R23  Facility 2018-07-21 04:45:00 2018-07-21 13:15:00       104
5     R23  Inverter 2018-07-21 13:16:00 2018-07-21 14:25:00       105
6     R23  Inverter 2018-07-25 14:20:00 2018-07-25 16:40:00       106
8     R23  Facility 2018-08-02 01:05:00 2018-08-02 09:28:00       108
9     R27  Facility 2018-09-14 10:00:00 2018-09-16 16:00:00         1
10    R27  Facility 2018-05-19 13:04:00 2018-05-19 15:44:00         2
11    R27     Other 2018-09-19 07:00:00 2018-10-11 20:00:00         3
12    R27  Facility 2018-10-13 12:00:00 2018-10-13 17:00:00         4

        WOType                                        GeneralDesc
```

(continues on next page)

(continued from previous page)

```
1   Preventive              Site offline due to grid disturbance
2   Corrective                Plant trip due to grid disturbance
3   Corrective           Site trip due to cause grid disturbance
4   Vegetation              Site tripped due to grid disturbance
5   Corrective      Inverter failed to start following plant trip
6   Corrective    inverter offline due to high ambient temp fault
8   Corrective                 Site trip due to grid disturbance
9   corrective  hurricane florence outages/response. complete ...
10  vegetation   Vegetation maintenance activities were performed
11  corrective  hurricane response. perform site inspection to...
12  preventive                         Monthly visual inspection
```

Rather than dropping rows, assigning "today's" time-stamp for missing end-dates to consider an open ticket

[25]:
```
om_data_datena_t, addressed = preprocess.om_nadate_process(om_data_checked_s, om_col_
↪dict, om_dendflag='today')
om_data_datena_t
```

[25]:
```
   randid     Asset          date_start            date_end  WONumber  \
1     R23  Facility 2018-05-19 13:04:00 2018-05-19 15:44:00       101
2     R23  Facility 2018-06-15 06:46:00 2018-06-15 10:30:00       102
3     R23  Facility 2018-06-18 11:20:00 2018-06-18 14:03:00       103
4     R23  Facility 2018-07-21 04:45:00 2018-07-21 13:15:00       104
5     R23  Inverter 2018-07-21 13:16:00 2018-07-21 14:25:00       105
6     R23  Inverter 2018-07-25 14:20:00 2018-07-25 16:40:00       106
7     R23  Inverter 2018-08-01 11:45:00 2023-01-12 12:26:26       107
8     R23  Facility 2018-08-02 01:05:00 2018-08-02 09:28:00       108
9     R27  Facility 2018-09-14 10:00:00 2018-09-16 16:00:00         1
10    R27  Facility 2018-05-19 13:04:00 2018-05-19 15:44:00         2
11    R27     Other 2018-09-19 07:00:00 2018-10-11 20:00:00         3
12    R27  Facility 2018-10-13 12:00:00 2018-10-13 17:00:00         4
13    R27     other 2018-10-14 11:00:00 2023-01-12 12:26:26         5

        WOType                                       GeneralDesc
1   Preventive              Site offline due to grid disturbance
2   Corrective                Plant trip due to grid disturbance
3   Corrective           Site trip due to cause grid disturbance
4   Vegetation              Site tripped due to grid disturbance
5   Corrective      Inverter failed to start following plant trip
6   Corrective    inverter offline due to high ambient temp fault
7   Corrective                  Inverter major underperformance
8   Corrective                 Site trip due to grid disturbance
9   corrective  hurricane florence outages/response. complete ...
10  vegetation   Vegetation maintenance activities were performed
11  corrective  hurricane response. perform site inspection to...
12  preventive                         Monthly visual inspection
13  preventive                         Monthly visual inspection
```

4. Pre-visualizing preparation

4.1 Print out overview of the overlap of OM and production data

`prod_summary` indicates how many time stamps overlapped with OM data versus the total number of time stamps, broken down by site. `om_data` indicates the lower and upper time bounds on OM data and the number of events, broken down by site.

[26]:
```
prod_summary, om_summary = utils.summarize_overlaps(prod_data_datena_d, om_data_datena_t,
↪ prod_col_dict, om_col_dict)
```

Production

[27]:
```
prod_summary
```

[27]:
```
        Actual # Time Stamps  Max # Time Stamps
randid
R23                      387                387
R27                      665                665
```

O&M

[28]:
```
om_summary
```

[28]:
```
       Earliest Event Start    Latest Event End  Total Events
randid
R23     2018-07-21 04:45:00 2023-01-12 12:26:26             5
R27     2018-09-14 10:00:00 2018-10-13 17:00:00             3
```

4.2 Extract overlapping data

[29]:
```
prod_data_clean, om_data_clean = utils.overlapping_data(prod_data_datena_d, om_data_
↪datena_d, prod_col_dict, om_col_dict)
```

Print post-processed production data frame

[30]:
```
prod_data_clean
```

[30]:
```
     randid                Date     Energy  Irradiance
0       R23 2018-07-19 01:00:00     1000.0         NaN
1       R23 2018-07-19 02:00:00     1000.0         NaN
2       R23 2018-07-19 03:00:00     1000.0         NaN
3       R23 2018-07-19 04:00:00     1000.0         NaN
4       R23 2018-07-19 05:00:00     1000.0         NaN
...     ...                 ...        ...         ...
1015    R27 2018-10-13 22:45:00  5289528.0         0.0
1016    R27 2018-10-13 23:00:00  5289528.0         0.0
1017    R27 2018-10-13 23:15:00  5289528.0         NaN
1018    R27 2018-10-13 23:30:00  5289528.0         NaN
```

(continues on next page)

(continued from previous page)

```
1019    R27 2018-10-13 23:45:00  5289528.0         NaN

[1020 rows x 4 columns]
```

Print post-processed O&M data frame

[31]:
```
om_data_clean
```

[31]:
```
  randid     Asset          date_start            date_end  WONumber  \
0    R23  Facility 2018-07-21 04:45:00 2018-07-21 13:15:00       104
1    R23  Inverter 2018-07-21 13:16:00 2018-07-21 14:25:00       105
2    R23  Inverter 2018-07-25 14:20:00 2018-07-25 16:40:00       106
3    R23  Facility 2018-08-02 01:05:00 2018-08-02 09:28:00       108
4    R27  Facility 2018-09-14 10:00:00 2018-09-16 16:00:00         1
5    R27     Other 2018-09-19 07:00:00 2018-10-11 20:00:00         3
6    R27  Facility 2018-10-13 12:00:00 2018-10-13 17:00:00         4

       WOType                                       GeneralDesc
0  Vegetation                Site tripped due to grid disturbance
1  Corrective      Inverter failed to start following plant trip
2  Corrective    inverter offline due to high ambient temp fault
3  Corrective                   Site trip due to grid disturbance
4  corrective  hurricane florence outages/response. complete ...
5  corrective  hurricane response. perform site inspection to...
6  preventive                          Monthly visual inspection
```

4.3 Calculate reference production data using IEC standards

[32]:
```
prod_data_clean_iec = iec.iec_calc(prod_data_clean, prod_col_dict, metadata, metad_col_
↪dict, gi_ref=1000.)
```

Expected energy is calculated based on irradiance information and shows up as a new column in the production data frame.

[33]:
```
prod_data_clean_iec.head(n=15)
```

[33]:
```
   randid                Date     Energy  Irradiance   IEC_pstep
0     R23 2018-07-19 01:00:00   1000.000         NaN         NaN
1     R23 2018-07-19 02:00:00   1000.000         NaN         NaN
2     R23 2018-07-19 03:00:00   1000.000         NaN         NaN
3     R23 2018-07-19 04:00:00   1000.000         NaN         NaN
4     R23 2018-07-19 05:00:00   1000.000         NaN         NaN
5     R23 2018-07-19 06:00:00   1000.000         NaN         NaN
6     R23 2018-07-19 07:00:00   1032.712     28.6245    71.56125
7     R23 2018-07-19 08:00:00   1217.521    136.8305   342.07625
8     R23 2018-07-19 09:00:00   1889.859    347.5645   868.91125
9     R23 2018-07-19 10:00:00   3073.485    565.9015  1414.75375
10    R23 2018-07-19 11:00:00   4662.416    754.6965  1886.74125
11    R23 2018-07-19 12:00:00   6518.864    896.4945  2241.23625
12    R23 2018-07-19 13:00:00   8469.309    984.3710  2460.92750
13    R23 2018-07-19 14:00:00  10059.862   1018.6565  2546.64125
14    R23 2018-07-19 15:00:00  11122.754    339.1815   847.95375
```

4.4 Calculating a comparison of production data relative to baseline

Calculate as a ratio (actual/baseline)

[34]:
```
prod_data_quant = utils.prod_quant(prod_data_clean_iec, prod_col_dict, comp_type='norm',␣
↪ecumu=True)
prod_data_quant.head(10)
```

[34]:
```
  randid                Date    Energy  Irradiance   IEC_pstep  Energy_pstep  \
0    R23 2018-07-19 01:00:00  1000.000         NaN         NaN           NaN
1    R23 2018-07-19 02:00:00  1000.000         NaN         NaN         0.000
2    R23 2018-07-19 03:00:00  1000.000         NaN         NaN         0.000
3    R23 2018-07-19 04:00:00  1000.000         NaN         NaN         0.000
4    R23 2018-07-19 05:00:00  1000.000         NaN         NaN         0.000
5    R23 2018-07-19 06:00:00  1000.000         NaN         NaN         0.000
6    R23 2018-07-19 07:00:00  1032.712     28.6245    71.56125        32.712
7    R23 2018-07-19 08:00:00  1217.521    136.8305   342.07625       184.809
8    R23 2018-07-19 09:00:00  1889.859    347.5645   868.91125       672.338
9    R23 2018-07-19 10:00:00  3073.485    565.9015  1414.75375      1183.626

   Compared
0       NaN
1       NaN
2       NaN
3       NaN
4       NaN
5       NaN
6  0.457119
7  0.540257
8  0.773771
9  0.836630
```

Calculate as a difference (baseline-actual)

[35]:
```
prod_data_quant = utils.prod_quant(prod_data_clean_iec, prod_col_dict, comp_type='diff',␣
↪ecumu=True)
prod_data_quant.head(10)
```

[35]:
```
  randid                Date    Energy  Irradiance   IEC_pstep  Energy_pstep  \
0    R23 2018-07-19 01:00:00  1000.000         NaN         NaN           NaN
1    R23 2018-07-19 02:00:00  1000.000         NaN         NaN         0.000
2    R23 2018-07-19 03:00:00  1000.000         NaN         NaN         0.000
3    R23 2018-07-19 04:00:00  1000.000         NaN         NaN         0.000
4    R23 2018-07-19 05:00:00  1000.000         NaN         NaN         0.000
5    R23 2018-07-19 06:00:00  1000.000         NaN         NaN         0.000
6    R23 2018-07-19 07:00:00  1032.712     28.6245    71.56125        32.712
7    R23 2018-07-19 08:00:00  1217.521    136.8305   342.07625       184.809
8    R23 2018-07-19 09:00:00  1889.859    347.5645   868.91125       672.338
9    R23 2018-07-19 10:00:00  3073.485    565.9015  1414.75375      1183.626

    Compared
0        NaN
1        NaN
2        NaN
```

(continues on next page)

(continued from previous page)

```
3        NaN
4        NaN
5        NaN
6   38.84925
7  157.26725
8  196.57325
9  231.12775
```

5. Visualizations

5.1 Visualizing overlapping production and O&M data

Making directories to store generated visualizations.

[36]:
```
#User should modify paths as needed
main_fldr = 'analysis'
if os.path.isdir(main_fldr):
    shutil.rmtree(main_fldr)
prod_fldr = os.path.join(main_fldr, 'perf_plots')
site_fldr = os.path.join(main_fldr, 'site_plots')
os.makedirs(prod_fldr)
os.makedirs(site_fldr)
```

Making visualizations

[37]:
```
figs = visualize.visualize_om_prod_overlap(prod_data_quant, om_data_clean, prod_col_dict,
↪ om_col_dict, prod_fldr=prod_fldr, e_cumu=True, be_cumu=False, samp_freq='H', pshift=0.
↪0, baselineflag=True)
```

Display figure handles of overlapping data ("figs" contains one figure per site in overlapping data frames)

[38]:
```
for i in range(len(figs)):
    figs[i].show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json

Data type cannot be displayed: application/vnd.plotly.v1+json

5.2 Calculate additional O&M metrics and generate relevant plots

Calculating individual event duration and age of system at time of event occurence

[39]:
```
om_data_update = utils.om_summary_stats(om_data_clean, metadata, om_col_dict, metad_col_
↪dict)
om_data_update.head()
```

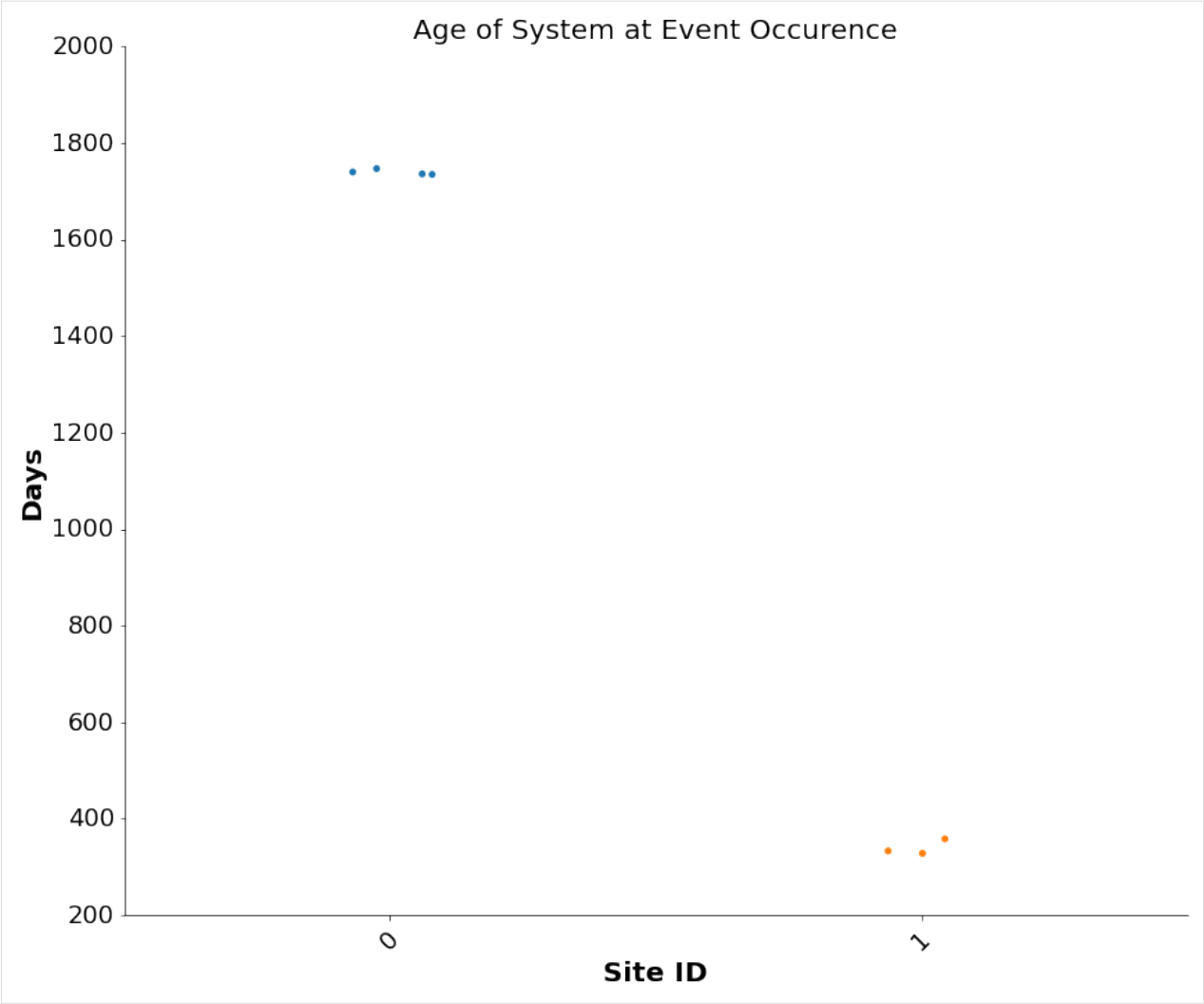
[39]:
```
  randid     Asset          date_start            date_end  WONumber  \
0    R23  Facility 2018-07-21 04:45:00 2018-07-21 13:15:00       104
1    R23  Inverter 2018-07-21 13:16:00 2018-07-21 14:25:00       105
2    R23  Inverter 2018-07-25 14:20:00 2018-07-25 16:40:00       106
3    R23  Facility 2018-08-02 01:05:00 2018-08-02 09:28:00       108
4    R27  Facility 2018-09-14 10:00:00 2018-09-16 16:00:00         1

       WOType                                       GeneralDesc  EventDur  \
0  Vegetation                Site tripped due to grid disturbance  8.500000
1  Corrective      Inverter failed to start following plant trip  1.150000
2  Corrective    inverter offline due to high ambient temp fault  2.333333
3  Corrective                   Site trip due to grid disturbance  8.383333
4  corrective  hurricane florence outages/response. complete ...  6.000000

   MonthStart        COD  AgeStart
0           7 2013-10-20      1735
1           7 2013-10-20      1736
2           7 2013-10-20      1740
3           8 2013-10-20      1747
4           9 2017-10-21       328
```

Setting Seaborn fig and font settings (inputs to count_fig and catscat_fig below)

[40]:
```
my_figsize = (12,10)
my_fontsize = 20
my_savedpi = 300
fig_sets = {'figsize': my_figsize,
            'fontsize': my_fontsize
            }
```

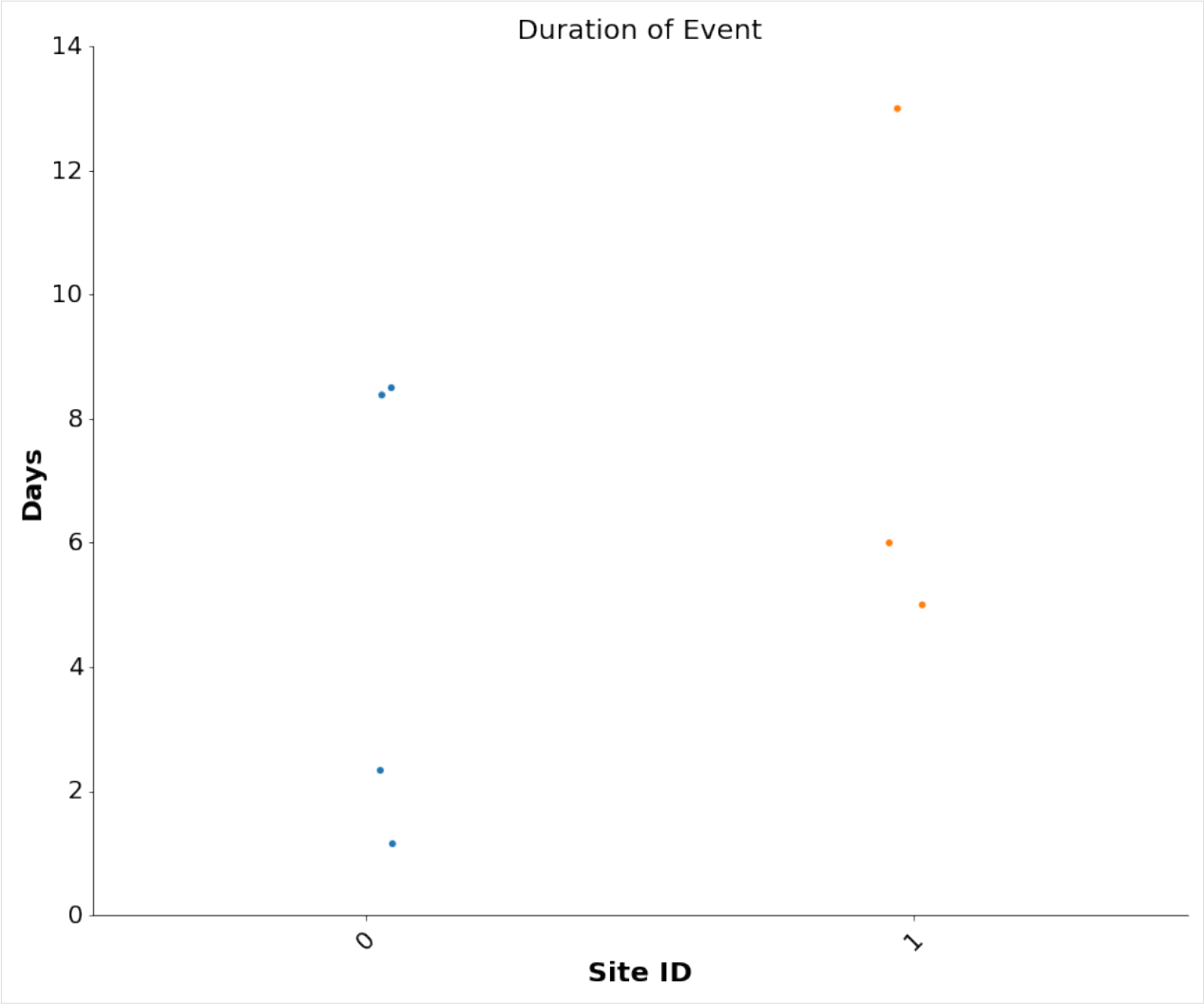
Creating scatter-plot of system age at beginning of each event, per site

[41]:
```
cat_varx = om_col_dict['siteid']
cat_vary= om_col_dict['agedatestart']
sv_nm = 'system_age.png'
myfig = visualize.visualize_categorical_scatter(om_data_update, om_col_dict, cat_varx,␣
↪cat_vary, fig_sets)
```



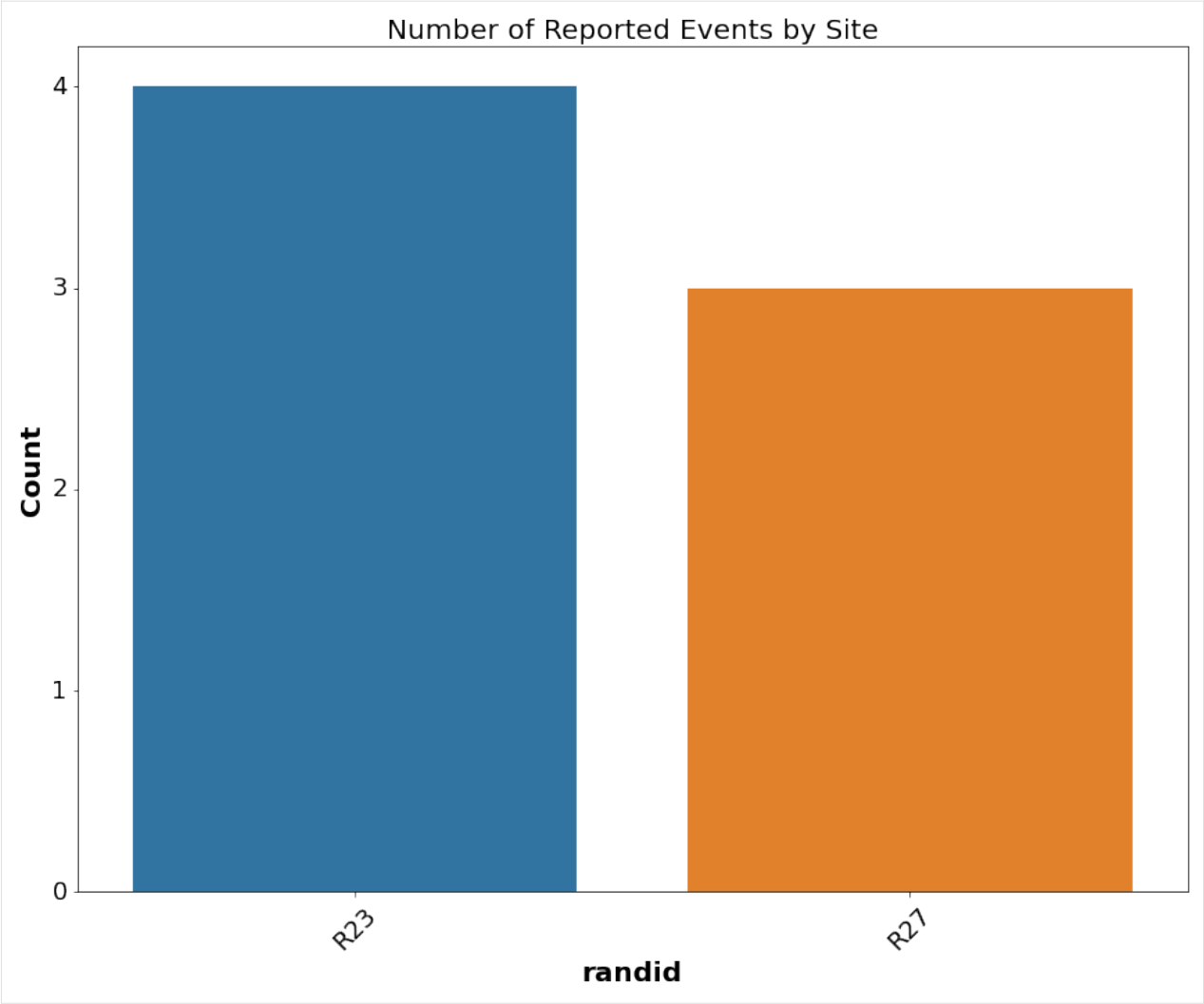
Creating scatter-plot of each event-duration, per site

[42]:
```
cat_varx = om_col_dict['siteid']
cat_vary= om_col_dict['eventdur']
sv_nm = 'event_dur.png'
myfig = visualize.visualize_categorical_scatter(om_data_update, om_col_dict, cat_varx,␣
↪cat_vary, fig_sets)
```



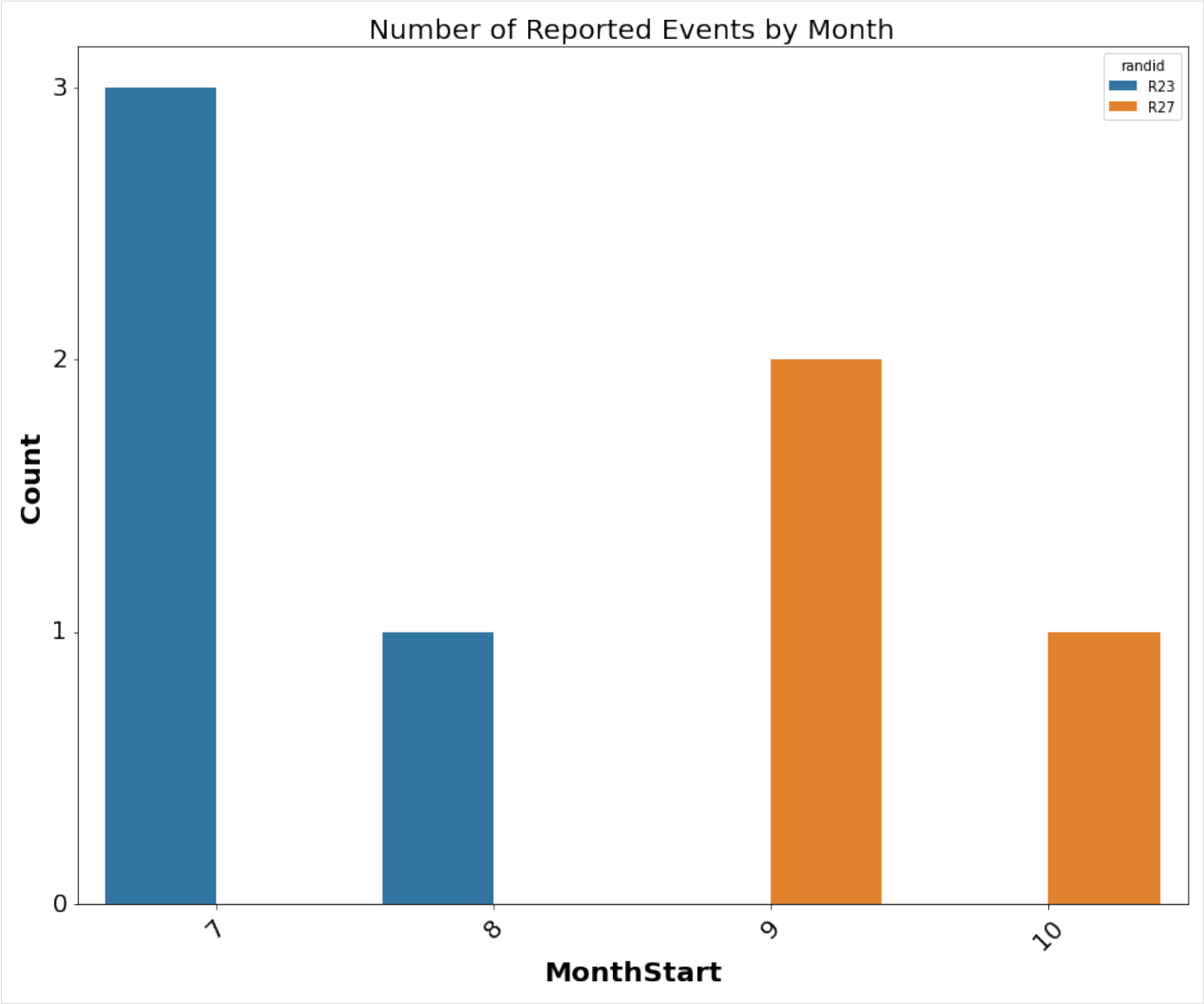
Count-plot of # of events, per site

[43]:
```
count_var = om_col_dict['siteid']
sv_nm = 'event_count_per_site.png'
myfig = visualize.visualize_counts(om_data_update, om_col_dict, count_var, fig_sets)
```



Count-plot of # of events, per month

```
[44]: count_var = om_col_dict['modatestart']
      sv_nm = 'event_count_per_month.png'
      myfig = visualize.visualize_counts(om_data_update, om_col_dict, count_var, fig_sets)
```



1.2.2 Text Module Tutorial

[56]:
```
import pandas as pd
import matplotlib.pyplot as plt

from pvops.text import utils
import text_class_example
```

Problem statements:

1. Text Preprocessing

Process the documents into concise, machine learning-ready documents. Additionally, extract dates from the text.

2. Text Classification

The written tickets are used to make an inference on the specified event descriptor.

Text processing

Import text data

[57]:
```
folder = 'example_data//'
filename = 'example_ML_ticket_data.csv'
df = pd.read_csv(folder+filename)
df.head(n=3)
```

[57]:
```
   Date_EventStart    Date_EventEnd     Asset  \
0   8/16/2018 9:00  8/22/2018 17:00  Combiner
1  9/17/2018 18:25   9/18/2018 9:50       Pad
2   8/26/2019 9:00  11/5/2019 17:00  Facility

                                     CompletionDesc                  Cause  \
0  cb 1.18 was found to have contactor issue woul...       0000 - Unknown.
1                  self resolved. techdispatched: no  004 - Under voltage.
2  all module rows washed, waiting for final repo...         0000 - Unknown

        ImpactLevel  randid
0  Underperformance      38
1  Underperformance      46
2  Underperformance      62
```

Establish settings

Specify column names which will be used in this pipeline.

[58]:
```
DATA_COLUMN = "CompletionDesc"   # Contains document
LABEL_COLUMN = "Asset"           # Establish event descriptor which will be inferenced␣
↪by classifiers
DATE_COLUMN = 'Date_EventStart'  # Date of ticket (start date, end date; any reflective␣
↪date will do), used in date extracting pipeline to replace information not specified␣
↪in ticket
```

Step 0: If needed, map raw labels to a cleaner set of labels

[59]:
```
asset_remap_filename = 'remappings_asset.csv'
REMAPPING_COL_FROM = 'in'
REMAPPING_COL_TO = 'out_'
remapping_df = pd.read_csv(folder+asset_remap_filename)
```

[60]:
```
remapping_col_dict = {
    'attribute_col': LABEL_COLUMN,
    'remapping_col_from': REMAPPING_COL_FROM,
    'remapping_col_to': REMAPPING_COL_TO
}

df_remapped_assets = utils.remap_attributes(df.iloc[30:].copy(), remapping_df.iloc[20:].
↪copy(), remapping_col_dict, allow_missing_mappings=True)

df = df_remapped_assets
```

[61]:
```
df[LABEL_COLUMN].value_counts()
```

[61]:
```
Asset
inverter                 26
facility                 24
tracker                   6
combiner                  4
substation                2
other                     2
transformer               1
ground-mount pv system    1
energy storage            1
energy meter              1
met station               1
pyranometer               1
Name: count, dtype: int64
```

Step 1: Establish example instance and render preliminary information about the tickets

[62]:
```
# Establish the class object (found in text_class_example.py)
print(df[LABEL_COLUMN].value_counts())

e = text_class_example.Example(df, LABEL_COLUMN)
e.summarize_text_data(DATA_COLUMN)
```

```
Asset
inverter                 26
facility                 24
tracker                   6
combiner                  4
substation                2
other                     2
transformer               1
```

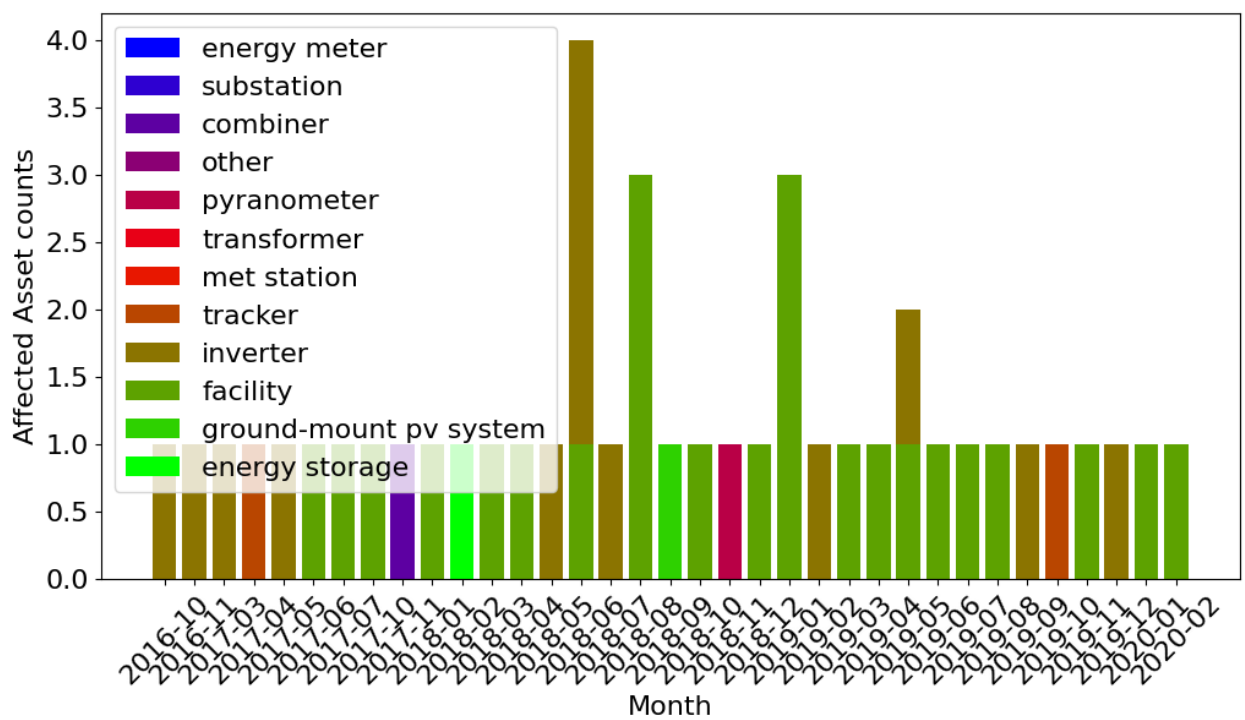
(continues on next page)

(continued from previous page)

```
ground-mount pv system     1
energy storage             1
energy meter               1
met station                1
pyranometer                1
Name: count, dtype: int64
DETAILS
  70 samples
  0 invalid documents
  29.16 words per sample on average
  Number of unique words 881
  2041.00 total words
```

Visualize timeseries of ticket publications

[63]:
```
fig = e.visualize_attribute_timeseries(DATE_COLUMN)
plt.show()
```



Functionality 1.1: Extract dates

[64]:
```
# Extract date from ticket, if any. This framework is not 100% correct.
dates_df = e.extract_dates(DATA_COLUMN, DATE_COLUMN, SAVE_DATE_COLUMN='ExtractedDates')
dates_df
```

[64]:
```
                                       CompletionDesc  \
0   8/39/19 inverter was faulted with lp15 (low pr...
1   11,july 2018 -upon arrival w-a6-2, inverter is...
2   arrived site checked into c4. i was able to pi...
3         c4 closed site remotely. techdispatched: no
4   inspection troubleshooting malfunctioning trac...
..                                                ...
65  cleared cleared alert however psi is -3 invert...
66             c4 closed remotely. techdispatched: no
67  pure power fixed damaged source circuits did f...
68  checked network connection to rm-1 didn't see ...
69  utility outage from 6/5 7am through 6/8 5:30pm...

                                       ExtractedDates
0                                [2019-08-17 07:35:00]
1   [2018-07-11 18:55:00, 2018-06-02 18:55:00, 201...
2                                [2020-05-26 14:45:00]
3                                                   []
4                                                   []
..                                                ...
65                               [2016-11-03 09:28:00]
66                                                  []
67         [2019-04-16 09:00:00, 2019-03-16 15:15:00]
68                                                  []
69         [2017-06-05 07:17:00, 2017-06-08 17:30:00]

[70 rows x 2 columns]
```

Functionality 1.2: Preprocess data for the Machine Learning classification

[65]:
```
preprocessed_df = e.prep_data_for_ML(DATA_COLUMN, DATE_COLUMN)
preprocessed_df
```

[65]:
```
                                       CompletionDesc  \
0   either reboot datalogger worked, issue resolve...
1                                . techdispatched: no
2                inverter resolved. techdispatched: no
3   10/2/19 e-1, row 51, e1-3-51-1. tracker tracki...
4   confirmed that cb 1.1.6 was turned off. verifi...
..                                                ...
59             c4 closed remotely. techdispatched: no
60  switchgear breaker for 2.6 was tripped. breake...
61                               . techdispatched: no
62                      resolved.. techdispatched: no
63  8/39/19 inverter was faulted with lp15 (low pr...
```

(continues on next page)

(continued from previous page)

```
                                            CleanDesc
0   either reboot datalogger worked issue resolved...
1                                     techdispatched
2                   inverter resolved techdispatched
3   row tracker tracking wrong lubed gear boxes tr...
4   confirmed cb turned verified voltage array tur...
..                                                ...
59                     closed remotely techdispatched
60  switchgear breaker tripped breaker inverter tr...
61                                    techdispatched
62                            resolved techdispatched
63  inverter faulted lp low pressure inverter show...

[64 rows x 2 columns]
```

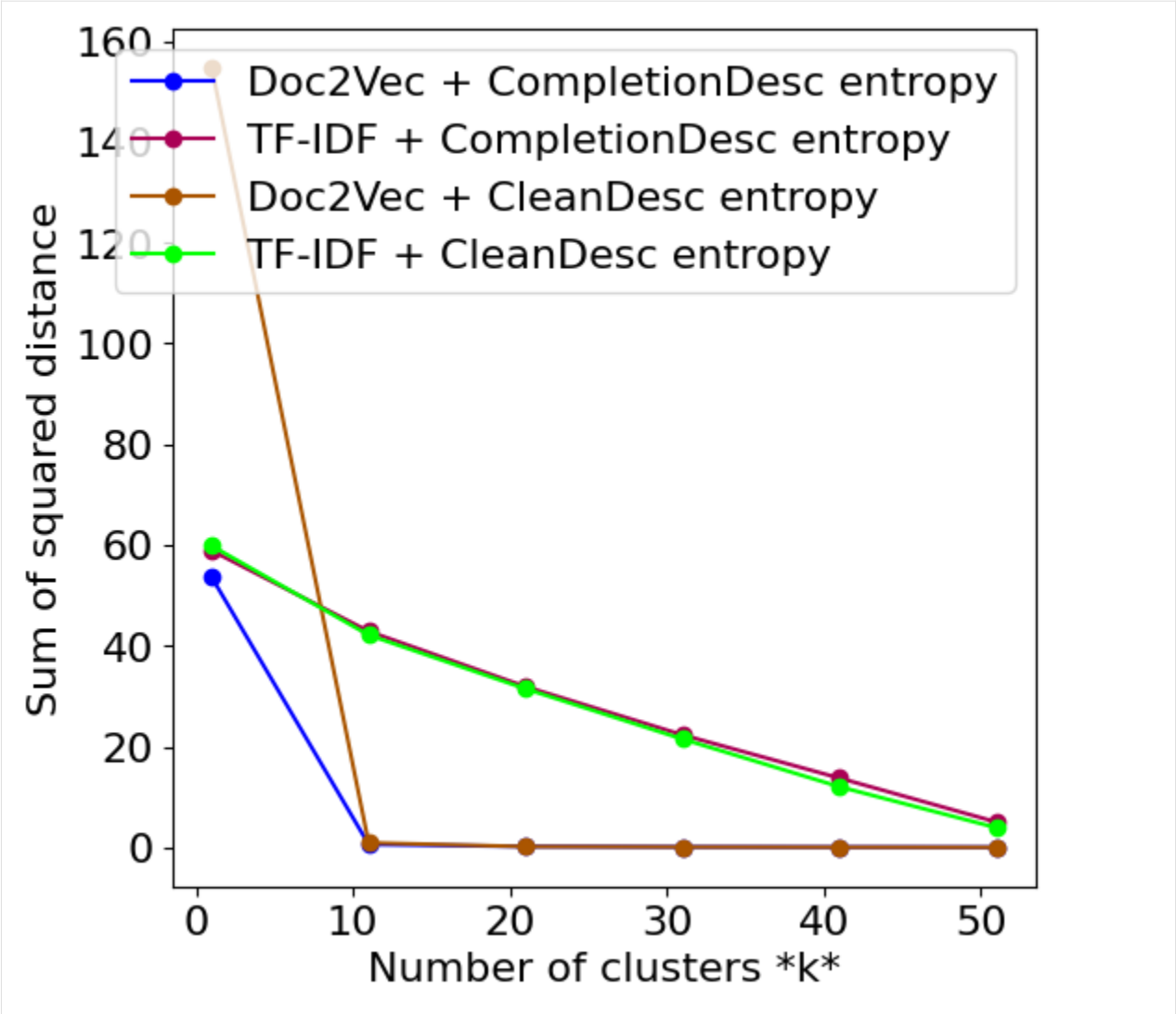
Results of text processing

[66]:
```
print("Pre-text processing")
e.summarize_text_data(DATA_COLUMN)

print("\nPost-text processing")
e.summarize_text_data('CleanDesc')
```

```
Pre-text processing
DETAILS
  64 samples
  0 invalid documents
  27.95 words per sample on average
  Number of unique words 778
  1789.00 total words

Post-text processing
DETAILS
  64 samples
  0 invalid documents
  17.31 words per sample on average
  Number of unique words 489
  1108.00 total words
```

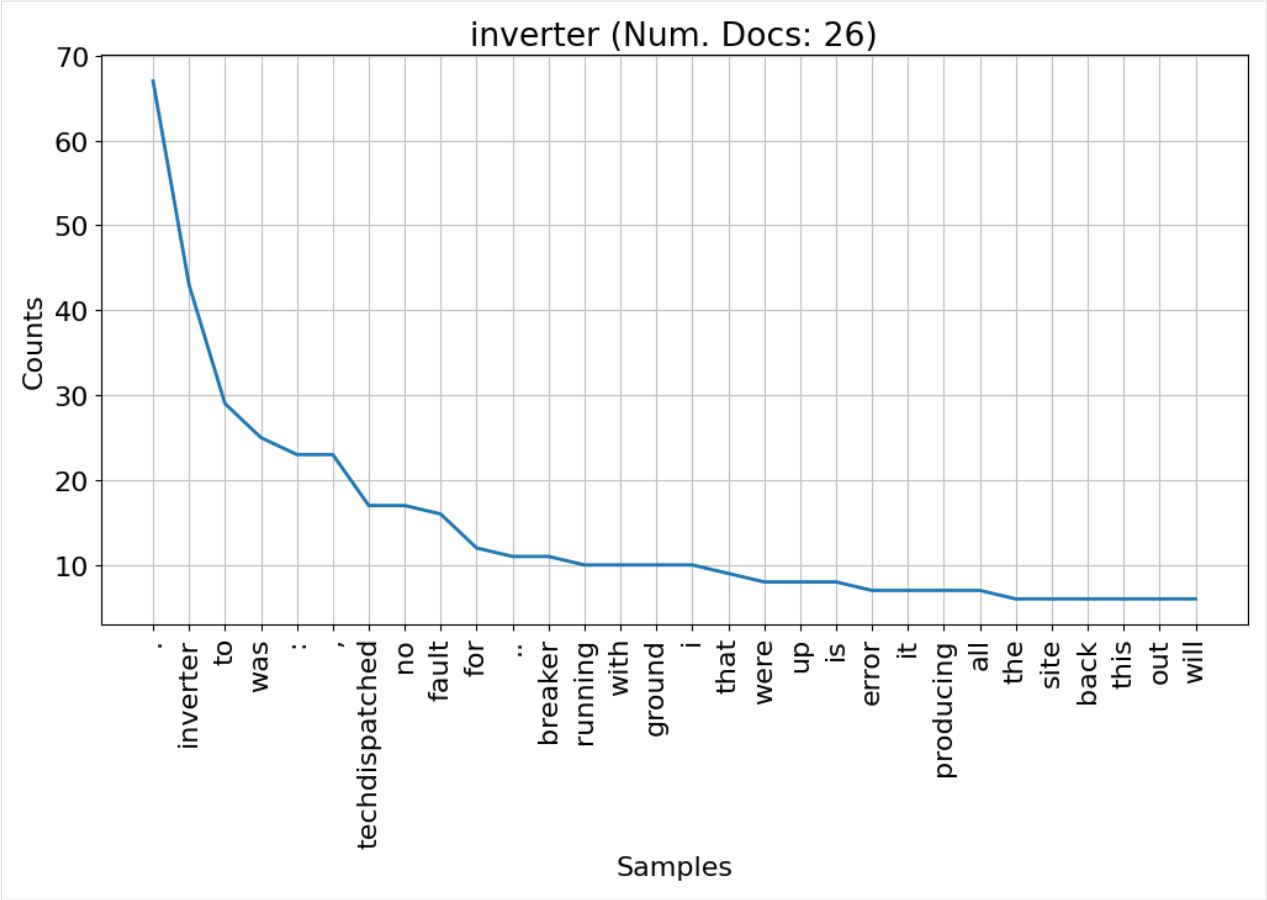
Visualizing entropy of clustering technique pre- and post- processing

[67]:
```
fig = e.visualize_cluster_entropy([DATA_COLUMN, 'CleanDesc'])
plt.show()
```

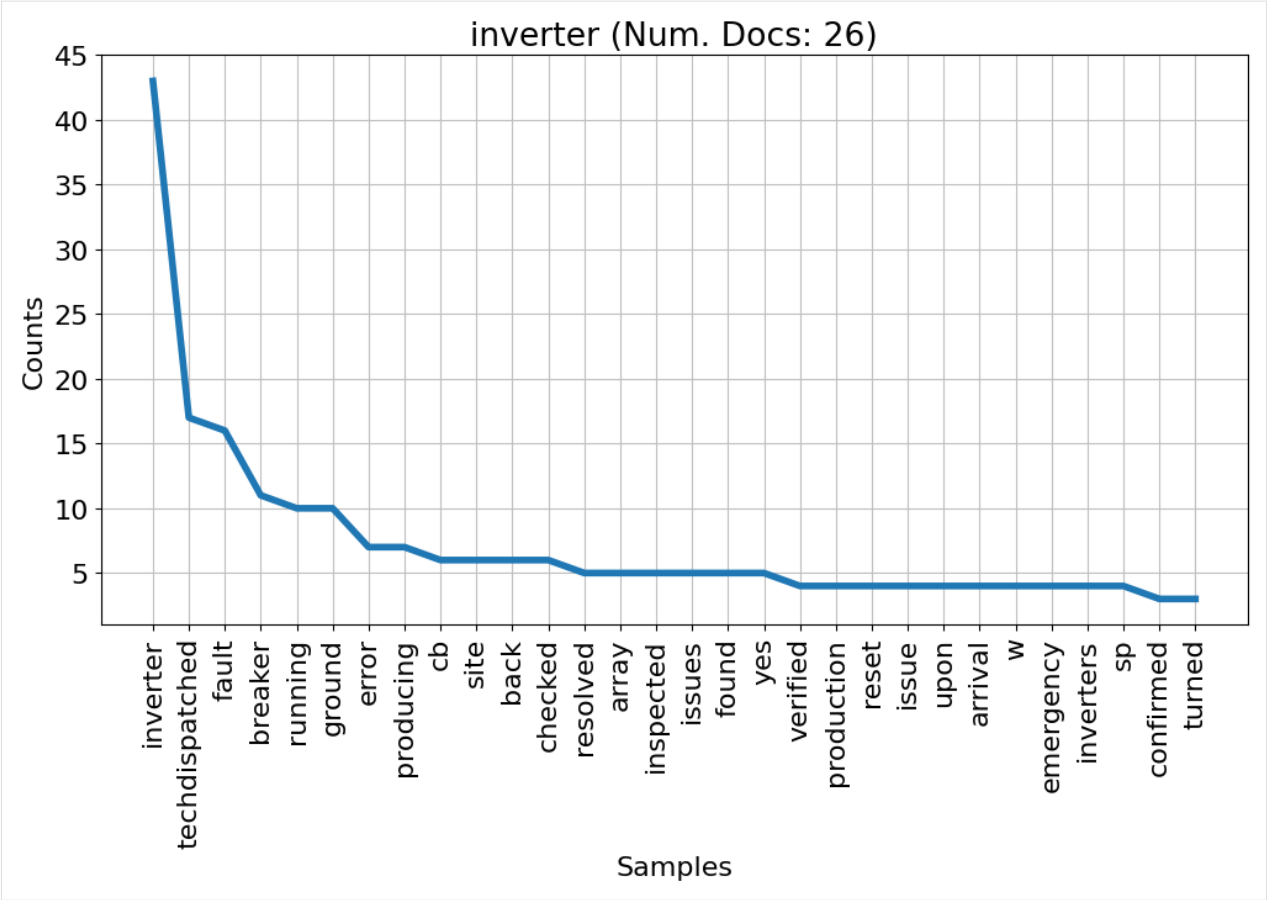


Functionality 1.3: Frequency plot

[68]:
```
# Frequency plot on unprocessed data
fig = e.visualize_freqPlot(LBL_CAT='inverter', DATA_COLUMN=DATA_COLUMN)
plt.show()
```



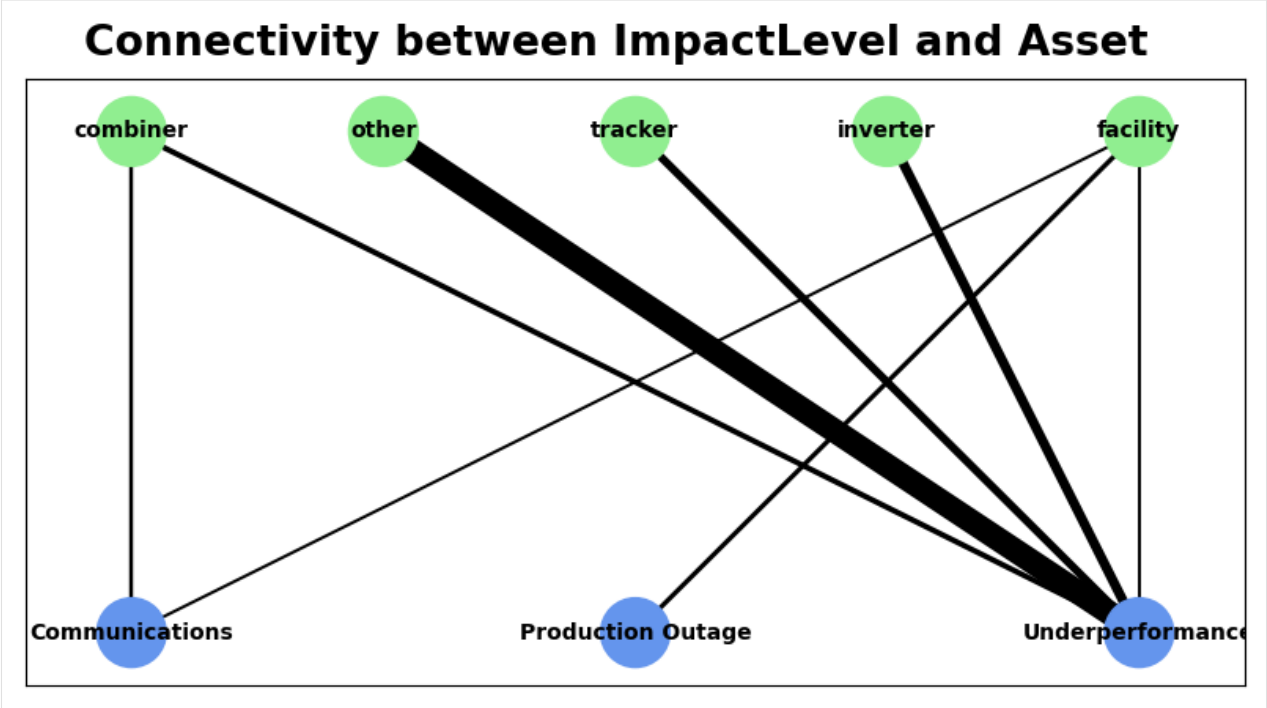
[69]:
```
# Frequency plot on processed data
fig = e.visualize_freqPlot(LBL_CAT='inverter',
                           # Optional, kwargs into nltk's FreqDist
                           graph_aargs = {
                               'linewidth':4
                           }
                       )
plt.show()
```



Hint: Use the below code to visualize frequency plots for all assets

```
set_labels = list(set(e.df[e.LABEL_COLUMN].tolist()))
for lbl in set_labels:
    fig = e.visualize_freqPlot(LBL_CAT=lbl)
    plt.show()
```

[70]:
```
# Only supports two attributes
om_col_dict = {
    'attribute1_col': 'Asset',
    'attribute2_col': 'ImpactLevel'
}

fig, G = e.visualize_attribute_connectivity(
    om_col_dict,
    figsize=[10,5],
    graph_aargs = {'with_labels':True,
    'font_weight':'bold',
    'node_size': 1000,
    'font_size':10}
)
```



Functionality 2.1: Conduct supervised classification on tickets using a cross-validated grid search

[71]:
```
# Setting few cross validation splits because of few example data
results, best_model = e.classify_supervised(n_cv_splits=2, embedding='tfidf')
print('best_model', best_model)
results
```

```
Starting ML analysis with TF-IDF embeddings
```

```
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
```

(continues on next page)

(continued from previous page)

```
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
```

(continues on next page)

(continued from previous page)

```
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪linear_model/_sag.py:350: ConvergenceWarning: The max_iter was reached which means the␣
↪coef_ did not converge
  warnings.warn(
```

```
best_model Pipeline(steps=[('tfidf', TfidfVectorizer(ngram_range=(1, 3))),
                ('clf', AdaBoostClassifier())])
```

[71]:

```
                     estimator min_score mean_score max_score std_score  \
67           AdaBoostClassifier  0.445812   0.487584  0.529356  0.041772
66           AdaBoostClassifier  0.445812   0.487584  0.529356  0.041772
65           AdaBoostClassifier  0.445812   0.487584  0.529356  0.041772
64           AdaBoostClassifier  0.445812   0.487584  0.529356  0.041772
62           AdaBoostClassifier  0.445812   0.487584  0.529356  0.041772
..                          ...       ...        ...       ...       ...
6                           SVC  0.234722   0.234722  0.234722       0.0
5                           SVC  0.234722   0.234722  0.234722       0.0
4                           SVC  0.221591   0.228157  0.234722  0.006566
39  PassiveAggressiveClassifier       NaN        NaN       NaN       NaN
40  PassiveAggressiveClassifier       NaN        NaN       NaN       NaN

   mean_fit_time clf__C clf__max_iter tfidf__ngram_range tfidf__stop_words  \
67       0.20235    NaN           NaN             (1, 3)              None
66       0.10263    NaN           NaN             (1, 3)              None
65      0.207092    NaN           NaN             (1, 3)              None
64      0.103079    NaN           NaN             (1, 3)              None
62      0.100246    NaN           NaN             (1, 3)              None
..           ...    ...           ...                ...               ...
6       0.006326    1.0           NaN             (1, 3)              None
5       0.006342    1.0           NaN             (1, 3)              None
4       0.005903    1.0           NaN             (1, 3)              None
39        0.0048    0.0           NaN             (1, 3)              None
40      0.004441    0.0           NaN             (1, 3)              None

    ... clf__splitter clf__alpha clf__batch_size clf__hidden_layer_sizes  \
67  ...           NaN        NaN             NaN                     NaN
66  ...           NaN        NaN             NaN                     NaN
65  ...           NaN        NaN             NaN                     NaN
64  ...           NaN        NaN             NaN                     NaN
62  ...           NaN        NaN             NaN                     NaN
..  ...           ...        ...             ...                     ...
6   ...           NaN        NaN             NaN                     NaN
5   ...           NaN        NaN             NaN                     NaN
4   ...           NaN        NaN             NaN                     NaN
39  ...           NaN        NaN             NaN                     NaN
40  ...           NaN        NaN             NaN                     NaN

   clf__learning_rate clf__solver      clf__loss clf__n_estimators  \
67                 0.8         NaN            NaN                100
```

(continues on next page)

(continued from previous page)

```
66                0.8        NaN            NaN               50
65                0.9        NaN            NaN              100
64                0.9        NaN            NaN               50
62                1.0        NaN            NaN               50
..                ...        ...            ...              ...
6                 NaN        NaN            NaN              NaN
5                 NaN        NaN            NaN              NaN
4                 NaN        NaN            NaN              NaN
39                NaN        NaN          hinge              NaN
40                NaN        NaN  squared_hinge              NaN

   clf__max_samples clf__algorithm
67              NaN        SAMME.R
66              NaN        SAMME.R
65              NaN        SAMME.R
64              NaN        SAMME.R
62              NaN        SAMME.R
..              ...            ...
6               NaN            NaN
5               NaN            NaN
4               NaN            NaN
39              NaN            NaN
40              NaN            NaN

[68 rows x 25 columns]
```

Functionality 1.4: Conduct unsupervised clustering on tickets using a cross-validated grid search

[78]:

```
# Setting few cross validation splits because of few example data
results, best_model = e.classify_unsupervised(n_cv_splits=2, embedding='tfidf')
print('best_model', best_model)
results
```

```
Starting ML analysis with TF-IDF embeddings
```

```
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪cluster/_birch.py:726: ConvergenceWarning: Number of subclusters found (1) by BIRCH is␣
↪less than (11). Decrease the threshold.
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪cluster/_birch.py:726: ConvergenceWarning: Number of subclusters found (1) by BIRCH is␣
↪less than (11). Decrease the threshold.
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪cluster/_birch.py:726: ConvergenceWarning: Number of subclusters found (1) by BIRCH is␣
↪less than (11). Decrease the threshold.
  warnings.warn(
/home/klbonne/.pyenv/versions/3.11.5/envs/pvops/lib/python3.11/site-packages/sklearn/
↪cluster/_birch.py:726: ConvergenceWarning: Number of subclusters found (1) by BIRCH is␣
↪less than (11). Decrease the threshold.
  warnings.warn(
```

```
best_model Pipeline(steps=[('tfidf', TfidfVectorizer(ngram_range=(1, 3))),
                ('to_dense', DataDensifier()), ('clf', Birch(n_clusters=11))])
```

[78]:

```
              estimator min_score mean_score max_score std_score  \
7                 Birch  0.629666   0.647877  0.666088  0.018211
4                 Birch  0.629666   0.647877  0.666088  0.018211
1   AffinityPropagation   0.42079   0.522839  0.624887  0.102048
3   AffinityPropagation   0.42079   0.517806  0.614823  0.097016
0   AffinityPropagation  0.425789   0.507145  0.588502  0.081356
2   AffinityPropagation  0.425789   0.503147  0.580505  0.077358
13               KMeans  0.303348   0.434765  0.566182  0.131417
15               KMeans  0.368506   0.407506  0.446506     0.039
10               KMeans   0.22441   0.317554  0.410697  0.093144
14               KMeans  0.268359   0.303465  0.338572  0.035107
12               KMeans  0.268081   0.298927  0.329774  0.030847
11               KMeans  0.289608    0.29647  0.303332  0.006862
16      MiniBatchKMeans  0.265281   0.277444  0.289608  0.012163
5                 Birch  0.157604    0.23402  0.310436  0.076416
8                 Birch  0.157604    0.23402  0.310436  0.076416
20      MiniBatchKMeans  0.110475   0.183592   0.25671  0.073117
17      MiniBatchKMeans  0.036041   0.160629  0.285217  0.124588
21      MiniBatchKMeans       0.0   0.131439  0.262878  0.131439
19      MiniBatchKMeans       0.0   0.086007  0.172015  0.086007
18      MiniBatchKMeans  0.065271   0.085739  0.106208  0.020469
23            MeanShift       0.0        0.0       0.0       0.0
24            MeanShift       0.0        0.0       0.0       0.0
6                 Birch       0.0        0.0       0.0       0.0
22            MeanShift       0.0        0.0       0.0       0.0
9                 Birch       0.0        0.0       0.0       0.0
25            MeanShift       0.0        0.0       0.0       0.0

   mean_fit_time clf__damping clf__max_iter tfidf__ngram_range  \
7          0.007          NaN           NaN             (1, 3)
4       0.007483          NaN           NaN             (1, 3)
1       0.010927          0.5           600             (1, 3)
3       0.010343          0.9           600             (1, 3)
0       0.010901          0.5           200             (1, 3)
2       0.013031          0.9           200             (1, 3)
13       0.01097          NaN           NaN             (1, 3)
15      0.057041          NaN           NaN             (1, 3)
10       0.06142          NaN           NaN             (1, 3)
14      0.034123          NaN           NaN             (1, 3)
12      0.221901          NaN           NaN             (1, 3)
11      0.098513          NaN           NaN             (1, 3)
16      0.015522          NaN           NaN             (1, 3)
5       0.008387          NaN           NaN             (1, 3)
8       0.008248          NaN           NaN             (1, 3)
20      0.010481          NaN           NaN             (1, 3)
17      0.028212          NaN           NaN             (1, 3)
21      0.015208          NaN           NaN             (1, 3)
19      0.010238          NaN           NaN             (1, 3)
18      0.040977          NaN           NaN             (1, 3)
23      0.050849          NaN           600             (1, 3)
```

(continues on next page)

(continued from previous page)

```
24      0.016261          NaN           300             (1, 3)
6       0.006244          NaN           NaN             (1, 3)
22      0.045921          NaN           300             (1, 3)
9       0.005972          NaN           NaN             (1, 3)
25      0.015578          NaN           600             (1, 3)

   tfidf__stop_words clf__branching_factor clf__n_clusters clf__threshold  \
7               None                   100              11            0.5
4               None                    50              11            0.5
1               None                   NaN             NaN            NaN
3               None                   NaN             NaN            NaN
0               None                   NaN             NaN            NaN
2               None                   NaN             NaN            NaN
13              None                   NaN              11            NaN
15              None                   NaN              11            NaN
10              None                   NaN              11            NaN
14              None                   NaN              11            NaN
12              None                   NaN              11            NaN
11              None                   NaN              11            NaN
16              None                   NaN              11            NaN
5               None                    50              11           0.75
8               None                   100              11           0.75
20              None                   NaN              11            NaN
17              None                   NaN              11            NaN
21              None                   NaN              11            NaN
19              None                   NaN              11            NaN
18              None                   NaN              11            NaN
23              None                   NaN             NaN            NaN
24              None                   NaN             NaN            NaN
6               None                    50              11            1.0
22              None                   NaN             NaN            NaN
9               None                   100              11            1.0
25              None                   NaN             NaN            NaN

    clf__init clf__n_init clf__bandwidth clf__bin_seeding
7         NaN         NaN            NaN              NaN
4         NaN         NaN            NaN              NaN
1         NaN         NaN            NaN              NaN
3         NaN         NaN            NaN              NaN
0         NaN         NaN            NaN              NaN
2         NaN         NaN            NaN              NaN
13     random          10            NaN              NaN
15     random         100            NaN              NaN
10  k-means++          10            NaN              NaN
14     random          50            NaN              NaN
12  k-means++         100            NaN              NaN
11  k-means++          50            NaN              NaN
16  k-means++           3            NaN              NaN
5         NaN         NaN            NaN              NaN
8         NaN         NaN            NaN              NaN
20     random          10            NaN              NaN
17  k-means++          10            NaN              NaN
```

(continues on next page)

(continued from previous page)

```
21     random          20            NaN              NaN
19     random           3            NaN              NaN
18  k-means++          20            NaN              NaN
23        NaN         NaN           None            False
24        NaN         NaN           None             True
6         NaN         NaN            NaN              NaN
22        NaN         NaN           None            False
9         NaN         NaN            NaN              NaN
25        NaN         NaN           None             True
```

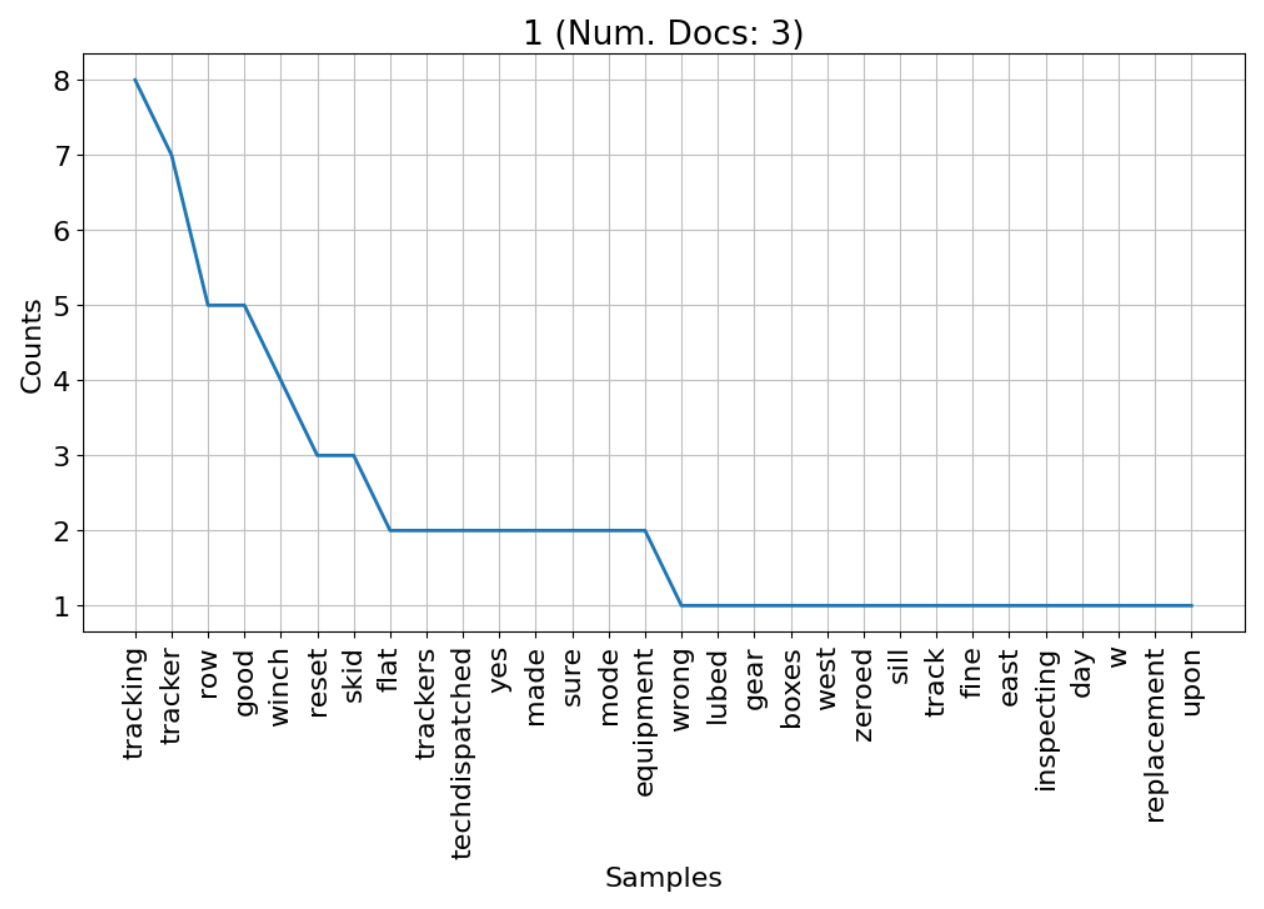
[73]:
```
e.predict_best_model(ml_type = 'supervised')
```

```
Best algorithm found:
 Pipeline(steps=[('tfidf', TfidfVectorizer(ngram_range=(1, 3))),
                ('clf', AdaBoostClassifier())])
Predictions stored to Supervised_Pred_Asset in `df` attribute
Score: 0.5729910714285715
```

[74]:
```
e.predict_best_model(ml_type = 'unsupervised')
```

```
Best algorithm found:
 Pipeline(steps=[('tfidf', TfidfVectorizer(ngram_range=(1, 3))),
                ('to_dense', DataDensifier()),
                ('clf', AffinityPropagation(damping=0.9, max_iter=600))])
Predictions stored to Unsupervised_Pred_Asset in `df` attribute
Score: 0.42789964962778615
```

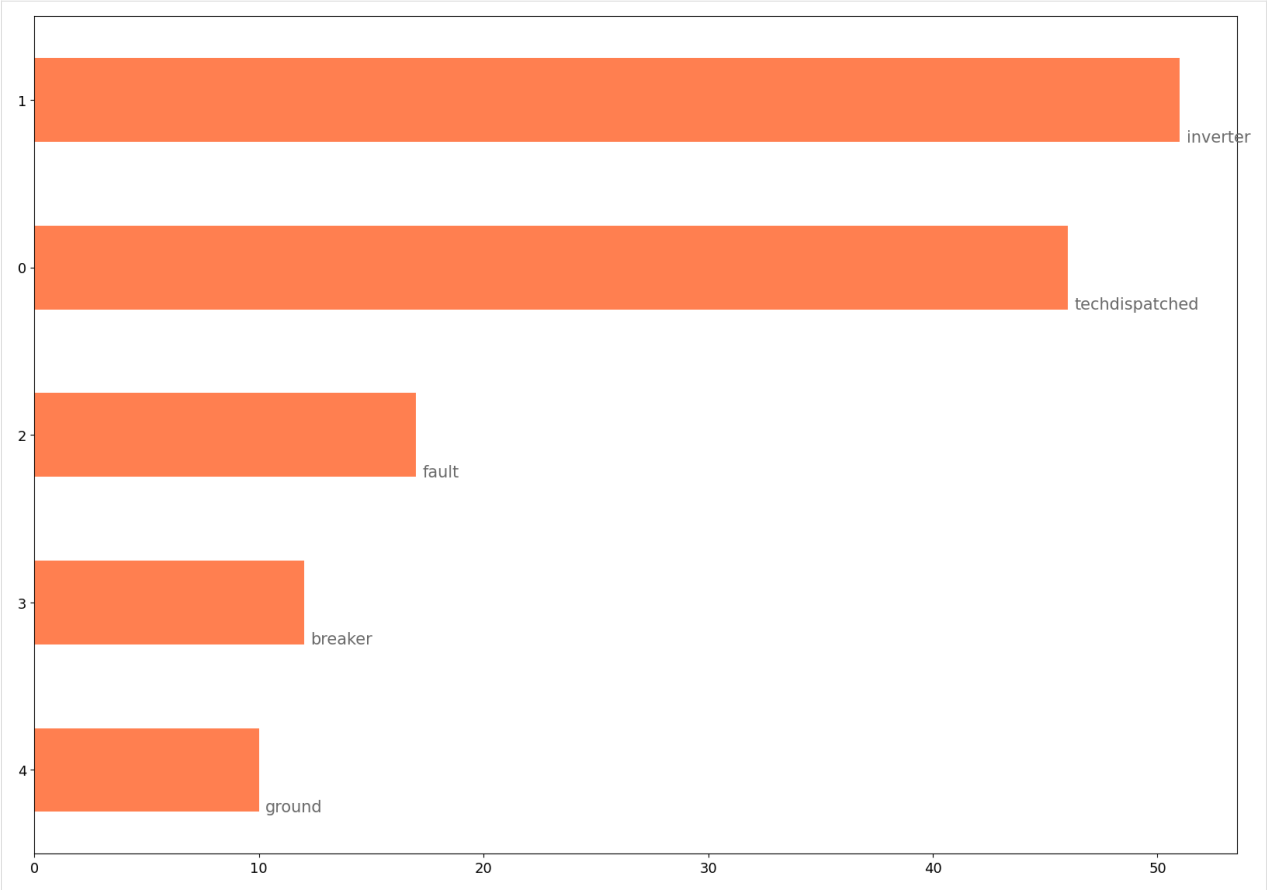
[75]:
```
e.LABEL_COLUMN = 'Unsupervised_Pred_Asset'
e.visualize_freqPlot(LBL_CAT = 1)
```



[75]:
```
FreqDist({'tracking': 8, 'tracker': 7, 'row': 5, 'good': 5, 'winch': 4, 'reset': 3, 'skid
↪': 3, 'flat': 2, 'trackers': 2, 'techdispatched': 2, ...})
```

Visualize Word Clusters

[76]:
```
fig = e.visualize_document_clusters(min_frequency=10, DATA_COLUMN='CleanDesc')
plt.show()
```



Seeing the popularity of `techdispatched`, one might consider adding `techdispatched` to the stopwords list

1.2.3 Timeseries Tutorial

This notebook demonstrates the use of `timeseries` module capabilities to process data for analysis and build expected energy models.

[20]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
import warnings
warnings.filterwarnings("ignore")
```

[21]:
```
from pvops.timeseries import preprocess
from pvops.timeseries.models import linear, iec
from pvops.text2time import utils as t2t_utils, preprocess as t2t_preprocess
```

[22]:
```
example_prodpath = os.path.join('example_data', 'example_prod_with_covariates.csv')
```

Load in data

[23]:
```
prod_data = pd.read_csv(example_prodpath, on_bad_lines='skip', engine='python')
```

Production data is loaded in with the columns: - date: a time stamp - randid: site ID - generated_kW: generated energy - expected_kW: expected energy - irrad_poa_Wm2: irradiance - temp_amb_C: ambient temperature - wind_speed_ms: wind speed - temp_mod_C: module temperature

[24]:
```
prod_data.head()
```

[24]:
```
                  date randid  generated_kW   expected_kW  irrad_poa_Wm2  \
0  2018-04-01 07:00:00    R15         0.475      0.527845        0.02775
1  2018-04-01 08:00:00    R15      1332.547   1685.979445       87.91450
2  2018-04-01 09:00:00    R15      6616.573   7343.981135      367.90350
3  2018-04-01 10:00:00    R15      8847.800  10429.876422      508.28700
4  2018-04-01 11:00:00    R15     11607.389  12981.228814      618.79450

   temp_amb_C  wind_speed_ms  temp_mod_C
0      16.570         4.2065     14.1270
1      16.998         4.1065     15.8610
2      20.168         4.5095     24.5745
3      21.987         4.9785     30.7740
4      23.417         4.6410     35.8695
```

[25]:
```
metadata = pd.DataFrame()
metadata['randid'] = ['R15', 'R10']
metadata['dcsize'] = [20000, 20000]
metadata.head()
```

[25]:
```
  randid  dcsize
0    R15   20000
1    R10   20000
```

Column dictionaries

Create production and metadata column dictionary with format {pvops variable: user-specific column names}. This establishes a connection between the user's data columns and the pvops library.

[26]:
```
prod_col_dict = {'siteid': 'randid',
                 'timestamp': 'date',
                 'powerprod': 'generated_kW',
                 'irradiance':'irrad_poa_Wm2',
                 'temperature':'temp_amb_C', # Optional parameter, used by one of the␣
↪modeling structures
                 'baseline': 'IEC_pstep', #user's name choice for new column (baseline␣
↪expected energy defined by user or calculated based on IEC)
                 'dcsize': 'dcsize', #user's name choice for new column (System DC-size,␣
↪extracted from meta-data)
                 'compared': 'Compared',#user's name choice for new column
                 'energy_pstep': 'Energy_pstep', #user's name choice for new column
                 'capacity_normalized_power': 'capacity_normalized_power', #user's name␣
↪choice for new column
```

(continues on next page)

(continued from previous page)

```
}

metad_col_dict = {'siteid': 'randid',
                  'dcsize': 'dcsize'}
```

Data Formatting

Use the `prod_date_convert` function to convert date information to python datetime objects and use `prod_nadate_process` to handle data entries with no date information - here we use `pnadrop=True` to drop such entries.

[27]:
```
prod_data_converted = t2t_preprocess.prod_date_convert(prod_data, prod_col_dict)
prod_data_datena_d, _ = t2t_preprocess.prod_nadate_process(prod_data_converted, prod_col_
↪dict, pnadrop=True)
```

Assign production data index to timestamp data, using column dictionary to translate to user columns.

[28]:
```
prod_data_datena_d.index = prod_data_datena_d[prod_col_dict['timestamp']]

min(prod_data_datena_d.index), max(prod_data_datena_d.index)
```

[28]:
```
(Timestamp('2018-04-01 07:00:00'), Timestamp('2019-03-31 18:00:00'))
```

[29]:
```
prod_data_datena_d
```

[29]:
```
                                date randid  generated_kW   expected_kW  \
date
2018-04-01 07:00:00 2018-04-01 07:00:00    R15         0.475      0.527845
2018-04-01 08:00:00 2018-04-01 08:00:00    R15      1332.547   1685.979445
2018-04-01 09:00:00 2018-04-01 09:00:00    R15      6616.573   7343.981135
2018-04-01 10:00:00 2018-04-01 10:00:00    R15      8847.800  10429.876422
2018-04-01 11:00:00 2018-04-01 11:00:00    R15     11607.389  12981.228814
...                                   ...    ...           ...           ...
2019-03-31 14:00:00 2019-03-31 14:00:00    R10     19616.000  20179.797303
2019-03-31 15:00:00 2019-03-31 15:00:00    R10     19552.000  20149.254116
2019-03-31 16:00:00 2019-03-31 16:00:00    R10     19520.000  20192.672439
2019-03-31 17:00:00 2019-03-31 17:00:00    R10     17968.000  19141.761319
2019-03-31 18:00:00 2019-03-31 18:00:00    R10     11616.000  12501.159607

                     irrad_poa_Wm2  temp_amb_C  wind_speed_ms  temp_mod_C
date
2018-04-01 07:00:00        0.02775   16.570000         4.2065   14.127000
2018-04-01 08:00:00       87.91450   16.998000         4.1065   15.861000
2018-04-01 09:00:00      367.90350   20.168000         4.5095   24.574500
2018-04-01 10:00:00      508.28700   21.987000         4.9785   30.774000
2018-04-01 11:00:00      618.79450   23.417000         4.6410   35.869500
...                            ...         ...            ...         ...
2019-03-31 14:00:00      967.10350   -4.129722         5.9935   35.860694
2019-03-31 15:00:00      983.94500   -4.214444         5.1770   38.549306
2019-03-31 16:00:00     1011.77950   -3.943611         4.2430   42.848889
2019-03-31 17:00:00      895.76800   -4.124167         3.1885   42.321806
2019-03-31 18:00:00      649.48700   -4.663056         2.2540   38.596944
```

(continues on next page)

(continued from previous page)

```
[8755 rows x 8 columns]
```

Data Preprocessing

Preprocess data with `prod_inverter_clipping_filter` using the threshold model. This adds a mask column to the dataframe where `True` indicates a row to be removed by the filter.

[30]:
```
masked_prod_data = preprocess.prod_inverter_clipping_filter(prod_data_datena_d, prod_col_
↪dict, metadata, metad_col_dict, 'threshold', freq=60)

filtered_prod_data = masked_prod_data[masked_prod_data['mask'] == False]

print(f"Detected and removed {sum(masked_prod_data['mask'])} rows with inverter clipping.
↪")
```

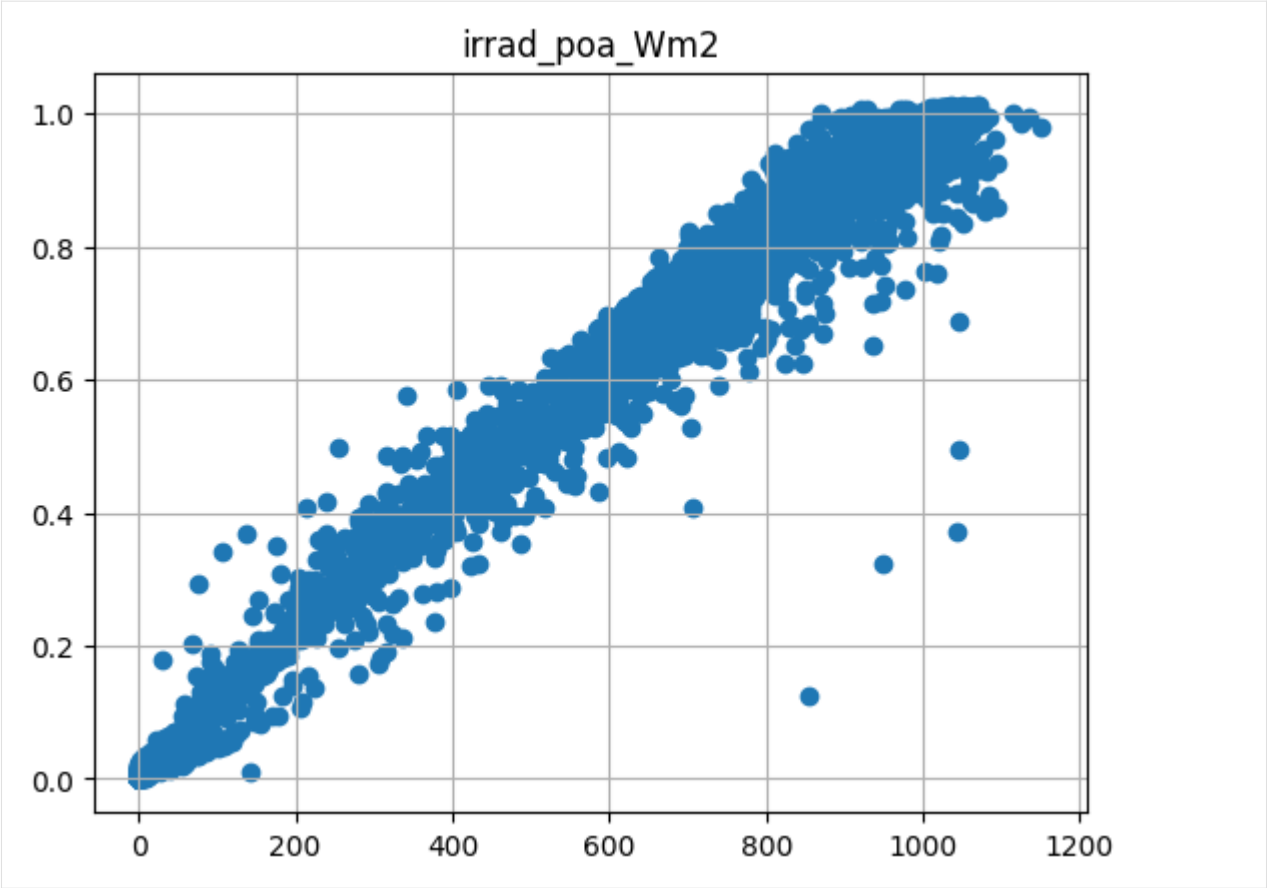
```
Detected and removed 24 rows with inverter clipping.
```

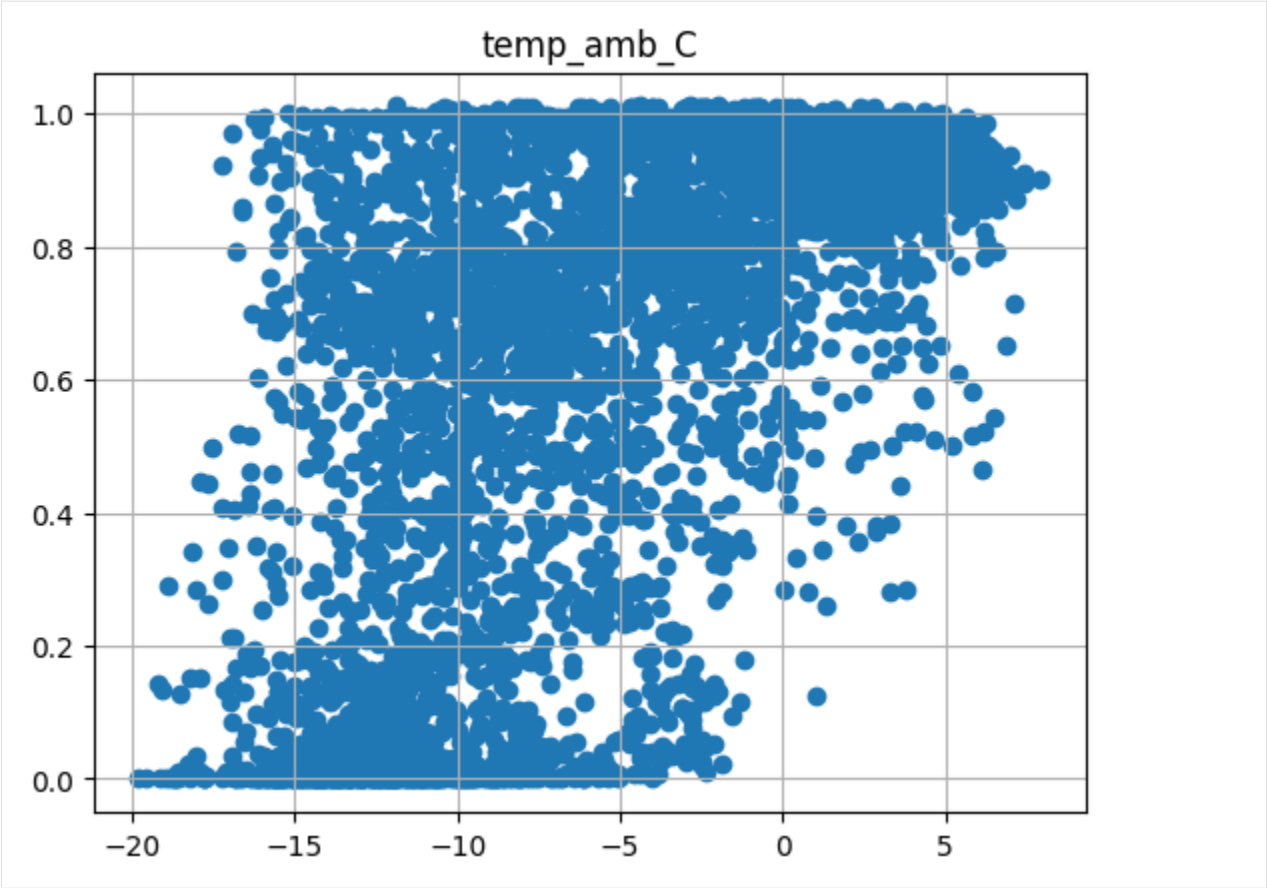
Normalize the power by capacity

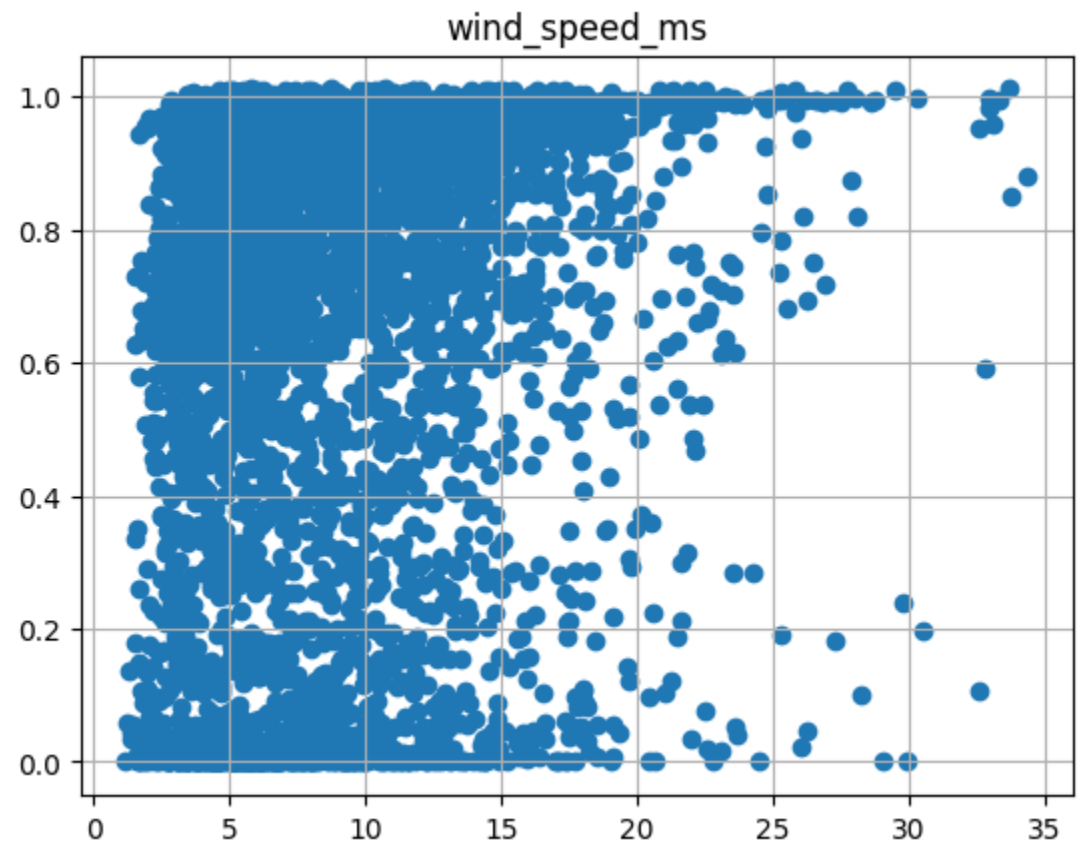
[31]:
```
for site in metadata[metad_col_dict['siteid']].unique():
    site_metad_mask = metadata[metad_col_dict['siteid']] == site
    site_prod_mask = filtered_prod_data[prod_col_dict['siteid']] == site
    dcsize = metadata.loc[site_metad_mask, metad_col_dict['dcsize']].values[0]
    filtered_prod_data.loc[site_prod_mask, [prod_col_dict['capacity_normalized_power']]]␣
↪= filtered_prod_data.loc[site_prod_mask, prod_col_dict['powerprod']] / dcsize
```

Visualize the power signal versus covariates for one site with normalized values.

[32]:
```
temp = filtered_prod_data[filtered_prod_data['randid'] == 'R10']
for xcol in ['irrad_poa_Wm2', 'temp_amb_C', 'wind_speed_ms']:
    plt.scatter(temp[xcol], temp[prod_col_dict['capacity_normalized_power']])
    plt.title(xcol)
    plt.grid()
    plt.show()
```


irrad_poa_Wm2
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000
1200


temp_amb_C
1.0
0.8
0.6
0.4
0.2
0.0
−20
−15
−10
−5
0
5



Dynamic linear modeling

We train a linear model using site data (irradiance, temperature, and windspeed)

[33]:
```
model_prod_data = filtered_prod_data.dropna(subset=['irrad_poa_Wm2', 'temp_amb_C', 'wind_
↪speed_ms']+[prod_col_dict['powerprod']])
# Make sure to only pass data for one site! If sites are very similar, you can consider␣
↪providing both sites.
model, train_df, test_df = linear.modeller(prod_col_dict,
                                           kernel_type='polynomial',
                                           #time_weighted='month',
                                           X_parameters=['irrad_poa_Wm2', 'temp_amb_C'],
↪#, 'wind_speed_ms'],
                                           Y_parameter='generated_kW',
                                           prod_df=model_prod_data,
                                           test_split=0.05,
                                           degree=3,
                                           verbose=1)
```

```
train {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
Design matrix shape: (8293, 108)
test {2, 3}
Design matrix shape: (437, 108)
```

(continues on next page)

(continued from previous page)

```
Begin training.
[OLS] Mean squared error: 679448.77
[OLS] Coefficient of determination: 0.99
[OLS] 108 coefficient trained.
[RANSAC] Mean squared error: 812510.79
[RANSAC] Coefficient of determination: 0.98

Begin testing.
[OLS] Mean squared error: 7982573.16
[OLS] Coefficient of determination: 0.85
[OLS] 108 coefficient trained.
[RANSAC] Mean squared error: 81461712.61
[RANSAC] Coefficient of determination: -0.55
```

Define a plotting function

[34]:
```
from sklearn.metrics import mean_squared_error, r2_score

def plot(model, prod_col_dict, data_split='test', npts=50):

    def print_info(real,pred,name):
        mse = mean_squared_error(real, pred)
        r2 = r2_score(real, pred)
        print(f'[{name}] Mean squared error: %.2f'
                % mse)
        print(f'[{name}] Coefficient of determination: %.2f'
                % r2)

    fig,(ax) = plt.subplots(figsize=(14,8))

    if data_split == 'test':
        df = test_df
    elif data_split == 'train':
        df = train_df

    measured = model.estimators['OLS'][f'{data_split}_y'][:npts]

    ax2 = ax.twinx()
    ax2.plot(model.estimators['OLS'][f'{data_split}_index'][:npts], df[prod_col_dict[
↪'irradiance']].values[:npts], 'k', label='irradiance')

    ax.plot(model.estimators['OLS'][f'{data_split}_index'][:npts], df['expected_kW'].
↪values[:npts], label='partner_expected')
    print_info(measured, df['expected_kW'].values[:npts], 'partner_expected')

    ax.plot(model.estimators['OLS'][f'{data_split}_index'][:npts], measured, label=
↪'measured')
    for name, info in model.estimators.items():
        predicted = model.estimators[name][f'{data_split}_prediction'][:npts]
        ax.plot(model.estimators[name][f'{data_split}_index'][:npts], predicted,␣
↪label=name)
        print_info(measured, predicted, name)
```

(continues on next page)

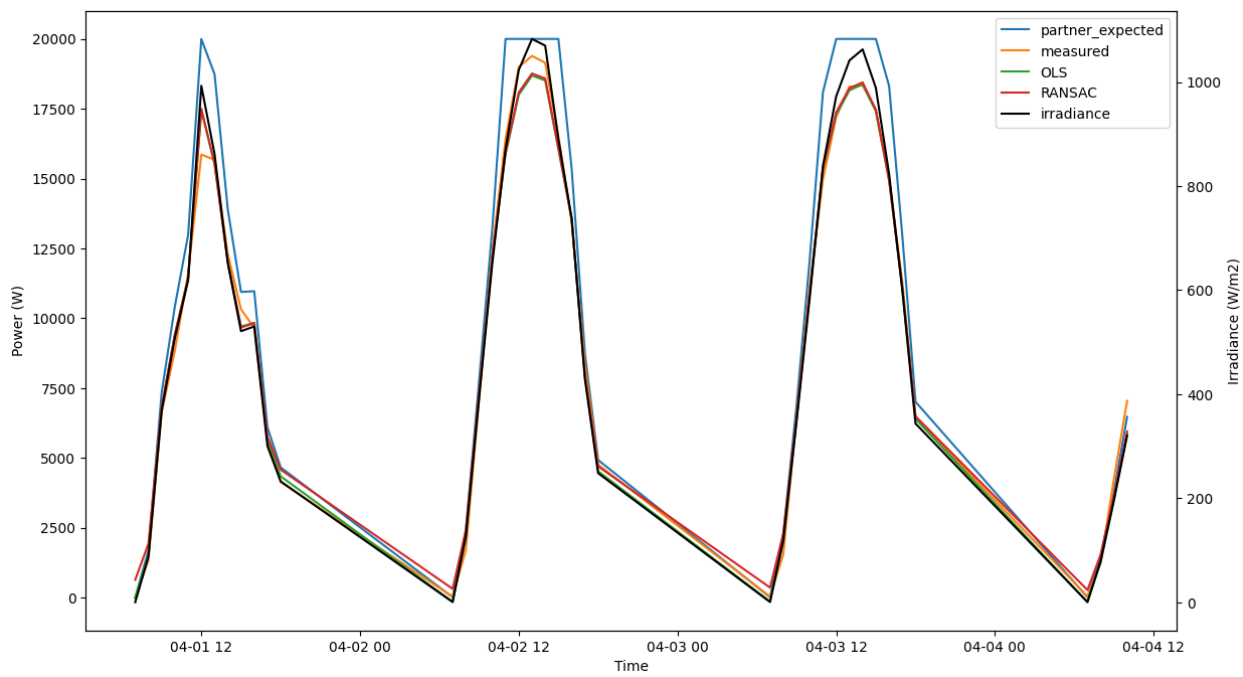
(continued from previous page)

```
    ax2.set_ylabel("Irradiance (W/m2)")
    ax.set_ylabel("Power (W)")
    ax.set_xlabel('Time')
    handles, labels = [(a+b) for a, b in zip(ax.get_legend_handles_labels(), ax2.get_
↪legend_handles_labels())]
    ax.legend(handles, labels, loc='best')
    plt.show()
```

Observe performance

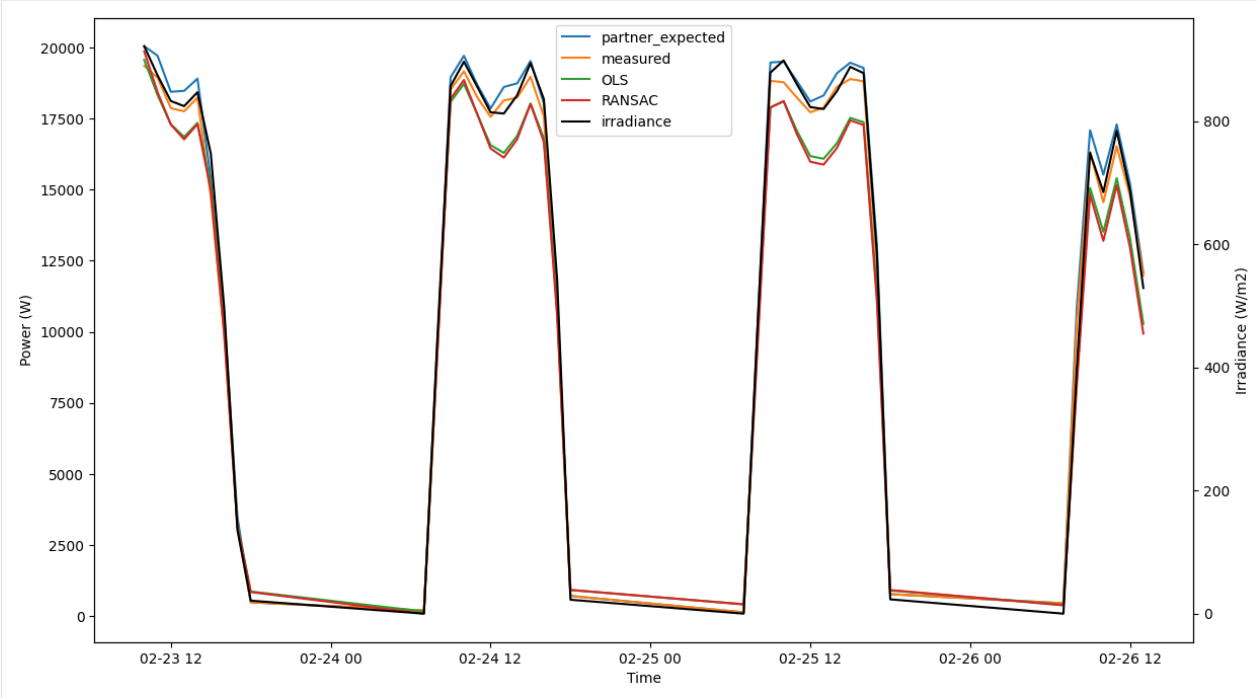
[35]:
```
plot(model, prod_col_dict, data_split='train', npts=40)
```

```
[partner_expected] Mean squared error: 2883252.79
[partner_expected] Coefficient of determination: 0.93
[OLS] Mean squared error: 225151.82
[OLS] Coefficient of determination: 0.99
[RANSAC] Mean squared error: 253701.63
[RANSAC] Coefficient of determination: 0.99
```



[36]:
```
plot(model, prod_col_dict, data_split='test', npts=40)
```

```
[partner_expected] Mean squared error: 273446.60
[partner_expected] Coefficient of determination: 0.99
[OLS] Mean squared error: 1037605.79
[OLS] Coefficient of determination: 0.98
[RANSAC] Mean squared error: 1300872.49
[RANSAC] Coefficient of determination: 0.97
```



1.2.4 Timeseries AIT Tutorial

The goal of this notebook is to use the trained AIT model to calculate expected energy levels based on field data. First we will load in and clean the data and after the expected energy is calculated, we will create comparitive visualizations.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
```

[2]:
```
from pvops.timeseries import preprocess
from pvops.timeseries.models import linear, iec, AIT
from pvops.text2time import utils as t2t_utils, preprocess as t2t_preprocess
```

Load in data

[3]:
```
example_OMpath = os.path.join('example_data', 'example_om_data2.csv')
example_prodpath = os.path.join('example_data', 'example_prod_with_covariates.csv')
example_metapath = os.path.join('example_data', 'example_metadata2.csv')
```

[4]:
```
prod_data = pd.read_csv(example_prodpath, on_bad_lines='skip', engine='python')
```

[5]:
```
prod_data.head(5)
```

[5]:
```
                  date randid  generated_kW   expected_kW  irrad_poa_Wm2  \
0  2018-04-01 07:00:00    R15         0.475      0.527845        0.02775
```

(continues on next page)

(continued from previous page)

```
1  2018-04-01 08:00:00    R15      1332.547   1685.979445       87.91450
2  2018-04-01 09:00:00    R15      6616.573   7343.981135      367.90350
3  2018-04-01 10:00:00    R15      8847.800  10429.876422      508.28700
4  2018-04-01 11:00:00    R15     11607.389  12981.228814      618.79450

   temp_amb_C  wind_speed_ms  temp_mod_C
0      16.570         4.2065     14.1270
1      16.998         4.1065     15.8610
2      20.168         4.5095     24.5745
3      21.987         4.9785     30.7740
4      23.417         4.6410     35.8695
```

[6]:
```
metadata = pd.DataFrame()
metadata['randid'] = ['R15', 'R10']
metadata['dcsize'] = [25000, 25000]
metadata.head()
```

[6]:
```
  randid  dcsize
0    R15   25000
1    R10   25000
```

Column dictionaries

Create production and metadata column dictionary with format {pvops variable: user-specific column names}. This establishes a connection between the user's data columns and the pvops library.

[7]:
```
prod_col_dict = {'siteid': 'randid',
                 'timestamp': 'date',
                 'powerprod': 'generated_kW',
                 'energyprod': 'generated_kW',
                 'irradiance':'irrad_poa_Wm2',
                 'temperature':'temp_amb_C', # Optional parameter, used by one of the␣
↪modeling structures
                 'baseline': 'AIT', #user's name choice for new column (baseline expected␣
↪energy defined by user or calculated based on IEC)
                 'dcsize': 'dcsize', #user's name choice for new column (System DC-size,␣
↪extracted from meta-data)
                 'compared': 'Compared',#user's name choice for new column
                 'energy_pstep': 'Energy_pstep', #user's name choice for new column
                 'capacity_normalized_power': 'capacity_normalized_power', #user's name␣
↪choice for new column
}

metad_col_dict = {'siteid': 'randid',
                  'dcsize': 'dcsize'}
```

Data Formatting

Use the `prod_date_convert` function to convert date information to python datetime objects and use `prod_nadate_process` to handle data entries with no date information - here we use `pnadrop=True` to drop such entries.

[8]:
```
prod_data_converted = t2t_preprocess.prod_date_convert(prod_data, prod_col_dict)
prod_data_datena_d, _ = t2t_preprocess.prod_nadate_process(prod_data_converted, prod_col_
↪dict, pnadrop=True)
```

Assign production data index to timestamp data, using column dictionary to translate to user columns.

[9]:
```
prod_data_datena_d.index = prod_data_datena_d[prod_col_dict['timestamp']]

min(prod_data_datena_d.index), max(prod_data_datena_d.index)
```

[9]:
```
(Timestamp('2018-04-01 07:00:00'), Timestamp('2019-03-31 18:00:00'))
```

Data Preprocessing

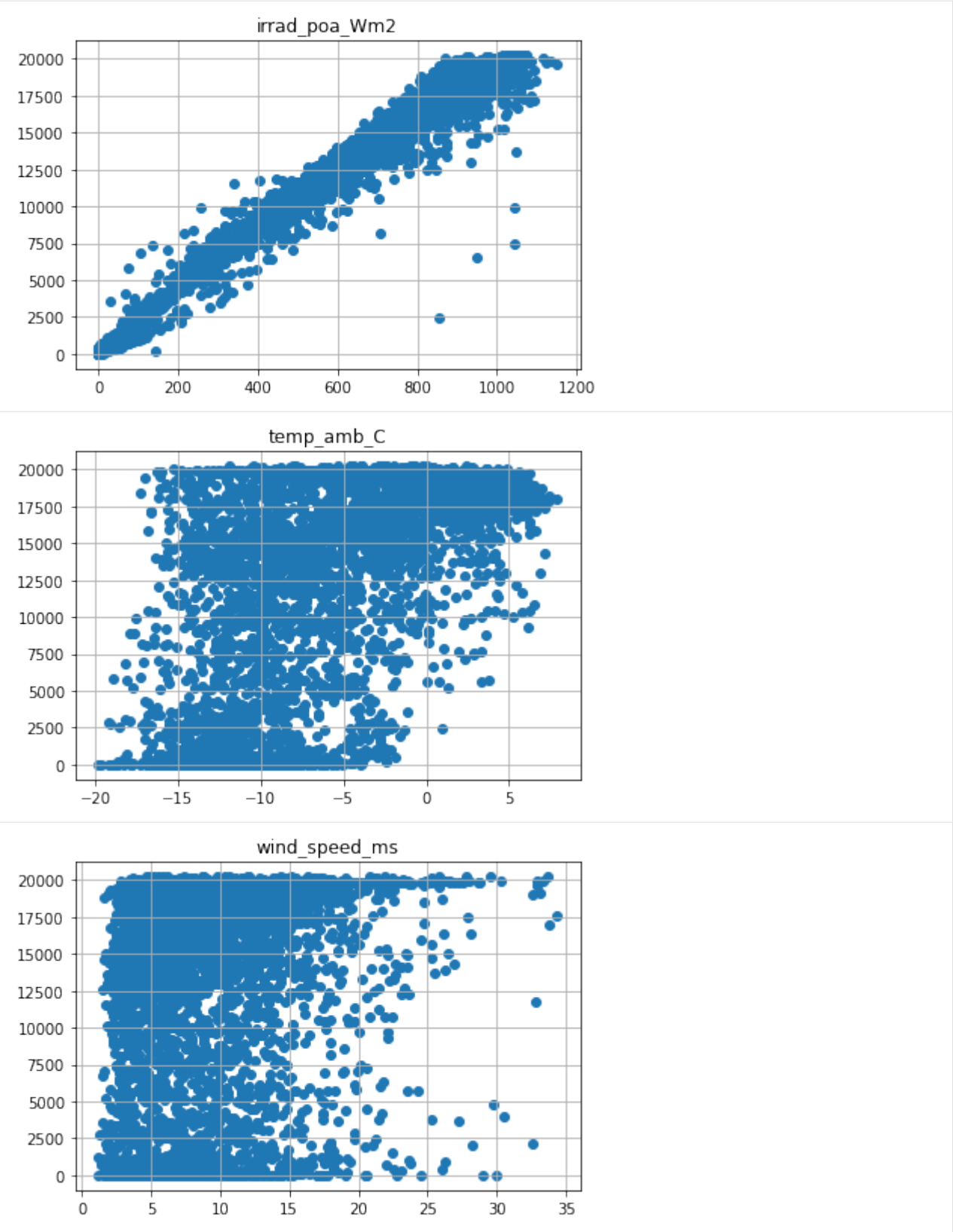
Preprocess data with `prod_inverter_clipping_filter` using the threshold model. This adds a mask column to the dataframe where `True` indicates a row to be removed by the filter.

[10]:
```
masked_prod_data = preprocess.prod_inverter_clipping_filter(prod_data_datena_d, prod_col_
↪dict, metadata, metad_col_dict, 'threshold', freq=60)

filtered_prod_data = masked_prod_data[masked_prod_data['mask'] == False].copy()
del filtered_prod_data['mask']

print(f"Detected and removed {sum(masked_prod_data['mask'])} rows with inverter clipping.
↪")
```

```
Detected and removed 24 rows with inverter clipping.
```

Visualize the power signal versus covariates (irradiance, ambient temp, wind speed) for one site

[11]:
```
temp = filtered_prod_data[filtered_prod_data['randid'] == 'R10']
for xcol in ['irrad_poa_Wm2', 'temp_amb_C', 'wind_speed_ms']:
    plt.scatter(temp[xcol], temp[prod_col_dict['powerprod']])
    plt.title(xcol)
    plt.grid()
    plt.show()
```



Add a dcsize column to production data and populate using site metadata.

[12]:
```
filtered_prod_data.head(5)
# metad.to_dict()
```

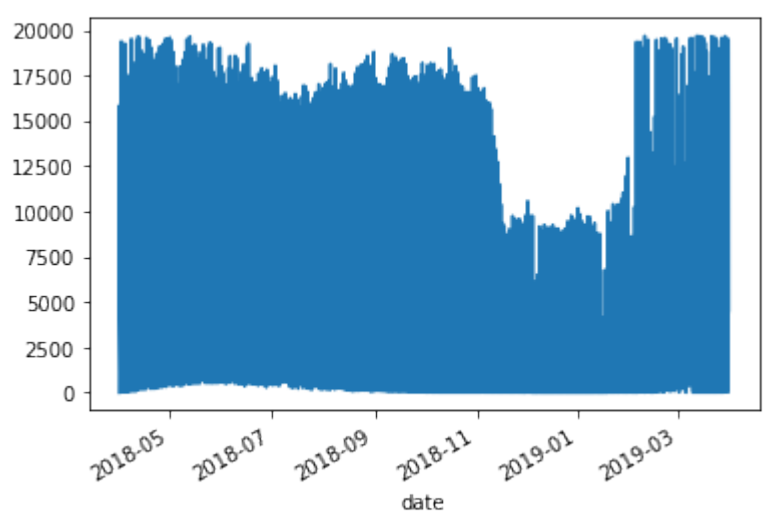
[12]:
```
                                    date randid  generated_kW   expected_kW  \
date
2018-04-01 07:00:00 2018-04-01 07:00:00    R15         0.475      0.527845
2018-04-01 08:00:00 2018-04-01 08:00:00    R15      1332.547   1685.979445
2018-04-01 09:00:00 2018-04-01 09:00:00    R15      6616.573   7343.981135
2018-04-01 10:00:00 2018-04-01 10:00:00    R15      8847.800  10429.876422
2018-04-01 11:00:00 2018-04-01 11:00:00    R15     11607.389  12981.228814

                     irrad_poa_Wm2  temp_amb_C  wind_speed_ms  temp_mod_C
date
2018-04-01 07:00:00        0.02775      16.570         4.2065     14.1270
2018-04-01 08:00:00       87.91450      16.998         4.1065     15.8610
2018-04-01 09:00:00      367.90350      20.168         4.5095     24.5745
2018-04-01 10:00:00      508.28700      21.987         4.9785     30.7740
2018-04-01 11:00:00      618.79450      23.417         4.6410     35.8695
```

[13]:
```
# Create 'dcsize' column first with site IDs
filtered_prod_data[prod_col_dict['dcsize']] = filtered_prod_data[prod_col_dict['siteid']]

# prepare dictionary for replace function
metad = metadata.copy()
metad.set_index('randid',inplace = True)

# replace site IDs with corresponding DC size
filtered_prod_data.replace(metad.to_dict(), inplace=True)
```

Visualize energy production for a specific site

[14]:
```
filtered_prod_data.loc[filtered_prod_data['randid'] == 'R15',prod_col_dict['energyprod
↪']].plot()
```

[14]:
```
<AxesSubplot: xlabel='date'>
```



Drop rows where important columns are na

[15]:
```
model_prod_data = filtered_prod_data.dropna(subset=['irrad_poa_Wm2', 'temp_amb_C', 'wind_
↪speed_ms', 'dcsize', prod_col_dict['energyprod']])
model_prod_data.head(5)
```

[15]:
```
                                 date randid  generated_kW   expected_kW  \
date
2018-04-01 07:00:00 2018-04-01 07:00:00    R15         0.475      0.527845
2018-04-01 08:00:00 2018-04-01 08:00:00    R15      1332.547   1685.979445
2018-04-01 09:00:00 2018-04-01 09:00:00    R15      6616.573   7343.981135
2018-04-01 10:00:00 2018-04-01 10:00:00    R15      8847.800  10429.876422
2018-04-01 11:00:00 2018-04-01 11:00:00    R15     11607.389  12981.228814

                     irrad_poa_Wm2  temp_amb_C  wind_speed_ms  temp_mod_C  \
date
2018-04-01 07:00:00        0.02775      16.570         4.2065     14.1270
2018-04-01 08:00:00       87.91450      16.998         4.1065     15.8610
2018-04-01 09:00:00      367.90350      20.168         4.5095     24.5745
2018-04-01 10:00:00      508.28700      21.987         4.9785     30.7740
2018-04-01 11:00:00      618.79450      23.417         4.6410     35.8695

                     dcsize
date
2018-04-01 07:00:00   25000
2018-04-01 08:00:00   25000
2018-04-01 09:00:00   25000
2018-04-01 10:00:00   25000
2018-04-01 11:00:00   25000
```

Dynamic linear modeling

Here we use the AIT model to calculate expected energy based on field data. This is appended to model_prod_data as a new column named 'AIT'.

[16]:
```
model_prod_data = AIT.AIT_calc(model_prod_data, prod_col_dict)
```

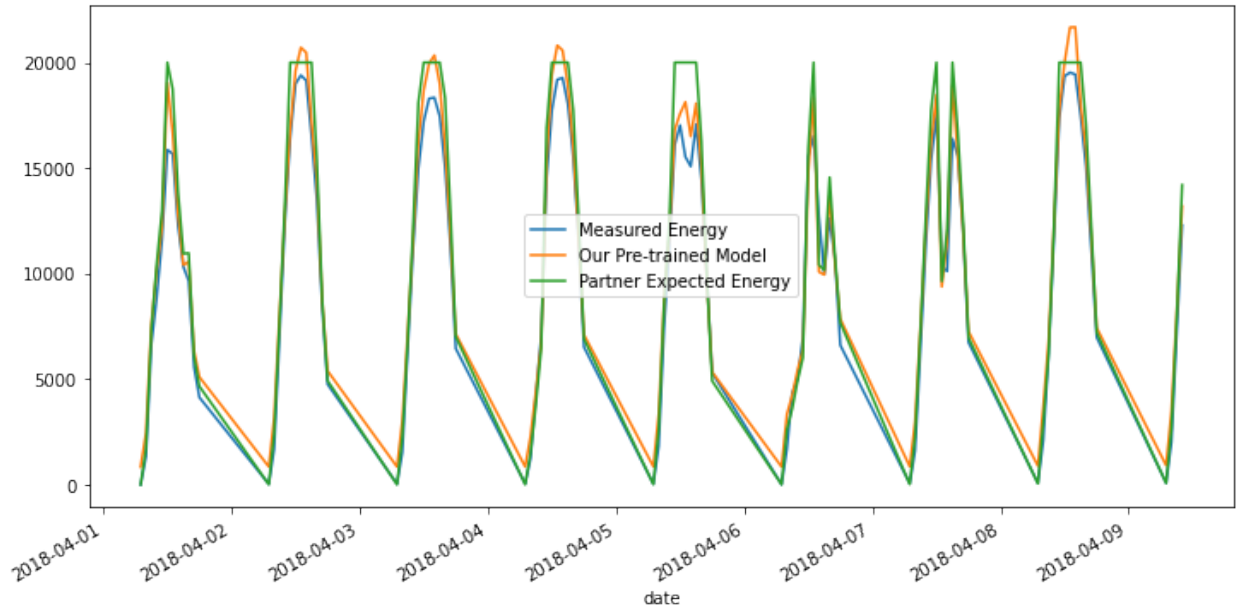
```
The fit has an R-squared of -569972861.7979063 and a log RMSE of 9.515344605202777
```

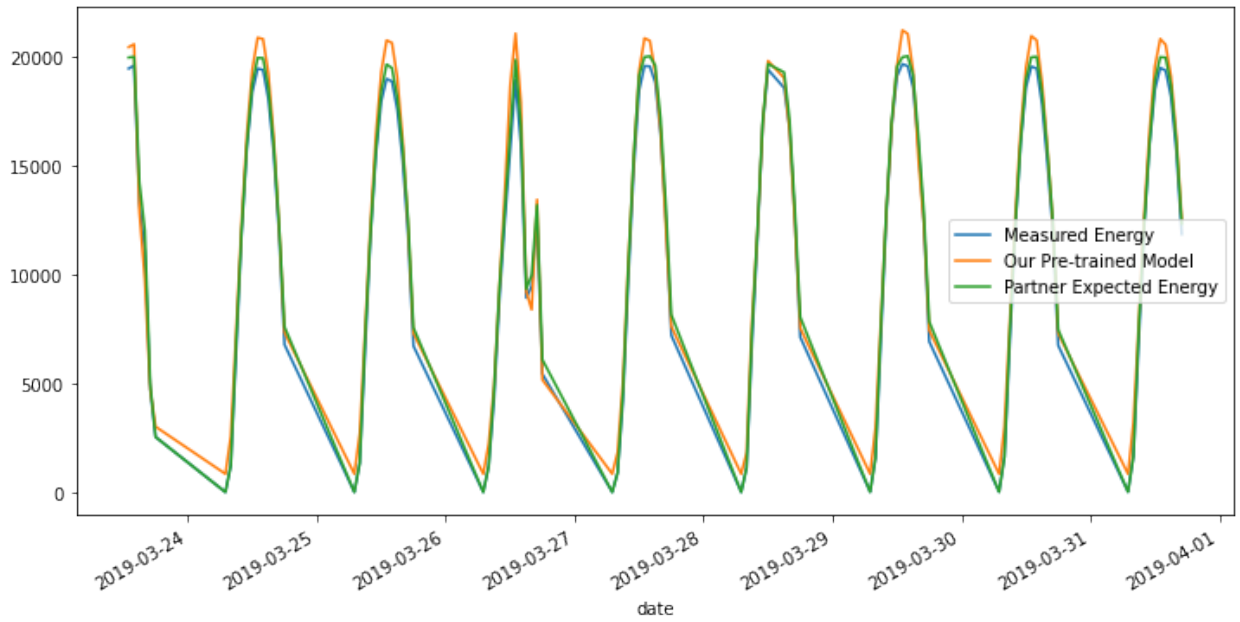
Visualize results

We visualize the measured hourly energy, our pre-trained model's expected energy, and the results of a partner-produced expected energy over various time-scales.

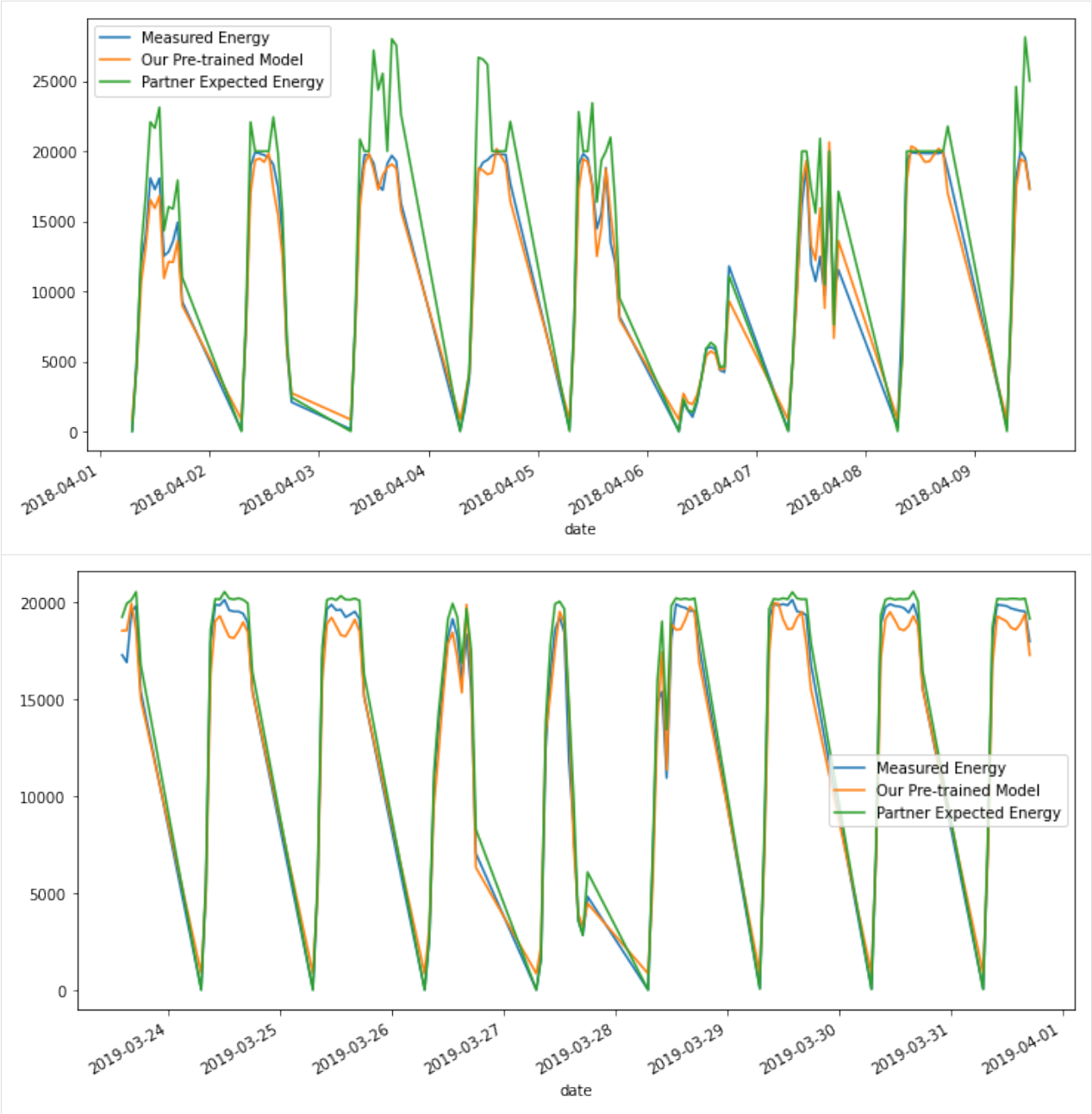
[17]:
```
# defining a plotting utility function
def plot(data, randid, from_idx=0, to_idx=1000):
    data.copy()
    # Just making the visualization labels better here.. for this example's data␣
↪specific.
    data.rename(columns={'generated_kW': 'Measured Energy',
                         'AIT': 'Our Pre-trained Model',
                         'expected_kW': 'Partner Expected Energy'}, inplace=True)
    data[data['randid']==randid][['Measured Energy', 'Our Pre-trained Model', 'Partner␣
↪Expected Energy']].iloc[from_idx:to_idx].plot(figsize=(12,6))
```

[18]:
```
plot(model_prod_data, "R15", from_idx=0, to_idx=100)
plot(model_prod_data, "R15", from_idx=-100, to_idx=-1)
```





[19]:
```
plot(model_prod_data, "R10", from_idx=0, to_idx=100)
plot(model_prod_data, "R10", from_idx=-100, to_idx=-1)
```



1.2.5 Timeseries IV Simulation

[90]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sys
import os

from pvops.iv import timeseries_simulator
from pvops.timeseries import preprocess
```

(continues on next page)

(continued from previous page)

```
from pvops.timeseries.models import linear
from pvops.text2time import utils as t2t_utils, preprocess as t2t_preprocess
```

Next we load in example data. We only simulate for irradiance above 200 select one of the two sites to focus on. Additionally, we drop rows without the data needed for simulation.

[91]:
```
example_prodpath = os.path.join('example_data', 'example_prod_with_covariates.csv')
env_df = pd.read_csv(example_prodpath)
env_df.index = pd.to_datetime(env_df["date"])
env_df = env_df.sort_index()

# Only simulate where irradiance > 200
env_df = env_df[env_df['irrad_poa_Wm2'] > 200]
# Two sites have data here so we choose one
env_df = env_df[env_df['randid'] == 'R15']
# Remove any NaN environmental specifications
env_df = env_df.dropna(subset=['irrad_poa_Wm2','temp_amb_C'])
env_df.head()
```

[91]:
```
                                    date randid  generated_kW   expected_kW  \
date
2018-04-01 09:00:00  2018-04-01 09:00:00    R15      6616.573   7343.981135
2018-04-01 10:00:00  2018-04-01 10:00:00    R15      8847.800  10429.876422
2018-04-01 11:00:00  2018-04-01 11:00:00    R15     11607.389  12981.228814
2018-04-01 12:00:00  2018-04-01 12:00:00    R15     15860.138  20000.000000
2018-04-01 13:00:00  2018-04-01 13:00:00    R15     15672.136  18737.585012

                     irrad_poa_Wm2  temp_amb_C  wind_speed_ms  temp_mod_C
date
2018-04-01 09:00:00       367.9035     20.1680         4.5095     24.5745
2018-04-01 10:00:00       508.2870     21.9870         4.9785     30.7740
2018-04-01 11:00:00       618.7945     23.4170         4.6410     35.8695
2018-04-01 12:00:00       992.8930     24.3280         4.3235     44.8640
2018-04-01 13:00:00       861.8760     25.1885         5.1810     44.3385
```

[92]:
```
metadata = pd.DataFrame()
metadata['randid'] = ['R15', 'R10']
metadata.head()
```

[92]:
```
  randid
0    R15
1    R10
```

[93]:
```
#Format for dictionaries is {pvops variable: user-specific column names}
prod_col_dict = {
    'siteid': 'randid',
    'timestamp': 'date',
    'powerprod': 'generated_kW',
    'irradiance':'irrad_poa_Wm2',
    'temperature':'temp_amb_C', # Optional parameter, used by one of the modeling␣
↪structures
    'baseline': 'IEC_pstep', #user's name choice for new column (baseline expected energy␣
↪defined by user or calculated based on IEC)
```

(continues on next page)

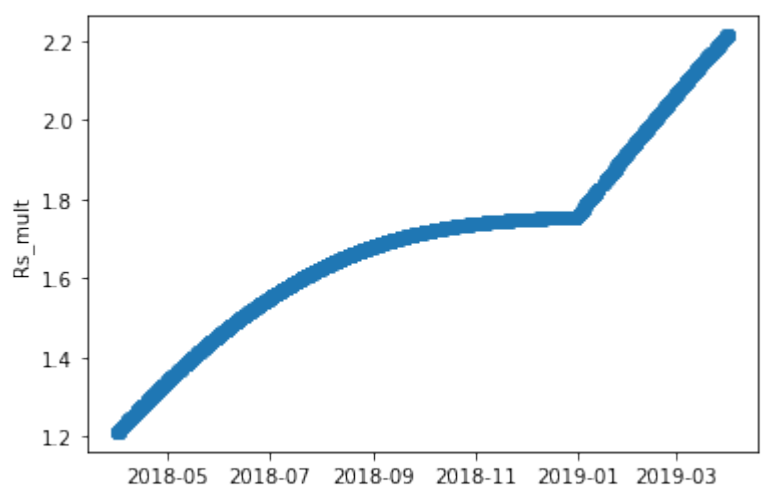
(continued from previous page)

```
    'dcsize': 'dcsize', #user's name choice for new column (System DC-size, extracted␣
↪from meta-data)
    'compared': 'Compared', #user's name choice for new column
    'energy_pstep': 'Energy_pstep' #user's name choice for new column
    }

metad_col_dict = {'siteid': 'randid'}
```

[94]:
```
failureA = timeseries_simulator.TimeseriesFailure()
longterm_fcn_dict = {
    'Rs_mult': "degrade"
}
annual_fcn_dict = {
    'Rs_mult': lambda x : ( 0.3 * np.sin(np.pi * x) )
}

failureA.trend(longterm_fcn_dict=longterm_fcn_dict,
               annual_fcn_dict=annual_fcn_dict,
               degradation_rate=1.005)

iv_col_dict = {'irradiance': 'irrad_poa_Wm2',
               'temperature': 'temp_amb_C'
               }

env_df['identifier'] = env_df.index.strftime("%Y-%m-%d %H:%M:%S")

time_simulator = timeseries_simulator.IVTimeseriesGenerator()
condition_dicts = time_simulator.generate(env_df, [failureA], iv_col_dict, 'identifier',␣
↪plot_trends=False)
```

[95]:
```
plt.scatter(time_simulator.specs_df.index, time_simulator.specs_df['Rs_mult'])
plt.ylabel('Rs_mult')
```

[95]:
```
Text(0, 0.5, 'Rs_mult')
```



[96]:
```
time_simulator.add_time_conditions('complete', nmods=12)
time_simulator.simulate()
```

```
Simulating cells:   0%|          | 0/3307 [00:00<?, ?it/s]/home/klbonne/.local/bin/
↪anaconda3/envs/pvops_dev/lib/python3.8/site-packages/scipy/optimize/_zeros_py.py:466:␣
↪RuntimeWarning: some failed to converge after 100 iterations
  warnings.warn(msg, RuntimeWarning)
Simulating cells: 100%|| 3307/3307 [01:17<00:00, 42.75it/s]
Adding up simulations: 100%|| 3306/3306 [00:53<00:00, 61.60it/s]
Adding up other definitions: 100%|| 3307/3307 [00:00<00:00, 29882.31it/s]
```

[97]:
```
sims_df = time_simulator.sims_to_df(focus=['string'], cutoff=True)
```

[98]:
```
sims_df.head()
```

[98]:
```
                                              current  \
0  [3.3373719452150903, 3.3359903398092783, 3.334...
1  [4.614599256674858, 4.612678148222023, 4.61080...
2  [5.621465807867254, 5.6191204837065545, 5.6168...
3  [9.020136734061941, 9.016314598640065, 9.01258...
4  [7.834888228861011, 7.831598650286882, 7.82838...

                                              voltage         E        T  \
0  [3.836930773104541e-12, 11.97867132786855, 23...  367.9035  20.1680
1  [3.836930773104541e-12, 12.058127583686556, 23...  508.2870  21.9870
2  [3.8331560148208155e-12, 12.093588347866122, 2...  618.7945  23.4170
3  [3.836930773104541e-12, 12.288852807799348, 24...  992.8930  24.3280
4  [3.836930773104541e-12, 12.182321719206548, 24...  861.8760  25.1885

                      mode   level
0  str_2018-04-01 09:00:00  string
1  str_2018-04-01 10:00:00  string
2  str_2018-04-01 11:00:00  string
3  str_2018-04-01 12:00:00  string
4  str_2018-04-01 13:00:00  string
```

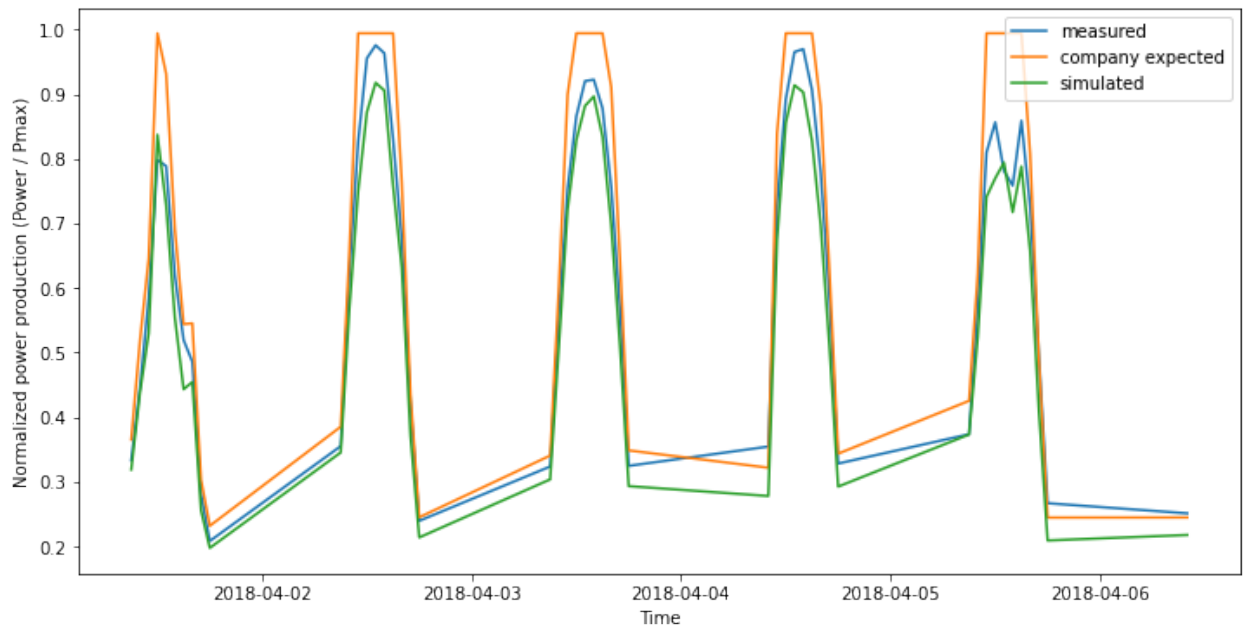
[99]:
```
pmaxs = np.array([max(np.array(row['current']) * np.array(row['voltage']))
            for ind,row in sims_df.iterrows()])

env_df["simulated_power"] = pmaxs
```

[100]:
```
def plot(input_df, npts_viz=50):
    df = input_df.copy()
    df['generated_kW'] = df['generated_kW'] / df['generated_kW'].max()
    df['expected_kW'] = df['expected_kW'] / df['expected_kW'].max()
    df['simulated_power'] = df['simulated_power'] / df['simulated_power'].max()

    plt.figure(figsize=(12,6))
    plt.plot(df.index[0:npts_viz], df['generated_kW'].iloc[0:npts_viz], label='measured')
    plt.plot(df.index[0:npts_viz], df['expected_kW'].iloc[0:npts_viz], label='company␣
↪expected')
    plt.plot(df.index[0:npts_viz], df['simulated_power'].iloc[0:npts_viz], label=
↪'simulated')
```

(continues on next page)

(continued from previous page)

```
    plt.legend()
    plt.xlabel("Time")
    plt.ylabel("Normalized power production (Power / Pmax)")
    plt.show()

plot(env_df)
```



Timeseries simulations

Using the environmental conditions and the physics-based estimation as input, we inference an output.

[101]:
```
prod_data_converted = t2t_preprocess.prod_date_convert(env_df, prod_col_dict)
prod_data_datena_d, _ = t2t_preprocess.prod_nadate_process(prod_data_converted, prod_col_
↪dict, pnadrop=True)
prod_data_datena_d.index = prod_data_datena_d[prod_col_dict['timestamp']]
```

[102]:
```
masked_prod_data = preprocess.prod_inverter_clipping_filter(prod_data_datena_d, prod_col_
↪dict, metadata, metad_col_dict, 'threshold', freq=60)

filtered_prod_data = masked_prod_data.loc[masked_prod_data['mask'] == False,:]
```

```
/home/klbonne/Documents/GitHub/pvOps/pvops/timeseries/preprocess.py:322: FutureWarning:␣
↪In a future version, `df.iloc[:, i] = newvals` will attempt to set the values inplace␣
↪instead of always setting a new array. To retain the old behavior, use either `df[df.
↪columns[i]] = newvals` or, if columns are non-unique, `df.isetitem(i, newvals)`
  prod_df.loc[site_prod_mask, "mask"] = pvanalytics.features.clipping.threshold(
```

[103]:
```
model_prod_data = filtered_prod_data.dropna(subset=['irrad_poa_Wm2', 'temp_amb_C', 'wind_
↪speed_ms']+[prod_col_dict['powerprod']])
```

[104]:
```
from sklearn.metrics import mean_squared_error, r2_score

def plot_inference(model, prod_col_dict, data_split='test', npts=50):

    def print_info(real,pred,name):
        mse = mean_squared_error(real, pred)
        r2 = r2_score(real, pred)
        print(f'[{name}] Mean squared error: %.2f'
                % mse)
        print(f'[{name}] Coefficient of determination: %.2f'
                % r2)

    fig,(ax) = plt.subplots(figsize=(14,8))

    if data_split == 'test':
        df = test_df
    elif data_split == 'train':
        df = train_df

    measured = model.estimators['OLS'][f'{data_split}_y'][:npts]

    ax2 = ax.twinx()
    ax2.plot(model.estimators['OLS'][f'{data_split}_index'][:npts], df[prod_col_dict[
↪'irradiance']].values[:npts], 'k', label='irradiance')

    ax.plot(model.estimators['OLS'][f'{data_split}_index'][:npts], df['expected_kW'].
↪values[:npts], label='partner_expected')
    print_info(measured, df['expected_kW'].values[:npts], 'partner_expected')

    ax.plot(model.estimators['OLS'][f'{data_split}_index'][:npts], measured, label=
↪'measured')
    for name, info in model.estimators.items():
        predicted = model.estimators[name][f'{data_split}_prediction'][:npts]
        ax.plot(model.estimators[name][f'{data_split}_index'][:npts], predicted,␣
↪label=name)
        print_info(measured, predicted, name)

    ax2.set_ylabel("Irradiance (W/m2)")
    ax.set_ylabel("Power (W)")
    ax.set_xlabel('Time')
    handles, labels = [(a+b) for a, b in zip(ax.get_legend_handles_labels(), ax2.get_
↪legend_handles_labels())]
    ax.legend(handles, labels, loc='best')
    plt.show()
```

Power modeling using environmental conditions

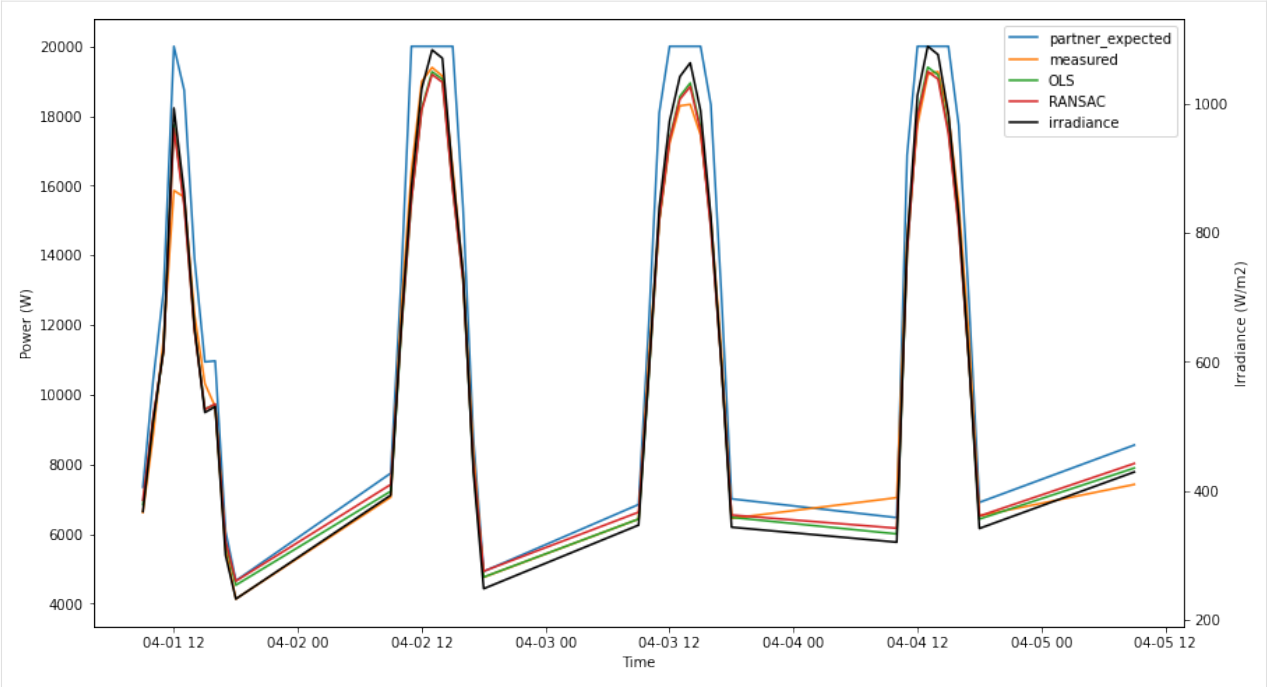
[105]:
```
# Make sure to only pass data for one site! If sites are very similar, you can consider␣
↪providing both sites.
model, train_df, test_df = linear.modeller(
                                            prod_col_dict,
                                            kernel_type='default',
                                            time_weighted='month',
                                            X_parameters=['irrad_poa_Wm2', 'temp_amb_C'],
                                            Y_parameter='generated_kW',
                                            prod_df=model_prod_data,
                                            test_split=0.05,
                                            degree=3,
                                            verbose=1)
```

```
train {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
test {3}

Begin training.
[OLS] Mean squared error: 933524.95
[OLS] Coefficient of determination: 0.96
[OLS] 24 coefficient trained.
[RANSAC] Mean squared error: 1052555.85
[RANSAC] Coefficient of determination: 0.95

Begin testing.
[OLS] Mean squared error: 510178.49
[OLS] Coefficient of determination: 0.98
[OLS] 24 coefficient trained.
[RANSAC] Mean squared error: 1743431.22
[RANSAC] Coefficient of determination: 0.93
```

```
/home/klbonne/.local/bin/anaconda3/envs/pvops_dev/lib/python3.8/site-packages/
↪statsmodels/regression/linear_model.py:1934: RuntimeWarning: divide by zero␣
↪encountered in double_scalars
  return np.sqrt(eigvals[0]/eigvals[-1])
```

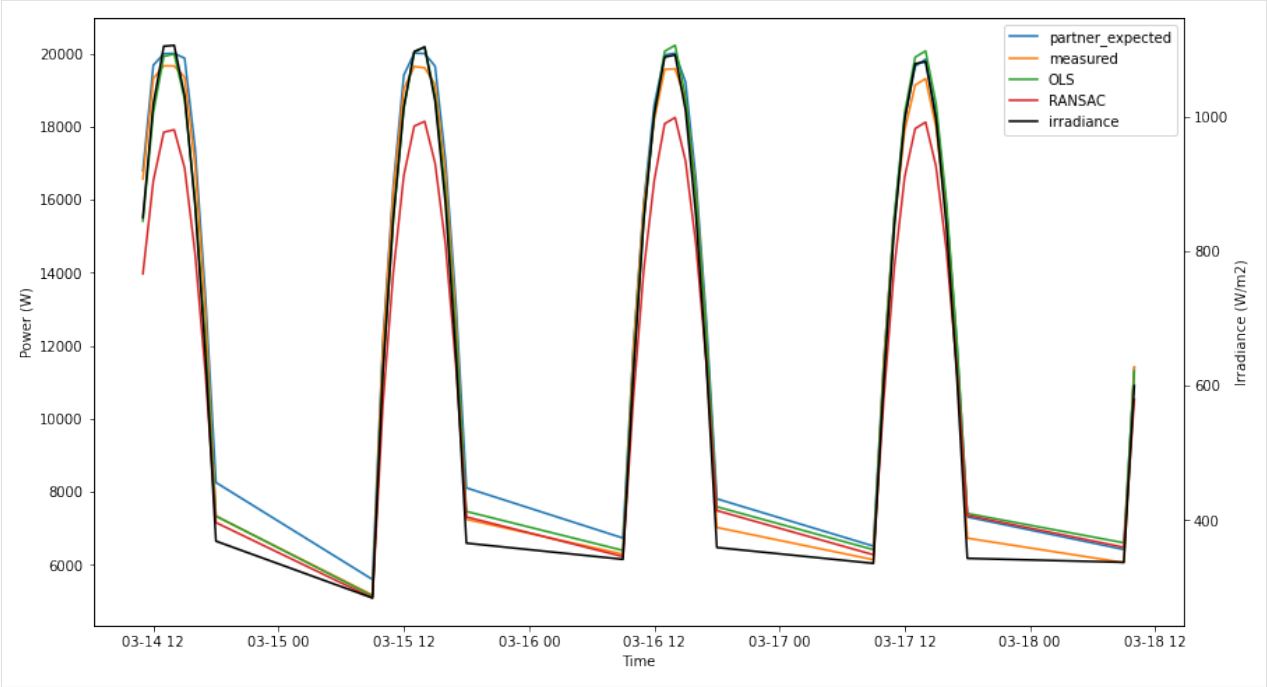
[106]:
```
plot_inference(model, prod_col_dict, data_split='train', npts=40)
```

```
[partner_expected] Mean squared error: 3433925.51
[partner_expected] Coefficient of determination: 0.85
[OLS] Mean squared error: 270424.03
[OLS] Coefficient of determination: 0.99
[RANSAC] Mean squared error: 268835.82
[RANSAC] Coefficient of determination: 0.99
```



[107]:
```
plot_inference(model, prod_col_dict, data_split='test', npts=40)
```

```
[partner_expected] Mean squared error: 180731.98
[partner_expected] Coefficient of determination: 0.99
[OLS] Mean squared error: 275562.47
[OLS] Coefficient of determination: 0.99
[RANSAC] Mean squared error: 2293599.51
[RANSAC] Coefficient of determination: 0.90
```



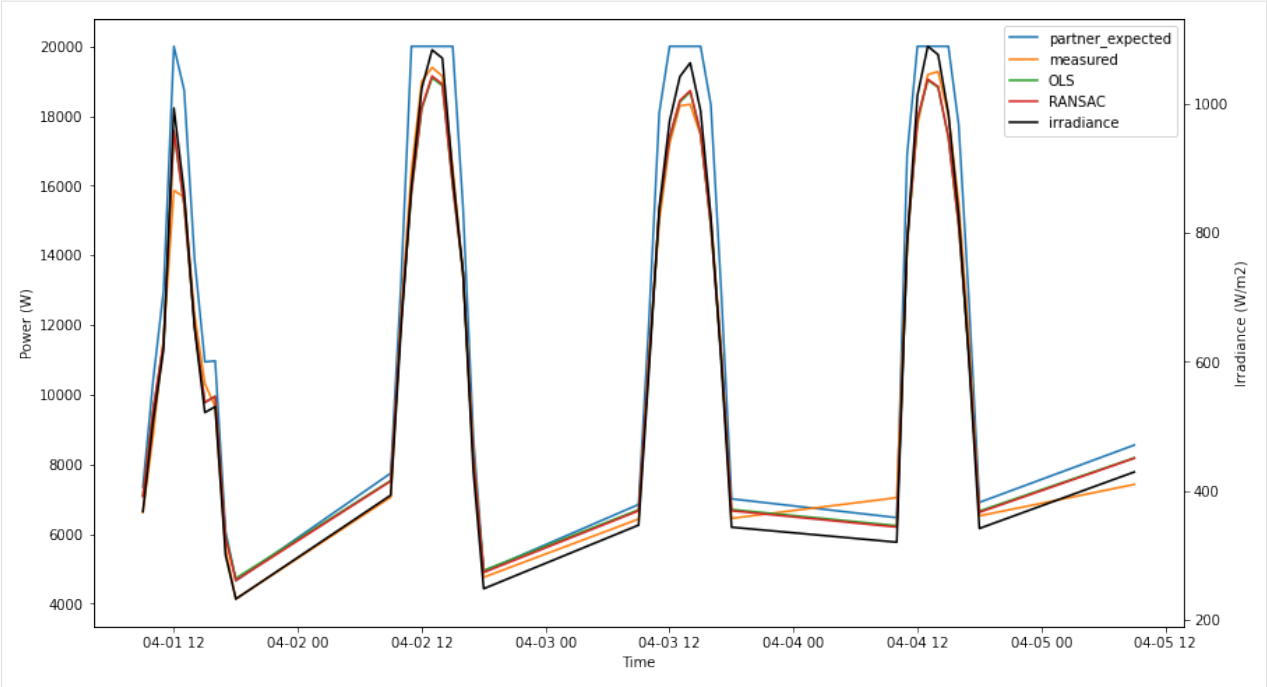
Power modeling using physics simulations as input

[108]:
```
# Make sure to only pass data for one site! If sites are very similar, you can consider␣
↪providing both sites.
model, train_df, test_df = linear.modeller(
                                            prod_col_dict,
                                            kernel_type='default',
                                            time_weighted='month',
                                            X_parameters=['irrad_poa_Wm2', 'temp_amb_C',
↪'simulated_power'],
                                            Y_parameter='generated_kW',
                                            prod_df=model_prod_data,
                                            test_split=0.05,
                                            degree=3,
                                            verbose=1)
```

```
train {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
test {3}

Begin training.
[OLS] Mean squared error: 754429.15
[OLS] Coefficient of determination: 0.97
[OLS] 36 coefficient trained.
[RANSAC] Mean squared error: 788074.79
[RANSAC] Coefficient of determination: 0.97

Begin testing.
[OLS] Mean squared error: 482864.39
[OLS] Coefficient of determination: 0.98
[OLS] 36 coefficient trained.
[RANSAC] Mean squared error: 1005442.42
[RANSAC] Coefficient of determination: 0.96
```

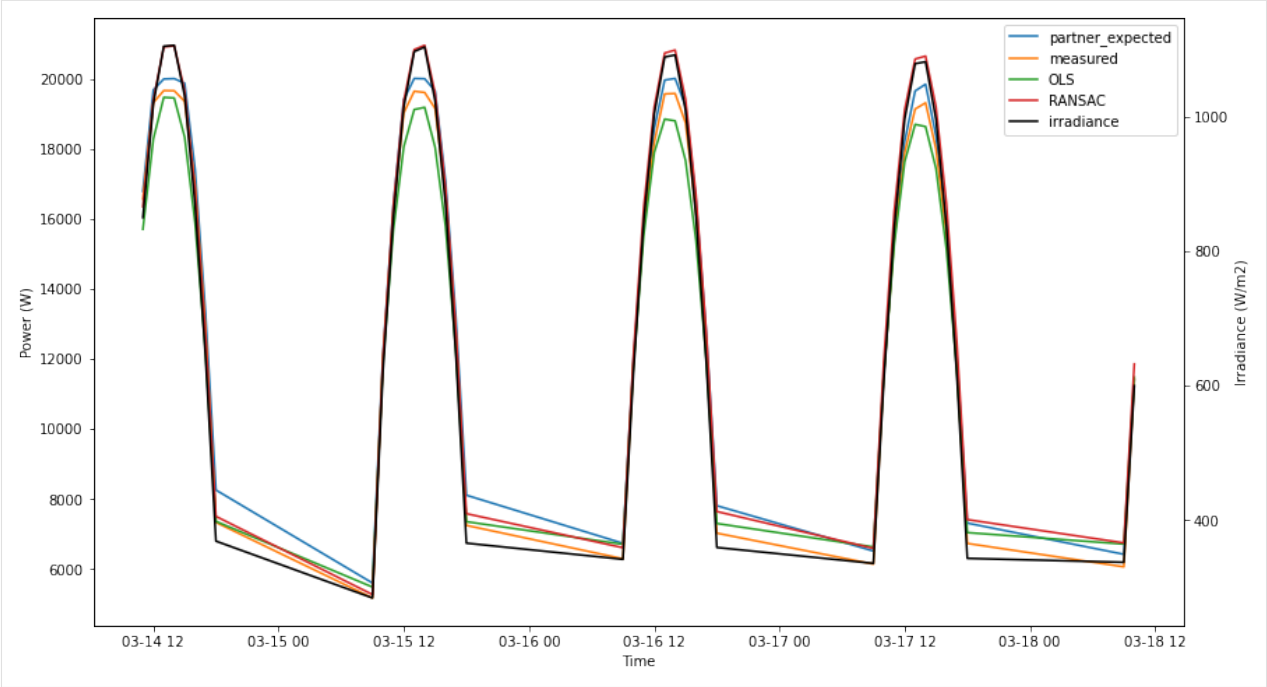
[109]:
```
plot_inference(model, prod_col_dict, data_split='train', npts=40)
```

```
[partner_expected] Mean squared error: 3433925.51
[partner_expected] Coefficient of determination: 0.85
[OLS] Mean squared error: 241325.46
[OLS] Coefficient of determination: 0.99
[RANSAC] Mean squared error: 232528.26
[RANSAC] Coefficient of determination: 0.99
```



[110]:
```
plot_inference(model, prod_col_dict, data_split='test', npts=40)
```

```
[partner_expected] Mean squared error: 180731.98
[partner_expected] Coefficient of determination: 0.99
[OLS] Mean squared error: 355754.91
[OLS] Coefficient of determination: 0.98
[RANSAC] Mean squared error: 566091.79
[RANSAC] Coefficient of determination: 0.98
```



1.2.6 IV Simulator Tutorial

A python class which simulates PV faults in IV curves.

Step 1: Start off by instantiating the object.

[1]:
```
from pvops.iv import simulator

sim = simulator.Simulator()
```

Step 2: Definition of Faults

Two methods exist to define a fault: 1. Use a pre-existing definition (defined by authors) by calling `add_preset_conditions`. 2. Manually define a fault by calling `add_manual_condition`.

Preset definition of faults

[2]:
```
heavy_shading = {'identifier':'heavy_shade',
                    'E': 400,
                    'Tc': 20}
light_shading = {'identifier':'light_shade',
                     'E': 800}
sim.add_preset_conditions('landscape', heavy_shading, rows_aff = 2)
sim.add_preset_conditions('portrait', heavy_shading, cols_aff = 2)
sim.add_preset_conditions('pole', heavy_shading, light_shading = light_shading, width =␣
↪2, pos = None)

sim.print_info()
```

```
Condition list: (Cell definitions)
        [pristine]: 1 definition(s)
        [heavy_shade]: 1 definition(s)
        [light_shade]: 1 definition(s)

Modcell types: (Cell mappings on module)
        [pristine]: 1 definition(s)
        [landscape_2rows]: 1 definition(s)
        [portrait_2cols]: 1 definition(s)
        [pole_2width]: 1 definition(s)

String definitions (Series of modcells)
        No instances.
```

Manual definition of faults

To define a fault manually, you must provide two specifications: 1. Mapping of cells onto a module, which we call a `modcell`. 2. Definition of cell conditions, stored in `condition_dict`.

[3]:
```
modcells = { 'another_example':  [[0,0,0,0,0,0,0,0,0,0,  # Using 2D list (aka, multiple␣
↪conditions as input)
                                    1,1,1,1,1,1,1,1,1,1,
                                    1,1,1,0,0,0,0,1,1,1,
                                    1,1,1,0,0,0,0,1,1,1,
                                    1,1,1,0,0,0,0,1,1,1,
                                    0,0,0,0,0,0,0,0,0,0],

                                  [1,1,1,1,1,1,1,1,1,1,
                                    0,0,0,0,0,0,0,0,0,0,
                                    0,0,0,1,1,1,1,0,0,0,
                                    0,0,0,1,1,1,1,0,0,0,
                                    0,0,0,1,1,1,1,0,0,0,
                                    1,1,1,1,1,1,1,1,1,1]]
            }
condition_dict = {0: {},
                  1: {'identifier': 'heavy_shade',
                      'E': 405,
                     }
                 }
sim.add_manual_conditions(modcells, condition_dict)

sim.print_info()
```

```
Condition list: (Cell definitions)
        [pristine]: 1 definition(s)
        [heavy_shade]: 2 definition(s)
        [light_shade]: 1 definition(s)

Modcell types: (Cell mappings on module)
        [pristine]: 1 definition(s)
        [landscape_2rows]: 1 definition(s)
        [portrait_2cols]: 1 definition(s)
        [pole_2width]: 1 definition(s)
        [another_example]: 2 definition(s)

String definitions (Series of modcells)
        No instances.
```

Step 3: Generate many samples via latin hypercube sampling

```
Pass in dictionaries which describe a distribution.
    {PARAMETER: {'mean': MEAN_VAL,
                 'std': STDEV_VAL,
                 'low': LOW_VAL,
                 'upp': UPP_VAL
                }
    }

PARAMETER: parameter defined in condition_dict
```

If all values are provided, a truncated gaussian distribution is used

If *low* and *upp* not specified, then a gaussian distribution is used

[4]:
```
N = 10
dicts = {'E': {'mean': 400,
               'std': 500,
               'low': 200,
               'upp': 600},

         'Tc':{'mean': 30,
               'std': 10}}


sim.generate_many_samples('heavy_shade', N, distributions = dicts)

dicts = {'E': {'mean': 800,
               'std': 500,
               'low': 600,
               'upp': 1000}}

sim.generate_many_samples('light_shade', N, distributions = dicts)

sim.print_info()
```

```
Condition list: (Cell definitions)
        [pristine]: 1 definition(s)
        [heavy_shade]: 12 definition(s)
        [light_shade]: 11 definition(s)

Modcell types: (Cell mappings on module)
        [pristine]: 1 definition(s)
        [landscape_2rows]: 1 definition(s)
        [portrait_2cols]: 1 definition(s)
        [pole_2width]: 1 definition(s)
        [another_example]: 2 definition(s)

String definitions (Series of modcells)
        No instances.
```

Step 4: Define a string as an assimilation of modcells

Define a dictionary with `keys` as the string name and values as a list of module names.

```
{STRING_IDENTIFIER: LIST_OF_MODCELL_NAMES}
```

Use `sim.modcells.keys()` to get list of modules defined thusfar, or look at *modcell types* list in function call `sim.print_info()`

[3]:
```
sim.modcells.keys()
```

[3]:
```
dict_keys(['pristine', 'landscape_2rows', 'portrait_2cols', 'pole_2width'])
```

[5]:
```
sim.build_strings({'pole_bottom_mods': ['pristine', 'pristine', 'pristine', 'pristine',
↪'pristine', 'pristine',
                                        'pole_2width', 'pole_2width', 'pole_2width',
↪'pole_2width', 'pole_2width', 'pole_2width'],
                   'portrait_2cols_3bottom_mods': ['pristine', 'pristine', 'pristine',
↪'pristine', 'pristine', 'pristine',
                                                   'pristine', 'pristine', 'pristine',
↪'portrait_2cols', 'portrait_2cols', 'portrait_2cols']})
```

Step 5: Simulate!

`sim.simulate()` simulates all cells, substrings, modules, and strings defined in steps 2 - 4

[6]:
```
import time
start_t = time.time()
sim.simulate()
print(f'\nSimulations completed after {round(time.time()-start_t,2)} seconds')

sim.print_info()
```

```
Simulating cells:   0%|
↪         | 0/3 [00:00<?, ?it/s]c:\users\mwhopwo\appdata\local\programs\python\python36\
↪lib\site-packages\scipy\optimize\zeros.py:463: RuntimeWarning: some failed to converge
↪after 100 iterations
  warnings.warn(msg, RuntimeWarning)
Simulating cells: 100%|| 3/3 [00:01<00:00,  2.93it/s]
Adding up simulations: 100%|| 2/2 [00:09<00:00,  4.66s/it]
Adding up other definitions: 100%|| 5/5 [00:01<00:00,  3.62it/s]


Simulations completed after 11.74 seconds
Condition list: (Cell definitions)
        [pristine]: 1 definition(s)
        [heavy_shade]: 12 definition(s)
        [light_shade]: 11 definition(s)

Modcell types: (Cell mappings on module)
        [pristine]: 1 definition(s)
        [landscape_2rows]: 1 definition(s)
        [portrait_2cols]: 1 definition(s)
        [pole_2width]: 1 definition(s)
```

(continues on next page)

(continued from previous page)

```
        [another_example]: 2 definition(s)

String definitions (Series of modcells)
        [pole_bottom_mods]: 132 definition(s)
        [portrait_2cols_3bottom_mods]: 12 definition(s)
```

Step 6: Visualization suite

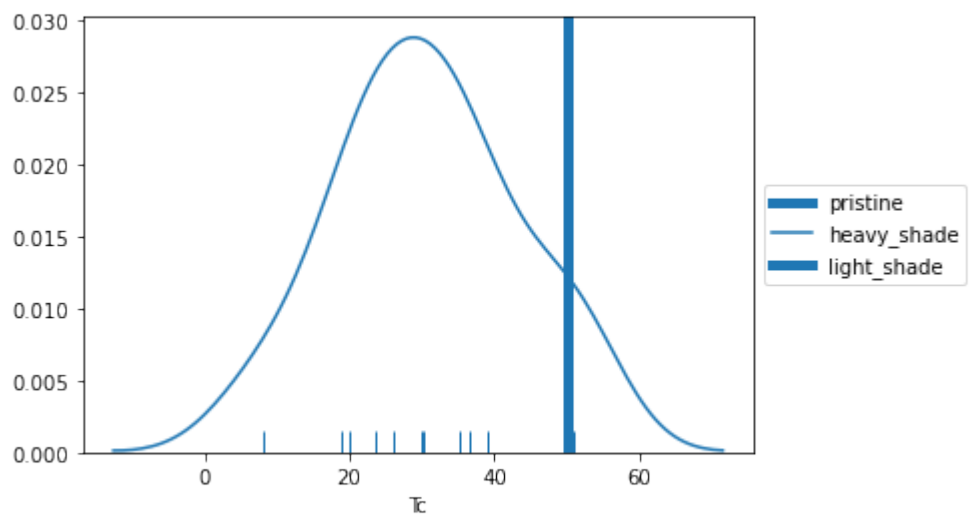
1) Plot distribution of cell-condition parameter definitions defined in steps 2 and 3

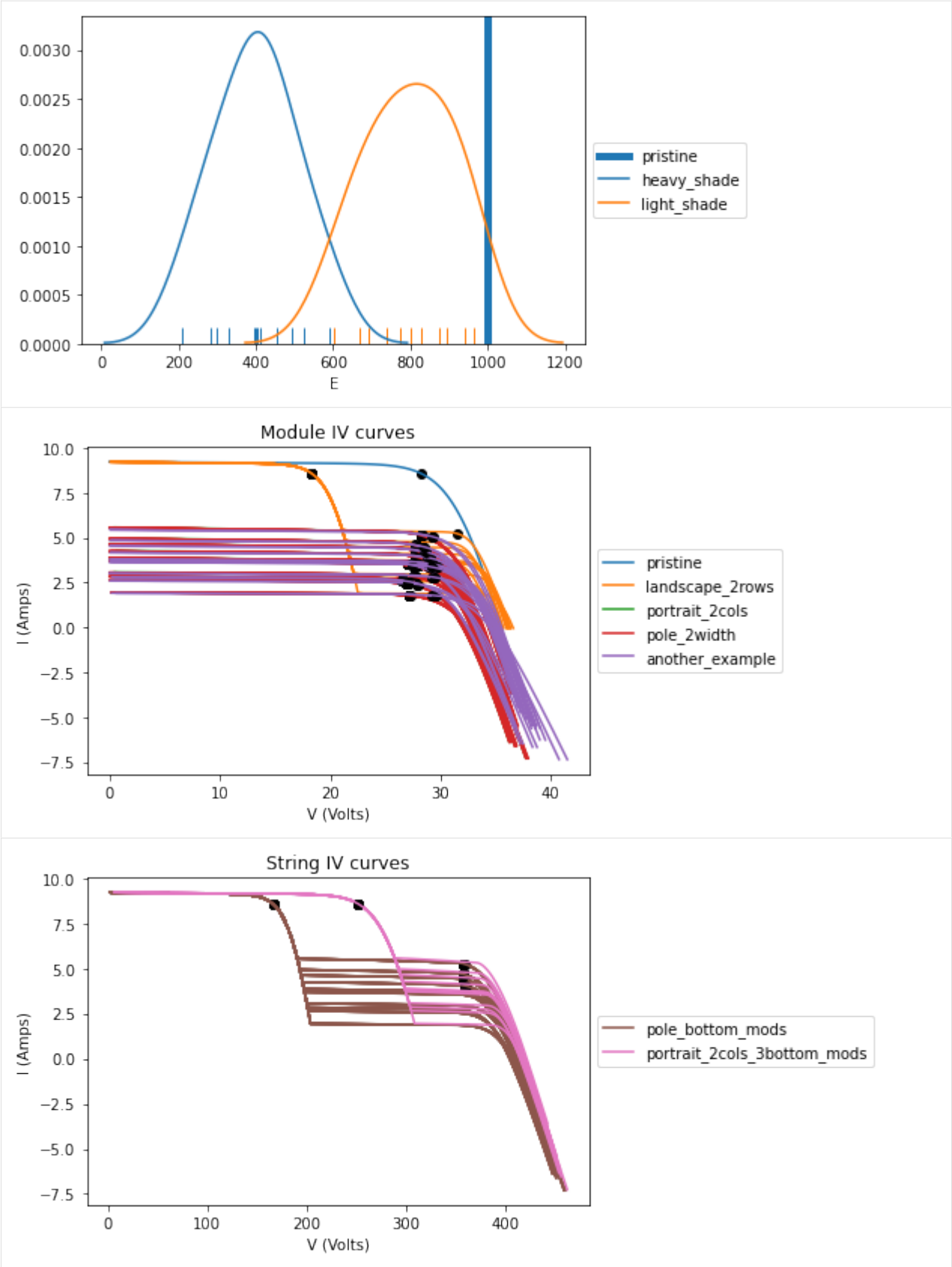
1a) TODO: The truncated gaussians should show cutoff at tails

2) Plot module-level IV curves

3) Plot string-level IV curves

[7]:
```
sim.visualize()
```




0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
0
200
400
600
800
1000
1200
E
pristine
heavy_shade
light_shade
Module IV curves
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
I (Amps)
0
10
20
30
40
V (Volts)
pristine
landscape_2rows
portrait_2cols
pole_2width
another_example
String IV curves
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
I (Amps)
0
100
200
300
400
V (Volts)
pole_bottom_mods
portrait_2cols_3bottom_mods

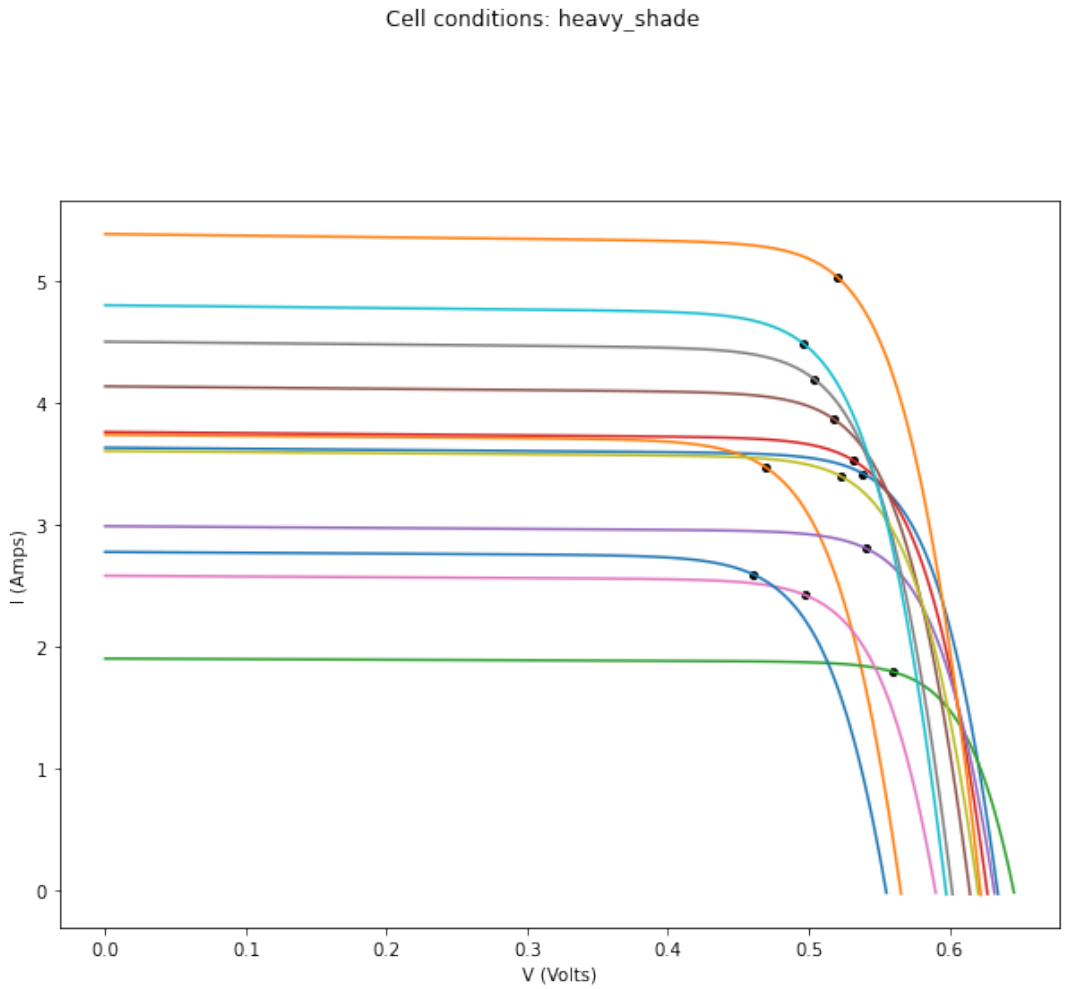
Step 6 cont'd: Visualize cell IV curves and settings

```
visualize_cell_level_traces(cell_identifier, cutoff = True, table = True)
```

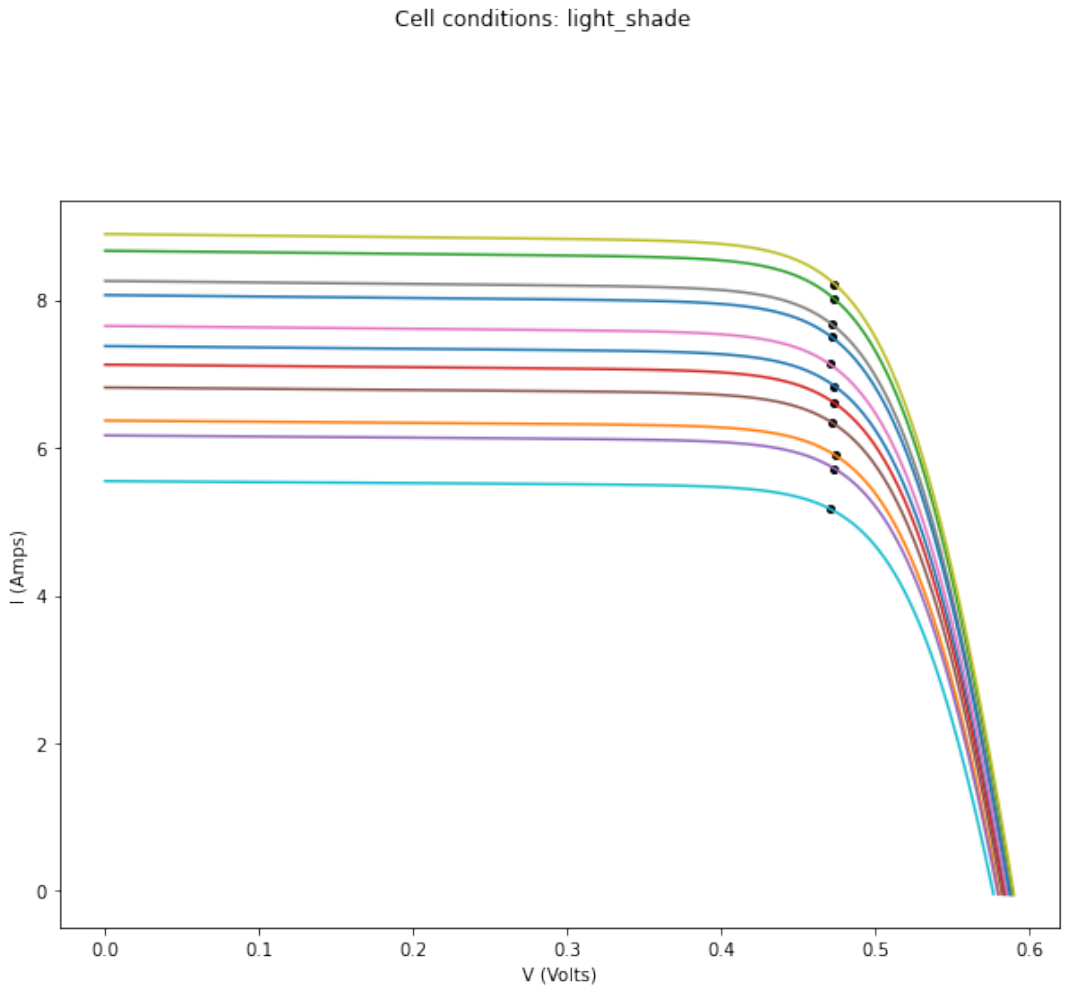
Automatically turns off table if the cell_identifier's number of definitions > 20

[8]:
```
sim.visualize_cell_level_traces('heavy_shade', cutoff = True, table = True)
sim.visualize_cell_level_traces('light_shade', cutoff = True, table = True)
```

[8]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x000001A4685F5D68>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x000001A4686300B8>],
      dtype=object)
```



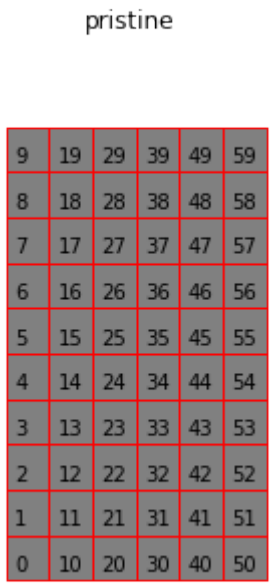
	E	Tc	Rsh_mult	Rs_mult	Io_mult	Il_mult	nnsvth_mult
condition1	400	20	1	1	1	1	1
condition2	405	50	1	1	1	1	1
condition3	210.5	8.1	1	1	1	1	1
condition4	413.42	23.61	1	1	1	1	1
condition5	329.14	18.89	1	1	1	1	1
condition6	453.06	30.07	1	1	1	1	1
condition7	281.76	35.09	1	1	1	1	1
condition8	491.5	36.46	1	1	1	1	1
condition9	395.68	25.96	1	1	1	1	1
condition10	523.5	39.15	1	1	1	1	1
condition11	300.53	50.83	1	1	1	1	1
condition12	590.3	29.81	1	1	1	1	1

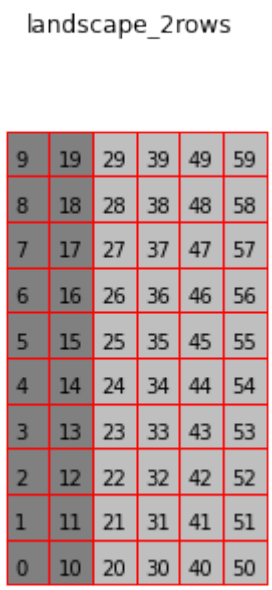


	E	Tc	Rsh_mult	Rs_mult	Io_mult	Il_mult	nnsvth_mult
condition1	800	50	1	1	1	1	1
condition2	690.59	50	1	1	1	1	1
condition3	940.1	50	1	1	1	1	1
condition4	772.75	50	1	1	1	1	1
condition5	668.78	50	1	1	1	1	1
condition6	739.14	50	1	1	1	1	1
condition7	829.56	50	1	1	1	1	1
condition8	895.71	50	1	1	1	1	1
condition9	964.71	50	1	1	1	1	1
condition10	601.67	50	1	1	1	1	1
condition11	874.99	50	1	1	1	1	1

Step 6 cont'd: Visualize modcells

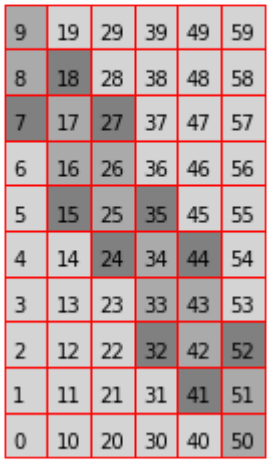
[9]:
```
for mod_identifier in sim.modcells.keys():
    sim.visualize_module_configurations(mod_identifier, title = mod_identifier)
```

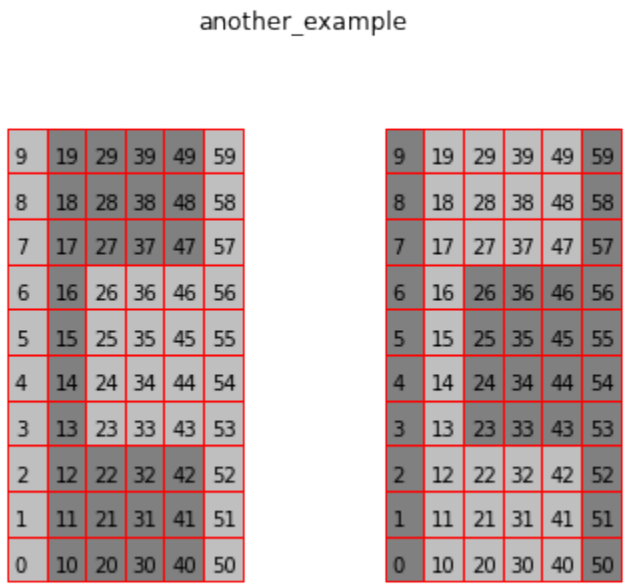




portrait_2cols

pole_2width





1.2.7 IV Failure Classification with Neural Networks

This notebook demonstrates the use of the `pvops.iv.models.nn` module for classification of faults.

[1]:
```
import os
import sys

from pvops.iv import simulator, extractor, preprocess
from pvops.iv.models import nn
```

Create iv column dictionary with format {pvops variable: user-specific column names}. This establishes a connection between the user's data columns and the pvops library.

[2]:
```
iv_col_dict = {
    "mode": "mode",
    "current": "current",            # Populated in simulator
    "voltage": "voltage",            # Populated in simulator
    "irradiance": "E",               # Populated in simulator
    "temperature": "T",              # Populated in simulator
    "power": "power",                # Populated in preprocess
    "derivative": "derivative",      # Populated in feature_generation
    "current_diff": "current_diff",  # Populated in feature_generation
}
```

Step 1: Collect your IV curves.

In this case, we simulate some curves, but you can replace this step by reading in your own data, if wanted.

[3]:
```
def define_failure_at_environment(sim, E, Tc, N_samples = 10):

    def namer(name):
        suffix = f"-{E}_{Tc}"
        return name + suffix

    sim.pristine_condition = {'identifier': 'pristine',
                              'E': E,
                              'Tc': Tc,
                              'Rsh_mult': 1,
                              'Rs_mult': 1,
                              'Io_mult': 1,
                              'Il_mult': 1,
                              'nnsvth_mult': 1,
                              }
    condition = {'identifier':namer('weathered_pristine')}
    sim.add_preset_conditions('complete', condition, save_name = namer('Complete_
↪weathered_pristine'))
    condition = {'identifier':namer('shade'),'Il_mult':0.6}
    sim.add_preset_conditions('complete', condition, save_name = namer('Complete_shading
↪'))
    condition = {'identifier':namer('cracking'),'Rs_mult':1.5}
    sim.add_preset_conditions('complete', condition, save_name = namer('Complete_cracking
↪'))

    dicts = {'Il_mult':{'mean': 0.6,
                        'std': 0.7,
                        'low': 0.33,
                        'upp': 0.95,
                        }
            }
    sim.generate_many_samples(namer('shade'), N_samples, dicts)

    dicts = {
            'Rs_mult':{'mean':1.3,
                        'std':0.6,
                        'low':1.1,
                        'upp':1.8
                        },
            'Rsh_mult':{'mean':0.5,
                        'std':0.6,
                        'low':0.3,
                        'upp':0.7
                        }
            }
    sim.generate_many_samples(namer('cracking'), N_samples, dicts)

    sim.build_strings({
                    namer('Partial Soiling (1M)'): [namer('Complete_weathered_pristine
```

(continues on next page)

(continued from previous page)

```
↪')]*11 + [namer('Complete_shading')]*1,
                    namer('Partial Soiling (6M)'): [namer('Complete_weathered_pristine
↪')]*6 + [namer('Complete_shading')]*6,
                    namer('Cell cracking (4M)'): [namer('Complete_weathered_pristine
↪')]*8 + [namer('Complete_cracking')]*4,
                    })
    return sim
```

[4]:
```
import numpy as np

sim = simulator.Simulator(
                pristine_condition = {
                                        'identifier': 'pristine',
                                        'E': 1000,
                                        'Tc': 50,
                                        'Rsh_mult': 1,
                                        'Rs_mult': 1,
                                        'Io_mult': 1,
                                        'Il_mult': 1,
                                        'nnsvth_mult': 1,
                                    })

sim.build_strings({'Pristine array': ['pristine']*12})

Tc = 35
for E in np.arange(200,1100,100):
    for Tc in np.arange(35,60,5):
        define_failure_at_environment(sim, E, Tc, N_samples = 25)
sim.simulate()
```

```
Simulating cells:   0%|          | 0/136 [00:00<?, ?it/s]/home/klbonne/.local/bin/
↪anaconda3/envs/pvops_dev/lib/python3.8/site-packages/scipy/optimize/_zeros_py.py:466:␣
↪RuntimeWarning: some failed to converge after 100 iterations
  warnings.warn(msg, RuntimeWarning)
Simulating cells: 100%|| 136/136 [00:39<00:00,  3.43it/s]
Adding up simulations: 100%|| 136/136 [00:45<00:00,  3.02it/s]
Adding up other definitions: 100%|| 136/136 [00:00<00:00, 584452.20it/s]
```

[5]:
```
df = sim.sims_to_df(focus=['string'], cutoff=True)
df.head()
```

[5]:
```
                                              current  \
0  [9.214203000284689, 9.210691359222285, 9.20726...
1  [1.828797118809955, 1.8281227256242998, 1.8274...
2  [1.828797118809955, 1.8281227256242998, 1.8274...
3  [1.828797118809955, 1.8281227256242998, 1.8274...
4  [1.828797118809955, 1.8281227256242998, 1.8274...

                                              voltage       E     T  \
0  [3.834932371660216e-12, 11.207809937682319, 22...  1000.0  50.0
1  [9.598528688982285, 19.45539770828682, 29.0763...   200.0  35.0
```

(continues on next page)

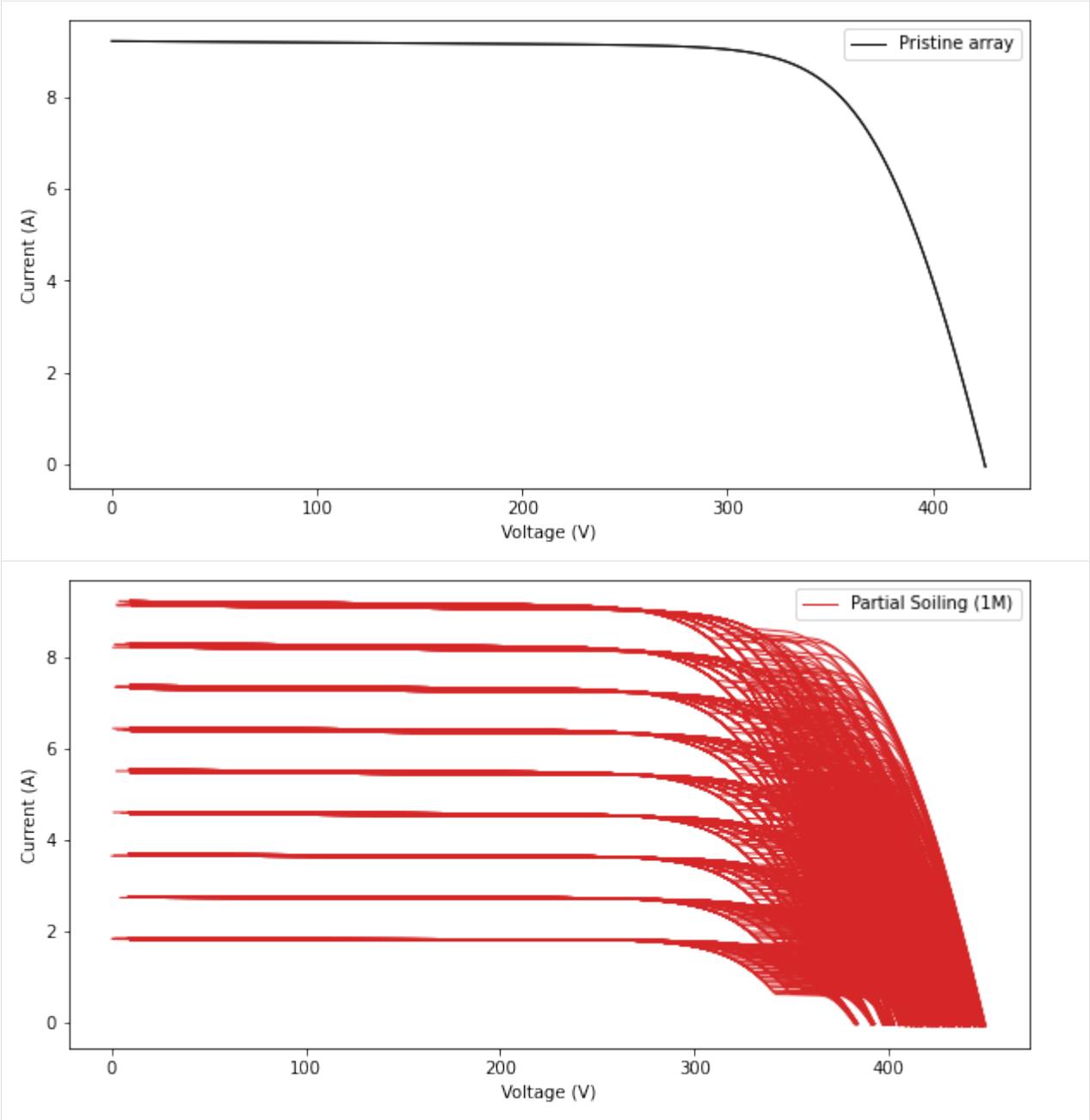
(continued from previous page)

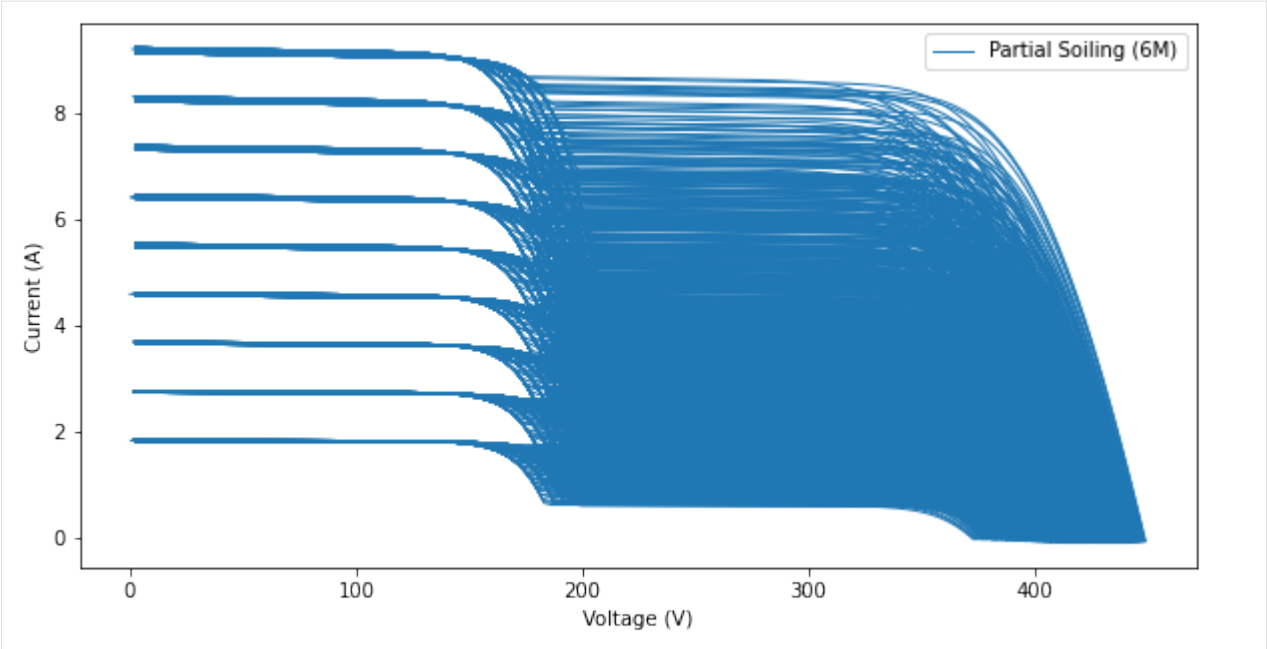
```
2  [9.598528688982285, 19.45539770828682, 29.0763...   200.0  35.0
3  [9.598528688982285, 19.45539770828682, 29.0763...   200.0  35.0
4  [9.598528688982285, 19.45539770828682, 29.0763...   200.0  35.0

                          mode   level
0               Pristine array  string
1  Partial Soiling (1M)-200_35  string
2  Partial Soiling (1M)-200_35  string
3  Partial Soiling (1M)-200_35  string
4  Partial Soiling (1M)-200_35  string
```

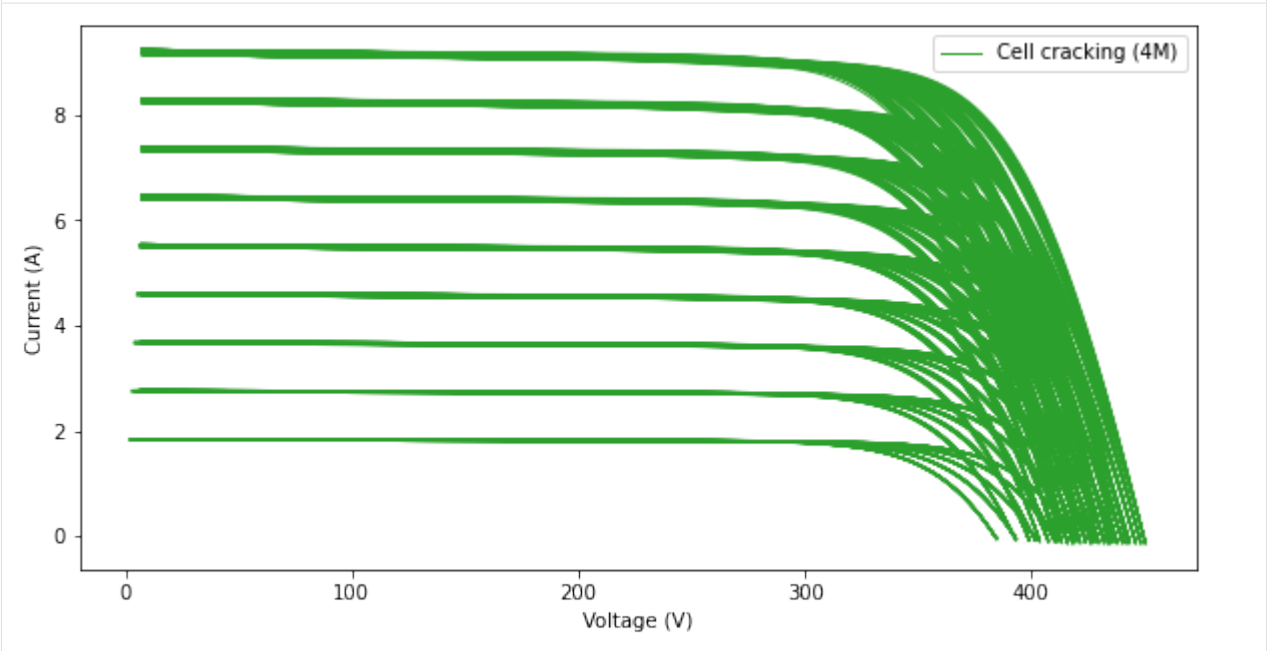
[6]:
```
# Convert modes to be the discrete failure modes
df['mode'] = [x.split('-')[0] for x in df['mode']]
```

Visualize generated samples for each failure mode.

[15]:
```
import matplotlib.pyplot as plt

colors = ['k', 'tab:red', 'tab:blue', 'tab:green']

# plt.figure(figsize=(12,8))
unique_modes = df['mode'].unique()
for md_idx, md in enumerate(unique_modes):
    plt.figure(figsize=(10,5))

    subdf = df[df['mode'] == md]
    i = 0
    for ind,row in subdf.iterrows():
        if i == 0:
            plt.plot(row['voltage'], row['current'], linewidth=1, color=colors[md_idx],␣
↪label=md)
        plt.plot(row['voltage'], row['current'], linewidth=1, color=colors[md_idx])
        i += 1
        plt.legend()
        plt.xlabel("Voltage (V)")
        plt.ylabel("Current (A)")
```


Pristine array
Current (A)
Voltage (V)
0
2
4
6
8
0
100
200
300
400
Partial Soiling (1M)
Current (A)
Voltage (V)
0
2
4
6
8
0
100
200
300
400





Next, process the data for irradiance and temperature correction, and normalize the axes. Shuffle the data in preparation for classification.

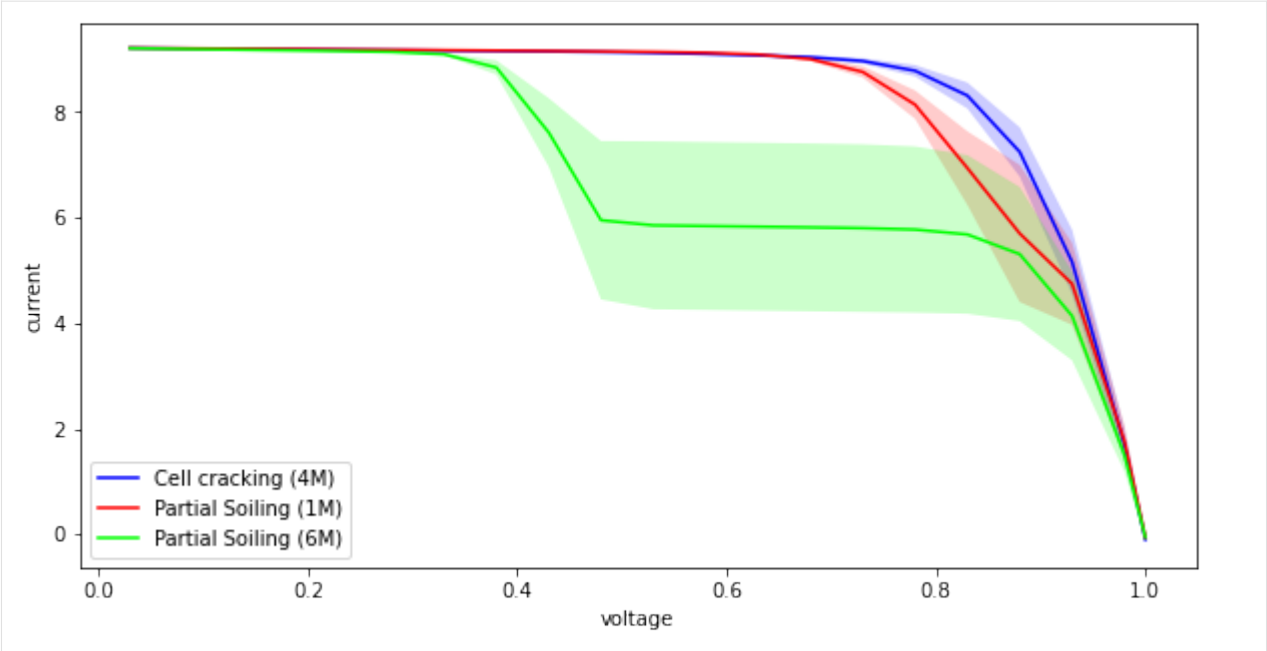
[8]:
```
prep_df = preprocess.preprocess(df, 0.05, iv_col_dict, resmpl_cutoff=0.03,
                                correct_gt=True, normalize_y=False,
                                CECmodule_parameters=sim.module_parameters,
                                n_mods=12, gt_correct_option=3)
# Shuffle
bigdf = prep_df.sample(frac=1).reset_index(drop=True)
bigdf.dropna(inplace=True)
bigdf.head(n=2)
```

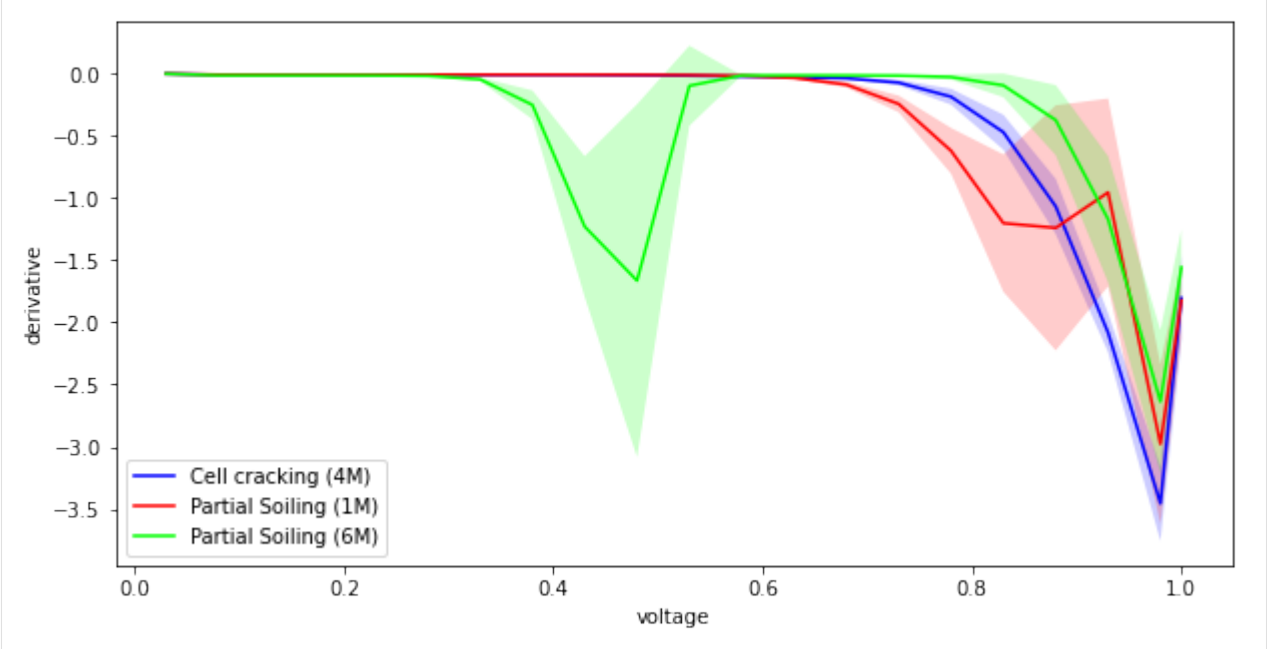
[8]:
```
                    mode                                                current  \
0  Partial Soiling (6M)  [9.208600987136746, 9.195882051686299, 9.18315...
1  Partial Soiling (6M)  [9.212759790227, 9.200363798497587, 9.18796074...

                                                 voltage  \
0  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
1  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...

                                                   power      E     T
0  [0.2762580296141024, 0.735670564134904, 1.1938...  600.0  50.0
1  [0.27638279370681, 0.736029103879807, 1.194434...  200.0  40.0
```

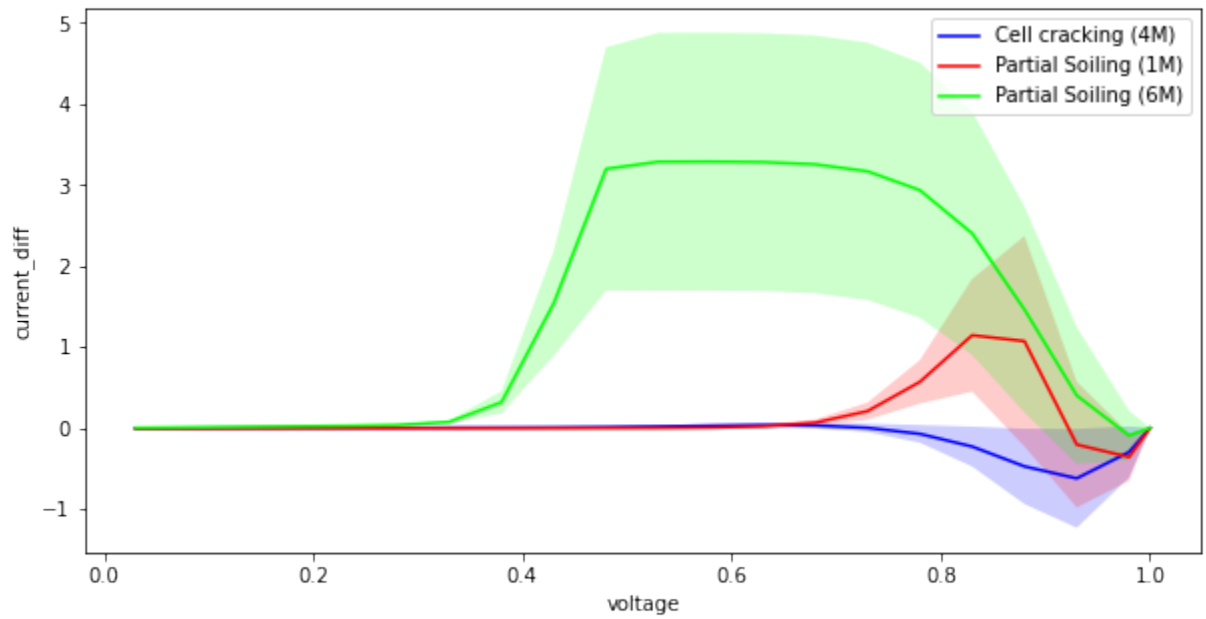
[9]:
```
# Feature generation
feat_df = nn.feature_generation(bigdf, iv_col_dict)
feat_df.head(n=2)
```

[9]:
```
                    mode                                                current  \
0  Partial Soiling (6M)  [9.208600987136746, 9.195882051686299, 9.18315...
1  Partial Soiling (6M)  [9.212759790227, 9.200363798497587, 9.18796074...

                                                 voltage  \
0  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
1  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...

                                                   power      E     T  \
0  [0.2762580296141024, 0.735670564134904, 1.1938...  600.0  50.0
1  [0.27638279370681, 0.736029103879807, 1.194434...  200.0  40.0

                                            current_diff  \
0  [0.0015998674001220792, 0.007648488303704681, ...
1  [-0.0025589356901321025, 0.0031667414924161363...

                                              derivative
0  [0.0, -0.012718935450447333, -0.01272509254493...
1  [0.0, -0.01239599172941297, -0.012403058265777...
```

[10]:
```
fig = nn.plot_profiles(feat_df,
                       iv_col_dict['voltage'],
                       iv_col_dict['current'],
                       iv_col_dict)
fig = nn.plot_profiles(feat_df,
                       iv_col_dict['voltage'],
                       iv_col_dict['derivative'],
                       iv_col_dict)
fig = nn.plot_profiles(feat_df,
                       iv_col_dict['voltage'],
                       iv_col_dict['current_diff'],
                       iv_col_dict)
```


current
voltage
Cell cracking (4M)
Partial Soiling (1M)
Partial Soiling (6M)


derivative
voltage
Cell cracking (4M)
Partial Soiling (1M)
Partial Soiling (6M)



[11]:
```
# To provide a clean output, we filter warnings. We encourage you to *remove* this␣
↪filter.
import logging
import tensorflow as tf
tf.get_logger().setLevel(logging.ERROR)

nn_config = {
    # NN parameters
    "model_choice": "1DCNN", # or "LSTM_multihead"
    "params": ['current', 'power', 'derivative', 'current_diff'],
    "dropout_pct": 0.5,
    "verbose": 1,
    # Training parameters
    "train_size": 0.9,
    "shuffle_split": True,
    "balance_tactic": 'truncate',
    "n_CV_splits": 5,
    "batch_size": 8,
    "max_epochs": 100,
    # LSTM parameters
    "use_attention_lstm": False,
    "units": 50,
    # 1DCNN parameters
    "nfilters": 64,
    "kernel_size": 12,
}

nn.classify_curves(feat_df, iv_col_dict, nn_config)
```

```
Balance data by mode:
        [Class Partial Soiling (1M)]: Resampled, 1170 == 1170
        [Class Partial Soiling (6M)]: Resampled, 1170 == 1170
```

(continues on next page)

(continued from previous page)

```
        [Class Cell cracking (4M)]: Resampled, 1170 == 1170
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d (Conv1D)             (None, 10, 64)            3136

 dropout (Dropout)           (None, 10, 64)            0

 flatten (Flatten)           (None, 640)               0

 dense (Dense)               (None, 100)               64100

 dense_1 (Dense)             (None, 3)                 303

=================================================================
Total params: 67,539
Trainable params: 67,539
Non-trainable params: 0
_________________________________________________________________
None
20/20 [==============================] - 0s 1ms/step - loss: 0.0025 - categorical_
↪accuracy: 1.0000
categorical_accuracy: 100.00%
20/20 [==============================] - 0s 960us/step - loss: 2.5271e-04 - categorical_
↪accuracy: 1.0000
categorical_accuracy: 100.00%
20/20 [==============================] - 0s 1ms/step - loss: 0.0010 - categorical_
↪accuracy: 1.0000
categorical_accuracy: 100.00%
20/20 [==============================] - 0s 2ms/step - loss: 1.8145e-04 - categorical_
↪accuracy: 1.0000
categorical_accuracy: 100.00%
20/20 [==============================] - 0s 2ms/step - loss: 2.1883e-05 - categorical_
↪accuracy: 1.0000
categorical_accuracy: 100.00%
44/44 [==============================] - 0s 1ms/step
                      precision    recall  f1-score   support

  Cell cracking (4M)       1.00      1.00      1.00       117
Partial Soiling (1M)       1.00      1.00      1.00       117
Partial Soiling (6M)       1.00      1.00      1.00       117

            accuracy                           1.00       351
           macro avg       1.00      1.00      1.00       351
        weighted avg       1.00      1.00      1.00       351

[[117   0   0]
 [  0 117   0]
 [  0   0 117]]
accuracy on test:  1.0
```

[11]:
```
(<pvops.iv.models.nn.IVClassifier at 0x7f149043b6a0>,
                      mode                                            current  \
546   Partial Soiling (6M)  [9.210855299765184, 9.198167892156352, 9.18547...
308   Partial Soiling (6M)  [9.203746007748741, 9.190451335406063, 9.17714...
1985  Partial Soiling (1M)  [9.213877334936674, 9.206654363996044, 9.19943...
1150  Partial Soiling (6M)  [9.209214372643126, 9.196397972209853, 9.18357...
1354  Partial Soiling (6M)  [9.204381949246688, 9.191145738106723, 9.17790...
...                    ...                                                ...
517     Cell cracking (4M)  [9.211188954334622, 9.201364333368181, 9.19153...
1913    Cell cracking (4M)  [9.214349750807305, 9.206652654143795, 9.19893...
1402  Partial Soiling (1M)  [9.218452597921566, 9.211327251671754, 9.20420...
585   Partial Soiling (6M)  [9.204333662518495, 9.191016941963325, 9.17769...
1599    Cell cracking (4M)  [9.216763526435322, 9.209357893492049, 9.20195...

                                                 voltage  \
546   [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
308   [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
1985  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
1150  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
1354  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
...                                                 ...
517   [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
1913  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
1402  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
585   [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
1599  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...

                                                   power       E     T  \
546   [0.2763256589929555, 0.7358534313725081, 1.194...   300.0  35.0
308   [0.2761123802324622, 0.7352361068324851, 1.193...   800.0  35.0
1985  [0.2764163200481002, 0.7365323491196836, 1.195...   800.0  40.0
1150  [0.2762764311792938, 0.7357118377767883, 1.193...   600.0  55.0
1354  [0.2761314584774006, 0.7352916590485379, 1.193...  1000.0  50.0
...                                                 ...     ...   ...
517   [0.27633566863003867, 0.7361091466694545, 1.19...   700.0  50.0
1913  [0.27643049252421914, 0.7365322123315037, 1.19...   300.0  55.0
1402  [0.27655357793764695, 0.7369061801337403, 1.19...   500.0  35.0
585   [0.27613000987555486, 0.735281355357066, 1.193...  1000.0  50.0
1599  [0.27650290579305964, 0.736748631479364, 1.196...   300.0  45.0

                                            current_diff  \
546   [-0.0006544452283154811, 0.005362647833651479,...
308   [0.006454846788127355, 0.013079204583940296, 0...
1985  [-0.003676480399805726, -0.0031238240060407207...
1150  [0.0009864818937419528, 0.007132567780150367, ...
1354  [0.005818905290180254, 0.012384801883280616, 0...
...                                                 ...
517   [-0.0009880997977536055, 0.0021662066218226528...
1913  [-0.004148896270436353, -0.0031221141537915997...
1402  [-0.008251743384697363, -0.007796711681750779,...
585   [0.005867192018373046, 0.012513598026679063, 0...
1599  [-0.006562671898453942, -0.00582735350204544, ...
```

(continues on next page)

(continued from previous page)

```
                                                derivative
546   [0.0, -0.012687407608831691, -0.01269405943970...
308   [0.0, -0.013294672342677671, -0.01330449252423...
1985  [0.0, -0.007222970940629736, -0.00722314079898...
1150  [0.0, -0.012816400433273145, -0.01282259372674...
1354  [0.0, -0.013236211139965093, -0.01324473877031...
...                                                 ...
517   [0.0, -0.009824620966440989, -0.00982576267010...
1913  [0.0, -0.007697096663509484, -0.00771783576816...
1402  [0.0, -0.007125346249811315, -0.00712545816528...
585   [0.0, -0.013316720555170747, -0.01332550767503...
1599  [0.0, -0.007405632943273233, -0.00740591956582...

[3159 rows x 8 columns],
                     mode                                                 current  \
982   Partial Soiling (6M)  [9.20744270094472, 9.194422397339585, 9.181395...
2403  Partial Soiling (1M)  [9.215729670026459, 9.208782958532584, 9.20183...
2457  Partial Soiling (6M)  [9.207748461504096, 9.19472827135731, 9.181700...
3099  Partial Soiling (1M)  [9.2123011775, 9.206447516934828, 9.1994454726...
456   Partial Soiling (6M)  [9.205895947108852, 9.192859018813232, 9.17981...
...                    ...                                                ...
1032  Partial Soiling (1M)  [9.210283333095331, 9.204441131746087, 9.19733...
2985  Partial Soiling (1M)  [9.214969316893278, 9.20771246203145, 9.200455...
2753  Partial Soiling (1M)  [9.2196017576896, 9.212665143862107, 9.2057283...
2015    Cell cracking (4M)  [9.211289735739712, 9.202459882066014, 9.19452...
1447  Partial Soiling (1M)  [9.21846129632336, 9.211350447604229, 9.204239...

                                                 voltage  \
982   [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
2403  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
2457  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
3099  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
456   [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
...                                                 ...
1032  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
2985  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
2753  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
2015  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...
1447  [0.03, 0.08, 0.13, 0.18000000000000002, 0.23, ...

                                                   power      E     T  \
982   [0.2762232810283416, 0.7355537917871667, 1.193...  700.0  50.0
2403  [0.27647189010079376, 0.7367026366826067, 1.19...  400.0  50.0
2457  [0.27623245384512285, 0.7355782617085848, 1.19...  600.0  45.0
3099  [0.276369035325, 0.7365158013547862, 1.1959279...  500.0  55.0
456   [0.27617687841326555, 0.7354287215050586, 1.19...  700.0  40.0
...                                                 ...    ...   ...
1032  [0.27630849999285995, 0.736355290539687, 1.195...  700.0  55.0
2985  [0.2764490795067983, 0.736616996962516, 1.1960...  800.0  35.0
2753  [0.27658805273068804, 0.7370132115089686, 1.19...  300.0  40.0
2015  [0.27633869207219136, 0.7361967905652812, 1.19...  600.0  55.0
1447  [0.2765538388897008, 0.7369080358083383, 1.196...  500.0  35.0
```

(continues on next page)

(continued from previous page)

```
                                          current_diff  \
982   [0.0027581535921488154, 0.009108142650418927, ...
2403  [-0.005528815489590499 5, -0.005252418542580628...
2457  [0.0024523930327724486, 0.008802268632694066, ...
3099  [-0.0021003229631322284, -0.002916976944824512...
456   [0.004304907428016591, 0.010671521176771392, 0...
...                                                   ...
1032  [-8.247855846299501e-05, -0.000910591756083434...
2985  [-0.004768462356409486, -0.004181922041446739 5...
2753  [-0.009400903152732454, -0.009134603872103852,...
2015  [-0.0010888812028433392, 0.0010706579239894154...
1447  [-0.008260441786491768, -0.00781990761422513, ...

                                            derivative
982   [0.0, -0.013020303605134842, -0.01302728621376...
2403  [0.0, -0.006946711493874602, -0.00694699702618...
2457  [0.0, -0.013020190146786348, -0.01302736006087...
3099  [0.0, -0.005853660565172447, -0.00700204423561...
456   [0.0, -0.013036928295619532, -0.01304477339346...
...                                                   ...
1032  [0.0, -0.005842201349244291 5, -0.0071043446507...
2985  [0.0, -0.007256854861827477 5, -0.0072569856979...
2753  [0.0, -0.006936613827493332, -0.00693677041243...
2015  [0.0, -0.008829853673697485, -0.00793823468386...
1447  [0.0, -0.007110848719131368 5, -0.0071109598167...

[351 rows x 8 columns])
```

1.2.8 IV: Brute-force diode parameter extraction

[1]:
```
from pvops.iv import simulator, extractor
```

Step 1: Collect your IV curves.

In this case, we simulate some curves, but you can replace this step by reading in your own data, if wanted.

[2]:
```
sim = simulator.Simulator(
                mod_specs = {
                            'Jinko_Solar_Co___Ltd_JKM270PP_60': {'ncols': 6,
                                                                 'nsubstrings': 3
                                                                }
                            }
)

sim.build_strings({'Unstressed': ['pristine']*12})

N = 100
dicts = {'E':       {'mean': 800,
```

(continues on next page)

(continued from previous page)

```
                        'std': 500,
                        'low': 400,
                        'upp': 1100
                    },
         'Tc':      {'mean': 30,
                      'std': 5,
                    },
         'Rs_mult': {'mean': 1.3,
                    'std': 0.6,
                    'low': 0.9,
                    'upp': 1.5
                    },
         'Rsh_mult': {'mean': 0.8,
                    'std': 0.9,
                    'low': 0.3,
                    'upp': 1.0
                    },
        'Il_mult': {'mean': 1.0,
                    'std': 0.5,
                    'low': 0.95,
                    'upp': 1.05
                    },
        'Io_mult': {'mean': 1.0,
                    'std': 0.6,
                    'low': 0.85,
                    'upp': 1.1
                    },
        'nnsvth_mult': {'mean': 1.0,
                        'std': 0.5,
                        'low': 0.85,
                        'upp': 1.1
                        }
        }
sim.generate_many_samples('pristine', N, dicts)
sim.simulate()

sim.print_info()
```

```
Simulating cells:   0%|          | 0/1 [00:00<?, ?it/s]/home/klbonne/.local/bin/
↪anaconda3/envs/pvops_dev/lib/python3.8/site-packages/scipy/optimize/_zeros_py.py:466:␣
↪RuntimeWarning: some failed to converge after 100 iterations
  warnings.warn(msg, RuntimeWarning)
Simulating cells: 100%|| 1/1 [00:02<00:00,  2.63s/it]
Adding up simulations: 100%|| 1/1 [00:01<00:00,  1.69s/it]
Adding up other definitions: 100%|| 1/1 [00:00<00:00, 10433.59it/s]
```

```
Condition list: (Cell definitions)
        [pristine]: 101 definition(s)

Modcell types: (Cell mappings on module)
        [pristine]: 1 definition(s)

String definitions (Series of modcells)
```

(continues on next page)

(continued from previous page)

```
        [Unstressed]: 101 definition(s)
```

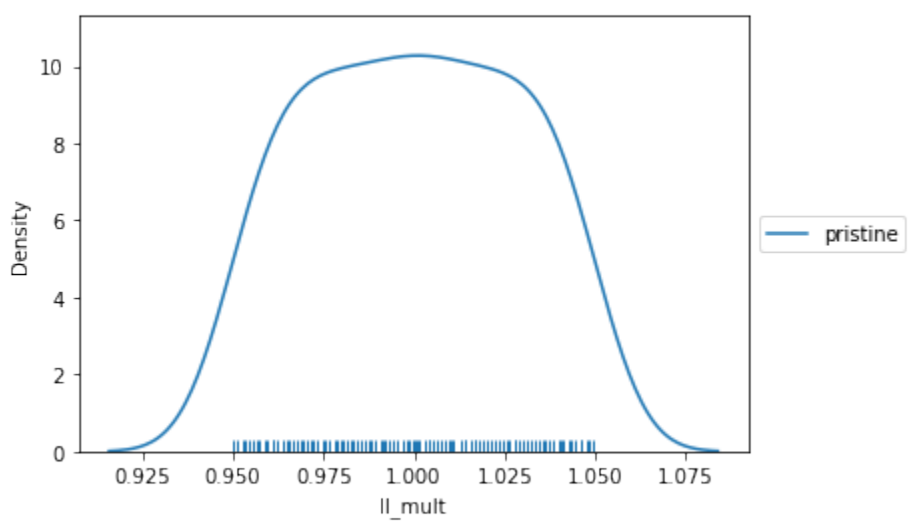
[3]:
```
sim.visualize()
```

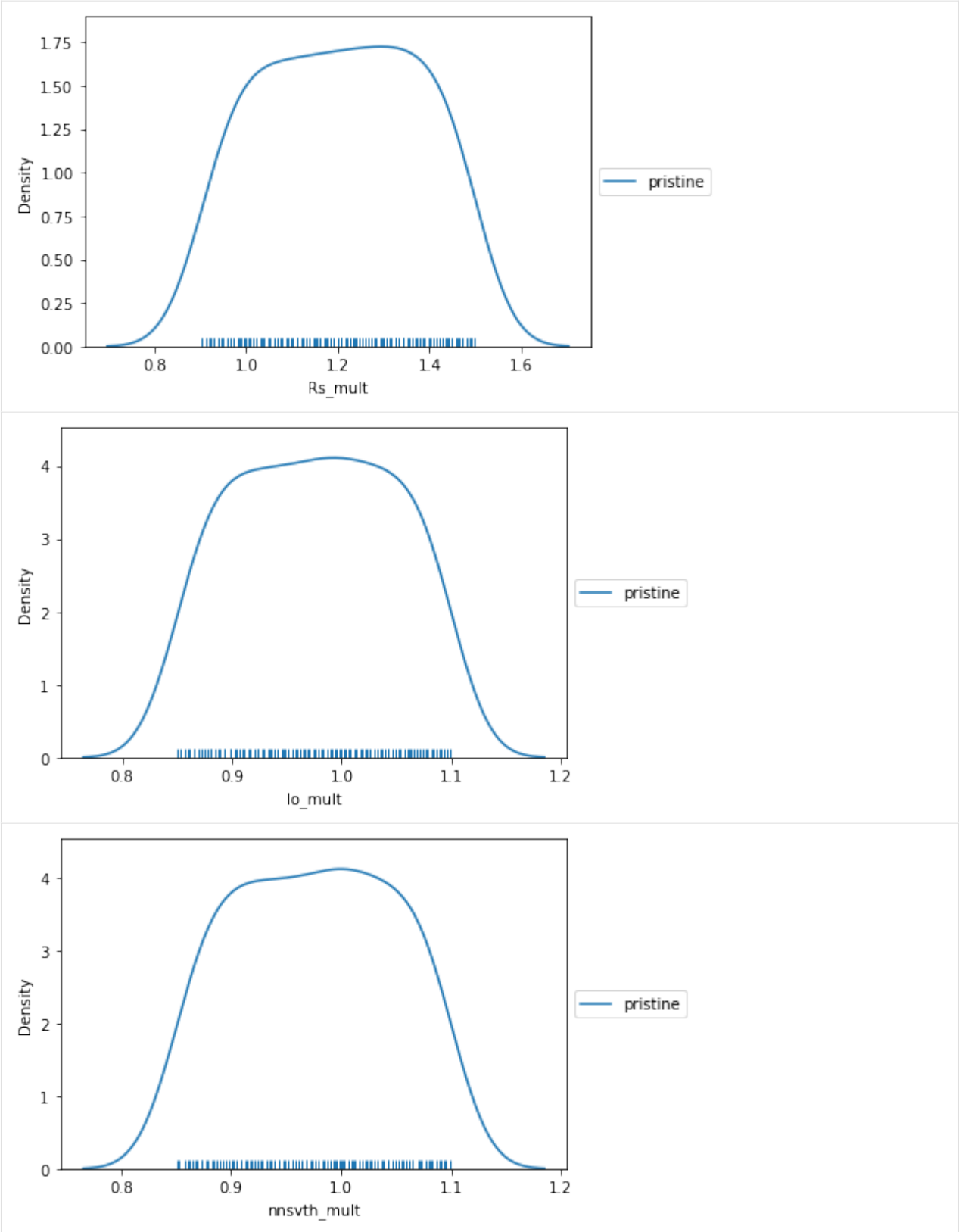
```
/home/klbonne/Documents/GitHub/pvOps/pvops/iv/simulator.py:1442: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  axs = sns.distplot(
/home/klbonne/Documents/GitHub/pvOps/pvops/iv/simulator.py:1445: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  axs = sns.distplot(
/home/klbonne/Documents/GitHub/pvOps/pvops/iv/simulator.py:1445: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  axs = sns.distplot(
/home/klbonne/Documents/GitHub/pvOps/pvops/iv/simulator.py:1445: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  axs = sns.distplot(
/home/klbonne/Documents/GitHub/pvOps/pvops/iv/simulator.py:1445: UserWarning:
```

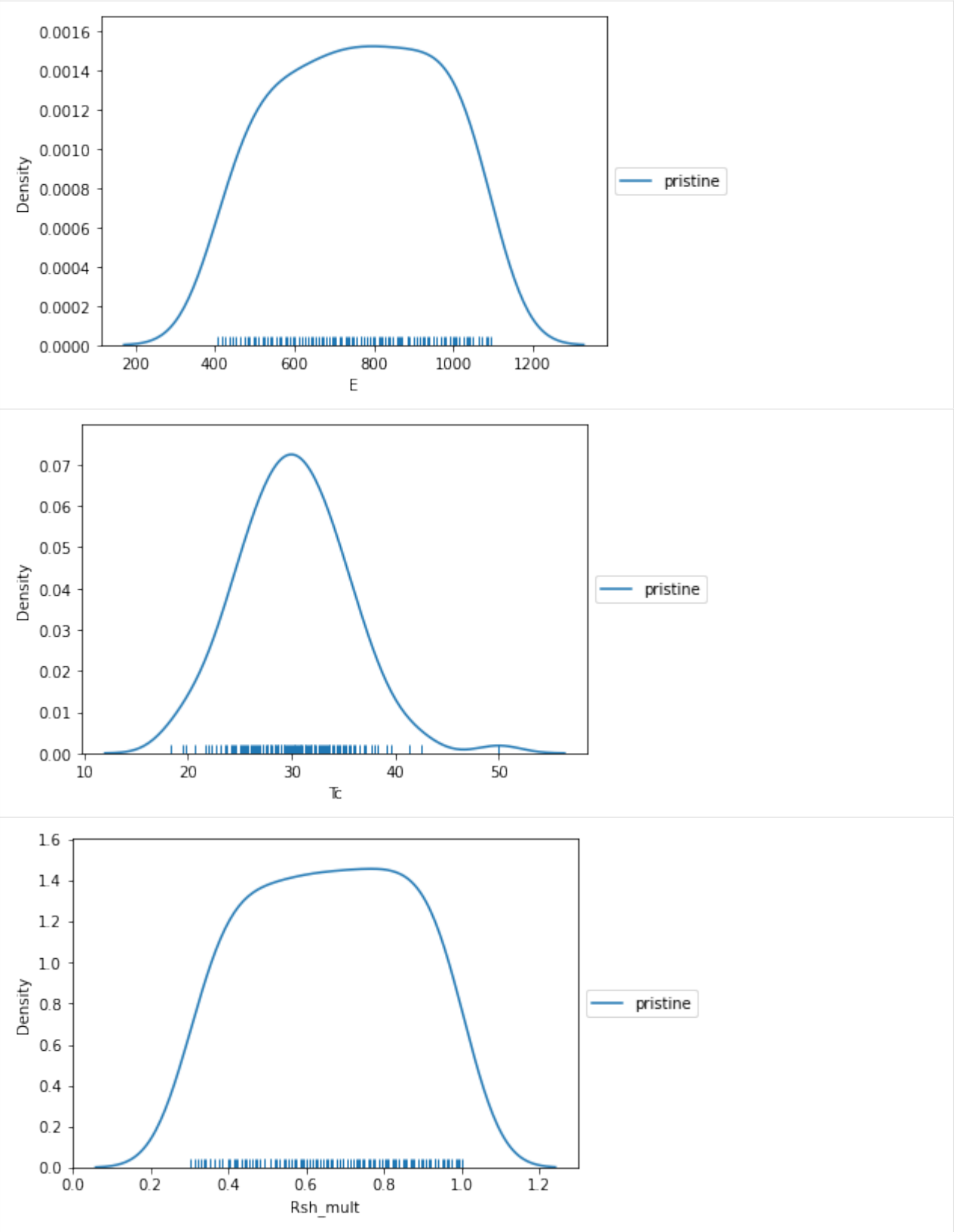
(continues on next page)

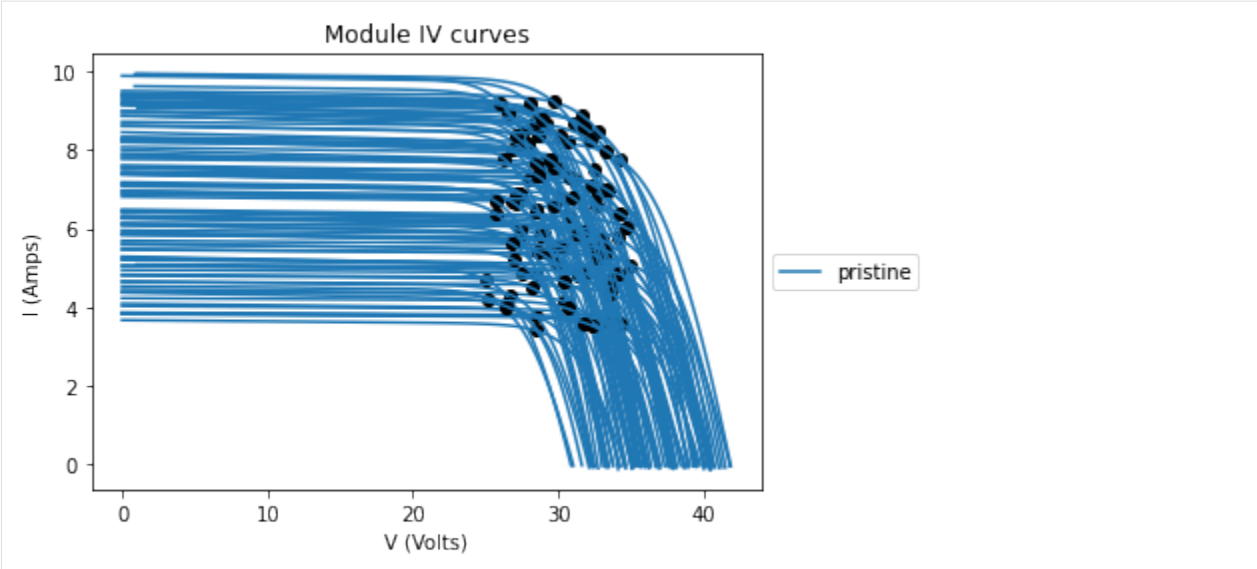
(continued from previous page)

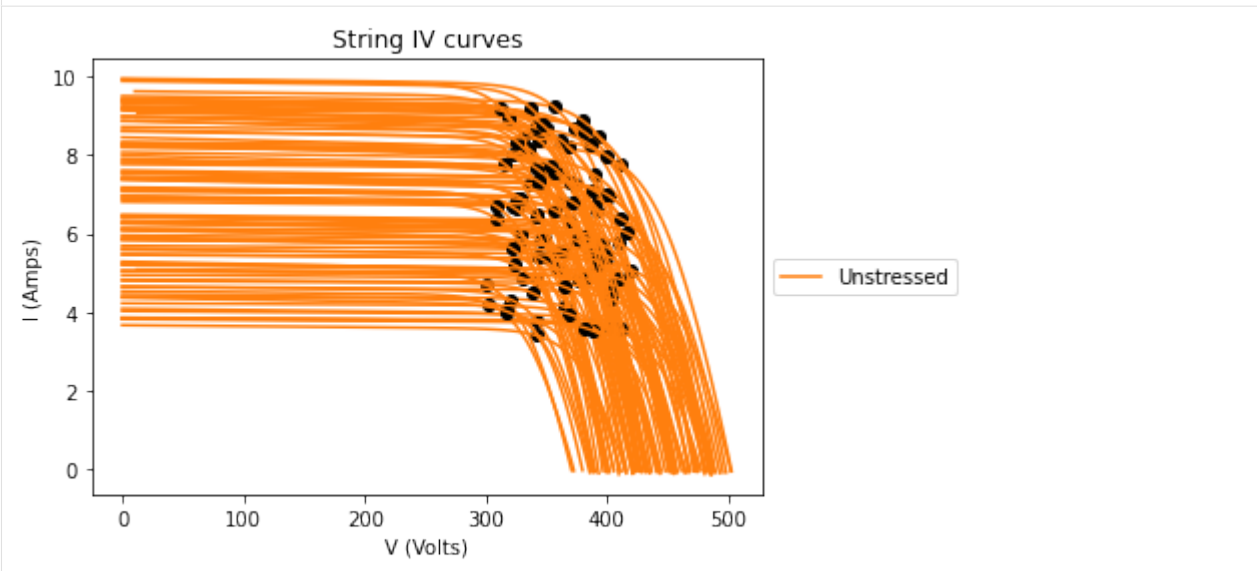
```
`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  axs = sns.distplot(
/home/klbonne/Documents/GitHub/pvOps/pvops/iv/simulator.py:1445: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  axs = sns.distplot(
/home/klbonne/Documents/GitHub/pvOps/pvops/iv/simulator.py:1445: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  axs = sns.distplot(
```




Density
Rs_mult
pristine
Density
Io_mult
pristine
Density
nnsvth_mult
pristine


0.0016
0.0014
0.0012
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
Density
200
400
600
800
1000
1200
E
pristine
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
Density
10
20
30
40
50
Tc
pristine
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Density
0.0
0.2
0.4
0.6
0.8
1.0
1.2
Rsh_mult
pristine





[4]:
```
df = sim.sims_to_df(focus=['string'], cutoff=True)
df.head()
```

[4]:
```
                                                 current  \
0  [9.214203000284689, 9.210691359222285, 9.20726...
1  [5.284107412718183, 5.2776846348732525, 5.2714...
2  [7.125172224252341, 7.119234129879513, 7.11343...
3  [6.431353183119788, 6.424342644781958, 6.41749...
4  [9.322058378622014, 9.31619129152217, 9.310464...

                                                voltage            E          T  \
0  [3.834932371660216e-12, 11.207809937682319, 22...  1000.000000  50.000000
1  [3.836930773104541e-12, 10.94261122291978, 21...   587.317020  30.336647
2  [3.838707129943941e-12, 12.742973777314244, 25...   775.239408  35.517139
3  [3.834932371660216e-12, 12.216345601755854, 24...   732.303708  35.722979
4  [3.835043393962678e-12, 11.380842316022822, 22...  1025.459067  31.791829
```

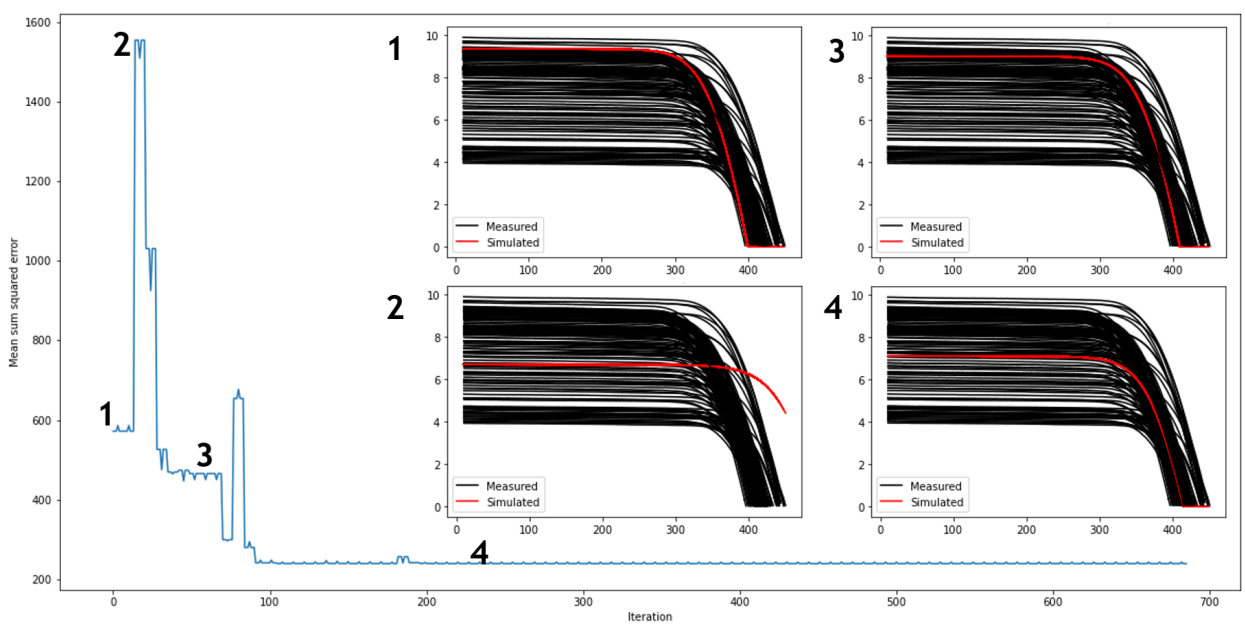
(continues on next page)

(continued from previous page)

```
         mode   level
0  Unstressed  string
1  Unstressed  string
2  Unstressed  string
3  Unstressed  string
4  Unstressed  string
```

Step 2: Conduct diode parameter extraction

[5]:
```
# Only created to retrieve cell_parameters
temporary_sim = simulator.Simulator(
                mod_specs = {
                            'Jinko_Solar_Co___Ltd_JKM270PP_60': {'ncols': 6,
                                                                 'nsubstrings': 3
                                                                }
                            }
)

extr = extractor.BruteForceExtractor(input_df=df,
                                     current_col='current',
                                     voltage_col='voltage',
                                     irradiance_col='E',
                                     temperature_col='T',
                                     T_type='cell',
                                     windspeed_col=None,
                                     Simulator_mod_specs=None,
                                     Simulator_pristine_condition=None)
```



1.3 API Documentation

1.3.1 text module

text.classify module

`pvops.text.classify.classification_deployer`(*X*, *y*, *n_splits*, *classifiers*, *search_space*, *pipeline_steps*, *scoring*, *greater_is_better=True*, *verbose=3*)

The classification deployer builds a classifier evaluator with an ingrained hyperparameter fine-tuning grid search protocol. The output of this function will be a data frame showing the performance of each classifier when utilizing a specific hyperparameter configuration.

To see an example of this method's application, see `tutorials//text_class_example.py`

Parameters

- **X** (*list of str*) – List of documents (str). The documents will be passed through the pipeline_steps, where they will be transformed into vectors.
- **y** (*list*) – List of labels corresponding with the documents in X
- **n_splits** (*int*) – Integer defining the number of splits in the cross validation split during training
- **classifiers** (*dict*) – Dictionary with key as classifier identifier (str) and value as classifier instance following sklearn's base model convention: sklearn_docs.

```
classifiers = {
    'LinearSVC' : LinearSVC(),
    'AdaBoostClassifier' : AdaBoostClassifier(),
    'RidgeClassifier' : RidgeClassifier()
}
```

See `supervised_classifier_defs.py` or `unsupervised_classifier_defs.py` for this package's defaults.

- **search_space** (*dict*) – Dictionary with classifier identifiers, as used in `classifiers`, mapped to its hyperparameters.

```
search_space = {
    'LinearSVC' : {
    'clf__C' : [1e-2,1e-1],
    'clf__max_iter':[800,1000],
    },
    'AdaBoostClassifier' : {
    'clf__n_estimators' : [50,100],
    'clf__learning_rate':[1.,0.9,0.8],
    'clf__algorithm' : ['SAMME.R']
    },
    'RidgeClassifier' : {
    'clf__alpha' : [0.,1e-3,1.],
    'clf__normalize' : [False,True]
    }
}
```

See `supervised_classifier_defs.py` or `unsupervised_classifier_defs.py` for this package's defaults.

- **pipeline_steps** (*list of tuples*) – Define embedding and machine learning pipeline. The last tuple must be `('clf', None)` so that the output of the pipeline is a prediction. For supervised classifiers using a TFIDF embedding, one could specify

```
pipeline_steps = [('tfidf', TfidfVectorizer()),
                  ('clf', None)]
```

For unsupervised clusterers using a TFIDF embedding, one could specify

```
pipeline_steps = [('tfidf', TfidfVectorizer()),
                  ('to_dense', DataDensifier.DataDensifier()),
                  ('clf', None)]
```

A densifier is required from some clusters, which fail if sparse data is passed.

- **scoring** (*sklearn callable scorer (i.e., any statistic that summarizes predictions relative to observations).*) – Example scorers include f1_score, accuracy, etc. Callable object that returns a scalar score created using sklearn.metrics.make_scorer For supervised classifiers, one could specify

```
scoring = make_scorer(f1_score, average = 'weighted')
```

For unsupervised classifiers, one could specify

```
scoring = make_scorer(homogeneity_score)
```

- **greater_is_better** (*bool*) – Whether the scoring parameter is better when greater (i.e. accuracy) or not.
- **verbose** (*int*) – Control the specificity of the prints. If greater than 1, a print out is shown when a new "best classifier" is found while iterating. Additionally, the verbosity during the grid search follows sklearn's definitions. The frequency of the messages increase with the verbosity level.

Returns

DataFrame – Summarization of results from all of the classifiers

`pvops.text.classify.`**`get_attributes_from_keywords`**(*om_df*, *col_dict*, *reference_df*, *reference_col_dict*)

Find keywords of interest in specified column of dataframe, return as new column value.

If keywords of interest given in a reference dataframe are in the specified column of the dataframe, return the keyword category, or categories. For example, if the string 'inverter' is in the list of text, return ['inverter'].

Parameters

- **om_df** (*pd.DataFrame*) – Dataframe to search for keywords of interest, must include text_col.
- **col_dict** (*dict of {str : str}*) – A dictionary that contains the column names needed:
  - data : string, should be assigned to associated column which stores the tokenized text logs
  - predicted_col : string, will be used to create keyword search label column
- **reference_df** (*DataFrame*) – Holds columns that define the reference dictionary to search for keywords of interest, Note: This function can currently only handle single words, no n-gram functionality.
- **reference_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names that describes how referencing is going to be done

- reference_col_from : string, should be assigned to associated column name in reference_df that are possible input reference values Example: pd.Series(['inverter', 'invert', 'inv'])
- reference_col_to : string, should be assigned to associated column name in reference_df that are the output reference values of interest Example: pd.Series(['inverter', 'inverter', 'inverter'])

Returns

om_df (*pd.DataFrame*) – Input df with new_col added, where each found keyword is its own row, may result in duplicate rows if more than one keywords of interest was found in text_col.

text.defaults module

`pvops.text.defaults.`**`supervised_classifier_defs`**`(`*settings_flag*`)`

Establish supervised classifier definitions which are non-specific to embeddor, and therefore, non-specific to the natural language processing application

Parameters

settings_flag (*str*) – Either 'light', 'normal' or 'detailed'; a setting which determines the number of hyperparameter combinations tested during the grid search. For instance, a dataset of 50 thousand samples may run for hours on the 'normal' setting but for days on 'detailed'.

Returns

- **search_space** (*dict*) – Hyperparameter instances for each clusterer
- **classifiers** (*dict*) – Contains sklearn classifiers instances

`pvops.text.defaults.`**`unsupervised_classifier_defs`**`(`*setting_flag*, *n_clusters*`)`

Establish supervised classifier definitions which are non-specific to embeddor, and therefore, non-specific to the natural language processing application

Parameters

- **setting_flag** (*str*) – Either 'normal' or 'detailed'; a setting which determines the number of hyperparameter combinations tested during the grid search. For instance, a dataset of 50,000 samples may run for hours on the 'normal' setting but for days on 'detailed'.
- **n_clusters** (*int,*) – Number of clusters to organize the text data into. Usually set to the number of unique categories within data.

Returns

- **search_space** (*dict*) – Hyperparameter instances for each clusterer
- **clusterers** (*dict*) – Contains sklearn cluster instances

text.nlp_utils module

class `pvops.text.nlp_utils.`**`DataDensifier`**

Bases: `BaseEstimator`

A data structure transformer which converts sparse data to dense data. This process is usually incorporated in this library when doing unsupervised machine learning. This class is built specifically to work inside a sklearn pipeline. Therefore, it uses the default `transform`, `fit`, `fit_transform` method structure.

fit(*X*, *y=None*)

Placeholder method to conform to the sklearn class structure.

Parameters

- **X** (*array*) – Input data
- **y** (*Not utilized.*)

Returns
DataDensifier object

fit_transform(*X*, *y=None*)

Performs same action as `DataDensifier.transform()`, which returns a dense array when the input is sparse.

Parameters

- **X** (*array*) – Input data
- **y** (*Not utilized.*)

Returns
dense array

transform(*X*, *y=None*)

Return a dense array if the input array is sparse.

Parameters
X (*array*) – Input data of numerical values. For this package, these values could represent embedded representations of documents.

Returns
dense array

class pvops.text.nlp_utils.**Doc2VecModel**(*vector_size=100*, *dm_mean=None*, *dm=1*, *dbow_words=0*, *dm_concat=0*, *dm_tag_count=1*, *dv=None*, *dv_mapfile=None*, *comment=None*, *trim_rule=None*, *callbacks=()*, *window=5*, *epochs=10*)

Bases: `BaseEstimator`

Performs a gensim Doc2Vec transformation of the input documents to create embedded representations of the documents. See gensim's Doc2Vec model for information regarding the hyperparameters.

fit(*raw_documents*, *y=None*)

Fits the Doc2Vec model.

fit_transform(*raw_documents*, *y=None*)

Utilizes the `fit()` and `transform()` methods in this class.

set_fit_request(*, *raw_documents: bool | None | str = '$UNCHANGED$'*) → *Doc2VecModel*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.

- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

raw_documents (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `raw_documents` parameter in `fit`.

Returns

self (*object*) – The updated object.

`set_transform_request`(*, *raw_documents: bool | None | str = '$UNCHANGED$'*) → *Doc2VecModel*

Request metadata passed to the `transform` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `transform` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `transform`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

raw_documents (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `raw_documents` parameter in `transform`.

Returns

self (*object*) – The updated object.

`transform`(*raw_documents*)

Transforms the documents into Doc2Vec vectors.

`pvops.text.nlp_utils.create_stopwords(lst_langs=['english'], lst_add_words=[], lst_keep_words=[])`

Concatenate a list of stopwords using both words grabbed from nltk and user-specified words.

Parameters

- **lst_langs** (*list*) – List of strings designating the languages for a nltk.corpus.stopwords.words query. If empty list is passed, no stopwords will be queried from nltk.
- **lst_add_words** (*list*) – List of words(e.g., "road" or "street") to add to stopwords list. If these words are already included in the nltk query, a duplicate will not be added.
- **lst_keep_words** (*list*) – List of words(e.g., "before" or "until") to remove from stopwords list. This is usually used to modify default stop words that might be of interest to PV.

Returns

list – List of alphabetized stopwords

`pvops.text.nlp_utils.summarize_text_data(om_df, colname)`

Display information about a set of documents located in a dataframe, including the number of samples, average number of words, vocabulary size, and number of words in total.

Parameters

- **om_df** (*DataFrame*) – A pandas dataframe containing O&M data, which contains at least the colname of interest
- **colname** (*str*) – Column name of column with text

Returns

dict – dictionary containing printed summary data

text.preprocess module

`pvops.text.preprocess.get_dates(document, om_df, ind, col_dict, print_info, infer_date_surrounding_rows=True)`

Extract dates from the input document.

This method is utilized within `preprocessor.py`. For an easy way to extract dates, utilize the preprocessor and set extract_dates_only = True.

Parameters

- **document** (*str*) – String representation of a document
- **om_df** (*DataFrame*) – A pandas dataframe containing O&M data, which contains at least the columns within col_dict.
- **ind** (*integer*) – Designates the row of the dataframe which is currently being observed. This is required because if the current row does not have a valid date in the *eventstart*, then an iterative search is conducted by first starting at the nearest rows.
- **col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the get_dates fn
  - data : string, should be assigned to associated column which stores the text logs
  - eventstart : string, should be assigned to associated column which stores the log submission datetime
- **print_info** (*bool*) – Flag indicating whether to print information about the preprocessing progress

- **infer_date_surrounding_rows** (*bool*) – If True, utilizes iterative search in dataframe to infer the datetime from surrounding rows if the current row's date value is nan If False, does not utilize the base datetime. Consequentially, today's date is used to replace the missing parts of the datetime. Recommendation: set True if you frequently publish documents and your dataframe is ordered chronologically

Returns
list – List of dates found in text

`pvops.text.preprocess.get_keywords_of_interest(document_tok, reference_df, reference_col_dict)`

Find keywords of interest in list of strings from reference dict.

If keywords of interest given in a reference dict are in the list of strings, return the keyword category, or categories. For example, if the string 'inverter' is in the list of text, return ['inverter'].

Parameters

- **document_tok** (*list of str*) – Tokenized text, functionally a list of string values.
- **reference_df** (*DataFrame*) – Holds columns that define the reference dictionary to search for keywords of interest, Note: This function can currently only handle single words, no n-gram functionality.
- **reference_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names that describes how referencing is going to be done
  - reference_col_from : string, should be assigned to associated column name in reference_df that are possible input reference values Example: pd.Series(['inverter', 'invert', 'inv'])
  - reference_col_to : string, should be assigned to associated column name in reference_df that are the output reference values of interest Example: pd.Series(['inverter', 'inverter', 'inverter'])

Returns
included_equipment (*list of str*) – List of keywords from reference_dict found in list_of_txt, can be more than one value.

`pvops.text.preprocess.preprocessor(om_df, lst_stopwords, col_dict, print_info=False, extract_dates_only=False)`

Preprocessing function which processes the raw text data into processed text data and extracts dates

Parameters

- **om_df** (*DataFrame*) – A pandas dataframe containing O&M data, which contains at least the columns within col_dict.
- **lst_stopwords** (*list*) – List of stop words which will be filtered in final preprocessing step
- **col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the get_dates fn
  - data : string, should be assigned to associated column which stores the text logs
  - eventstart : string, should be assigned to associated column which stores the log submission datetime
  - save_data_column : string, should be assigned to associated column where the processed text should be stored
  - save_date_column : string, should be assigned to associated column where the extracted dates from the text should be stored

- **print_info** (*bool*) – Flag indicating whether to print information about the preprocessing progress
- **extract_dates_only** (*bool*) – If True, return after extracting dates in each ticket If False, return with preprocessed text and extracted dates

Returns
df (*DataFrame*) – Contains the original columns as well as the processed data, located in columns defined by the inputs

`pvops.text.preprocess.text_remove_nondate_nums(document, PRINT_INFO=False)`

Conduct initial text processing steps to prepare the text for date extractions. Function mostly uses regex-based text substitution to remove numerical structures within the text, which may be mistaken as a date by the date extractor.

Parameters

- **document** (*str*) – String representation of a document
- **PRINT_INFO** (*bool*) – Flag indicating whether to print information about the preprocessing progress

Returns
string – string of processed document

`pvops.text.preprocess.text_remove_numbers_stopwords(document, lst_stopwords)`

Conduct final processing steps after date extraction

Parameters

- **document** (*str*) – String representation of a document
- **lst_stopwords** (*list*) – List of stop words which will be filtered in final preprocessing step

Returns
string – string of processed document

text.utils module

`pvops.text.utils.remap_attributes(om_df, remapping_df, remapping_col_dict, allow_missing_mappings=False, print_info=False)`

A utility function which remaps the attributes of om_df using columns
within remapping_df.

Parameters

- **om_df** (*DataFrame*) – A pandas dataframe containing O&M data, which needs to be remapped.
- **remapping_df** (*DataFrame*) – Holds columns that define the remappings
- **remapping_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names that describes how remapping is going to be done
  - attribute_col : string, should be assigned to associated column name in om_df which will be remapped
  - remapping_col_from : string, should be assigned to associated column name in remapping_df that matches original attribute of interest in om_df

  - remapping_col_to : string, should be assigned to associated column name in remapping_df that contains the final mapped entries
- **allow_missing_mappings** (*bool*) – If True, allow attributes without specified mappings to exist in the final dataframe. If False, only attributes specified in *remapping_df* will be in final dataframe.
- **print_info** (*bool*) – If True, print information about remapping.

Returns
DataFrame – dataframe with remapped columns populated

`pvops.text.utils.remap_words_in_text(om_df, remapping_df, remapping_col_dict)`

A utility function which remaps a text column of om_df using columns
within remapping_df.

Parameters

- **om_df** (*DataFrame*) – A pandas dataframe containing O&M note data
- **remapping_df** (*DataFrame*) – Holds columns that define the remappings
- **remapping_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names that describes how remapping is going to be done
  - data : string, should be assigned to associated column name in om_df which will have its text tokenized and remapped
  - remapping_col_from : string, should be assigned to associated column name in remapping_df that matches original attribute of interest in om_df
  - remapping_col_to : string, should be assigned to associated column name in remapping_df that contains the final mapped entries

Returns
DataFrame – dataframe with remapped columns populated

text.visualize module

`pvops.text.visualize.visualize_attribute_connectivity(om_df, om_col_dict, figsize=(20, 10), attribute_colors=['lightgreen', 'cornflowerblue'], edge_width_scalar=10, graph_aargs={})`

Visualize a knowledge graph which shows the frequency of combinations between attributes `ATTRIBUTE1_COL` and `ATTRIBUTE2_COL`

NOW USES BIPARTITE LAYOUT ATTRIBUTE2_COL is colored using a colormap.

Parameters

- **om_df** (*DataFrame*) – A pandas dataframe containing O&M data, which contains columns specified in om_col_dict
- **om_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names to be used in visualization:

```
{
    'attribute1_col' : string,
    'attribute2_col' : string
}
```

- **figsize** (*tuple*) – Figure size, defaults to (20,10)
- **attribute_colors** (*list[str]*) – List of two strings which designate the colors for Attribute1 and Attribute 2, respectively.
- **edge_width_scalar** (*numeric*) – Weight utilized to scale widths based on number of connections between Attribute 1 and Attribute 2. Larger values will produce larger widths, and smaller values will produce smaller widths.
- **graph_aargs** (*dict*) – Optional, arguments passed to networkx graph drawer. Suggested attributes to pass:
  - with_labels=True
  - font_weight='bold'
  - node_size=19000
  - font_size=35

Returns

- *Matplotlib axis,*
- *networkx graph*

`pvops.text.visualize.`**`visualize_attribute_timeseries`**(*om_df*, *om_col_dict*, *date_structure='%Y-%m'*, *figsize=(12, 6)*, *cmap_name='brg'*)

Visualize stacked bar chart of attribute frequency over time, where x-axis is time and y-axis is count, displaying separate bars for each label within the label column

Parameters

- **om_df** (*DataFrame*) – A pandas dataframe of O&M data, which contains columns in om_col_dict
- **om_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the get_dates fn
  - **label** (*string*), should be assigned to associated column name for the label/attribute of interest in om_df
  - **date** (*string*), should be assigned to associated column name for the dates relating to the documents in om_df
- **date_structure** (*str*) – Controls the resolution of the bar chart's timeseries Default : "%Y-%m". Can change to include finer resolutions (e.g., by including day, "%Y-%m-%d") or coarser resolutions (e.g., by year, "%Y")
- **figsize** (*tuple*) – Optional, figure size
- **cmap_name** (*str*) – Optional, color map name in matplotlib

Returns

Matplotlib figure instance

`pvops.text.visualize.visualize_classification_confusion_matrix(om_df, col_dict, title='')`

Visualize confusion matrix comparing known categorical values, and predicted categorical values.

Parameters

- **om_df** (*DataFrame*) – A pandas dataframe containing O&M data, which contains columns specified in om_col_dict
- **col_dict** (*dict of {str : str}*) – A dictionary that contains the column names needed:
  - data : string, should be assigned to associated column which stores the tokenized text logs
  - attribute_col : string, will be assigned to attribute column and used to create new attribute_col
  - predicted_col : string, will be used to create keyword search label column
- **title** (*str*) – Optional, title of plot

Returns

Matplotlib figure instance

`pvops.text.visualize.visualize_cluster_entropy(doc2vec, eval_kmeans, om_df, data_cols, ks, cmap_name='brg')`

Visualize entropy of embedding space parition. Currently only supports doc2vec embedding.

Parameters

- **doc2vec** (*Doc2Vec model instance*) – Instance of gensim.models.doc2vec.Doc2Vec
- **eval_kmeans** (*callable*) – Callable cluster fit function For instance,

```
def eval_kmeans(X,k):
    km = KMeans(n_clusters=k)
    km.fit(X)
    return km
```

- **om_df** (*DataFrame*) – A pandas dataframe containing O&M data, which contains columns specified in om_col_dict
- **data_cols** (*list*) – List of column names (str) which have text data.
- **ks** (*list*) – List of k parameters required for the clustering mechanic *eval_kmeans*
- **cmap_name** – Optional, color map

Returns

Matplotlib figure instance

`pvops.text.visualize.visualize_document_clusters(cluster_tokens, min_frequency=20)`

Visualize words most frequently occurring in a cluster. Especially useful when visualizing the results of an unsupervised partitioning of documents.

Parameters

- **cluster_tokens** (*list*) – List of tokenized documents
- **min_frequency** (*int*) – Minimum number of occurrences that a word must have in a cluster for it to be visualized

Returns

Matplotlib figure instance

`pvops.text.visualize.visualize_word_frequency_plot(tokenized_words, title='', font_size=16, graph_aargs={})`

Visualize the frequency distribution of words within a set of documents

Parameters

- **tokenized_words** (*list*) – List of tokenized words
- **title** (*str*) – Optional, title of plot
- **font_size** (*int*) – Optional, font size
- **aargs** – Optional, other parameters passed to nltk.FreqDist.plot()

Returns

Matplotlib figure instance

1.3.2 text2time module

text2time.preprocess module

These functions focus on pre-processing user O&M and production data to create visualizations of the merged data

`pvops.text2time.preprocess.data_site_na(pom_df, df_col_dict)`

Drops rows where site-ID is missing (NAN) within either production or O&M data.

Parameters

- **pom_df** (*DataFrame*) – A data frame corresponding to either the production or O&M data.
- **df_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names associated with the input *pom_df* and contains at least:
  - **siteid** (*string*), should be assigned to column name for user's site-ID

Returns

- **pom_df** (*DataFrame*) – An updated version of the input data frame, where rows with site-IDs of NAN are dropped.
- **addressed** (*DataFrame*) – A data frame showing rows from the input that were removed by this function.

`pvops.text2time.preprocess.om_date_convert(om_df, om_col_dict, toffset=0.0)`

Converts dates from string format to date time object in O&M dataframe.

Parameters

- **om_df** (*DataFrame*) – A data frame corresponding to O&M data.
- **om_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names associated with the O&M data, which consist of at least:
  - **datestart** (*string*), should be assigned to column name for O&M event start date in om_df
  - **dateend** (*string*), should be assigned to column name for O&M event end date in om_df
- **toffset** (*float*) – Value that specifies how many hours the O&M data should be shifted by in case time-stamps in production data and O&M data don't align as they should

Returns

DataFrame – An updated version of the input dataframe, but with time-stamps converted to localized (time-zone agnostic) date-time objects.

`pvops.text2time.preprocess.om_datelogic_check(om_df, om_col_dict, om_dflag='swap')`

Addresses issues with O&M dates where the start of an event is listed as occurring after its end. These row are either dropped or the dates are swapped, depending on the user's preference.

Parameters

- **om_df** (*DataFrame*) – A data frame corresponding to O&M data.
- **om_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names associated with the O&M data, which consist of at least:
  - **datestart** (*string*), should be assigned to column name for associated O&M event start date in om_df
  - **dateend** (*string*), should be assigned to column name for associated O&M event end date in om_df
- **om_dflag** (*str*) – A flag that specifies how to address rows where the start of an event occurs after its conclusion. A flag of 'drop' will drop those rows, and a flag of 'swap' swap the two dates for that row.

Returns

- **om_df** (*DataFrame*) – An updated version of the input dataframe, but with O&M data quality issues addressed to ensure the start of an event precedes the event end date.
- **addressed** (*DataFrame*) – A data frame showing rows from the input that were addressed by this function.

`pvops.text2time.preprocess.om_nadate_process(om_df, om_col_dict, om_dendflag='drop')`

Addresses issues with O&M dataframe where dates are missing (NAN). Two operations are performed : 1) rows are dropped where start of an event is missing and (2) rows where the conclusion of an event is NAN can either be dropped or marked with the time at which program is run, depending on the user's preference.

Parameters

- **om_df** (*DataFrame*) – A data frame corresponding to O&M data.
- **om_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names associated with the O&M data, which consist of at least:
  - **datestart** (*string*), should be assigned to column name for user's O&M event start-date
  - **dateend** (*string*), should be assigned to column name for user's O&M event end-date
- **om_dendflag** (*str*) – A flag that specifies how to address rows where the conclusion of an event is missing (NAN). A flag of 'drop' will drop those rows, and a flag of 'today' will replace the NAN with the time at which the program is run. Any other value will leave the rows untouched.

Returns

- **om_df** (*DataFrame*) – An updated version of the input dataframe, but with no missing timestamps in the O&M data.
- **addressed** (*DataFrame*) – A data frame showing rows from the input that were addressed by this function.

`pvops.text2time.preprocess.prod_date_convert(prod_df, prod_col_dict, toffset=0.0)`

Converts dates from string format to datetime format in production dataframe.

Parameters

- **prod_df** (*DataFrame*) – A data frame corresponding to production data.

- **prod_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names associated with the production data, which consist of at least:
  - **timestamp** (*string*), should be assigned to user's time-stamp column name
- **toffset** (*float*) – Value that specifies how many hours the production data should be shifted by in case time-stamps in production data and O&M data don't align as they should.

Returns

DataFrame – An updated version of the input dataframe, but with time-stamps converted to localized (time-zone agnostic) date-time objects.

`pvops.text2time.preprocess.`**`prod_nadate_process`**`(`*prod_df*, *prod_col_dict*, *pnadrop=False*`)`

Processes rows of production data frame for missing time-stamp info (NAN).

Parameters

- **prod_df** (*DataFrame*) – A data frame corresponding to production data.
- **prod_df_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names associated with the production data, which consist of at least:
  - **timestamp** (*string*), should be assigned to associated time-stamp column name in prod_df
- **pnadrop** (*bool*) – Boolean flag that determines what to do with rows where time-stamp is missing. A value of *True* will drop these rows. Leaving the default value of *False* will identify rows with missing time-stamps for the user, but the function will output the same input data frame with no modifications.

Returns

- **prod_df** (*DataFrame*) – The output data frame. If pflag = 'drop', an updated version of the input data frame is output, but rows with missing time-stamps are removed. If default value is maintained, the input data frame is output with no modifications.
- **addressed** (*DataFrame*) – A data frame showing rows from the input that were addressed or identified by this function.

text2time.utils module

These helper functions focus on performing secondary calcuations from the O&M and production data to create visualizations of the merged data

`pvops.text2time.utils.`**`interpolate_data`**`(`*prod_df*, *om_df*, *prod_col_dict*, *om_col_dict*, *om_cols_to_translate=['asset', 'prod_impact']*`)`

Provides general overview of the overlapping production and O&M data.

Parameters

- **prod_df** (*DataFrame*) – A data frame corresponding to the production data after having been processed by the perf_om_NA_qc function. This data frame needs the columns specified in prod_col_dict.
- **om_df** (*DataFrame*) – A data frame corresponding to the O&M data after having been processed by the perf_om_NA_qc function. This data frame needs the columns specified in om_col_dict.
- **prod_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the production data
  - **siteid** (*string*), should be assigned to associated site-ID column name in prod_df

  - **timestamp** (*string*), should be assigned to associated time-stamp column name in prod_df
  - **energyprod** (*string*), should be assigned to associated production column name in prod_df
  - **irradiance** (*string*), should be assigned to associated irradiance column name in prod_df
- **om_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the O&M data
  - **siteid** (*string*), should be assigned to associated site-ID column name in om_df
  - **datestart** (*string*), should be assigned to associated O&M event start-date column name in om_df
  - **dateend** (*string*), should be assigned to associated O&M event end-date column name in om_df
  - Others specified in om_cols_to_translate
- **om_cols_to_translate** (*list*) – List of om_col_dict keys to translate into prod_df

Returns

- **prod_output** (*DataFrame*) – A data frame that includes statistics for the production data per site in the data frame. Two statistical parameters are calculated and assigned to separate columns:
  - **Actual # Time Stamps** (*datetime.datetime*), total number of overlapping production time-stamps
  - **Max # Time Stamps** (*datetime.datetime*), maximum number of production time-stamps, including NANs
- **om_out** (*DataFrame*) – A data frame that includes statistics for the O&M data per site in the data frame. Three statistical parameters are calculated and assigned to separate columns:
  - **Earliest Event Start** (*datetime.datetime*), column that specifies timestamp of earliest start of all events per site.
  - **Latest Event End** (*datetime.datetime*), column that specifies timestamp for latest conclusion of all events per site.
  - **Total Events** (*int*), column that specifies total number of events per site

`pvops.text2time.utils.om_summary_stats(om_df, meta_df, om_col_dict, meta_col_dict)`

Adds columns to OM dataframe capturing statistics (e.g., event duration, month of occurrence, and age). Latter is calculated by using corresponding site commissioning date within the metadata dataframe.

Parameters

- **om_df** (*DataFrame*) – A data frame corresponding to the O&M data after having been pre-processed by the QC and overlappingDFs functions. This data frame needs to have the columns specified in om_col_dict.
- **meta_df** (*DataFrame*) – A data frame corresponding to the metadata that contains columns specified in meta_col_dict.
- **om_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the O&M data which consist of at least:
  - **siteid** (*string*), should be assigned to column name for associated site-ID
  - **datestart** (*string*), should be assigned to column name for associated O&M event start-date
  - **dateend** (*string*), should be assigned to column name for associated O&M event end-date

- **eventdur** (*string*), should be assigned to column name desired for calculated event duration (calculated here, in hours)
- **modatestart** (*string*), should be assigned to column name desired for month of event start (calculated here)
- **agedatestart** (*string*), should be assigned to column name desired for calculated age of site when event started (calculated here, in days)

- **meta_col_dict** (*dict*) – A dictionary that contains the column names relevant for the metadata
  - **siteid** (*string*), should be assigned to associated site-ID column name in meta_df
  - **COD** (*string*), should be asigned to column name corresponding to associated commisioning dates for all sites captured in om_df

Returns

om_df (*DataFrame*) – An updated version of the input dataframe, but with three new columns added for visualizations: event duration, month of event occurrence, and age of system at time of event occurrence. See om_col_dict for mapping of expected variables to user-defined variables.

`pvops.text2time.utils.`**`overlapping_data`**`(`*prod_df*, *om_df*, *prod_col_dict*, *om_col_dict*`)`

Finds the overlapping time-range between the production data and O&M data for any given site. The outputs are a truncated version of the input data frames, that contains only data with overlapping dates between the two DFs.

Parameters

- **prod_df** (*DataFrame*) – A data frame corresponding to the production data after having been processed by the perf_om_NA_qc function. This data frame needs the columns specified in prod_col_dict. The time-stamp column should not have any NANs for proper operation of this function.
- **om_df** (*DataFrame*) – A data frame corresponding to the O&M data after having been processed by the perf_om_NA_qc function. This data frame needs the columns specified in om_col_dict. The time-stamp columns should not have any NANs for proper operation of this function.
- **prod_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the production data
  - **siteid** (*string*), should be assigned to associated site-ID column name in prod_df
  - **timestamp** (*string*), should be assigned to associated time-stamp column name in prod_df
  - **energyprod** (*string*), should be assigned to associated production column name in prod_df
  - **irradiance** (*string*), should be assigned to associated irradiance column name in prod_df
- **om_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the O&M data
  - **siteid** (*string*), should be assigned to associated site-ID column name in om_df
  - **datestart** (*string*), should be assigned to associated O&M event start-date column name in om_df
  - **dateend** (*string*), should be assigned to associated O&M event end-date column name in om_df

Returns

- **prod_df** (*DataFrame*) – Production data frame similar to the input data frame, but truncated to only contain data that overlaps in time with the O&M data.

- **om_df** (*DataFrame*) – O&M data frame similar to the input data frame, but truncated to only contain data that overlaps in time with the production data.

`pvops.text2time.utils.prod_anomalies(prod_df, prod_col_dict, minval=1.0, repval=nan, ffill=True)`

For production data with cumulative energy entries, 1) addresses time-stamps where production unexpectedly drops to near zero and 2) replaces unexpected production drops with NANs or with user-specified value. If unexpected production drops are replaced with NANs and if 'ffill' is set to 'True' in the input argument, a forward-fill method is used to replace the unexpected drops.

Parameters

- **prod_df** (*DataFrame*) – A data frame corresponding to production data were production is logged on a cumulative basis.
- **prod_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names associated with the production data, which consist of at least:
  - **energyprod** (*string*), should be assigned to the associated cumulative production column name in prod_df
- **minval** (*float*) – Cutoff value for production data that determines where anomalies are defined. Any production values below minval will be addressed by this function. Default minval is 1.0
- **repval** (*float*) – Value that should replace the anomalies in a cumulative production data format. Default value is numpy's NAN.
- **ffill** (*boolean*) – Boolean flag that determines whether NANs in production column in prod_df should be filled using a forward-fill method.

Returns

- **prod_df** (*DataFrame*) – An updated version of the input dataframe, but with zero production values converted to user's preference.
- **addressed** (*DataFrame*) – A data frame showing rows from the input that were addressed by this function.

`pvops.text2time.utils.prod_quant(prod_df, prod_col_dict, comp_type, ecumu=True)`

Compares performance of observed production data in relation to an expected baseline

Parameters

- **prod_df** (*DataFrame*) – A data frame corresponding to the production data after having been processed by the QC and overlappingDFs functions. This data frame needs at least the columns specified in prod_col_dict.
- **prod_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the production data
  - **siteid** (*string*), should be assigned to associated site-ID column name in prod_df
  - **timestamp** (*string*), should be assigned to associated time-stamp column name in prod_df
  - **energyprod** (*string*), should be assigned to associated production column name in prod_df
  - **baseline** (*string*), should be assigned to associated expected baseline production column name in prod_df
  - **compared** (*string*), should be assigned to column name desired for quantified production data (calculated here)
  - **energy_pstep** (*string*), should be assigned to column name desired for energy per time-step (calculated here)

- **comp_type** (*str*) – Flag that specifies how the energy production should be compared to the expected baseline. A flag of 'diff' shows the subtracted difference between the two (baseline - observed). A flag of 'norm' shows the ratio of the two (observed/baseline)
- **ecumu** (*bool*) – Boolean flag that specifies whether the production (energy output) data is input as cumulative information ("True") or on a per time-step basis ("False").

Returns

DataFrame – A data frame similar to the input, with an added column for the performance comparisons

`pvops.text2time.utils.summarize_overlaps(prod_df, om_df, prod_col_dict, om_col_dict)`

Provides general overview of the overlapping production and O&M data.

Parameters

- **prod_df** (*DataFrame*) – A data frame corresponding to the production data after having been processed by the perf_om_NA_qc function. This data frame needs the columns specified in prod_col_dict.
- **om_df** (*DataFrame*) – A data frame corresponding to the O&M data after having been processed by the perf_om_NA_qc function. This data frame needs the columns specified in om_col_dict.
- **prod_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the production data
  - **siteid** (*string*), should be assigned to associated site-ID column name in prod_df
  - **timestamp** (*string*), should be assigned to associated time-stamp column name in prod_df
  - **energyprod** (*string*), should be assigned to associated production column name in prod_df
  - **irradiance** (*string*), should be assigned to associated irradiance column name in prod_df
- **om_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the O&M data
  - **siteid** (*string*), should be assigned to associated site-ID column name in om_df
  - **datestart** (*string*), should be assigned to associated O&M event start-date column name in om_df
  - **dateend** (*string*), should be assigned to associated O&M event end-date column name in om_df

Returns

- **prod_output** (*DataFrame*) – A data frame that includes statistics for the production data per site in the data frame. Two statistical parameters are calculated and assigned to separate columns:
  - **Actual # Time Stamps** (*datetime.datetime*), total number of overlapping production time-stamps
  - **Max # Time Stamps** (*datetime.datetime*), maximum number of production time-stamps, including NANs
- **om_out** (*DataFrame*) – A data frame that includes statistics for the O&M data per site in the data frame. Three statistical parameters are calculated and assigned to separate columns:
  - **Earliest Event Start** (*datetime.datetime*), column that specifies timestamp of earliest start of all events per site.

- **Latest Event End** (datetime.datetime*), column that specifies timestamp for latest conclusion of all events per site.
- **Total Events** (*int*), column that specifies total number of events per site

text2time.visualize module

These functions focus on visualizing the processed O&M and production data

`pvops.text2time.visualize.visualize_categorical_scatter(om_df, om_col_dict, cat_varx, cat_vary, fig_sets)`

Produces a seaborn categorical scatter plot to show the relationship between an O&M numerical column and a categorical column using sns.catplot()

Parameters

- **om_df** (*DataFrame*) – A data frame corresponding to the O&M data after having been preprocessed to address NANs and date consistency, and after applying the `om_summary_stats` function. This data frame needs at least the columns specified in om_col_dict.
- **om_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the O&M data
  - **eventdur** (*string*), should be assigned to column name desired for repair duration. This column is calculated by `om_summary_stats`
  - **agedatestart** (*string*), should be assigned to column name desired for age of site when event started. This column is calculated by `om_summary_stats`
- **cat_varx** (*str*) – Column name that contains categorical variable to be plotted
- **cat_vary** (*str*) – Column name that contains numerical variable to be plotted
- **fig_sets** (*dict*) – A dictionary that contains the settings to be used for the figure to be generated, and those settings should include:
  - **figsize** (*tuple*), which is a tuple of the figure settings (e.g. *(12,10)*)
  - **fontsize** (*int*), which is the desired font-size for the figure

Returns

None

`pvops.text2time.visualize.visualize_counts(om_df, om_col_dict, count_var, fig_sets)`

Produces a seaborn countplot of an O&M categorical column using sns.countplot()

Parameters

- **om_df** (*DataFrame*) – A data frame corresponding to the O&M data after having been preprocessed to address NANs and date consistency, and after applying the `om_summary_stats` function. This data frame needs at least the columns specified in om_col_dict.
- **om_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the O&M data
  - **siteid** (*string*), should be assigned to column name for associated site-ID in om_df.
  - **modatestart** (*string*), should be assigned to column name desired for month of event start. This column is calculated by `om_summary_stats`
- **count_var** (*str*) – Column name that contains categorical variable to be plotted

- **fig_sets** (*dict*) – A dictionary that contains the settings to be used for the figure to be generated, and those settings should include:
  - **figsize** (*tuple*), which is a tuple of the figure settings (e.g. *(12,10)*)
  - **fontsize** (*int*), which is the desired font-size for the figure

Returns
None

`pvops.text2time.visualize.visualize_om_prod_overlap(prod_df, om_df, prod_col_dict, om_col_dict, prod_fldr, e_cumu, be_cumu, samp_freq='H', pshift=0.0, baselineflag=True)`

Creates Plotly figures of performance data overlaid with coinciding O&M tickets. A separate figure for each site in the production data frame (prod_df) is generated.

Parameters

- **prod_df** (*DataFrame*) – A data frame corresponding to the performance data after (ideally) having been processed by the perf_om_NA_qc and overlappingDFs functions. This data frame needs to contain the columns specified in prod_col_dict.
- **om_df** (*DataFrame*) – A data frame corresponding to the O&M data after (ideally) having been processed by the perf_om_NA_qc and overlappingDFs functions. This data frame needs to contain the columns specified in om_col_dict.
- **prod_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the production data
  - **siteid** (*string*), should be assigned to associated site-ID column name in prod_df
  - **timestamp** (*string*), should be assigned to associated time-stamp column name in prod_df
  - **energyprod** (*string*), should be assigned to associated production column name in prod_df
  - **irradiance** (*string*), should be assigned to associated irradiance column name in prod_df. Data should be in [W/m^2].
- **om_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the O&M data
  - **siteid** (*string*), should be assigned to column name for user's site-ID
  - **datestart** (*string*), should be assigned to column name for user's O&M event start-date
  - **dateend** (*string*), should be assigned to column name for user's O&M event end-date
  - **workID** (*string*), should be assigned to column name for user's O&M unique event ID
  - **worktype** (*string*), should be assigned to column name for user's O&M ticket type (corrective, predictive, etc)
  - **asset** (*string*), should be assigned to column name for affected asset in user's O&M ticket
- **prod_fldr** (*str*) – Path to directory where plots should be saved.
- **e_cumu** (*bool*) – Boolean flag that specifies whether the production (energy output) data is input as cumulative information ("True") or on a per time-step basis ("False").
- **be_cumu** (*bool*) – Boolean that specifies whether the baseline production data is input as cumulative information ("True") or on a per time-step basis ("False").
- **samp_freq** (*str*) – Specifies how the performance data should be resampled. String value is any frequency that is valid for pandas.DataFrame.resample(). For example, a value of 'D' will resample on a daily basis, and a value of 'H' will resample on an hourly basis.

- **pshift** (*float*) – Value that specifies how many hours the performance data should be shifted by to help align performance data with O&M data. Mostly necessary when resampling frequencies are larger than an hour
- **baselineflag** (*bool*) – Boolean that specifies whether or not to display the baseline (i.e., expected production profile) as calculated with the irradiance data using the baseline production data. A value of 'True' will display the baseline production profile on the generated Plotly figures, and a value of 'False' will not.

Returns

list – List of Plotly figure handles generated by function for each site within prod_df.

1.3.3 timeseries module

timeseries.preprocess module

`pvops.timeseries.preprocess.establish_solar_loc(prod_df, prod_col_dict, meta_df, meta_col_dict)`

Adds solar position column using pvLib.

Parameters

- **prod_df** (*DataFrame*) – A data frame corresponding to production data containing a datetime index.
- **prod_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names associated with the production data, which consist of at least:
  - **siteid** (*string*), should be assigned to site-ID column name in prod_df
- **meta_df** (*DataFrame*) – A data frame corresponding to site metadata. At the least, the columns in meta_col_dict be present. The index must contain the site IDs used in prod_df.
- **meta_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the meta-data
  - **longitude** (*string*), should be assigned to site's longitude
  - **latitude** (*string*), should be assigned to site's latitude

Returns

- *Original dataframe (copied) with new timeseries solar position data using*
- *the same column name definitions provided in pvLib.*

`pvops.timeseries.preprocess.normalize_production_by_capacity(prod_df, prod_col_dict, meta_df, meta_col_dict)`

Normalize power by capacity. This preprocessing step is meant as a step prior to a modeling attempt where a model is trained on multiple sites simultaneously.

Parameters

- **prod_df** (*DataFrame*) – A data frame corresponding to production data.
- **prod_df_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names associated with the production data, which consist of at least:
  - **energyprod** (*string*), should be assigned to production data in prod_df
  - **siteid** (*string*), should be assigned to site-ID column name in prod_df

- **capacity_normalized_power** (*string*), should be assigned to a column name where the normalized output signal will be stored

- **meta_df** (*DataFrame*) – A data frame corresponding to site metadata. At the least, the columns in meta_col_dict be present.
- **meta_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the meta-data
  - **siteid** (*string*), should be assigned to site-ID column name
  - **dcsize** (*string*), should be assigned to column name corresponding to site's DC size

Returns

prod_df (*DataFrame*) – normalized production data

`pvops.timeseries.preprocess.`**`prod_inverter_clipping_filter`**(*prod_df*, *prod_col_dict*, *meta_df*, *meta_col_dict*, *model*, ***kwargs*)

Filter rows of production data frame according to performance and data quality

Parameters

- **prod_df** (*DataFrame*) – A data frame corresponding to production data.
- **prod_df_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names associated with the production data, which consist of at least:
  - **timestamp** (*string*), should be assigned to associated time-stamp column name in prod_df
  - **siteid** (*string*), should be assigned to site-ID column name in prod_df
  - **powerprod** (*string*), should be assigned to associated power production column name in prod_df
- **meta_df** (*DataFrame*) – A data frame corresponding to site metadata. At the least, the columns in meta_col_dict be present.
- **meta_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the meta-data
  - **siteid** (*string*), should be assigned to site-ID column name
  - **latitude** (*string*), should be assigned to column name corresponding to site's latitude
  - **longitude** (*string*), should be assigned to column name corresponding to site's longitude
- **model** (*str*) – A string distinguishing the inverter clipping detection model programmed in pvanalytics. Available options: ['geometric', 'threshold', 'levels']
- **kwargs** – Extra parameters passed to the relevant pvanalytics model. If none passed, defaults are used.

Returns

prod_df (*DataFrame*) – If drop=True, a filtered dataframe with clipping periods removed is returned.

`pvops.timeseries.preprocess.`**`prod_irradiance_filter`**(*prod_df*, *prod_col_dict*, *meta_df*, *meta_col_dict*, *drop=True*, *irradiance_type='ghi'*, *csi_max=1.1*)

Filter rows of production data frame according to performance and data quality.

THIS METHOD IS CURRENTLY IN DEVELOPMENT.

Parameters

- **prod_df** (*DataFrame*) – A data frame corresponding to production data.

- **prod_df_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names associated with the production data, which consist of at least:
  - **timestamp** (*string*), should be assigned to associated time-stamp column name in prod_df
  - **siteid** (*string*), should be assigned to site-ID column name in prod_df
  - **irradiance** (*string*), should be assigned to associated irradiance column name in prod_df
  - **clearsky_irr** (*string*), should be assigned to clearsky irradiance column name in prod_df
- **meta_df** (*DataFrame*) – A data frame corresponding to site metadata. At the least, the columns in meta_col_dict be present.
- **meta_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the meta-data
  - **siteid** (*string*), should be assigned to site-ID column name
  - **latitude** (*string*), should be assigned to column name corresponding to site's latitude
  - **longitude** (*string*), should be assigned to column name corresponding to site's longitude
- **irradiance_type** (*str*) – A string description of the irradiance_type which was passed in prod_df. Options: *ghi*, *dni*, *dhi*. In future, *poa* may be a feature.
- **csi_max** (*int*) – A pvanalytics parameter of maximum ratio of measured to clearsky (clearsky index).

Returns

- **prod_df** (*DataFrame*) – A dataframe with new **clearsky_irr** column. If drop=True, a filtered prod_df according to clearsky.
- **clearsky_mask** (*series*) – Returns True for each value where the clearsky index is less than or equal to csi_mask

timeseries models

timeseries.models.linear module

`class pvops.timeseries.models.linear.DefaultModel(time_weighted=None, estimators=None, verbose=0, X_parameters=[])`

Bases: `Model`, `TimeWeightedProcess`

Generate a simple model using the input data, without any data transposition.

`construct(X, y, data_split='train')`

`class pvops.timeseries.models.linear.Model(estimators=None)`

Bases: `object`

Linear model kernel

`predict()`

Predict using the model.

`train()`

Train the model.

class pvops.timeseries.models.linear.**PolynomialModel**(*degree=2*, *estimators=None*, *time_weighted=None*, *verbose=0*, *X_parameters=[]*, *exclude_params=[]*)

Bases: *Model*, *TimeWeightedProcess*

Add all interactions between terms with a degree.

construct(*X*, *y*, *data_split='train'*)

class pvops.timeseries.models.linear.**TimeWeightedProcess**(*verbose=0*)

Bases: object

Generate time-oriented dummy variables for linear regression. Available timeframes include "month", "season", and "hour".

time_weight(*X*, *time_weighted='season'*, *data_split='train'*)

pvops.timeseries.models.linear.**modeller**(*prod_col_dict*, *kernel_type='default'*, *time_weighted='month'*, *X_parameters=[]*, *Y_parameter=None*, *estimators=None*, *prod_df=None*, *test_split=0.2*, *train_df=None*, *test_df=None*, *degree=3*, *exclude_params=[]*, *verbose=0*)

Wrapper method to conduct the modelling of the timeseries data.

To input the data, there are two options.

- Option 1: include full production data in *prod_df* parameter and *test_split* so that the test split is conducted
- Option 2: conduct the test-train split prior to calling the function and pass in data under *test_df* and *train_df*

Parameters

- **prod_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the production data
  - siteid (*string*), should be assigned to site-ID column name in prod_df
  - **timestamp** (*string*), should be assigned to time-stamp column name in prod_df
  - **irradiance** (*string*), should be assigned to irradiance column name in prod_df, where data should be in [W/m^2]
  - **baseline** (*string*), should be assigned to preferred column name to capture model calculations in prod_df
  - **dcsize**, (*string*), should be assigned to preferred column name for site capacity in prod_df
  - **powerprod**, (*string*), should be assigned to the column name holding the power or energy production. This will be used as the output column if Y_parameter is not passed.
- **kernel_type** (*str*) – Type of kernel type for the statistical model
  - 'default', establishes a kernel where one component is instantiated in the model for each feature.
  - 'polynomial', a paraboiloidal polynomial with a dynamic number of covariates (Xs) and degrees (n). For example, with 2 covariates and a degree of 2, the formula would be: Y(, X) = _0 + _1 X_1 + _2 X_2 + _3 X_1 X_2 + _4 X_1^2 + _5 X_2^2
- **time_weighted** (*str or None*) – Interval for time-based feature generation. For each interval in this time-weight, a dummy variable is established in the model prior to training. Options include:
  - if 'hour', establish discrete model components for each hour of day

  - if 'month', establish discrete model components for each month
  - if 'season', establish discrete model components for each season
  - if None, no time-weighted dummy-variable generation is conducted.
- **X_parameters** (*list of str*) – List of prod_df column names used in the model
- **Y_parameter** (*str*) – Optional, name of the y column. Defaults to prod_col_dict['powerprod'].
- **estimators** (*dict*) – Optional, dictionary with key as regressor identifier (str) and value as a dictionary with key "estimator" and value the regressor instance following sklearn's base model convention: sklearn_docs.

```
estimators = {'OLS': {'estimator': LinearRegression()},
              'RANSAC': {'estimator': RANSACRegressor()}
              }
```

- **prod_df** (*DataFrame*) – A data frame corresponding to the production data used for model development and evaluation. This data frame needs at least the columns specified in prod_col_dict.
- **test_split** (*float*) – A value between 0 and 1 indicating the proportion of data used for testing. Only utilized if *prod_df* is specified. If you want to specify your own test-train splits, pass values to *test_df* and *train_df*.
- **test_df** (*DataFrame*) – A data frame corresponding to the test-split of the production data. Only needed if *prod_df* and *test_split* are not specified.
- **train_df** (*DataFrame*) – A data frame corresponding to the test-split of the production data. Only needed if *prod_df* and *test_split* are not specified.
- **degree** (*int*) – Utilized for 'polynomial' and 'polynomial_log' *kernel_type* options, this parameter defines the highest degree utilized in the polynomial kernel.
- **exclude_params** (*list*) – A list of parameter definitions (defined as lists) to be excluded in the model. For example, if want to exclude a parameter in a 4-covariate model that uses 1 degree on first covariate, 2 degrees on second covariate, and no degrees for 3rd and 4th covariates, you would specify a exclude_params as `[ [1,2,0,0] ]`. Multiple definitions can be added to list depending on how many terms need to be excluded.

  If a time_weighted parameter is selected, a time weighted definition will need to be appended to *each* exclusion definition. Continuing the example above, if one wants to exclude "hour 0" for the same term, then the exclude_params must be `[ [1,2,0,0,0] ]`, where the last 0 represents the time-weighted partition setting.
- **verbose** (*int*) – Define the specificity of the print statements during this function's execution.

Returns

- *model* – which is a `pvops.timeseries.models.linear.Model` object, has a useful attribute
- *estimators* – which allows access to model performance and data splitting information.
- *train_df* – which is the training split of prod_df
- *test_df* – which is the testing split of prod_df

`pvops.timeseries.models.linear.`**`predicter`**(*model*, *df*, *Y_parameter*, *X_parameters*, *prod_col_dict*, *verbose=0*)

timeseries.models.AIT module

class pvops.timeseries.models.AIT.**AIT**

Bases: *Processer*, *Predictor*

predict(*prod_df*, *prod_col_dict*)

predict_subset(*prod_df*, *scaler*, *model_terms*, *prod_col_dict*)

pvops.timeseries.models.AIT.**AIT_calc**(*prod_df*, *prod_col_dict*)

Calculates expected energy using measured irradiance based on trained regression model from field data. Plane-of-array irradiance is recommended when using the pre-trained AIT model.

Parameters

- **prod_df** (*DataFrame*) – A data frame corresponding to the production data
- **prod_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the production data
  - **irradiance** (*string*), should be assigned to irradiance column name in prod_df, where data should be in [W/m^2]
  - **dcsize**, (*string*), should be assigned to preferred column name for site capacity in prod_df
  - **energyprod**, (*string*), should be assigned to the column name holding the power or energy production. If this is passed, an evaluation will be provided.
  - **baseline**, (*string*), should be assigned to preferred column name to capture the calculations in prod_df

Example

```
production_col_dict = {'irradiance': 'irrad_poa_Wm2',
                      'ambient_temperature': 'temp_amb_C',
                      'dcsize': 'capacity_DC_kW',
                      'energyprod': 'energy_generated_kWh',
                      'baseline': 'predicted'
                      }
data = AIT_calc(data, production_col_dict)
```

Returns

DataFrame – A data frame for production data with a new column, the predicted energy

class pvops.timeseries.models.AIT.**Predictor**

Bases: `object`

Predictor class

apply_additive_polynomial_model(*model_terms*, *Xs*)

Predict energy using a model derived by pvOps.

Parameters

- **df** (*dataframe*) – Data containing columns with the values in the *prod_col_dict*
- **model_terms** (*list of tuples*) – Contain model coefficients and powers. For example,

```
[(0.29359785963294494, [1, 0]),
(0.754806343190528, [0, 1]),
(0.396833207207238, [1, 1]),
(-0.0588375219110795, [0, 0])]
```

- **prod_col_dict** (*dict*) – Dictionary mapping nicknamed parameters to the named parameters in the dataframe *df*.

Returns
Array of predicted energy values

`evaluate`(*real*, *pred*)

`class` `pvops.timeseries.models.AIT.`**`Processer`**

Bases: `object`

`check_data`(*data*, *prod_col_dict*)

timeseries.models.iec module

`pvops.timeseries.models.iec.`**`iec_calc`**(*prod_df*, *prod_col_dict*, *meta_df*, *meta_col_dict*, *gi_ref=1000.0*)

Calculates expected energy using measured irradiance based on IEC calculations.

Parameters

- **prod_df** (*DataFrame*) – A data frame corresponding to the production data after having been processed by the perf_om_NA_qc and overlappingDFs functions. This data frame needs at least the columns specified in prod_col_dict.
- **prod_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the production data
  - **siteid** (*string*), should be assigned to site-ID column name in prod_df
  - **timestamp** (*string*), should be assigned to time-stamp column name in prod_df
  - **irradiance** (*string*), **plane-of-array**. Should be assigned to irradiance column name in prod_df, where data should be in [W/m^2].
  - **baseline** (*string*), should be assigned to preferred column name to capture IEC calculations in prod_df
  - **dcsize**, (*string*), should be assigned to preferred column name for site capacity in prod_df
- **meta_df** (*DataFrame*) – A data frame corresponding to site metadata. At the least, the columns in meta_col_dict be present.
- **meta_col_dict** (*dict of {str : str}*) – A dictionary that contains the column names relevant for the meta-data
  - **siteid** (*string*), should be assigned to site-ID column name
  - **dcsize** (*string*), should be assigned to column name corresponding to site capacity, where data is in [kW]
- **gi_ref** (*float*) – reference plane of array irradiance in W/m^2 at which a site capacity is determined (default value is 1000 [W/m^2])

Returns
DataFrame – A data frame for production data with a new column, iecE, which is the predicted energy calculated based on the IEC standard using measured irradiance data

1.3.4 iv module

iv.extractor module

Derive the effective diode parameters from a set of input curves.

class pvops.iv.extractor.**BruteForceExtractor**(*input_df*, *current_col*, *voltage_col*, *irradiance_col*, *temperature_col*, *T_type*, *windspeed_col=None*, *Simulator_mod_specs=None*, *Simulator_pristine_condition=None*)

Bases: `object`

Process measured IV curves to extract diode parameters. Requires a set of curves to create Isc vs Irr and Voc vs Temp vs Isc(Irr)

Parameters

- **input_df** (*DataFrame*) – Contains IV curves with a datetime index
- **current_col** (*string*) – Indicates column where current values in IV curve are located; each cell is an array of current values in a single IV curve
- **voltage_col** (*string*) – Indicates column where voltage values in IV curve are located; each cell is an array of voltage values in a single IV curve
- **irradiance_col** (*string*) – Indicates column where irradiance value (W/m2)
- **temperature_col** (*string*) – Indicates column where temperature value (C)
- **T_type** (*string*) – Describe input temperature, either 'ambient' or 'module' or 'cell'

create_string_object(*iph*, *io*, *rs*, *rsh*, *nnsvth*)

f_multiple_samples(*params*)

fit_params(*cell_parameters*, *n_mods*, *bounds_func*, *user_func=None*, *verbose=0*)

Fit diode parameters from a set of IV curves.

Parameters

- **cell_parameters** (*dict*) – Cell-level parameters, usually extracted from the CEC database, which will be used as the initial guesses in the optimization process.
- **n_mods** (*int*) – if int, defines the number of modules in a string(1=simulate a single module)
- **bounds_func** (*function*) – Function to establish the bounded search space See below for an example:

```
def bounds_func(iph,io,rs,rsh,nnsvth,perc_adjust=0.5):
    return ((iph - 0.5*iph*perc_adjust, iph + 2*iph*perc_adjust),
            (io - 40*io*perc_adjust, io + 40*io*perc_adjust),
            (rs - 20*rs*perc_adjust, rs + 20*rs*perc_adjust),
            (rsh - 150*rsh*perc_adjust, rsh + 150*rsh*perc_adjust),
            (nnsvth - 10*nnsvth*perc_adjust, nnsvth +␣
↪10*nnsvth*perc_adjust))
```

- **user_func** (*function*) – Optional, a function similar to *self.create_string_object* which has the following inputs: *self, iph, io, rs, rsh, nnsvth*. This can be used to extract unique failure parameterization.
- **verbose** (*int*) – if verbose >= 1, print information about fitting if verbose >= 2, plot information about each iteration

iv.physics_utils module

`pvops.iv.physics_utils.T_to_tcell(POA, T, WS, T_type, a=-3.56, b=-0.075, delTcnd=3)`

Ambient temperature to cell temperature according to NREL weather-correction. See Dierauf *et al.* [DGK+13].

Parameters

- **Tamb** (*numerical,*) – Ambient temperature, in Celcius
- **WS** (*numerical,*) – Wind speed at height of 10 meters, in m/s
- **a, b, delTcnd** (*numerical,*) – Page 12 in [DGK+13]
- **T_type** (*string,*) – Describe input temperature, either 'ambient' or 'module'

Returns

numerical – Cell temperature, in Celcius

`pvops.iv.physics_utils.add_series(voltage_1, current_1, voltage_2=None, current_2=None, v_bypass=None)`

Adds two IV curves in series.

Parameters

- **voltage_1** (*numeric*) – Voltage for first IV curve [V]
- **current_1** (*numeric*) – Current for first IV curve [A]
- **voltage_2** (*numeric or None, default None*) – Voltage for second IV curve [V]
- **current_1** (*numeric or None, default None*) – Voltage for second IV curve [A]
- **v_bypass** (*float or None, default None*) – Forward (positive) turn-on voltage of bypass diode, e.g., 0.5V [V]

Returns

- **voltage** (*numeric*) – Voltage for combined IV curve [V]
- **current** (*numeric*) – Current for combined IV curve [V]

Note: Current for the combined IV curve is the sorted union of the current of the two input IV curves. At current values in the other IV curve, voltage is determined by linear interpolation. Voltage at current values outside an IV curve's range is determined by linear extrapolation.

If *voltage_2* and *current_2* are None, returns *(voltage_1, current_1)* to facilitate starting a loop over IV curves.

`pvops.iv.physics_utils.bypass(voltage, v_bypass)`

Limits voltage to greater than -v_bypass.

Parameters

- **voltage** (*numeric*) – Voltage for IV curve [V]
- **v_bypass** (*float or None, default None*) – Forward (positive) turn-on voltage of bypass diode, e.g., 0.5V [V]

Returns

voltage (*numeric*) – Voltage clipped to greater than -v-bpass

`pvops.iv.physics_utils.calculate_IVparams(v, c)`

Calculate parameters of IV curve.

This needs to be reworked: extrapolate parameters from linear region instead of hardcoded regions.

Parameters

- **x** (*numpy array*) – X-axis data
- **y** (*numpy array*) – Y-axis data
- **npts** (*int*) – Optional, number of points to resample curve
- **deg** (*int*) – Optional, polyfit degree

Returns

Dictionary of IV curve parameters

`pvops.iv.physics_utils.gt_correction(v, i, gact, tact, cecparams, n_units=1, option=3)`

IV Trace Correction using irradiance and temperature. Three correction options are provided, two of which are from an IEC standard.

Parameters

- **v** (*numpy array*) – Voltage array
- **i** (*numpy array*) – Current array
- **gact** (*float*) – Irradiance upon measurement of IV trace
- **tact** (*float*) – Temperature (C or K) upon measuremnt of IV trace
- **cecparams** (*dict*) – CEC database parameters, as extracted by *pvops.iv.utils.get_CEC_params*.
- **n_units** (*int*) – Number of units (cells or modules) in string, default 1
- **option** (*int*) – Correction method choice. See method for specifics.

Returns

- *vref* – Corrected voltage array
- *iref* – Corrected current array

`pvops.iv.physics_utils.intersection(x1, y1, x2, y2)`

Compute intersection of curves, y1=f(x1) and y2=f(x2). Adapted from https://stackoverflow.com/a/5462917

Parameters

- **x1** (*numpy array*) – X-axis data for curve 1
- **y1** (*numpy array*) – Y-axis data for curve 1
- **x2** (*numpy array*) – X-axis data for curve 2
- **y2** (*numpy array*) – Y-axis data for curve 2

Returns

intersection coordinates

`pvops.iv.physics_utils.iv_cutoff(Varr, Iarr, val)`

Cut IV curve greater than voltage *val* (usually 0)

Parameters

- **V** (*numpy array*) – Voltage array

- **I** (*numpy array*) – Current array
- **val** (*numeric*) – Filter threshold

Returns
V_cutoff, I_cutoff

`pvops.iv.physics_utils.smooth_curve(x, y, npts=50, deg=12)`

Smooth curve using a polyfit

Parameters

- **x** (*numpy array*) – X-axis data
- **y** (*numpy array*) – Y-axis data
- **npts** (*int*) – Optional, number of points to resample curve
- **deg** (*int*) – Optional, polyfit degree

Returns

- *smoothed x array*
- *smoothed y array*

`pvops.iv.physics_utils.voltage_pts(npts, v_oc, v_rbd)`

Provide voltage points for an IV curve.

Points range from v_brd to v_oc, with denser spacing at both limits. v=0 is included as the midpoint.

Based on method PVConstants.npts from pvmismatch

Parameters

- **npts** (*integer*) – Number of points in voltage array.
- **v_oc** (*float*) – Open circuit voltage [V]
- **v_rbd** (*float*) – Reverse bias diode voltage (negative value expected) [V]

Returns
array [V]

iv.preprocess module

`pvops.iv.preprocess.preprocess(input_df, resmpl_resolution, iv_col_dict, resmpl_cutoff=0.03, correct_gt=False, normalize_y=True, CECmodule_parameters=None, n_mods=None, gt_correct_option=3)`

IV processing function which supports irradiance & temperature correction

Parameters

- **input_df** (*DataFrame*)
- **resmpl_resolution** (*float*)
- **iv_col_dict** (*dict*)
- **resmpl_cutoff** (*float*)
- **correct_gt** (*bool*)
- **normalize_y** (*bool*)
- **CECmodule_parameters** (*None*)

- **n_mods** (*int*)
- **gt_correct_option** (*int*)

Returns
df (*DataFrame*)

iv.simulator module

class pvops.iv.simulator.**Simulator**(*mod_specs={'Jinko_Solar_Co___Ltd_JKM270PP_60': {'ncols': 6, 'nsubstrings': 3}}, pristine_condition={'E': 1000, 'Il_mult': 1, 'Io_mult': 1, 'Rs_mult': 1, 'Rsh_mult': 1, 'Tc': 50, 'identifier': 'pristine', 'nnsvth_mult': 1}, replacement_5params={'I_L_ref': None, 'I_o_ref': None, 'R_s': None, 'R_sh_ref': None, 'a_ref': None}, num_points_in_IV=200, simulation_method=2*)

Bases: `object`

An object which simulates Photovoltaic (PV) current-voltage (IV) curves with failures

Parameters

- **mod_specs** (*dict*) – Define the module and some definitions of that module which are not included in the the CEC database. The *key* in this dictionary is the name of the module in the CEC database. The *values* are *ncols*, which is the number of columns in the module, and *nsubstrings*, which is the number of substrings.
- **pristine_condition** (*dict*) – Defines the pristine condition. A full condition is defined as a dictionary with the following key/value pairs:

```
{
    'identifier': IDENTIFIER_NAME, # (str) Name used to define␣
↪condition
    'E': IRRADIANCE, # (numeric) Value of irradiance (Watts per␣
↪meter-squared)
    'Tc': CELL_TEMPERATURE, # (numeric) Multiplier usually less than␣
↪1
                            # to simulate a drop in Rsh
    'Rsh_mult': RSH_MULTIPLIER, # (numeric) Multiplier usually less␣
↪than 1
                                # to simulate a drop in RSH
    'Rs_mult': RS_MULTIPLIER, # (numeric) Multiplier usually less␣
↪than 1
                              # to simulate an increase in RS
    'Io_mult': IO_MULTIPLIER, # (numeric) Multiplier usually less␣
↪than 1
                              # to simulate a drop in IO
    'Il_mult': IL_MULTIPLIER, # (numeric) Multiplier usually less␣
↪than 1
                              # to simulate a drop in IL
    'nnsvth_mult': NNSVTH_MULTIPLIER, # (numeric) Multiplier usually␣
↪less
                                      # than 1 to simulate a drop in␣
↪NNSVTH, and therefore a_ref
    'modname': MODULE_NAME_IN_CECDB # (str) Module name in CEC␣
↪database
```

(continues on next page)

(continued from previous page)

```
                                        # (e.g. Jinko_Solar_Co___Ltd_
↪JKMS260P_60)
}
```

- **replacement_5params** (*dict*) – Optional, replace the definitions of the five electrical parameters, which normally are extracted from the CEC database. These parameters can be determined by the `IVProcessor` class

  Key/value pairs:

```
{
    'I_L_ref': None,
    'I_o_ref': None,
    'R_s': None,
    'R_sh_ref': None,
    'a_ref': None
}
```

- **simulation_method** (*int*) – Module simulation method (1 or 2)

  1) Avalanche breakdown model, as hypothesized in Ref.[1]

  2) : Add-on to method 1, includes a rebalancing of the I_sc prior to adding in series

Variables

- **multilevel_ivdata** (*dict*) – Dictionary containing the simulated IV curves
  - For nth-definition of string curves, `multilevel_ivdata['string']['STRING IDENTIFIER'][n]`
  - For nth-definition of module curves, `multilevel_ivdata['module']['MODULE IDENTIFIER'][n]`
  - For nth-definition of substring (substr_id = 1,2,3,...) curves, `multilevel_ivdata['module']['MODULE IDENTIFIER']['substr{sbstr_id}'][n]`
- **pristine_condition** (*dict*) – Dictionary of conditions defining the pristine case
- **module_parameters** (*dict*) – Dictionary of module-level parameters
- **cell_parameters** (*dict*) – Dictionary of cell-level parameters

BISHOP88_simulate_module(*mod_key*)

PVOPS_simulate_module(*mod_key*)

add_manual_conditions(*modcell*, *condition_dict*)

Create cell-level fault conditions manually

Parameters

- **modcell** (*dict*) – Key: name of the condition Value: list,
  - 1D list: Give a single situation for this condition
  - 2D list: Give multiple situations for this condition

[1] "Computer simulation of the effects of electrical mismatches in photovoltaic cell interconnection circuits" JW Bishop, Solar Cell (1988) DOI: 10.1016/0379-6787(88)90059-2

– A list where each value signifies a cell's condition.

If key is same as an existing key, the list is appended to list of scenarios which that key owns

- **condition_dict** (*dict*) – Define the numerical value written in modcell

Note: If the variable is not defined, values will default to those specified in the pristine condition, defined in __init__.

A full condition is defined as:

```
{ID: {'identifier': IDENTIFIER_NAME, # (str) Name used to define␣
↪condition
      'E': IRRADIANCE, # (numeric) Value of irradiance (Watts per␣
↪meter-squared)
      'Tc': CELL_TEMPERATURE, # (numeric) Value of cell temperature␣
↪(Celcius)
      'Rsh_mult': RSH_MULTIPLIER, # (numeric) Multiplier usually␣
↪less than 1
                                    # to simulate a drop in Rsh
      'Rs_mult': RS_MULTIPLIER, # (numeric) Multiplier usually␣
↪greater than 1
                                  # to simulate increase in Rs
      'Io_mult': IO_MULTIPLIER, # (numeric) Multiplier usually less␣
↪than 1
                                  # to simulate a drop in IO
      'Il_mult': IL_MULTIPLIER, # (numeric) Multiplier usually less␣
↪than 1
                                  # to simulate a drop in IL
      'nnsvth_mult': NNSVTH_MULTIPLIER # (numeric) Multiplier␣
↪usually less than 1 to
                                         # simulate a drop in NNSVTH,␣
↪and therefore a_ref
    }
}
```

add_preset_conditions(*fault_name*, *fault_condition*, *save_name=None*, ***kwargs*)

Create cell-level fault conditions from presets defined by authors

Parameters

- **fault_name** (*str*) – Options:
  - 'complete': entire module has fault_condition (e.g. Full module shading) Requires no other specifications e.g. add_preset_conditions('complete', fault_condition)
  - 'landscape': entire rows are affected by fault_condition (e.g. interrow shading) Requires specification of rows_aff e.g. add_preset_conditions('landscape', fault_condition, rows_aff = 2)
  - 'portrait': entire columns are affected by fault_condition (e.g. vegetation growth shading) Requires specification of cols_aff
    - e.g. add_preset_conditions('portrait', fault_condition, cols_aff = 2)

- 'pole': Place pole shadow over module Requires specification of width (integer), which designates the width of main shadow and requires light_shading fault_condition specification which specifies less intense shading on edges of shadow
  - Optional: pos = (left, right) designates the start and end of the pole shading, where left is number in the first column and right is number in last column if pos not specified, the positions are chosen randomly e.g. add_preset_conditions('pole', fault_condition, light_shading = light_fault_condition, width = 2, pos = (5, 56))
- 'bird_droppings': Random positions are chosen for bird_dropping simulations
  - Optional specification is n_droppings. If not specified, chosen as random number between 1 and the number of cells in a column e.g. add_preset_conditions('bird_droppings', fault_condition, n_droppings = 3)

- **fault_location** (*dict*) – Same dict as one shown in __init__.
- **kwargs** (*variables dependent on which fault_name you choose, see above*)

Tip: For a wider spectrum of cases, run all of these multiple times. Each time it's run, the case is saved

build_strings(*config_dict*)

Pass a dictionary into object memory

e.g. For 6 modules faulted with modcell specification 'complete'

```
config_dict = {
    'faulting_bottom_mods': [
        'pristine', 'pristine', 'pristine',
        'pristine', 'pristine', 'pristine',
        'complete', 'complete', 'complete',
        'complete', 'complete', 'complete'
        ]
    }
```

generate_many_samples(*identifier*, *N*, *distributions=None*, *default_sample=None*)

For cell *identifier*, create *N* more samples by randomly sampling a gaussian or truncated gaussian distribution.

Parameters

- **identifier** (*str*) – Cell identifier to upsample
- **N** (*int*) – Number of samples to generate
- **distributions** (*dict*) – Dictionary of distribution definitions, either gaussian or truncated gaussian. Each definition must note a 'mean' and 'std', however if 'low' and 'upp' thresholds are also included, then a truncated gaussian distribution will be generated.

  One does not need to define distributions for all parameters, only those that you want altered.

```
distributions = {
    'Rsh_mult':{'mean':None,
                'std': None,
                'low': None,
```

(continues on next page)

(continued from previous page)

```
                 'upp': None},
    'Rs_mult': {'mean':None,
                 'std': None,
                 'low': None,
                 'upp': None},
        ...
        # All keys in self.acceptible_keys
    }
```

- **default_sample** – If provided, use this sample to replace the parameters which do not have distributions specified. Else, uses the pristine condition as a reference.

print_info()

Display information about the established definitions

reset_conditions()

Reset failure conditions

sims_to_df(*focus=['string', 'module']*, *cutoff=False*)

Return the failure definitions as a dataframe.

Parameters

- **focus** (*list of string*) – Subset the definitions to a level of the system Currently available: 'substring', 'module', 'string'
- **cutoff** (*bool*) – Cutoff curves to only return on positive voltage domain

Returns

- *Dataframe with columns* –
  - 'current': IV trace current
  - 'voltage': IV trace voltage
  - 'E': Average irradiance for all samples used to build this array
  - 'T': Average cell temperature for all samples used to build this array
  - 'mode': failure name
  - 'level': level of system (i.e. module, string), as defined by the input *focus* parameter
- **#TODO** (*create focus for cell. For now, one can do it manually themselves.*)

simulate(*sample_limit=None*)

Simulate the cell, substring, module, and string-level IV curves using the defined conditions

Parameters

sample_limit (*int*) – Optional, used when want to restrict number of combinations of failures at the string level.

simulate_module(*mod_key*)

Wrapper method which simulates a module depending on the defined simulation_method.

Parameters

mod_key (*str*) – Module name as defined in condiction_dict and modcells

simulate_modules()

Simulates all instantiated modules

visualize(*lim=False*)

Run visualization suite to visualize information about the simulated curves.

visualize_cell_level_traces(*cell_identifier*, *cutoff=True*, *table=True*, *axs=None*)

Visualize IV curves for cell_identifier and tabulate the definitions.

Parameters

- **cell_identifier** (*str*) – Cell identifier. Call *self.print_info()* for full list.
- **cutoff** (*bool*) – If True, only visualize IV curves in positive voltage domain
- **table** (*bool*) – If True, append table to bottom of figure
- **axs** (*maplotlib axes*) – Matplotli subplots axes

Returns

matplotlib axes

visualize_module_configurations(*module_identifier*, *title=None*, *n_plots_atonce=3*)

Visualize failure locations on a module.

Parameters

- **module_identifier** (*int*) – Module identifier. Call *self.print_info()* for full list.
- **title** (*str*) – Optional, add this title to figure.
- **n_plots_atonce** (*int*) – Number of plots to render in a single figure.

Returns

- *matplotlib axes*
- **TODO** (*MAKE COLOR either 1) same as condition color from other tables*) – 2) colored by intensity of param definition, given a param (e.g. 'E')

visualize_multiple_cells_traces(*list_cell_identifiers*, *cutoff=True*)

Visualize multiple cell traces

Parameters

- **list_cell_identifiers** (*list*) – list of cell identifiers. call *self.print_info()* for full list.
- **cutoff** (*bool*) – If True, only visualize IV curves in positive voltage domain

Returns

matplotlib axes

visualize_specific_iv(*ax=None*, *string_identifier=None*, *module_identifier=None*, *substring_identifier=None*, *cutoff=True*, *correct_gt=False*)

Visualize a string, module, or substring IV curve. If the object has multiple definitions, all definitions will be plotted

Parameters

- **ax** (*matplotlib axes*) – Optional, pass an axes to add visualization
- **string_identifier** (*str*) – Optional, Identification of string definition
- **module_identifier** (*str*) – Optional, Identification of module definition
- **substring_identifier** (*str*) – Optional, Identification of module definition
- **cutoff** (*bool*) – If True, only visualize IV curves in positive voltage domain

- **correct_gt** (*bool*) – If True, correct curves according to irradiance and temperature Here, cutoff must also be True.

Returns
matplotlib axes

`pvops.iv.simulator.create_df(Varr, Iarr, POA, T, mode)`

Builds a dataframe from the given parameters

Parameters

- **Varr**
- **Iarr**
- **POA**
- **T**
- **mode**

Returns
df (*DataFrame*)

iv.utils module

`pvops.iv.utils.get_CEC_params(name, mod_spec)`

Query module-level parameters from CEC database and derive cell-level parameters.

Utilizing methods from pvsystem.retrieve_sam('CECMod')

Parameters

- **name** (*string*) – Representing module name in CEC database
- **mod_specs** (*dict*) – Provide 'ncols' and 'nsubstrings'

Returns
module_parameters (dict), cell_parameters (dict)

iv.timeseries_simulator module

`class pvops.iv.timeseries_simulator.IVTimeseriesGenerator(**iv_sim_kwargs)`

Bases: *Simulator*

`add_time_conditions(preset_mod_mapping, nmods=12)`

`generate(env_df, failures, iv_col_dict, identifier_col, plot_trends=False)`

Simulate a PV system

Parameters

- **env_df** (*dataframe*) – DataFrame containing irradiance ("E") and temperature ("T") columns
- **failures** (*list*) – List of timeseries_simulator.TimeseriesFailure objects

`class pvops.iv.timeseries_simulator.TimeseriesFailure`

Bases: `object`

add_interpolation(*specs_df*, *plot_trends=False*)

Add failure properties to specs_df

trend(*longterm_fcn_dict=None*, *annual_fcn_dict=None*, *daily_fcn_dict=None*, ***kwargs*)

Define a failure's trend across intraday (trending with time of day) and longterm timeframes.

Parameters

- **longterm_fcn_dict** (*dict*) – A dictionary where keys are the diode-multipliers in IVSimulator ('Rsh_mult', 'Rs_mult', 'Io_mult', 'Il_mult', 'nnsvth_mult') and values are either a function or a string. If a function, the function should be a mathematical operation as a *function of the number of float years since operation start*, a value on domain [0,inf), and outputs the chosen diode-multiplier's values across this timeseries. If a string, must use a pre-defined definition:

  – 'degrade' : degrade over time at specified rate. Specify rate by passing a definition for *degradation_rate*

  For example,

```
# 2 Ways of Doing Same Thing

# Method 1
longterm_fcn_dict = {
    'Rs_mult': lambda x : 1.005 * x
}
f = Failure()
f.trend(longterm_fcn_dict)

# Method 2
longterm_fcn_dict = {
    'Rs_mult': 'degrade'
}
f = Failure()
f.trend(longterm_fcn_dict,
        degradation_rate=1.005)
```

- **annual_fcn_dict** (*dict*) – A dictionary where keys are the diode-multipliers in IVSimulator ('Rsh_mult', 'Rs_mult', 'Io_mult', 'Il_mult', 'nnsvth_mult') and values are either a function or a string. If a function, the function should be a mathematical operation as a *function of the percentage through this year*, a value on domain [0,1], and outputs the chosen diode-multiplier's values across this timeseries. If a string, must use a pre-defined definition:

- **daily_fcn_dict** (*function or str*) – A dictionary where keys are the diode-multipliers in IVSimulator ('Rsh_mult', 'Rs_mult', 'Io_mult', 'Il_mult', 'nnsvth_mult') and values are either a function or a string. If a function, the function should be a mathematical operation as a *function of the percentage through this day*, a value on domain [0,1], and outputs the chosen diode-multiplier's values across this timeseries. If a string, must use a pre-defined definition:

iv.models.nn module

class pvops.iv.models.nn.**IVClassifier**(*nn_config*)

Bases: object

predict(*batch_size=8*)

Predict using the trained model.

Parameters

batch_size (*int*) – Number of samples per gradient update

structure(*train*, *test*)

Structure the data according to the chosen network model's input structure.

Parameters

- **train** (*dataframe*) – Train data containing IV data and associated features
- **test** (*dataframe*) – Test data containing IV data and associated features
- **nn_config** (*dict*) – Parameters used for the IV trace classifier.

train()

Train neural network with stratified KFold.

pvops.iv.models.nn.**balance_df**(*df*, *iv_col_dict*, *balance_tactic='truncate'*)

Balance data so that an equal number of samples are found at each unique *ycol* definition.

Parameters

- **bigdf** (*dataframe*) – Dataframe containing the *ycol* column.
- **iv_col_dict** (*dict*) – Dictionary containing at least the following definition: - **mode** (*str*), column in *df* which holds the definitions which must contain a balanced number of samples for each unique definition.
- **balance_tactic** (*str*) – mode balancing tactic, either "truncate" or "gravitate". Truncate will utilize the exact same number of samples for each category. Gravitate will sway the original number of samples towards a central target.

Returns

balanced_df (*DataFrame*) – balanced according to the *balance_tactic*.

pvops.iv.models.nn.**classify_curves**(*df*, *iv_col_dict*, *nn_config*)

Build and evaluate an IV trace failure *mode* classifier.

Parameters

- **df** (*dataframe*) – Data with columns in *iv_col_dict*
- **iv_col_dict** (*dict*) – Dictionary containing definitions for the column names in *df* **mode** (*str*): column name for failure mode identifier
- **nn_config** (*dict*) – Parameters used for the IV trace classifier.

pvops.iv.models.nn.**feature_generation**(*bigdf*, *iv_col_dict*, *pristine_mode_identifier='Pristine array'*)

Generate features of an IV curve data set including: 1. Current differential between a sample IV curve and a pristine IV curve 2. Finite difference of the IV curve along the y-axis, indicating the slope of the cuve.

Parameters

- **bigdf** (*dataframe*) – Dataframe holding columns from *iv_col_dict*, except for the *derivative* and *current_diff* which are calculated here.

- **iv_col_dict** (*dict*) – Dictionary containing definitions for the column names in *df*
  - **current** (*str*): column name for IV current arrays.
  - **voltage** (*str*): column name for IV voltage arrays.
  - **mode** (*str*): column name for failure mode identifier.
  - **irradiance** (*str*): column name for the irradiance definition.
  - **temperature** (*str*): column name for the temperature definition.
  - **derivative** (*str*): column name for the finite derivative, as calculated in this function.
  - **current_diff** (*str*): column name for current differential, as calculated in *get_diff_array*.
- **pristine_mode_identifier** (*str*) – Pristine array identifier. The pristine curve is utilized in `get_diff_array`. If multiple rows exist at this `pristine_mode_identifier`, the one with the highest irradiance and lowest temperature definitions is chosen.

Returns

- **all_V** (*array*) – Combined voltage array
- **diff** (*array*) – Current differential calculation

`pvops.iv.models.nn.`**`get_diff_array`**(*sample_V*, *sample_I*, *pristine_V*, *pristine_I*, *debug=False*)

Generate IV current differential between sample and pristine.

Parameters

- **sample_V** (*array*) – Voltage array for a sample's IV curve
- **sample_I** (*array*) – Current array for a sample's IV curve
- **pristine_V** (*array*) – Voltage array for a pristine IV curve
- **pristine_I** (*array*) – Current array for a pristine IV curve

Returns

- **all_V** (*array*) – Combined voltage array
- **diff** (*array*) – Current differential calculation

`pvops.iv.models.nn.`**`plot_profiles`**(*df*, *colx*, *coly*, *iv_col_dict*, *cmap_name='brg'*)

Plot curves also with area colorizations to display the deviation in definitions.

Parameters

- **df** (*dataframe*) – Dataframe containing the *colx*, *coly*, and iv_col_dict['mode'] column
- **colx** (*str*) – Column containing x-axis array of data on each sample.
- **coly** (*str*) – Column containing y-axis array of data on each sample.
- **iv_col_dict** (*dict*) – Dictionary containing at least the following definition: - **mode** (*str*), column in *df* which holds the definitions which must contain a balanced number of samples for each unique definition.
- **cmap_name** (*str*) – Matplotlib colormap.

Returns

matplotlib figure

1.4 Developing pvOps

1.4.1 Installation

To maintain a local installation, developers should use the following commands:

```
git clone https://github.com/sandialabs/pvOps.git
cd pvops
pip install -e .
```

1.4.2 Testing

To test locally, run:

```
pytest pvops
```

at the root of the repository. Note that this requires the installation of pytest.

1.4.3 Linting

Pvops uses flake8 to maintain code standards. To lint locally using the same filters required by pvops CI/CD pipeline, run the following command at the root of the repository:

```
flake8 . --count --statistics --show-source --ignore=E402,E203,E266,E501,W503,F403,F401,
↪E402,W291,E302,W391,W292,F405,E722,W504,E121,E125,E712
```

Note that this requires the installation of flake8.

1.4.4 Documentation

Building docs

To build docs locally, navigate to `pvops/docs` and run:

```
make html
```

After building, the static html files can be found in `_build/html`.

Docstrings

The pvOps documentation adheres to NumPy style docstrings. Not only does this help to keep a consistent style, but it is also necessary for the API documentation to be parsed and displayed correctly. For an example of what this should look like:

```
def func(arg1, arg2):
"""Summary line.

Extended description of function.
```

(continues on next page)

(continued from previous page)

```
    Parameters
    ----------
    arg1 : int
        Description of arg1
    arg2 : str
        Description of arg2

    Returns
    -------
    bool
        Description of return value

    """
    return True
```

Additional examples can be found in the napoleon documentation.

Extending Documentation

When adding new functionality to the repository, it is important to check that it is being properly documented in the API documentation. Most of this is automatic. For example, if a function is added to `pvops.text.visualize` with a proper docstring, there is no more work to do. However, when new files are created they must be added to the appropriate page in `docs/pages/apidoc` so that the automatic documentation recognizes it.

New pages should be placed into `docs/pages`, and linked to in `index.html`, or another page. It is recommended to use absolute paths (starting from the root of the documentation) when linking anything.

1.5 Contributing

Thank you for wanting to contribute to this library! We will try to make this an easy process for you. It is recommended that you read the *development* page so that you can lint and test before submitting code. Checking that your PR passes the required testing and linting procedures will speed up the acceptance of your PR.

1.5.1 Issues and bug reporting

To report issues or bugs please create a new issue on the pvops issues page. Before submitting your bug report, please perform a cursory search to see if the problem has been already reported. If it has been reported, and the issue is still open, add a comment to the existing issue instead of opening a new issue.

Guidelines for effective bug reporting

- Use a clear descriptive title for the issue.
- Describe the steps to reproduce the problem, the behavior you observed after following the steps, and the expected behavior.
- If possible, provide a simple example of the bug using pvOps example data.
- When relevant, provide information on your computing environment (operating system, python version, pvOps version or commit).

- For runtime errors, provide a function call stack.

1.5.2 Contributing code

Software developers, within the core development team and external collaborators, are expected to follow standard practices to document and test new code. Software developers interested in contributing to the project are encouraged to create a Fork of the project and submit a Pull Request (PR) using GitHub. Pull requests will be reviewed by the core development team. Create a PR or help with other PRs which are in the library by referencing pvops PR page.

Guidelines for preparing and submitting pull-requests

- Use a clear descriptive title for your pull-requests
- Describe if your submission is a bugfix, documentation update, or a feature enhancement. Provide a concise description of your proposed changes.
- Provide references to open issues, if applicable, to provide the necessary context to understand your pull request
- Make sure that your pull-request merges cleanly with the *master* branch of pvOps. When working on a feature, always create your feature branch off of the latest *master* commit
- Ensure that appropriate documentation and tests accompany any added features.

1.6 What's New

These are new features and improvements of note in each release.

1.6.1 0.3.0 (November 9 2023)

This release incorporates new functions and addresses depreciated commands in some of the package dependencies.

Functionality

- Updated visualize_attribute_connectivity to use bipartite graph layout (updated function).
- IV related dependencies moved to an installation extra (install using *pip install pvops[iv]*).
- Removed deprecated normalization parameters in ML pipeline (bug fix).
- Updated code to fix deprecation/future warnings.

Testing

- Added Python 3.11 to the test environment.

Documentation

- Fix small typos in index.rst.
- Renamed references to examples as tutorials for consistency.
- Updated docs to refer to modules as modules, rather than packages.
- Updated RTD config to install doc requirements using the package installation extra
- Removed redundant boilerplate in development.rst
- Update tested versions in documentation
- Added links to tutorials where appropriate in the user guide.
- Added a simplified version of the module overview table from the JOSS manuscript to the homepage of the documentation.
- Added statement of need to homepage
- Fixed image embed in tutorial
- Added dates to what's new sections
- Expanded patch notes to include recent tags.
- Deleted WIP docs pages to remove "not included in any toctree" errors.
- Added nbsphinx gallery view to tutorials page.
- Added more content to abbreviations page.

Tutorials

- Rename pvOps examples to tutorials for consistency throughout repository.
- Linked to tutorials in README.
- Added a description of data in timeseries tutorial.
- Removed redundant plots in timeseries tutorial.

Other

- Added copyright and license attributes to pvops.
- Removed manifest.in (not needed).
- Removed docs/init.py (not a module).
- Chose more appropriate author/copyright in setup.py and conf.py.
- Added version to pvops (pvops.__version__ now exists).
- Removed external licenses (determined to be unnecessary by legal).
- Renamed citation file and updated version number.
- Added noxfile for dev task running.
- Removed unused docker files
- Add standard python files to gitignore
- Removed redundant requirements files

- Pinned documentation related requirements

1.6.2 0.2.0 (August 9 2023)

This release incorporates new functions and addresses depreciated commands in some of the package dependencies.

Documentation

- Doc pages "makeover" in preparation for JOSS publication
- Added additional context and detail to example notebooks.
- Added module guides
- Added contributing pages

New Features

- Added *get_attributes_from_keywords* to text.classify
- Added *get_keywords_of_interest* to text.preprocess
- Added *remap_words_in_text* to text.visualize

1.6.3 0.1.9 (November 21 2022)

Includes updated documentation and fixes for dependency issues

Documentation

- Docstrings polished across the package.
- Resolved documentation build errors and warnings

1.6.4 0.1.8 (Jan 14 2022)

Includes a data-derived expected energy model trained using machine learning methods. Associated example is also within the documentation.

Functionality

- Added AIT model

Other

- Add citation.cif

1.6.5 0.1.7 (September 20 2021)

Updated functions for data processing (text and timeseries) analysis. Also includes IV curve functions

1.6.6 Beta

New features and bug fixes are predominant in the beta versions.

New features

- IV trace classification framework built according to literature (PR #25)
- Timeseries IV simulation for highly customizable degradation of system parameters (PR #28)
- Leverage pvlib solarposition package to populate content per site (PR #32)
- Add coefficient-level evaluations linear models (PR #32)
- Give user ability to input own test-train splits to linear modeller (PR #32)
- Remap attributes function must retain the unaltered attributes (PR #32)
- Interpolate O&M data onto production data where overlaps exist (PR #32)

Bug fixes

- Basic package fixes to README (PR #27) and documentation configuration (PR #24)
- Fix IV simulator bug for edge case where two IV curves added have equal I_{sc} (PR #30)
- Neural network configuration referencing in 1D CNN (PR #32)

Docs

- Update how to reference pvOps (PR #33)

Tests

- Removed python 3.6 test support due to https://github.com/actions/setup-python/issues/162.

1.6.7 Alpha

The original release of pvOps consists mostly of new features.

New features

- *text* module added which conducts natural language processing on Operations & Maintenance (O&M) tickets, or other.
- *text2time* module investigates the relationship between the production timeseries data and the O&M tickets.
- *timeseries* module conducts timeseries preprocessing and modeling
- *iv* incorporates the ability to simulate current-voltage (IV) curves under different environmental, load, and failure conditions.

Documentation

- Built original website
- Add whatsnew
- Add jupyter notebook embeddings

Testing

- Built comprehensive tests with pytest
- Connected tests to automated testing pipeline

1.7 References

1.7.1 Citing Us

If using this package, please cite us using the following

```
Bonney et al., (2023). pvOps: a Python package for empirical analysis of photovoltaic␣
↪field data.
Journal of Open Source Software, 8(91), 5755, https://doi.org/10.21105/joss.05755
```

In BibTex format:

```
@article{Bonney2023,
  doi = {10.21105/joss.05755},
  url = {https://doi.org/10.21105/joss.05755},
  year = {2023},
  publisher = {The Open Journal},
  volume = {8},
  number = {91},
  pages = {5755},
  author = {Kirk L. Bonney and Thushara Gunda and Michael W. Hopwood and Hector Mendoza␣
↪and Nicole D. Jackson},
```

(continues on next page)

(continued from previous page)

```
title = {pvOps: a Python package for empirical analysis of photovoltaic field data},
journal = {Journal of Open Source Software} }
```

We also utilize content from other packages. See the NOTICE/ directory on our GitHub!

Additionally, some of our own content comes from published papers. See the following external references.

1.7.2 External references

BIBLIOGRAPHY

[Bis88] J.W. Bishop. Computer simulation of the effects of electrical mismatches in photovoltaic cell interconnection circuits. *Solar Cells*, 25(1):73–89, 1988. URL: https://www.sciencedirect.com/science/article/pii/0379678788900592, doi:https://doi.org/10.1016/0379-6787(88)90059-2.

[DJN+18] Michael G Deceglie, Dirk Jordan, Ambarish Nag, Christopher A Deline, and Adam Shinn. Rdtools: an open source python library for pv degradation analysis. Technical Report, National Renewable Energy Lab.(NREL), Golden, CO (United States), 2018.

[DGK+13] T. Dierauf, A. Growitz, S. Kurtz, J. L. B. Cruz, E. Riley, and C. Hansen. Weather-corrected performance ratio. 4 2013. URL: https://www.osti.gov/biblio/1078057, doi:10.2172/1078057.

[HHM18] William F Holmgren, Clifford W Hansen, and Mark A Mikofski. Pvlib python: a python package for modeling solar energy systems. *Journal of Open Source Software*, 3(29):884, 2018. doi:10.21105/joss.00884.

[HG22] Michael W. Hopwood and Thushara Gunda. Generation of data-driven expected energy models for photovoltaic systems. *Applied Sciences*, 2022. URL: https://www.mdpi.com/2076-3417/12/4/1872, doi:10.3390/app12041872.

[HGSW20] Michael W. Hopwood, Thushara Gunda, Hubert Seigneur, and Joseph Walters. Neural network-based classification of string-level iv curves from physically-induced failures of photovoltaic modules. *IEEE Access*, 8():161480–161487, 2020. doi:10.1109/ACCESS.2020.3021577.

[HSBS22] Michael W. Hopwood, Joshua S. Stein, Jennifer L. Braid, and Hubert P. Seigneur. Physics-based method for generating fully synthetic iv curve training datasets for machine learning classification of pv failures. *Energies*, 2022. URL: https://www.mdpi.com/1996-1073/15/14/5085, doi:10.3390/en15145085.

[KS16] Katherine A Klise and Joshua S Stein. Performance monitoring using pecos (v. 0.1). Technical Report, Sandia National Laboraties, 2016. doi:10.2172/1734479.

[MHG21] Hector Mendoza, Michael Hopwood, and Thushara Gunda. Pvops: improving operational assessments through data fusion. In *2021 IEEE 48th Photovoltaic Specialists Conference (PVSC)*, volume, 0112–0119. 2021. doi:10.1109/PVSC43889.2021.9518439.

[PKL+20] Benjamin G Pierce, Ahmad Maroof Karimi, JiQi Liu, Roger H French, and Jennifer L Braid. Identifying degradation modes of photovoltaic modules using unsupervised machine learning on electroluminescense images. In *2020 47th IEEE Photovoltaic Specialists Conference (PVSC)*, 1850–1855. IEEE, 2020. doi:10.1109/PVSC45281.2020.9301021.

PYTHON MODULE INDEX

p

INDEX

I

M

N

O

P

R

S

T

U

V