NEMSEER

Release 1.0.7

Abhijith (Abi) Prakash

Mar 13, 2024

CONTENTS

A package for downloading and handling historical National Electricity Market (NEM) forecast data produced by the Australian Energy Market Operator (AEMO).

CHAPTER

ONE

INSTALLATION

```
pip install nemseer
```

Many `nemseer` use-cases require NEMOSIS, which can also be installed using `pip`:

```
pip install nemosis
```

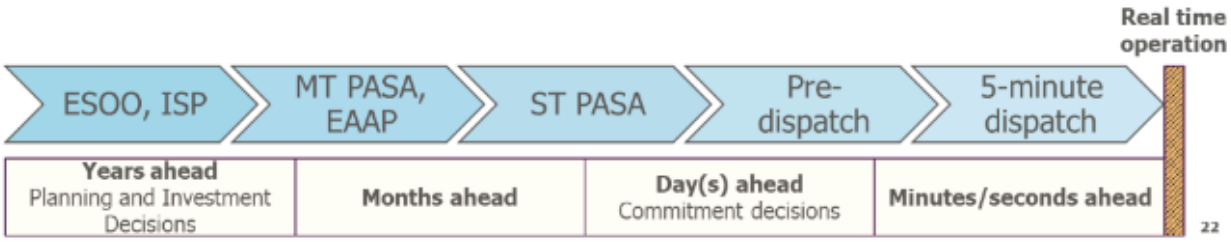
CHAPTER

TWO

OVERVIEW

`nemseer` allows you to access historical AEMO pre-dispatch and Projected Assessment of System Adequacy (PASA) forecast[1] data available through the MMSDM Historical Data SQLLoader. `nemseer` can then compile this data into pandas DataFrames or xarray Datasets.

An overview of `nemseer` functionality and potential use-cases are provided in the JOSS paper for this package.



Source: AEMC.

Source: Reserve services in the National Electricity Market, AEMC, 2021

Whereas PASA processes are primarily used to assess resource adequacy based on technical inputs and assumptions for resources in the market (i.e. used to answer questions such as *"can operational demand be met in the forecast horizon with a sufficient safety (reserve) margin?"*), pre-dispatch processes incorporate the latest set of market participant offers and thus produce regional prices forecasts for energy and frequency control ancillary services (FCAS). Overviews of the various pre-dispatch and PASA processes can be found in the glossary.

`nemseer` enables you to download and work with data for the following forecast types. Where available, AEMO process and table descriptions are linked:

1. 5-minute pre-dispatch (`P5MIN`: Table descriptions)
2. Pre-dispatch (`PREDISPATCH`: Table descriptions)
3. Pre-dispatch Projected Assessment of System Adequacy (`PDPASA`: Tables and Descriptions)
4. Short Term Projected Assessment of System Adequacy (`STPASA`: Table descriptions)
5. Medium Term Projected Assessment of System Adequacy (`MTPASA`: Table descriptions)

Another helpful reference for PASA information is AEMO's Reliability Standard Implementation Guidelines.

[1] We use the term *"forecast"* loosely, especially given that these *"forecasts"* change once participants update offer information (e.g. through rebidding) or submit revised resource availabilities and energy constraints. Both of these are intended outcomes of these *"ahead processes"*, which are run to provide system and market information to participants to inform their decision-making. However, to avoid confusion and to ensure consistency with the language used by AEMO, we use the terms *"forecast"* (or outputs) and *"forecast types"* (or ahead processes) in `nemseer`.

2.1 ST PASA Replacement Project

Note that the methodologies for PD PASA and ST PASA are being reviewed by AEMO. In particular, the ST PASA Replacement project will combine PD PASA and ST PASA into ST PASA. For more detail, refer to the final determination of the rule change and the AEMO ST PASA Replacement Project home page.

CHAPTER

THREE

USAGE

3.1 Glossary

The glossary contains overviews of the PASA and pre-dispatch processes, and descriptions of terminology used in `nemseer`.

3.2 Quick start

Check out the Quick start for guide on to use `nemseer`.

3.3 Examples

Some use case examples have been included in the Examples section of the documentation.

CHAPTER

FOUR

SUPPORT

If you are having an issue with this software that has not already been raised in the issues register, please raise a new issue.

CHAPTER

FIVE

CONTRIBUTING

Interested in contributing? Check out the contributing guidelines, which also includes steps to install `nemseer` for development.

Please note that this project is released with a Code of Conduct. By contributing to this project, you agree to abide by its terms.

CHAPTER

SIX

CITATION

If you use `nemseer`, please cite the JOSS paper for this package

If you use code or analysis from any of the demand error and/or price convergence examples in the documentation, please also cite `NEMOSIS` via this conference paper

CHAPTER

SEVEN

LICENSES

`nemseer` was created by Abhijith Prakash. It is licensed under the terms of GNU GPL-3.0-or-later licences.

The content within the documentation for this project is licensed under a Creative Commons Attribution 4.0 International License.

CHAPTER

EIGHT

CREDITS

`nemseer` was created with `cookiecutter` and the `py-pkgs-cookiecutter` template.

Development of `nemseer` was funded by the UNSW Digital Grid Futures Institute.

8.1 Contributor Acknowledgements

Thanks to:

- Nicholas Gorman for reviewing `nemseer` code
- Krisztina Katona for reviewing and improving the glossary
- Dylan McConnell for assistance in interpreting PASA run types
- Declan Heim for suggesting improvements to `nemseer` examples

CHAPTER

NINE

FULL TABLE OF CONTENTS

9.1 Quick start

`nemseer` lets you download raw historical forecast data from the *MMSDM Historical Data SQLLoader*, cache it in the *parquet* format and use `nemseer` to assemble and filter forecast data into a `pandas.DataFrame` or `xarray.Dataset` for further analysis. Assembled queries can optionally be saved to a *processed cache*.

9.1.1 Core concepts and information for users

Glossary

Refer to the *glossary* for an overview of key terminology used in `nemseer`. This includes descriptions of datetimes accepted as inputs in `nemseer`:

- *run_start*
- *run_end*
- *forecasted_start*
- *forecasted_end*

Note: AEMO ahead process tables with forecasted results typically have *three* datetime columns:

1. A *forecasted time* which the forecast outputs pertain to
2. A nominal *run time*. For most forecast types, this is reported in the `RUN_DATETIME` column.
3. An *actual run time*
    - The *actual* run time can differ from the *nominal* time. For example:
        - The 18:15 `P5MIN` run (`RUN_DATETIME`) may actually be run/published at 18:10 (`LASTCHANGED`)
        - The 18:30 `PREDISPATCH` run (`PREDISPATCHSEQNO`, which is parsed into `PREDISPATCH_RUN_DATETIME` by `nemseer`) may actually be run/published at 18:02 (`LASTCHANGED`)

The glossary also provides an overview of the various ahead processes run by AEMO, including:

- *P5MIN*
- *PREDISPATCH*
- *PDPASA*
- *STPASA*

- *MTPASA*

Parquet

Parquet files can be loaded using data analysis packages such as pandas, and work well with packages for handling large on-memory/cluster datasets (e.g. polars and dask). Parquet offers efficient data compression and columnar data storage, which can mean faster queries from file. Parquet files also store file metadata (which can include table schema).

Types of compiled data

`nemseer` has functionality that allows a user to compile data into two types of in-memory data structures:

- pandas DataFrames. Pandas is a widely-used Python package for manipulating data.
- Multi-dimensional xarray Datasets. xarray is intended for handling and querying data across multiple dimensions (e.g. the regional price forecast for a particular *forecasted time* from a range of *run times*)
  - For more information, refer to the *Getting started* section of the xarray documentation. The xarray tutorial is also an excellent resource.
  - Converting to xarray can be *memory-intensive*.

Managing memory

Some queries via `nemseer` may require a large amount of memory to complete. While memory use is query-specific, we suggest that `nemseer` be used on a system with at least 8GB of RAM. 16GB+ is preferable.

However, there are some things you can try if you do run into issues with memory. The suggestions below also apply to large queries on powerful computers:

1. You can use `nemseer` to simply download raw data as CSVs or to then cache data in the parquet format. Once you have a cache, you can use tools like polars or dask to process chunks of data in parallel. You may be able to reduce peak memory usage this way. It should be noted that `nemseer` converts a single AEMO CSV into a single parquet file. That is, it does not partition the parquet store.
2. Conversion to `xarray.Dataset` can be memory intensive. As this usually occurs when the data to be converted has a high number of dimensions (as determined by `nemseer`), `nemseer` will print a warning prior to attempting to convert any such data. While xarray integrates with dask, this functionality is contingent on loading data from a netCDF file.

Processed cache

The *processed_cache* is optional, but may be useful for some users. Specifying a path for this argument will lead to `nemseer` saving queries (i.e. requested data filtered based on user-supplied *run times* and *forecasted times*) as *parquet* (if the `pandas.DataFrame` data structure is specified) or netCDF (if the `xarray.Dataset` data structure is specified).

If subsequent `nemseer` queries include this *processed_cache*, `nemseer` will check file metadata of the relevant file types to see if a particular table query has already been saved. If it has, `nemseer` will compile data from the *processed_cache*.

Note: Because `nemseer` looks at metadata stored *in* each file, it does not care about the file name as long as file extensions are preserved (i.e. `*.parquet`, `*.nc`). As such, files in the *processed_cache* can be renamed from default file names assigned by `nemseer`.

Warning: Saving to netCDF will let you load xarray objects into memory. However, saving these datasets to netCDF files may take up large amounts of hard disk space.

Deprecated tables

If tables have been deprecated, `nemseer` will print a warning when the table is being downloaded. Deprecated tables are documented *here*.

9.1.2 What can I query?

`nemseer` has functionality to tell you what you can query. This includes valid *forecast types*, *months and years* for which data is available and requestable *tables*.

Note: While these functions allow you to explicitly query this information, it's worth noting that functions for *compiling data* and *downloading raw data* validate inputs and provide feedback when invalid inputs (such as invalid forecast types or data date ranges) are supplied.

Forecast types

You can access valid *forecast types* with the command below.

```
>>> import nemseer
>>> nemseer.forecast_types
('P5MIN', 'PREDISPATCH', 'PDPASA', 'STPASA', 'MTPASA')
```

Date range of available data

The years and months available via AEMO's *MMSDM Historical Data SQLLoader* can be queried as follows.

```
>>> import nemseer
>>> nemseer.get_data_daterange()
{...}
```

Table availability

You can also see which tables are available for a given year, month and *forecast type*.

Below, we fetch *pre-dispatch* tables available for January 2022 (i.e. this month would include or be between *run_start* and *run_end*):

```
>>> import nemseer
>>> nemseer.get_tables(2022, 1, "PREDISPATCH")
['CASESOLUTION', 'CONSTRAINT', 'CONSTRAINT_D', 'INTERCONNECTORRES', 'INTERCONNECTORRES_D
↪', 'INTERCONNECTR_SENS_D', 'LOAD', 'LOAD_D', 'MNSPBIDTRK', 'OFFERTRK', 'PRICE',
↪'PRICESENSITIVITIE_D', 'PRICE_D', 'REGIONSUM', 'REGIONSUM_D', 'SCENARIODEMAND',
↪'SCENARIODEMANDTRK']
```

Descriptions of tables and columns

AEMO's MMS Data Model documentation describes the tables and columns that are available via `nemseer`.

To use this documentation, first find the package that corresponds to the *forecast type* you are interested in. Some of the package names differ from the *forecast type*s that `nemseer` uses. For example, tables for *STPASA* are obtained using the `nemseer` forecast_type`STPASA`, whereas the same tables are documented by AEMO within the `STPASA_SOLUTION` package.

Once you locate the right package, you can find descriptions of the tables made available by `nemseer`.

`PREDISPATCH` tables

For some pre-dispatch table (`CONSTRAINT`, `LOAD`, `PRICE`, `INTERCONNECTORRES` and `REGIONSUM`), there are two types of tables. Those ending with `_D` only contain the latest forecast for a particular interval, whereas those without `_D` have all relevant forecasts for an interval of interest.

9.1.3 Compiling data

The main use case of `nemseer` is to download raw data (if it is not available in the *raw_cache*) and then compile it into a data format for further analysis/processing. To do this, `nemseer` has *`compile_data`*.

This function:

1. Downloads the relevant raw data[1] and converts it into *parquet* in the *raw_cache*.
2. If it's supplied, interacts with a *processed_cache* (see *below*).
3. Returns a dictionary consisting of compiled `pandas.DataFrame`s or `xarray.Dataset`s (i.e. assembled and filtered based on the supplied *run times* and *forecasted times*) mapped to their corresponding table name.

For example, we can compile *STPASA* forecast data contained in the `CASESOLUTION` and `CONSTRAINTSOLUTION` tables. The query below will filter *run times* between "2021/02/01 00:00" and "2021/02/28 00:00" and *forecasted times* between 09:00 on March 1 and 12:00 on March 3. The returned `dict` maps each of the requested tables to their corresponding assembled and filtered datasets. These datasets are `pandas.DataFrame` as `data_format="df"` (this is the default for this argument).

```
>>> import nemseer
>>> data = nemseer.compile_data(
... run_start="2021/02/01 00:00",
... run_end="2021/02/28 00:00",
... forecasted_start="2021/03/01 09:00",
... forecasted_end="2021/03/01 12:00",
... forecast_type="STPASA",
... tables=["CASESOLUTION", "CONSTRAINTSOLUTION"],
... raw_cache="./nemseer_cache/",
... data_format="df",
... )
INFO: Downloading and unzipping CASESOLUTION for 2/2021
INFO: Downloading and unzipping CONSTRAINTSOLUTION for 2/2021
INFO: Converting PUBLIC_DVD_STPASA_CASESOLUTION_202102010000.CSV to parquet
INFO: Converting PUBLIC_DVD_STPASA_CONSTRAINTSOLUTION_202102010000.CSV to parquet
```

(continues on next page)

[1] As explained by the glossary note for AEMO's *MMSDM Historical Data SQLLOader*, `nemseer` accesses monthly raw data. The month in the raw data filename corresponds to the month in which the forecast was run. As such, a *forecasted_start* and *forecasted_end* in the same month may actually require two raw data files (i.e. *run_start* and *run_end* may be in different months).

(continued from previous page)

```
>>> data.keys()
dict_keys(['CASESOLUTION', 'CONSTRAINTSOLUTION'])
```

In the example above we include argument names, but these can be omitted.

You can also just query a single table, such as the query below:

```
>>> import nemseer
>>> data = nemseer.compile_data(
... "2021/02/01 00:00",
... "2021/02/28 00:00",
... "2021/03/01 09:00",
... "2021/03/01 12:00",
... "STPASA",
... "REGIONSOLUTION",
... "./nemseer_cache/",
... )
INFO: Downloading and unzipping REGIONSOLUTION for 2/2021
INFO: Converting PUBLIC_DVD_STPASA_REGIONSOLUTION_202102010000.CSV to parquet
>>> data.keys()
dict_keys(['REGIONSOLUTION'])
```

Note: `nemseer` also accepts datetimes with seconds specified, so long as the seconds are `00`. This is because the datetime fields that are relevant to `nemseer` functionality are specified to the nearest minute.

With datetimes specified down to seconds, you can the same datetimes for `nemseer` as you would for other related tools, such as `NEMOSIS` or `NEMED`.

We can also compile data to an `xarray.Dataset`. To do this, we need to set `data_format="xr"`:

```
>>> import nemseer
>>> data = nemseer.compile_data(
... "2021/02/01 00:00",
... "2021/02/28 00:00",
... "2021/02/28 00:30",
... "2021/02/28 00:55",
... "P5MIN",
... "REGIONSOLUTION",
... "./nemseer_cache/",
... data_format="xr",
... )
INFO: Downloading and unzipping REGIONSOLUTION for 2/2021
INFO: Converting PUBLIC_DVD_P5MIN_REGIONSOLUTION_202102010000.CSV to parquet
INFO: Converting REGIONSOLUTION data to xarray.
>>> data.keys()
dict_keys(['REGIONSOLUTION'])
>>> type(data['REGIONSOLUTION'])
<class 'xarray.core.dataset.Dataset'>
```

Compiling data to a processed cache

As outlined *above*, compiled data can be saved to the *processed_cache* as parquet (if `data_format` = "df") or as netCDF files (if `data_format` = "xr").

If the same *processed_cache* is supplied to subsequent queries, `nemseer` will check whether any portion of the subsequent query has already been saved in the *processed_cache*. If it has, `nemseer` will load data from the *processed_cache*, thereby bypassing any download/raw data compilation.

With a supplied *processed_cache*, we can save the query to parquet (`data_format` = "df") or to netCDF (`data_format` = "xr"):

```
>>> import nemseer
>>> data = nemseer.compile_data(
... "2021/02/01 00:00",
... "2021/02/28 00:00",
... "2021/03/01 09:00",
... "2021/03/01 12:00",
... "STPASA",
... "REGIONSOLUTION",
... "./nemseer_cache/",
... processed_cache="./processed_cache/",
... )
INFO: Query raw data already downloaded to nemseer_cache
INFO: Writing REGIONSOLUTION to the processed cache as parquet
```

And if this saved query is a portion of another subsequent query, `nemseer` will load data from the *processed_cache*:

```
>>> import nemseer
>>> data = nemseer.compile_data(
... "2021/02/01 00:00",
... "2021/02/28 00:00",
... "2021/03/01 09:00",
... "2021/03/01 12:00",
... "STPASA",
... ["CASESOLUTION", "REGIONSOLUTION"],
... "./nemseer_cache/",
... processed_cache="./processed_cache/",
... )
INFO: Query raw data already downloaded to nemseer_cache
INFO: Compiling REGIONSOLUTION data from the processed cache
INFO: Writing CASESOLUTION to the processed cache as parquet
```

Validation and feedback

compile_data will validate user inputs and provide feedback on valid inputs. Specifically, it validates:

1. Basic datetime chronologies (e.g. *run_end* not before *run_start*)
2. Whether the requested *forecast type* and table type(s) are valid
3. Whether the requested *run times* and `forecasted times` are valid for the requested *forecast type*. In other words, forecasts that are run between *run_start* and *run_end* only produce data for a certain range of *forecasted times*. This varies between *forecast types*. For more information, refer to the forecast-specific datetime *validators*.

Getting valid run times for a set of forecasted times

If you're interested in forecast data for a particular datetime range (i.e. between *forecasted_start* and *forecasted_end*) but not sure what the valid *run times* for this range are, you can use `generate_runtimes`.

This function returns the first *run_start* and last *run_end* between which forecast outputs for the *forecasted times* are available.

In the example below, we request *run times* that contain data for the *forecasted times* used in the *compiling data examples*:

```
>>> import nemseer
>>> nemseer.generate_runtimes("2021/03/01 09:00", "2021/03/01 12:00", "STPASA")
('2021/02/22 14:00', '2021/02/28 14:00')
```

You can see that in the *compiling data examples* we had a wider *run time* range. This is fine since filtering will only retain *run times* that contain the requested *forecasted times*. The inverse is not true: `compile_data` will raise errors if the requested *forecasted times* are not valid/do not have forecast outputs for the requested *run times*.

9.1.4 Downloading raw data

You can download raw data[Page 22, 1] to a cache using `download_raw_data()`. This function only downloads data to the *raw_cache*.

CSVs can be retained by specifying `keep_csv=True`.

Unlike *compiling data*, only one set of datetimes needs to be provided (though these datetimes are keyword arguments for this function):

1. Provide `forecasted_start` and `forecasted_end` only. `nemseer` will determine the appropriate `run_start` and `run_end` for this forecasted range (via `nemseer.generate_runtimes()`) and download the corresponding raw data.
2. Provide `run_start` and `run_end` only. Dummy forecasted times are used.

```
>>> import nemseer
>>> nemseer.download_raw_data(
... forecast_type="P5MIN",
... tables="REGIONSOLUTION",
... raw_cache="./nemseer_cache/",
... forecasted_start="2020/01/02 00:00",
... forecasted_end="2020/01/02 00:30",
... keep_csv=False
... )
INFO: Downloading and unzipping REGIONSOLUTION for 1/2020
INFO: Converting PUBLIC_DVD_P5MIN_REGIONSOLUTION_202001010000.CSV to parquet
```

Alternatively, provide *run times*:

```
>>> import nemseer
>>> nemseer.download_raw_data(
... forecast_type="P5MIN",
... tables="REGIONSOLUTION",
... raw_cache="./nemseer_cache/",
... run_start="2021/01/02 00:00",
... run_end="2021/01/02 00:30",
```

(continues on next page)

(continued from previous page)

```
... keep_csv=False
... )
INFO: Downloading and unzipping REGIONSOLUTION for 1/2021
INFO: Converting PUBLIC_DVD_P5MIN_REGIONSOLUTION_202101010000.CSV to parquet
```

9.2 Glossary

run_start
: Forecasts with run times at or after this datetime are queried.

run_end
: Forecasts with run times before or at this datetime are queried.

run time
run times
run_time
: The time at which a forecast is *nominally* run.

- *P5MIN*: Every 5 minutes beginning on the hour
- *PREDISPATCH*/*PDPASA*: Every 30 minutes beginning on the hour
- *STPASA*: On the hour, either every hour or every two hours
  - Frequency of runs was increased in 2021
- *MTPASA*: Run every week on Tuesdays, datetime of run will vary

forecasted_start
: Forecasts pertaining to times at or after this datetime are retained.

forecasted_end
: Forecasts pertaining to times before or at this datetime are retained.

forecasted time
forecasted times
forecasted_time
: The time to which a forecast's outputs pertain.

forecast type
forecast types
: The term used in `nemseer` to refer to AEMO's ahead processes (as outlined in *pre-dispatch* and *PASA*).

actual run time
: The actual time at which the forecast run is executed/published. This is often reported in the `LASTCHANGED` column.

MTPASA
STPASA
PDPASA
PASA
: Projected Assessment of System Adequacy. PASA processes are focused on assessing reliability/resource adequacy.

- `PDPASA` is run with a similar frequency and horizon to (30-minute) *pre-dispatch*, and `STPASA` is run at least every two hours (it is currently run every hour) following the horizon covered by `PDPASA`. Both attempt to

maximise generation reserves available to the system given forecasts for demand and variable renewable energy generation, a simplified set of forecasted network constraints and participant-submitted resource availabilities and energy constraints. Together, they assess reliability for the next 7 trading days[1]. There are 3 run types for `PDPASA` and `STPASA` (N.B. the descriptions below are based on an interpretation of AEMO documentation as it does not explicitly describe the run types that appear in `PDPASA`/`STPASA` tables):

1. `RELIABILITY_LRC` likely corresponds to the Capacity Adequacy (CA) run. In the CA run, 10% probability of exceedance regional demand traces are used to calculate the surplus reserve concurrently available to each region. This is then used to assess whether a Low Reserve Condition (LRC) may arise[2].
2. `OUTAGE_LRC`. Whilst `RELIABILITY_LRC` models full interconnector availability, `OUTAGE_LRC` is believed to model interconnector outages.
3. `LOR`. This almost certainly corresponds to the Maximum Surplus Capacity (MSC) run. In the MSC run, 50% probability of exceedance regional demand traces are used to calculate the maximum spare capacity available in each region[Page 27, 2]. Lack of Reserve conditions (LOR) are then determined based on the regional spare capacity available relative to 3 thresholds (LOR1 being the least severe and LOR3 being the most), each of which is set by one or more generation contingencies or the Forecast Uncertainty Measure[3].

Along with *pre-dispatch* processes, PASA processes are used to identify LOR conditions. In the event of projected supply scarcity (i.e. forecasted LOR2 or LOR3), AEMO will estimate a latest time to intervene. If AEMO deems the market response to be insufficient by this time, it can exercise the Reliability and Emergency Reserve Trader (RERT), issue directions or issue instructions (i.e. instruct network service providers to commence load shedding)[4].

- Using participant-submitted resource availabilities, forecasted network constraints and resource short-run marginal costs (SRMC), `MTPASA` outputs are reported for each day following aggregation of the results of a market simulation run at half-hourly resolution. These outputs consist of system reliability forecasts (i.e. reporting unserved energy from the Reliability Run and loss of load probability from the Loss of Load Probability Run) that extend out for the next 24 months:

  1. In the Reliability Run, forecast uncertainty is addressed by using a range of reference weather years (at least 8). For each of these reference weather years, AEMO uses at least two percentiles (i.e. probabilities of exceedence) of demand traces that are based on historical data and adjusted for future trends (e.g. accounting for growth in energy consumption or rooftop solar photovoltaics), as well as historically observed generation traces for wind and solar. Then, each of these trace combinations are run using at least 100 random forced outage patterns[5].
  2. In the Loss of Load Probability Run, traces for "abstract" days are constructed based on monthly high demand and low variable renewable energy generation conditions observed over the different reference weather years. The Loss of Load Probability Run is used to determine the days that have a higher risk of load shedding[5].

If the expected annual unserved energy, averaged across simulations in the Reliability Run, exceeds the maximum level specified by the reliability standard, a Low Reserve Condition (LRC) is identified. In response to LRCs, AEMO can direct generators to reschedule outages or contract for longer notice RERT[4].

[1] Australian Energy Market Commission. Updating Short Term PASA, Rule determination. Technical report, May 2022.
[2] Australian Energy Market Operator. Short Term PASA Process Description. Technical report, March 2012.
[3] Australian Energy Market Operator, 2018. Reserve Level Declaration Guidelines.
[4] Australian Energy Market Operator, 2021. Short Term Reserve Management
[5] Australian Energy Market Operator, 2021. Medium Term PASA Process Description.

PD
5MPD
PREDISPATCH
P5MIN
5-minute pre-dispatch
pre-dispatch

Pre-dispatch processes consists of (30-minute) pre-dispatch (PREDISPATCH) and 5-minute pre-dispatch (5MPD or P5MIN). To add to any confusion, when people or documents refer to "pre-dispatch", they are often referring to PREDISPATCH. The use of submitted participant offers distinguishes pre-dispatch processes from PASA processes. These are used alongside forecasts for constraints, demand and variable renewable energy generation to forecast dispatch conditions and regional prices for energy and FCAS. Along with *PDPASA* and *STPASA*, pre-dispatch processes are used to identify Lack of Reserve (LOR) conditions. If AEMO deems the market response to be insufficient by this time, it can exercise the Reliability and Emergency Reserve Trader (RERT), issue directions or issue instructions (i.e. instruct network service providers to commence load shedding)[4].

- PREDISPATCH forecasts are generated every half hour at half-hourly resolution until the end of the last *trading day* for which bid band price submission has closed (this occurs at 1230 EST)[6].
- P5MIN is run for every dispatch interval for the next hour.
- For differences between P5MIN, PREDISPATCH and actual dispatch, refer to pages 32 and 33 of this reference[10]. Notable differences are:
  - P5MIN and PREDISPATCH explore the impact of demand forecast error on regional energy prices and interconnector flows through a sensitivity analysis[7]. Only sensitivites for PREDISPATCH are available via the *MMSDM Historical Data SQLLOader*.
  - The Fast-start Inflexibility Profiles (FSIP) are accommodated in P5MIN but not PREDISPATCH[8].
  - Unit Daily Energy constraints are used in P5MIN and PREDISPATCH, presumably in similar manner to their use in the PASA processes[9].
  - The Economic Participation Factor (EPF) and Intervention Pricing calculations are not performed in either P5MIN or PREDISPATCH[9].
  - Unit dispatch targets from these pre-dispatch processes are not downloaded to Automatic Generation Control (AGC)[9].

market day
trading day

From 0400 (exclusive) to 0400 (inclusive) on the next day (i.e. `(0400 Day 1, 0400 Day 2]`).

MMSDM Historical Data SQLLOader
SQLLoader

An archive of historical market data used by `nemseer` for historical forecast data queries. Data is organised by year and month. The month corresponds to the month in which the forecast was run (i.e. if the month lies between *run_start* and *run_end*).

- Within each month, there exists a directory for most of the data queried by `nemseer` (DATA), including pre-dispatch data with the most recent forecast run, and directories for *"complete"* pre-dispatch and 5-minute pre-dispatch data (PREDISP_ALL_DATA and P5MIN_ALL_DATA, respectively).
  - For pre-dispatch, the complete directory contains data with all forecast runs pertaining to a particular time.

[6] Australian Energy Market Operator, 2021. Pre-dispatch operating procedure.
[10] Katona, Krisztina and Sklibosios Nikitopoulos, Christina and Schloegl, Erik, A Price Mechanism Survey of the Australian National Electricity Market (April 25, 2023). Available at SSRN: https://ssrn.com/abstract=4428450 or http://dx.doi.org/10.2139/ssrn.4428450
[7] Australian Energy Market Operator, 2021. Pre-Dispatch Sensitivities.
[8] Australian Energy Market Operator, 2021. Pre-Dispatch.
[9] Australian Energy Market Operator, 2010. Pre-Dispatch Process Description.

- As the data in the complete P5MIN directory appears to be the same as that in DATA, nemseer does not use this directory.

raw_cache
Directory to which nemseer downloads cleaned raw data. Cleaning by nemseer includes:

- Removing file metadata from the start and end of the file
- Parsing datetimes, including parsing PREDISPATCHSEQNO (in :term:PREDISPATCH tables) into a new datetime column
- Caching raw data in a parquet format, which enables column-based queries and uses less disk space than CSV An invalid/corrupted files list (.invalid_aemo_files.txt) is also maintained in this directory if an invalid/corrupted zip is queried via nemseer. This prevents nemseer from downloading/compiling invalid/corrupted data from AEMO's database.

processed_cache
Directory to which nemseer queries can be saved to (if provided to the relevant functions). The data for each table in a query (which is described by a set of *run times* and *forecasted times*) is either saved to a parquet file if the user specifies they want pandas DataFrame structures, or a netCDF file if the user specifies they want xarray Dataset structures. Each of these files is also saved with query metadata, which nemseer will check during subsequent queries. If the query metadata of any files in the processed cache corresponds to that of the current query, data will be loaded from that file in the processed cache. Note that as the metadata is saved *within* the file, files in the processed cache can be renamed without affecting this functionality (so long as file extensions are preserved).

9.3 Core functionality

9.3.1 Functions

nemseer.**compile_data**(*run_start: str*, *run_end: str*, *forecasted_start: str*, *forecasted_end: str*, *forecast_type: str*, *tables: Union[str, List[str]]*, *raw_cache: str*, *processed_cache: Optional[str] = None*, *data_format: str = 'df'*) → Optional[Union[Dict[str, DataFrame], Dict[str, Dataset]]]

Compiles queried data from raw_cache and/or processed_cache.

For each queried table, this function:

1. If required, downloads raw forecast data for the table and converts to the requested data structure.
2. Otherwise, compiles table data from either of or both of the caches.
3. Applies user-requested filtering to *run times* and *forecasted times* to any raw data.

If data_format = "df" (default), a pandas.DataFrame is returned. Otherwise, if data_format = "xr", a xarray.Dataset is returned.

Examples

See *compiling data examples*.

Parameters

- **run_start** (*str*) – Forecast runs at or after this datetime are queried.
- **run_end** (*str*) – Forecast runs before or at this datetime are queried.
- **forecasted_start** (*str*) – Forecasts pertaining to times at or after this datetime are retained.
- **forecasted_end** (*str*) – Forecasts pertaining to times before or at this datetime are retained.
- **forecast_type** (*str*) – One of *nemseer.forecast_types*
- **tables** (*Union[str, List[str]]*) – Table or tables required. A single table can be supplied as a string. Multiple tables can be supplied as a list of strings.
- **raw_cache** (*str*) – Path to create or reuse as *raw_cache*. Files are downloaded to this directory and cached data is maintained in the parquet format.
- **processed_cache** (*optional*) – Path to build or reuse *processed_cache*. Should be distinct from `raw_cache`
- **data_format** (*str*) – Default is 'df', which returns `pandas DataFrame`. Can also request 'xr', which returns `xarray.Dataset`.

Return type

Optional[*Union*[*Dict*[str, *DataFrame*], *Dict*[str, *Dataset*]]]

nemseer.**download_raw_data**(*forecast_type: str*, *tables: Union[str, List[str]]*, *raw_cache: str*, *run_start: Optional[str] = None*, *run_end: Optional[str] = None*, *forecasted_start: Optional[str] = None*, *forecasted_end: Optional[str] = None*, *keep_csv: bool = False*) → None

Downloads raw forecast data from NEMWeb MMSDM Historical Data SQLLoader

Downloads raw forecast data. Accepts a datetime pair, which can be either of:

1. `run_start` and `run_end`
2. `forecasted_start` and `forecasted_end`

Examples

See *downloading raw data examples*.

Parameters

- **forecast_type** (*str*) – One of *nemseer.forecast_types*
- **tables** (*Union[str, List[str]]*) – Table or tables required. A single table can be supplied as a string. Multiple tables can be supplied as a list of strings.
- **raw_cache** (*str*) – Path to create or reuse as *raw_cache*. Files are downloaded to this directory and cached data is maintained in the parquet format.
- **run_start** (*Optional[str]*) – Forecast runs at or after this datetime are queried. If supplied, must be included with `run_end`.

- **run_end** (*Optional[str]*) – Forecast runs before or at this datetime are queried. If supplied, must be included with `run_start`.
- **forecasted_start** (*Optional[str]*) – Forecasts pertaining to times at or after this datetime are retained. If supplied, must be included with `forecasted_end`.
- **forecasted_end** (*Optional[str]*) – Forecasts pertaining to times before or at this datetime are retained. If supplied, must be included with `forecasted_start`.
- **keep_csv** (*bool*) – Default False. If True, downloaded csvs are retained in the *raw_cache*.

Raises
`ValueError` – If a valid pair of datetimes is not supplied, or if more than a valid pair of datetimes is supplied.

Return type
None

nemseer.**generate_runtimes**(*forecasted_start: str*, *forecasted_end: str*, *forecast_type: str*) → Tuple[str, str]

For a particular *forecast type*, generates the earliest *run_start* and the latest *run_end* that can be queried for the supplied *forecasted times*.

In other words, this function will return all *run times* for forecasts that cover the supplied *forecasted times*. As such, using the *run_start* and *run_end* returned by this function with *nemseer.compile_data()* will ensure most, if not all of the data for the seleected *forecasted times* is returned.

N.B. These have been determined based on AEMO documentation and actual data. This may not be accurate for all *forecast types*, e.g. *MTPASA* which is not run at a set time.

Examples

See *getting valid run times for a set of forecasted time*.

Parameters

- **forecasted_start** (*str*) – Forecasts pertaining to times at or after this datetime are retained.
- **forecasted_end** (*str*) – Forecasts pertaining to times before or at this datetime are retained.
- **forecast_type** (*str*) – One of *nemseer.forecast_types*

Returns
Tuple of *nemseer*-valid string datetimes that correspond to valid *run* times

Raises
`ValueError` – If supplied *forecasted* times are invalid.

Return type
Tuple[str, str]

nemseer.**get_data_daterange**() → Dict[int, List[int]]

Years and months with data on NEMWeb MMSDM Historical Data SQLLoader .. rubric:: Examples

See *querying date ranges*

Returns
Months mapped to each year. Data is available for each of these months.

Return type
Dict[int, *List*[int]]

nemseer.**get_tables**(*year: int*, *month: int*, *forecast_type: str*, *actual: bool = False*) → List[str]

Requestable tables of particular forecast type on MMSDM Historical Data SQLLoader

If `actual` = False, provides a list of tables that can be requested via *nemseer*.

If `actual` = True, returns actual tables available via NEMWeb, including all tables that are enumerated.

N.B.:

- Removes numbering from enumerated tables for *P5MIN* - e.g. *CONSTRAINTSOLUTION(x)* are all reduced to *CONSTRAINTSOLUTION*

Examples

See *querying table availability*

Parameters

- **year** (*int*) – Year
- **month** (*int*) – Month
- **forecast_type** (*str*) – One of *nemseer.forecast_types*
- **actual** (*bool*) –

Returns
List of tables associated with that forecast type for that period

Return type
List[str]

9.3.2 Data

Forecast types

Forecast types requestable through `nemseer`. See also *forecast types*, and *pre-dispatch* and *PASA*.

nemseer.**forecast_types = ('P5MIN', 'PREDISPATCH', 'PDPASA', 'STPASA', 'MTPASA')**

Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable's items.

If the argument is a tuple, the return value is the same object.

9.4 Examples

Below are some examples that explore what you can do with `NEMSEER`.

We recommend that you look through these in the order below, as the later examples reuse concepts/code from earlier ones.

9.4.1 Visualising demand forecast convergence using `nemseer` and `xarray`

In this example, we look at a couple of ways we can plot demand forecast convergence using a set of 5-minute pre-dispatch (*5MPD* or `P5MIN`) forecasts for the evening of 14/07/2022 for NSW. In particular, we'll look at the ways to plot using `xarray` data structures.

Key imports

NEM data tools:

- `NEMOSIS` for actual market data
  - Data obtained from `NEMOSIS` is contained within `pandas` DataFrames
- `NEMSEER` for historical forecast data
  - In this tutorial, we focus on what we can do with data obtained from `NEMSEER` in `xarray` data structures
  - The `xarray` tutorial is a great resource for learning how to use `xarray` for data handling and plotting

Plotting

- `matplotlib` for static plotting. Both `xarray` and `pandas` have implemented plotting methods using this library.
- `plotly` for interactive plots. In some cases, we will use `plotly` directly, but in others, we will use `hvplot`.
- `hvplot`, a high-level API for plotting. `hvplot` can use many backends (default is `bokeh`, but we will use `plotly`) and makes it easy to plot `xarray` data structures using the `.hvplot` accessor

```
from pathlib import Path

import nemosis
from nemseer import compile_data, generate_runtimes


import plotly.graph_objects as go

import matplotlib.pyplot as plt

import hvplot.xarray
hvplot.extension("plotly")

# for some advanced colour manipulation
from matplotlib.colors import to_hex
import numpy as np

# plotly rendering
import plotly.io as pio
import plotly.express as px
```

```
INFO: generated new fontManager
```

Study times

Here we'll define our datetime range that we're interested in:

- NEMOSIS only accepts datetime strings with seconds specified.
- NEMSEER will accept datetimes with seconds specified, so long as the seconds are 00. This is because the datetimes relevant to NEMSEER functionality are only specified to the nearest minute.

```
(start, end) = ("2022/07/14 16:55:00", "2022/07/14 19:00:00")
```

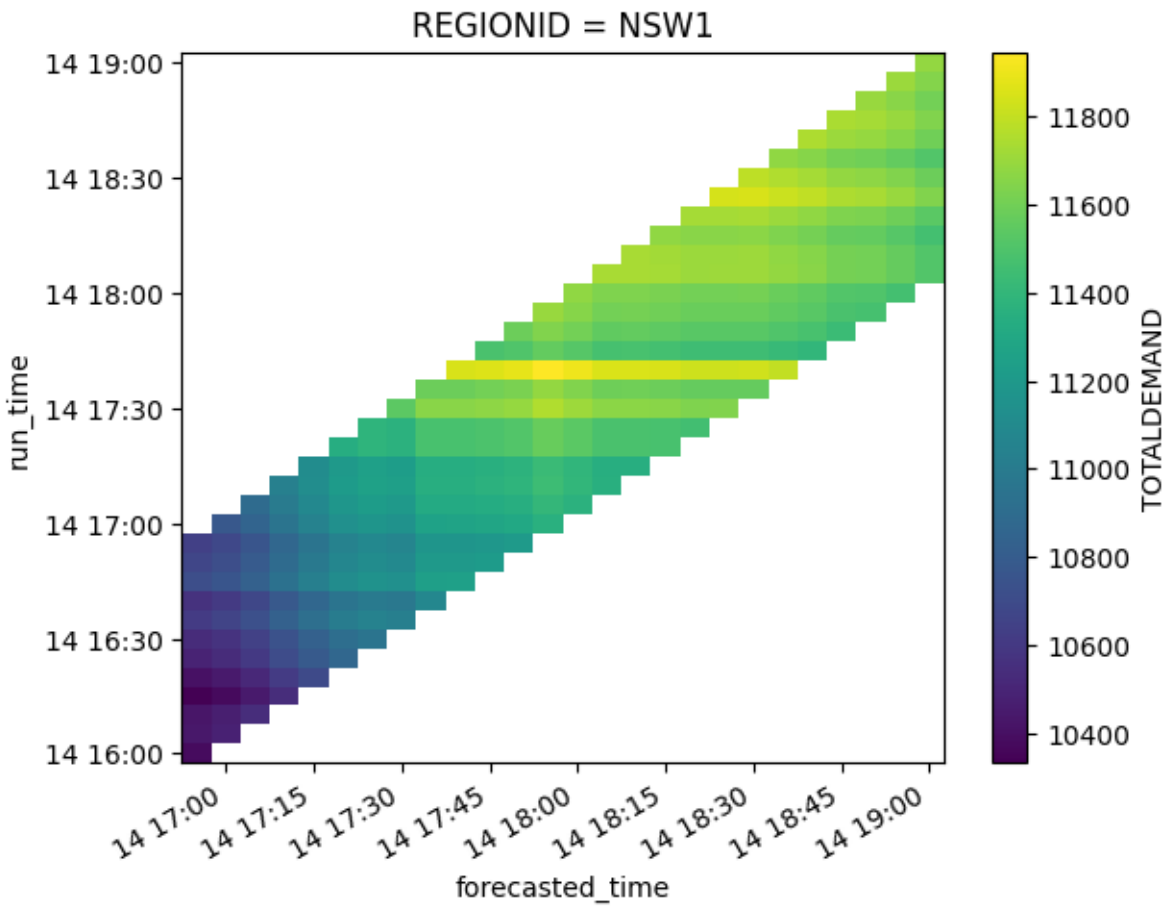
Get demand forecast data for NSW

```
p5_run_start, p5_run_end = generate_runtimes(start, end,
                                             "P5MIN")
p5_data = compile_data(p5_run_start, p5_run_end, start, end,
                       "P5MIN", "REGIONSOLUTION", raw_cache="nemseer_cache/",
                       data_format="xr")
p5_region = p5_data["REGIONSOLUTION"]
# .sel() is a lot like .loc() for pandas
# We then use ["TOTALDEMAND"] to acces that specific variable
p5_demand_forecasts = p5_region.sel(
    forecasted_time=slice(start, end),
    REGIONID="NSW1"
)["TOTALDEMAND"]
```

Basic plotting with xarray

We can call .plot() on an xarray data structure to create a plot. Because our data is 3D (run_time, forecasted_time, TOTALDEMAND), xarray creates a heatmap.

```
p5_demand_forecasts.plot();
```



We can also create an interactive version using `hvplot`:

```
hvhmap = p5_demand_forecasts.hvplot.heatmap(
    x="forecasted_time", y="run_time", C="TOTALDEMAND"
)
# you can view this chart by calling the chart variable
```

Integrating actual demand into our plots

To compare forecasts with actual demand data, we will use `NEMOSIS` to obtain actual demand data for NSW for this evening.

```
# create a folder for NEMOSIS data
nemosis_cache = Path("nemosis_cache/")
if not nemosis_cache.exists():
    nemosis_cache.mkdir()

# get demand data for NSW
nsw_demand = nemosis.dynamic_data_compiler(
  start, end, "DISPATCHREGIONSUM", nemosis_cache,
  filter_cols=["REGIONID", "INTERVENTION"],
  filter_values=(["NSW1"],[0])
```

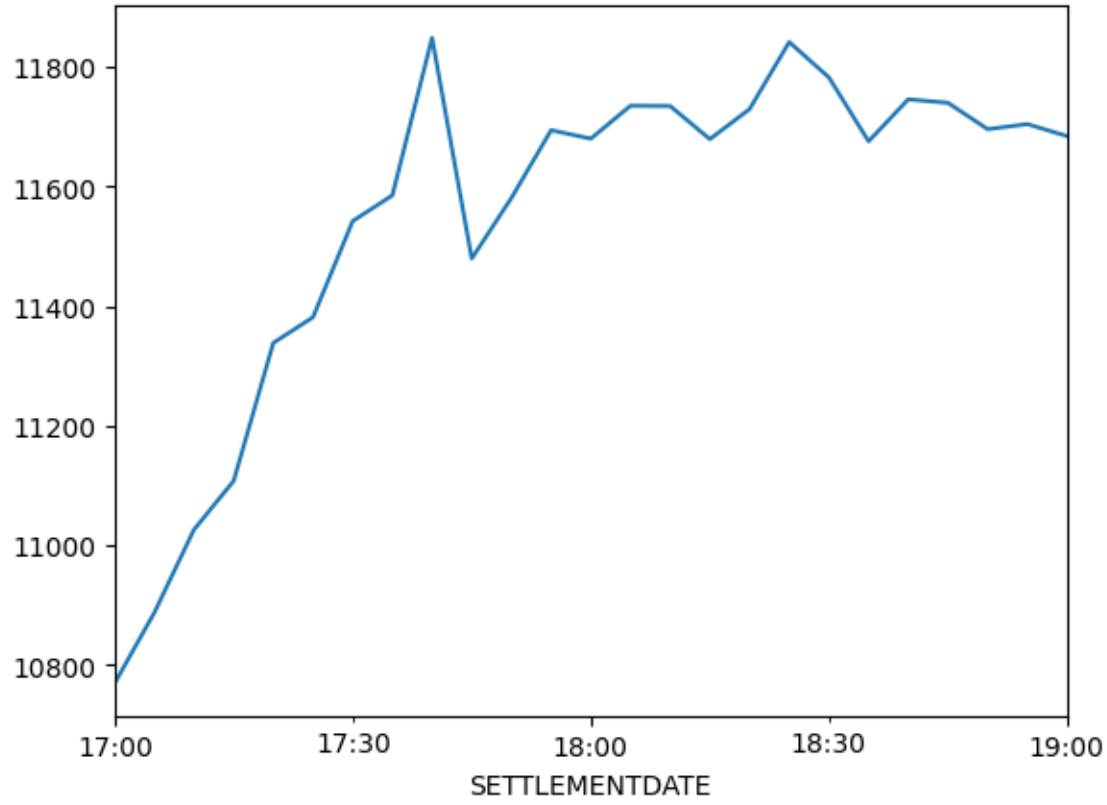
(continues on next page)

(continued from previous page)

```
)
nsw_demand = nsw_demand.set_index('SETTLEMENTDATE').sort_index()
```

pandas has plotting functionality that wraps matplotlib:

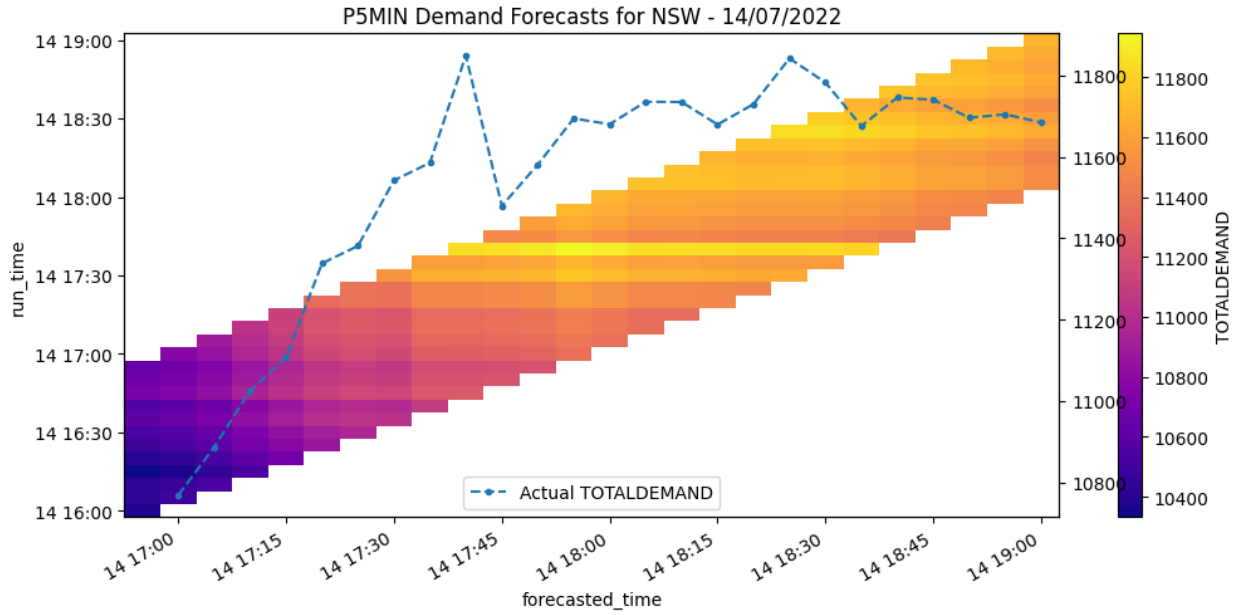
```
nsw_demand["TOTALDEMAND"].plot();
```



We'll now tie the actual data in with our forecasted data. The actual data will be a line chart, and the forecast data will be a heatmap. We'll first do this using matplotlib:

1. Create a matplotlib axis
2. Plot out heatmap onto this axis
3. Then create a secondary y-axis (via ax.twinx()) and plot our actual demand

```
fig, ax = plt.subplots(1, 1, figsize=(12, 5))
p5_demand_forecasts.plot(cmap="plasma", ax=ax)
ax_demand = ax.twinx()
ax_demand.plot(nsw_demand.index, nsw_demand["TOTALDEMAND"], ls="--", marker=".",
               label="Actual TOTALDEMAND")
ax_demand.legend(loc="lower center")
ax.set_title("P5MIN Demand Forecasts for NSW - 14/07/2022");
```



Faceted Plotting

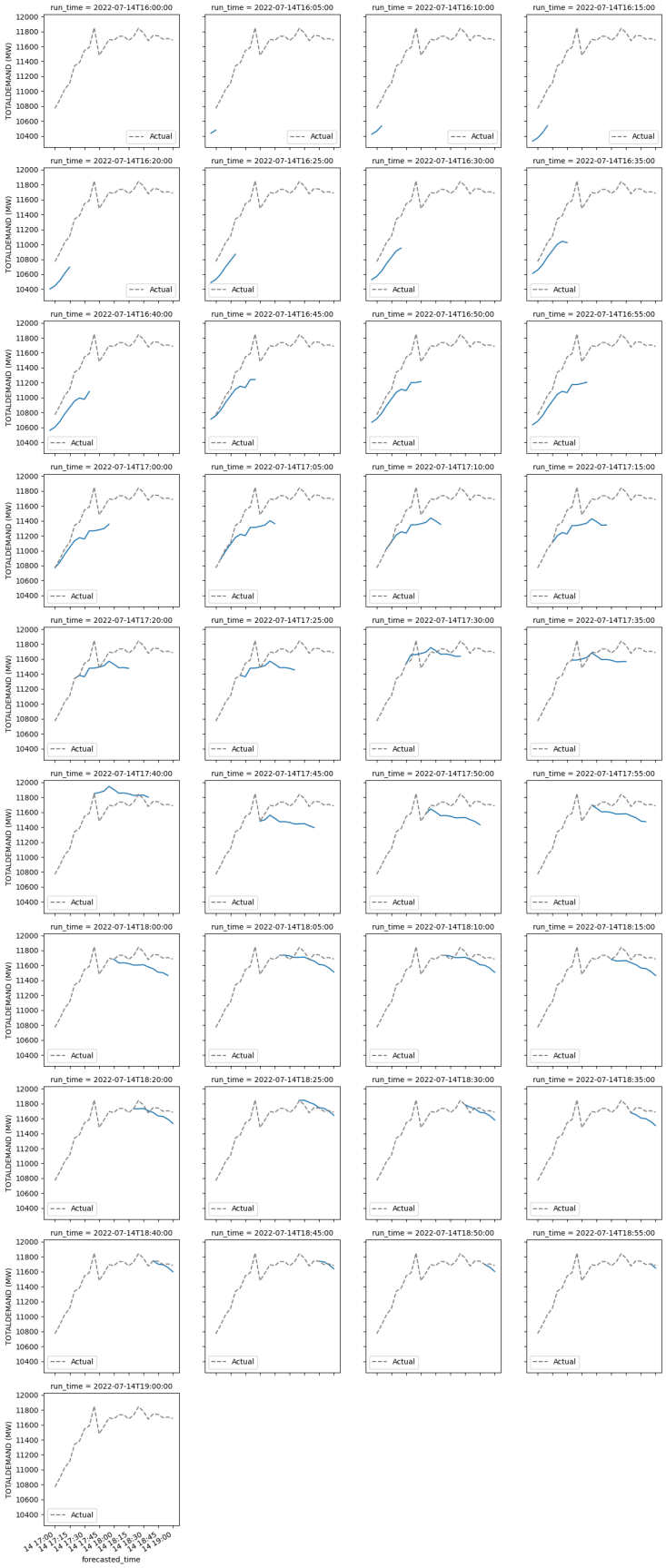
The forecast that was run at the same time as a demand spike seems to forecast relatively high demand when compared to adjacent forecast runs. Let's see if we can make that a bit clearer.

We'll create a set of faceted plots that separate different runtimes to look at this closely.

```
# This creates an xarray FacetGrid
fg = p5_demand_forecasts.plot(hue="run_time", col="run_time", col_wrap=4)
# We then iterate through the matplotlib axes to add the actual data
for ax in fg.axes.flat:
    ax.plot(
        nsw_demand.index, nsw_demand["TOTALDEMAND"], label="Actual", ls="--", color="gray
↪"
    )
    ax.legend()
fg.set_ylabels("TOTALDEMAND (MW)");
```

```
/tmp/ipykernel_791/95342703.py:4: DeprecationWarning:

self.axes is deprecated since 2022.11 in order to align with matplotlibs plt.subplots,␣
↪use self.axs instead.
```


run_time = 2022-07-14T16:00:00
run_time = 2022-07-14T16:05:00
run_time = 2022-07-14T16:10:00
run_time = 2022-07-14T16:15:00
run_time = 2022-07-14T16:20:00
run_time = 2022-07-14T16:25:00
run_time = 2022-07-14T16:30:00
run_time = 2022-07-14T16:35:00
run_time = 2022-07-14T16:40:00
run_time = 2022-07-14T16:45:00
run_time = 2022-07-14T16:50:00
run_time = 2022-07-14T16:55:00
run_time = 2022-07-14T17:00:00
run_time = 2022-07-14T17:05:00
run_time = 2022-07-14T17:10:00
run_time = 2022-07-14T17:15:00
run_time = 2022-07-14T17:20:00
run_time = 2022-07-14T17:25:00
run_time = 2022-07-14T17:30:00
run_time = 2022-07-14T17:35:00
run_time = 2022-07-14T17:40:00
run_time = 2022-07-14T17:45:00
run_time = 2022-07-14T17:50:00
run_time = 2022-07-14T17:55:00
run_time = 2022-07-14T18:00:00
run_time = 2022-07-14T18:05:00
run_time = 2022-07-14T18:10:00
run_time = 2022-07-14T18:15:00
run_time = 2022-07-14T18:20:00
run_time = 2022-07-14T18:25:00
run_time = 2022-07-14T18:30:00
run_time = 2022-07-14T18:35:00
run_time = 2022-07-14T18:40:00
run_time = 2022-07-14T18:45:00
run_time = 2022-07-14T18:50:00
run_time = 2022-07-14T18:55:00
run_time = 2022-07-14T19:00:00
TOTALDEMAND (MW)
Actual
forecasted_time

From this, it's a little clearer that demand forecasting for *5MPD* is heavily influenced by current demand. In actual fact, demand forecasts for the last 11 dispatch intervals in a *5MPD* forecast are based on a recursive application of an average percentage demand change, which is specific to a given dispatch interval and whether the day is a weekday or weekend. This average percentage demand change is calculated using demand data form the last two weeks. For more information, see this AEMO document on 5MPD demand forecasting and this AEMO document on demand terms in the EMMS data model.

Interactive Plots

Now we'll try and recreate some of the plots above using `plotly`.

`hvplot` helps us generate the chart we need from the `xarray` data structure. After that, we need to obtain a `plotly` object to work with to add additional traces.

```
# Generate interactive heatmap
hmap = p5_demand_forecasts.hvplot.heatmap(
    x="forecasted_time", y="run_time", C="TOTALDEMAND", cmap="plasma",
    title="P5MIN Demand Forecasts for NSW - 14/07/2022"
)
# Create plotly.go.Figure from hvplot data structure
fig = go.Figure(hvplot.render(hmap, backend="plotly"))
# add actual demand as a line trace
line = go.Scatter(
    x=nsw_demand.index, y=nsw_demand["TOTALDEMAND"], yaxis="y2",
    line={"color": "black", "dash": "dash"}, name="Actual TOTALDEMAND",
    )
fig.add_trace(line)
# update_layout defines the second y-axis and figure width and height
fig = fig.update_layout(
    xaxis=dict(domain=[0.1, 0.9]),
    yaxis2=dict(overlaying="y", title="Actual TOTALDEMAND (MW)", side="right"),
    height=300, width=700
)
# you can view this chart by calling the chart variable
# below, we load a pre-generated chart
```

`hvplot` has easy ways to integrate interactivity. We can trigger this by leaving one dimension as a degree of freedom (e.g. specifying x-axis as `forecasted_time`, y-axis as `TOTALDEMAND` thus leaving `run_time` as a degree of freedom).

We can also get `hvplot` and `plotly` to plot across the degree(s) of freedom simultaneously using `by=`:

```
run_time_iterations = p5_demand_forecasts.hvplot(by="run_time")
run_lines = go.Figure(hvplot.render(run_time_iterations, backend="plotly"))
# you can view this chart by calling the chart variable
# below, we load a pre-generated chart
```

This is quite hard to read. We can clean this up and use a sequential colour scheme to indicate forecast outputs from later run times:

```
# obtain data from hvplot
plotly_data = hvplot.render(run_time_iterations, backend="plotly")
# modify the colour of each trace using the Reds sequential colormap
for i, increment in enumerate(np.linspace(0, 1, len(plotly_data["data"]))):
    plotly_data["data"][i]["line"]["color"] = to_hex(plt.cm.Reds(increment))
```

(continues on next page)

(continued from previous page)

```
# create a plotly.go.Figure
overlay = go.Figure(plotly_data)
# add actual demand data
line = go.Scatter(
    x=nsw_demand.index, y=nsw_demand["TOTALDEMAND"], yaxis="y2",
    line={"color": "black", "dash": "dash"}, name="Actual TOTALDEMAND",
    )
overlay.add_trace(line)
# update the layout to specify the secondary y-axis
overlay = overlay.update_layout(
    xaxis=dict(domain=[0.1, 0.9]),
    yaxis2=dict(overlaying="y", title="Actual TOTALDEMAND (MW)", side="right"),
    height=300, width=700,
    title = "P5MIN Demand Forecasts for NSW - 14/07/2022"
)
# you can view this chart by calling the chart variable
# below, we load a pre-generated chart
```

9.4.2 Looking at 5-minute pre-dispatch demand forecast errors in 2021

In this example, we will take a look at 5-minute pre-dispatch (*5MPD*) demand forecast "error" (the difference between actual and forecasted demand) for 2021. AEMO runs *5MPD* to provide system and market information for the next hour.

We'll look at forecast "error" on a NEM-wide basis; that is, we will sum actual scheduled demand across all NEM regions and then compare that to the sum of forecast scheduled demand across all NEM regions.

The code below could be modified to do this analysis on a region by region basis (we do this with (30-minute) pre-dispatch demand forecasts in this example).

Key imports

```
# standard libraries
from datetime import datetime, timedelta
from pathlib import Path

# NEM data libraries
# NEMOSIS for actual demand data
# NEMSEER for forecast demand data
import nemosis
from nemseer import compile_data, download_raw_data, generate_runtimes

# data wrangling libraries
import numpy as np
import pandas as pd

# interactive plotting
import plotly.express as px
import plotly.io as pio
import plotly.graph_objects as go
```

(continues on next page)

(continued from previous page)

```
# progress bar for error computation
from tqdm.autonotebook import tqdm

# supress logging from NEMSEER and NEMOSIS
import logging

logging.getLogger("nemosis").setLevel(logging.WARNING)
logging.getLogger("nemseer").setLevel(logging.ERROR)
```

Plot styling

```
nemseer_template = dict(
    layout=go.Layout(
        font_family="Source Sans 3",
        title_font_size=24,
        title_x=0.05,
        plot_bgcolor="#f0f0f0",
        colorway=px.colors.qualitative.Bold,
    )
)
```

Defining our analysis start and end dates

```
analysis_start = "2021/01/01 00:05:00"
analysis_end = "2022/01/01 00:00:00"
```

Obtaining actual demand data from NEMOSIS

We will download DISPATCHREGIONSUM to access the TOTALDEMAND field (actual scheduled demand).

We'll first download the data we need and cache it so that it's ready for computation.

```
nemosis_cache = Path("nemosis_cache/")
if not nemosis_cache.exists():
    nemosis_cache.mkdir()
```

```
nemosis.cache_compiler(
    analysis_start, analysis_end, "DISPATCHREGIONSUM", nemosis_cache, fformat="parquet"
)
```

Obtaining forecast demand data from NEMSEER

We will download REGIONSOLUTION to access the TOTALDEMAND field in P5MIN forecasts.

We'll first download the data we need and cache it so that it's ready for computation.

```
download_raw_data(
    "P5MIN",
    "REGIONSOLUTION",
    "nemseer_cache/",
    forecasted_start=analysis_start,
    forecasted_end=analysis_end,
)
```

Calculating forecast error

Below we calculate demand forecast error for P5MIN forecasts using forecast demand data and actual demand data.

Attention: The *actual run time* of 5MPD is approximately 5 minutes before the nominal *run time*. We will adjust for this in this when calculating forecast ahead times. See the note in *this section*.

As data for the entire period is loaded into memory, adapt the length of the period you select to your machine specifications (e.g. a year's worth of forecast data consumed ~15GB on the test machine).

Forecast error calculation functions

The code below uses functionalities offered by NEMOSIS, NEMSEER and pandas to calculate demand forecast error.

```
def calculate_p5min_demand_forecast_error_vectorised(
    analysis_start: str, analysis_end: str
) -> pd.DataFrame:
    """
    Calculates P5MIN demand forecast error (Actual - Forecast) for all forecasts
    that are run for a given forecasted_time in a vectorised fashion.

    Args:
        forecasted_time: Datetime string in the form YYYY/mm/dd HH:MM:SS
    Returns:
        pandas DataFrame with forecast error in `TOTALDEMAND` columns, the ahead time
        of the forecast run in `ahead_time`, and the forecasted time in
        `forecasted_time`.
    """

    def get_forecast_data(analysis_start: str, analysis_end: str) -> pd.DataFrame:
        """
        Use NEMSEER to get 5MPD forecast data. Also omits any intervention periods.
        """
        # use NEMSEER functions to compile pre-cached data
        forecasts_run_start, forecasts_run_end = generate_runtimes(
            analysis_start, analysis_end, "P5MIN"
```

(continues on next page)

(continued from previous page)

```
        )
        forecast_df = compile_data(
            forecasts_run_start,
            forecasts_run_end,
            analysis_start,
            analysis_end,
            "P5MIN",
            "REGIONSOLUTION",
            "nemseer_cache/",
        )["REGIONSOLUTION"]
        # remove intervention periods
        forecast_df = forecast_df.query("INTERVENTION == 0")
        return forecast_df

    def get_actual_data(analysis_start: str, analysis_end: str) -> pd.DataFrame:
        """
        Use NEMOSIS to get actual data. Also omits any intervention periods
        """
        # NEMOSIS start time must precede end of interval of interest by 5 minutes
        nemosis_start = (
            datetime.strptime(analysis_start, "%Y/%m/%d %H:%M:%S")
            - timedelta(minutes=5)
        ).strftime("%Y/%m/%d %H:%M:%S")
        # use NEMOSIS to compile pre-cached data and filter out interventions
        actual_df = nemosis.dynamic_data_compiler(
            nemosis_start,
            analysis_end,
            "DISPATCHREGIONSUM",
            nemosis_cache,
            filter_cols=["INTERVENTION"],
            filter_values=([0],),
            fformat="parquet",
        )
        return actual_df

    def calculate_p5min_forecast_demand_error(
        actual_demand: pd.DataFrame, forecast_demand: pd.DataFrame
    ) -> pd.DataFrame:
        """
        Calculate P5MIN forecast demand error given actual and forecast demand

        Ahead time calculation reflects the fact that P5MIN actual run time is
        5 minutes before the nominal run time.
        """
        # left merge ensures all forecasted values have the corresponding actual value␣
↪merged in
        merged = pd.merge(
            forecast_demand, actual_demand, on="forecasted_time", how="left"
        )
        if len(merged) > len(forecast_demand):
            raise ValueError(
                "Merge should return DataFrame with dimensions of forecast data"
```

(continues on next page)

(continued from previous page)

```
            )
        # subtract 5 minutes from run time to get actual run time
        merged["ahead_time"] = merged["forecasted_time"] - (
            merged["RUN_DATETIME"] - timedelta(minutes=5)
        )
        forecast_error = (
            merged["TOTALDEMAND"] - merged["FORECAST_TOTALDEMAND"]
        ).rename("TOTALDEMAND")
        # create the forecast error DataFrame
        forecast_error = pd.concat(
            [forecast_error, merged["ahead_time"]], axis=1
        ).set_index(merged["forecasted_time"])
        return forecast_error

    # get forecast data
    forecast_df = get_forecast_data(analysis_start, analysis_end)
    # rename columns in preparation for merge
    forecast_df = forecast_df.rename(
        columns={
            "TOTALDEMAND": "FORECAST_TOTALDEMAND",
            "INTERVAL_DATETIME": "forecasted_time",
        }
    )
    # group by forecasted and run times, then sum demand across regions to get NEM-wide␣
↪demand
    forecast_demand = forecast_df.groupby(["forecasted_time", "RUN_DATETIME"])[
        "FORECAST_TOTALDEMAND"
    ].sum()
    forecast_demand = forecast_demand.reset_index()

    # get actual data
    actual_df = get_actual_data(analysis_start, analysis_end)
    # rename columns in preparation for merge
    actual_df = actual_df.rename(
        columns={
            "SETTLEMENTDATE": "forecasted_time",
            "TOTALDEMAND": "TOTALDEMAND",
        }
    )
    # group by forecasted time and then sum demand across regions to get NEM-wide demand
    actual_demand = (
        actual_df.groupby("forecasted_time")["TOTALDEMAND"].sum().reset_index()
    )

    # calculate forecast error
    forecast_error = calculate_p5min_forecast_demand_error(
        actual_demand, forecast_demand
    )
    return forecast_error
```

```
forecast_error = calculate_p5min_demand_forecast_error_vectorised(
    analysis_start, analysis_end
```

(continues on next page)

(continued from previous page)

```
)
```

Plotting forecast error percentiles for each ahead time

How does forecast error change based on how many minutes they are ahead of the time they are forecasting for?

Forecast error percentiles

We can compute forecast error percentiles across `ahead_times` (between 0 and 55 minutes for 5-minute pre-dispatch).

To do this, we will group the error DataFrame by `ahead_time`, compute the percentile and then add a column that indicates the computed percentile. We'll repeat this process across all percentiles of interest and then concatenate the results to form a single DataFrame for plotting.

```
percentile_data = []
for quantile in (0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95, 0.99):
    quantile_result = forecast_error.groupby(
        forecast_error["ahead_time"].dt.seconds / 60
    )["TOTALDEMAND"].quantile(quantile)
    percentile_result = pd.concat(
        [
            quantile_result,
            pd.Series(
                np.repeat(quantile * 100, len(quantile_result)),
                index=quantile_result.index,
                name="Percentile",
            ).astype(int),
        ],
        axis=1,
    )
    percentile_data.append(percentile_result)
percentile_df = pd.concat(percentile_data, axis=0).reset_index()
```

We can plot these quantiles for each ahead time.

It's interesting to note that there is only a slight positive bias in the 50th percentile forecast as the forecast ahead time approaches one hour.

```
ahead_percentile = px.line(
    percentile_df,
    x="ahead_time",
    y="TOTALDEMAND",
    color="Percentile",
    title="Hour-ahead (5MPD) NEM-wide Demand Forecast Error, 2021<br><sup>Error = Actual␣
↪- Forecast,"
    + "</sup>",
    labels={
        "TOTALDEMAND": "Demand Forecast Error (MW)",
        "ahead_time": "Forecast Ahead Time (minutes)",
    },
    template=nemseer_template,
```

(continues on next page)

(continued from previous page)

```
    color_discrete_map={
        1: "#E24A33",
        5: "#348ABD",
        10: "#988ED5",
        25: "#777777",
        50: "#FBC15E",
        75: "#777777",
        90: "#988ED5",
        95: "#348ABD",
        99: "#E24A33",
    },
)
ahead_percentile["layout"]["xaxis"]["autorange"] = "reversed"
```

Plotting the distributions of forecast errors by ahead time

We can look at the full distributions of forecast errors across ahead times.

But first, we'll remove "forecasts" at `ahead_time` = 5, as these correspond to actual dispatch conditions.

We'll also convert the Timedeltas into an integer, which will be helpful for plotting.

```
error_excluding_real_time = forecast_error[
    forecast_error["ahead_time"].dt.seconds > 300
]
error_excluding_real_time.loc[:, "ahead_time"] = (
    error_excluding_real_time.loc[:, "ahead_time"].dt.seconds / 60
).astype(int)
```

```
ahead_hist = px.histogram(
    error_excluding_real_time,
    x="TOTALDEMAND",
    color="ahead_time",
    template=nemseer_template,
)
ahead_hist.update_layout(
    legend_title_text="Ahead Time (mins)",
);
```

Plotting forecast error quantiles against time of day

How does forecast error change across the day?

Below, we repeat percentile calculations, but this time we group the data by the time of day.

From the chart below, we can see that, across the NEM, intra-hour demand forecasting errors tend to be larger during the morning and evening ramps.

```
TOD_percentile_data = []
for quantile in (0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95, 0.99):
    quantile_result = error_excluding_real_time.groupby(
        error_excluding_real_time.index.time
```

(continues on next page)

(continued from previous page)

```
    )["TOTALDEMAND"].quantile(quantile)
    percentile_result = pd.concat(
        [
            quantile_result,
            pd.Series(
                np.repeat(quantile * 100, len(quantile_result)),
                index=quantile_result.index,
                name="Percentile",
            ).astype(int),
        ],
        axis=1,
    )
    TOD_percentile_data.append(percentile_result)
TOD_percentile = pd.concat(TOD_percentile_data, axis=0).reset_index()
```

```
tod_percentile = px.line(
    TOD_percentile,
    x="index",
    y="TOTALDEMAND",
    color="Percentile",
    labels={
        "TOTALDEMAND": "Demand Forecast Error (MW)",
        "ahead_time": "Forecast Ahead Time (minutes)",
        "index": "Time of Day",
    },
    title="Hour-ahead (5MPD) NEM-wide Demand Forecast Error, 2021<br><sup>Error = Actual␣
↪- Forecast,"
    + " excludes forecast run at real time</sup>",
    template=nemseer_template,
    color_discrete_map={
        1: "#E24A33",
        5: "#348ABD",
        10: "#988ED5",
        25: "#777777",
        50: "#FBC15E",
        75: "#777777",
        90: "#988ED5",
        95: "#348ABD",
        99: "#E24A33",
    },
)
```

9.4.3 Looking at pre-dispatch demand forecast errors in 2021

In this example, we will take a look at (30-minute) pre-dispatch (*PREDISPATCH*) demand forecast "error" (the difference between "actual" - the demand used in dispatch - and forecasted demand) for 2021. Unlike 5PMD, pre-dispatch extends out to 39 hours ahead, so it's a good dataset to use to look at day-ahead forecast errors.

Key imports

```
# standard libraries
import logging
from datetime import datetime, timedelta
from pathlib import Path

# NEM data libraries
# NEMOSIS for actual demand data
# NEMSEER for forecast demand data
import nemosis
from nemseer import compile_data, download_raw_data, generate_runtimes

# data wrangling libraries
import numpy as np
import pandas as pd

# interactive plotting
import plotly.express as px
import plotly.io as pio
import plotly.graph_objects as go

# progress bar for error computation
from tqdm.autonotebook import tqdm

# silence NEMSEER and NEMOSIS logging
logging.getLogger("nemseer").setLevel(logging.ERROR)
logging.getLogger("nemosis").setLevel(logging.WARNING)
```

Plot styling

```
nemseer_template = dict(
    layout=go.Layout(
        font_family="Source Sans 3",
        title_font_size=24,
        title_x=0.05,
        plot_bgcolor="#f0f0f0",
        colorway=px.colors.qualitative.Bold,
    )
)
```

Defining our analysis start and end dates

```
analysis_start = "2021/01/01 00:00:00"
analysis_end = "2022/01/01 00:00:00"
```

Obtaining actual demand data from NEMOSIS

We will download DISPATCHREGIONSUM to access the TOTALDEMAND field.

We'll first download the data we need and cache it so that it's ready for computation.

```
nemosis_cache = Path("nemosis_cache/")
if not nemosis_cache.exists():
    nemosis_cache.mkdir()
```

```
nemosis.cache_compiler(
    analysis_start, analysis_end, "DISPATCHREGIONSUM", nemosis_cache, fformat="parquet"
)
```

Obtaining forecast demand data from NEMSEER

We will download REGIONSUM to access the TOTALDEMAND field in PREDISPATCH forecasts.

We'll first download the data we need and cache it so that it's ready for computation.

```
download_raw_data(
    "PREDISPATCH",
    "REGIONSUM",
    "nemseer_cache/",
    forecasted_start=analysis_start,
    forecasted_end=analysis_end,
)
```

Calculating regional forecast errors

Below we calculate demand forecast error for PREDISPATCH forecasts using forecast demand data and actual demand data.

Attention: The *actual run time* of PD is approximately 30 minutes before the nominal *run time*. We will adjust for this in this when calculating forecast ahead times. See the note in *this section*.

As data for the entire period is loaded into memory, adapt the length of the period you select to your machine specifications (e.g. a year's worth of forecast data consumed ~10GB on the test machine).

Forecast error calculation functions

The code below uses functionalities offered by `NEMOSIS`, `NEMSEER` and `pandas` to calculate demand forecast error.

```
def calculate_predispatch_demand_forecast_error_vectorised(
    analysis_start: str, analysis_end: str
) -> pd.DataFrame:
    """
    Calculates PD demand forecast error (Actual - Forecast) for all forecasts
    that are run for a given forecasted_time in a vectorised fashion.

    Args:
        forecasted_time: Datetime string in the form YYYY/mm/dd HH:MM:SS
    Returns:
        pandas DataFrame with forecast error in `TOTALDEMAND` columns, the ahead time
        of the forecast run in `ahead_time`, and the forecasted time in
        `forecasted_time`.
    """

    def get_forecast_data(analysis_start: str, analysis_end: str) -> pd.DataFrame:
        """
        Use NEMSEER to get PD forecast data. Also omits any intervention periods.
        """
        # use NEMSEER functions to compile pre-cached data
        forecasts_run_start, forecasts_run_end = generate_runtimes(
            analysis_start, analysis_end, "PREDISPATCH"
        )
        forecast_df = compile_data(
            forecasts_run_start,
            forecasts_run_end,
            analysis_start,
            analysis_end,
            "PREDISPATCH",
            "REGIONSUM",
            "nemseer_cache/",
        )["REGIONSUM"]
        # remove intervention periods
        forecast_df = forecast_df.query("INTERVENTION == 0")
        return forecast_df

    def get_actual_data(analysis_start: str, analysis_end: str) -> pd.DataFrame:
        """
        Use NEMOSIS to get actual data. Also omits any intervention periods
        """
        # NEMOSIS start time must precede end of interval of interest by 5 minutes
        nemosis_start = (
            datetime.strptime(analysis_start, "%Y/%m/%d %H:%M:%S")
            - timedelta(minutes=5)
        ).strftime("%Y/%m/%d %H:%M:%S")
        # use NEMOSIS to compile pre-cached data and filter out interventions
        actual_df = nemosis.dynamic_data_compiler(
            nemosis_start,
            analysis_end,
```

(continues on next page)

(continued from previous page)

```
            "DISPATCHREGIONSUM",
            nemosis_cache,
            filter_cols=["INTERVENTION"],
            filter_values=([0],),
            fformat="parquet",
        )
        return actual_df

    def calculate_pd_forecast_demand_error(
        actual_demand: pd.DataFrame, forecast_demand: pd.DataFrame
    ) -> pd.DataFrame:
        """
        Calculate PD forecast demand error given actual and forecast demand

        Ahead time calculation reflects the fact that PD actual run time is
        30 minutes before the nominal run time.
        """
        # merge the two types of demand
        merged = pd.merge(
            forecast_demand,
            actual_demand,
            on=["forecasted_time", "REGIONID"],
            how="left",
        )
        if len(merged) > len(forecast_demand):
            raise ValueError(
                "Merge should return DataFrame with dimensions of forecast data"
            )
        # subtract 30 minutes from run time to get actual run time
        merged["ahead_time"] = merged["forecasted_time"] - (
            merged["run_time"] - timedelta(minutes=30)
        )
        # calculate forecast error
        forecast_error = (
            merged["TOTALDEMAND"] - merged["FORECAST_TOTALDEMAND"]
        ).rename("TOTALDEMAND")
        # create the forecast error DataFrame
        forecast_error = pd.concat(
            [forecast_error, merged["ahead_time"], merged["REGIONID"]], axis=1
        ).set_index(merged["forecasted_time"])
        return forecast_error

    # get forecast data
    forecast_df = get_forecast_data(analysis_start, analysis_end)
    # rename columns in preparation for merge
    forecast_df = forecast_df.rename(
        columns={
            "TOTALDEMAND": "FORECAST_TOTALDEMAND",
            "DATETIME": "forecasted_time",
            "PREDISPATCH_RUN_DATETIME": "run_time",
        }
    )
```

(continues on next page)

(continued from previous page)

```
    forecast_demand = forecast_df[
        ["run_time", "forecasted_time", "REGIONID", "FORECAST_TOTALDEMAND"]
    ]

    # get actual data
    actual_df = get_actual_data(analysis_start, analysis_end)
    # rename columns in preparation for merge
    actual_df = actual_df.rename(
        columns={
            "SETTLEMENTDATE": "forecasted_time",
            "TOTALDEMAND": "TOTALDEMAND",
        }
    )
    actual_demand = actual_df[["forecasted_time", "REGIONID", "TOTALDEMAND"]]

    forecast_error = calculate_pd_forecast_demand_error(actual_demand, forecast_demand)
    return forecast_error
```

```
forecast_error = calculate_predispatch_demand_forecast_error_vectorised(
    analysis_start, analysis_end
)
```

Region-by-region error percentiles

Below we plot regional error percentiles for all ahead times.

```
region_ahead_percentiles = {}
for region in (regions := ("QLD1", "NSW1", "VIC1", "SA1", "TAS1")):
    percentile_data = []
    region_error = forecast_error.query("REGIONID==@region")
    for quantile in (0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95, 0.99):
        quantile_result = region_error.groupby(
            region_error["ahead_time"].dt.total_seconds() / (60**2)
        )["TOTALDEMAND"].quantile(quantile)
        percentile_result = pd.concat(
            [
                quantile_result,
                pd.Series(
                    np.repeat(quantile * 100, len(quantile_result)),
                    index=quantile_result.index,
                    name="Percentile",
                ).astype(int),
            ],
            axis=1,
        )
        percentile_data.append(percentile_result)
    percentile_df = pd.concat(percentile_data, axis=0).reset_index()
    region_ahead_percentiles[region] = percentile_df
```

```
figs = []
for region in regions:
    fig = px.line(
        region_ahead_percentiles[region],
        x="ahead_time",
        y="TOTALDEMAND",
        color="Percentile",
        title=f"PD {region} Demand Forecast Error, 2021<br><sup>Error = Actual - Forecast
↪</sup>",
        labels={
            "TOTALDEMAND": "Demand Forecast Error (MW)",
            "ahead_time": "Forecast Ahead Time (Hours, Actual Run Time)",
        },
        template=nemseer_template,
        color_discrete_map={
            1: "#E24A33",
            5: "#348ABD",
            10: "#988ED5",
            25: "#777777",
            50: "#FBC15E",
            75: "#777777",
            90: "#988ED5",
            95: "#348ABD",
            99: "#E24A33",
        },
    )
    fig["layout"]["xaxis"]["autorange"] = "reversed"
    figs.append(fig)
```

Why does the error drop off beyond ~24 hours?

A limited number of periods during the day are actually forecasted beyond 24 hours out.

`PREDISPATCH` is run until the end of the trading day for which bid price band submission has closed (1230 EST). So this means, for example:

- The 1300 PD (nominal) run will forecast out til 4AM two days away (39 hours)
- But the 1400 PD (nominal) run will still only forecast out til 4AM two days away (38 hours)
- And the 0800 PD (nominal) run the next day will still only forecast out til 4AM the next day (20 hours)

So because of this, the number of error samples drops off beyond 16 hours ahead (see figure below).

In addition, the runs closer to ~35 hours will be forecasts for periods in the early hours of the morning. These periods tend to have more predictable demand.

```
sample_count = px.line(
    forecast_error.groupby(forecast_error["ahead_time"].dt.total_seconds() / (60**2))[
        "TOTALDEMAND"
    ]
    .count()
    .rename("Computed Errors"),
    labels={"value": "Count of Samples"},
```

(continues on next page)

(continued from previous page)

```
    template=nemseer_template,
)
sample_count.update_layout(legend_title="", xaxis=dict(title="Ahead Time (hours)"));
```

NEM-wide Demand Forecast Error, less than 24 hours

Because of the reasons above, we'll focus on ahead times of up to 24 hours.

```
nem_error = (
    forecast_error.reset_index()
    .groupby(["forecasted_time", "ahead_time"])["TOTALDEMAND"]
    .sum()
    .reset_index()
)
nem_percentile_data = []
for quantile in (0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95, 0.99):
    nem_quantile_result = nem_error.groupby(
        nem_error["ahead_time"].dt.total_seconds() / (60**2)
    )["TOTALDEMAND"].quantile(quantile)
    nem_percentile_result = pd.concat(
        [
            nem_quantile_result,
            pd.Series(
                np.repeat(quantile * 100, len(nem_quantile_result)),
                index=nem_quantile_result.index,
                name="Percentile",
            ).astype(int),
        ],
        axis=1,
    )
    nem_percentile_data.append(nem_percentile_result)
nem_percentile_df = pd.concat(nem_percentile_data, axis=0).reset_index()
```

```
nemwide = px.line(
    nem_percentile_df.query("ahead_time < 24"),
    x="ahead_time",
    y="TOTALDEMAND",
    color="Percentile",
    title=f"Pre-dispatch NEM-wide Demand Forecast Error, 2021<br><sup>Error = Actual -␣
↪Forecast</sup>",
    labels={
        "TOTALDEMAND": "Demand Forecast Error (MW)",
        "ahead_time": "Forecast Ahead Time (Hours, Actual Run Time)",
    },
    template=nemseer_template,
    color_discrete_map={
        1: "#E24A33",
        5: "#348ABD",
        10: "#988ED5",
        25: "#777777",
        50: "#FBC15E",
```

(continues on next page)

(continued from previous page)

```
        75: "#777777",
        90: "#988ED5",
        95: "#348ABD",
        99: "#E24A33",
    },
)
nemwide["layout"]["xaxis"]["autorange"] = "reversed"
```

Distributions of Day-Ahead Demand Forecast Error by Region

We can see that the TOTALDEMAND day-ahead demand forecast error distribution is long-tailed for every region.

```
day_ahead = forecast_error[
    forecast_error["ahead_time"].dt.total_seconds() / (60**2) == 24
]
da_dists = px.histogram(
    day_ahead,
    x="TOTALDEMAND",
    facet_row="REGIONID",
    title="Pre-dispatch Demand Forecast Error, 2021<br><sup>Day-Ahead (24 hours ahead)</
↪sup>",
    template=nemseer_template,
)
da_dists.update_layout(xaxis=dict(title="Demand Forecast Error (MW)"));
```

9.4.4 Energy price convergence in 2021

In this example, we will take a look at energy price convergence.

To do this, we'll look at price forecasts from (30-minute) pre-dispatch (*PREDISPATCH*) and 5-minute pre-dispatch (*P5MIN*).

We'll also look at the relationship between demand forecast error and energy price convergence.

Note: While we use *price error* to refer to the difference between actual prices and forecast prices in some parts of this example, we prefer *price convergence*. The intention of `P5MIN` and `PREDISPATCH` is to provide AEMO and participants with up-to-date system and market information. A potential outcome of this is that participants change their decisions (e.g. rebidding, as we note here). As such, prices reported in `P5MIN` and `PREDISPATCH` should not be strictly interpreted as forecasts.

Key imports

```
# standard libraries
import logging
from datetime import datetime, timedelta
from pathlib import Path
from typing import Tuple

# NEM data libraries
# NEMOSIS for actual demand data
# NEMSEER for forecast demand data
import nemosis
from nemseer import compile_data, download_raw_data, generate_runtimes

# data wrangling libraries
import pandas as pd
import xarray as xr

# interactive plotting
from plotly.subplots import make_subplots
import plotly.express as px
import plotly.graph_objects as go
import plotly.io as pio

# static plotting
import matplotlib
import matplotlib.pyplot as plt

# silence NEMSEER and NEMOSIS logging
logging.getLogger("nemosis").setLevel(logging.WARNING)
logging.getLogger("nemseer").setLevel(logging.ERROR)
```

Plot Styling

```
nemseer_template = dict(
    layout=go.Layout(
        font_family="Source Sans 3",
        title_font_size=24,
        title_x=0.05,
        plot_bgcolor="#f0f0f0",
        colorway=px.colors.qualitative.Bold,
    )
)
plt.style.use(Path("styling", "matplotlib_styling.mplstyle"))
```

Defining our analysis start and end dates

```
analysis_start = "2021/01/01 00:00:00"
analysis_end = "2022/01/01 00:00:00"
```

Getting Data

In AEMO data tables, the energy price in $/MW/hr is usually found in the RRP column.

We will focus on 5-minute dispatch interval prices.

Note: Prior to midnight 30 September (commencement of 5-minute settlement), market settlement prices for energy were calculated as the average of the 6 dispatch interval prices in a half hour window.

Obtaining actual price data from NEMOSIS

We will download DISPATCHPRICE to access the RRP (energy price) field and cache it so that it's ready for computation.

```
nemosis_cache = Path("nemosis_cache/")
if not nemosis_cache.exists():
    nemosis_cache.mkdir()
```

```
nemosis.cache_compiler(
    analysis_start, analysis_end, "DISPATCHPRICE", nemosis_cache, fformat="parquet"
)
```

Obtaining actual demand data from NEMOSIS

We can download DISPATCHREGIONSUM to get actual demand values used in dispatch (TOTALDEMAND).

```
nemosis.cache_compiler(
    analysis_start, analysis_end, "DISPATCHREGIONSUM", nemosis_cache, fformat="parquet"
)
```

Obtaining forecast price data from NEMSEER

We will download PRICE to access the RRP field in PREDISPATCH forecasts, and REGIONSOLUTION to access the RRP field in P5MIN forecasts. We'll cache it so that it's ready for computation.

```
download_raw_data(
    "PREDISPATCH",
    "PRICE",
    "nemseer_cache/",
    forecasted_start=analysis_start,
    forecasted_end=analysis_end,
)
```

(continues on next page)

(continued from previous page)

```
download_raw_data(
    "P5MIN",
    "REGIONSOLUTION",
    "nemseer_cache/",
    forecasted_start=analysis_start,
    forecasted_end=analysis_end,
)
```

Obtaining forecast demand data from NEMSEER

We will also download demand data. This is contained within `REGIONSOLUTION` for `P5MIN`, but we need to get the table `REGIONSUM` for `PREDISPATCH`.

```
download_raw_data(
    "PREDISPATCH",
    "REGIONSUM",
    "nemseer_cache/",
    forecasted_start=analysis_start,
    forecasted_end=analysis_end,
)
```

Comparing price (and demand) convergence for a particular time

The code below (toggle to unhide) allows us to plot forecast convergence for price and demand across `PREDISPATCH` and `P5MIN`.

Note that we adjust the *nominal run time* to the *actual run time* for both forecast types.

```
def plot_price_comparison(time: str, regionid: str) -> go.Figure:
    """
    Creates a figure that compares price forecasts from PD & P5MIN against
    the actual price.

    Args:
        time: Datetime string in format YYYY/mm/dd HH:MM:SS
        regionid: One of ("NSW1", "QLD1", "VIC1", "TAS1", "SA1")
    Returns:
        plotly Figure
    """

    def get_actual_data(time: str) -> pd.DataFrame:
        """
        Gets actual price data
        """
        # get actual data from the hour beforehand to the interval of interest
        nemosis_window = (
            (
                datetime.strptime(time, "%Y/%m/%d %H:%M:%S") - timedelta(minutes=60)
            ).strftime("%Y/%m/%d %H:%M:%S"),
```

(continues on next page)

(continued from previous page)

```
            time,
        )
        nemosis_price = nemosis.dynamic_data_compiler(
            nemosis_window[0],
            nemosis_window[1],
            "DISPATCHPRICE",
            nemosis_cache,
            filter_cols=["INTERVENTION"],
            filter_values=([0],),
        )
        actual_price = nemosis_price[["SETTLEMENTDATE", "REGIONID", "RRP"]]
        return actual_price

    def get_forecast_data(time: str) -> Tuple[xr.DataArray, xr.DataArray]:
        """
        Gets P5 and PD forecast price data
        Also corrects nominal to actual run time
        """
        # get P5 and PD forecast data
        ## get PD data
        run_start, run_end = generate_runtimes(time, time, "PREDISPATCH")
        pd_price = compile_data(
            run_start,
            run_end,
            time,
            time,
            "PREDISPATCH",
            "PRICE",
            "nemseer_cache/",
            data_format="xr",
        )["PRICE"]["RRP"]
        ## calculate actual run time from run time, then swap out nominal for actual
        pd_price = pd_price.assign_coords(
            {"actual_run_time": pd_price.coords["run_time"] - pd.Timedelta(30, "T")}
        )
        pd_price = pd_price.swap_dims({"run_time": "actual_run_time"}).drop("run_time")
        ## get P5 data
        p5_price = compile_data(
            run_start,
            run_end,
            time,
            time,
            "P5MIN",
            "REGIONSOLUTION",
            "nemseer_cache/",
            data_format="xr",
        )["REGIONSOLUTION"]["RRP"]
        ## calculate actual run time from run time, then swap out nominal for actual
        p5_price = p5_price.assign_coords(
            {"actual_run_time": p5_price.coords["run_time"] - pd.Timedelta(5, "T")}
        )
        p5_price = p5_price.swap_dims({"run_time": "actual_run_time"}).drop("run_time")
```

(continues on next page)

(continued from previous page)

```
        return pd_price, p5_price

    actual_price = get_actual_data(time).query("REGIONID==@regionid")
    pd_price, p5_price = get_forecast_data(time)
    pd_price = pd_price.sel(REGIONID=regionid)
    p5_price = p5_price.sel(REGIONID=regionid)
    # create plotly figure
    fig = go.Figure(
        layout=dict(
            title=f"Energy Price Comparison: {regionid} {time} <br><sup>Actual run times␣
↪for forecasts</sup>",
            template=nemseer_template

        ),

    )
    fig.add_traces(
        [
            go.Scatter(
                x=pd_price.isel(forecasted_time=0).to_dataframe().index,
                y=pd_price.isel(forecasted_time=0).to_dataframe()["RRP"],
                name="PD",
                mode="lines",
            ),
            go.Scatter(
                x=p5_price.isel(forecasted_time=0).to_dataframe().index,
                y=p5_price.isel(forecasted_time=0).to_dataframe()["RRP"],
                name="P5",
                mode="lines",
            ),
            go.Scatter(
                x=actual_price["SETTLEMENTDATE"],
                y=actual_price["RRP"],
                name="Actual",
                mode="lines",
            ),
        ]
    )
    return fig
```

```
def plot_demand_comparison(time: str, regionid: str) -> go.Figure:
    """
    Creates a figure that compares demand forecasts from PD & P5MIN against
    the actual demand used in dispatch.

    Args:
        time: Datetime string in format YYYY/mm/dd HH:MM:SS
        regionid: One of ("NSW1", "QLD1", "VIC1", "TAS1", "SA1")
    Returns:
        plotly Figure
    """
```

(continues on next page)

(continued from previous page)

```
    def get_actual_data(time: str) -> pd.DataFrame:
        """
        Gets actual demand data
        """
        # get actual data from the hour beforehand to the interval of interest
        nemosis_window = (
            (
                datetime.strptime(time, "%Y/%m/%d %H:%M:%S") - timedelta(minutes=60)
            ).strftime("%Y/%m/%d %H:%M:%S"),
            time,
        )
        nemosis_demand = nemosis.dynamic_data_compiler(
            nemosis_window[0],
            nemosis_window[1],
            "DISPATCHREGIONSUM",
            nemosis_cache,
            filter_cols=["INTERVENTION"],
            filter_values=([0],),
        )
        actual_demand = nemosis_demand[["SETTLEMENTDATE", "REGIONID", "TOTALDEMAND"]]
        return actual_demand.sort_values("SETTLEMENTDATE")

    def get_forecast_data(time: str) -> Tuple[xr.DataArray, xr.DataArray]:
        """
        Gets P5 and PD forecast demand data
        Also corrects nominal to actual run time
        """
        # get P5 and PD forecast data
        ## get PD data
        run_start, run_end = generate_runtimes(time, time, "PREDISPATCH")
        pd_demand = compile_data(
            run_start,
            run_end,
            time,
            time,
            "PREDISPATCH",
            "REGIONSUM",
            "nemseer_cache/",
            data_format="xr",
        )["REGIONSUM"]["TOTALDEMAND"]
        ## calculate actual run time from run time, then swap out nominal for actual
        pd_demand = pd_demand.assign_coords(
            {"actual_run_time": pd_demand.coords["run_time"] - pd.Timedelta(30, "T")}
        )
        pd_demand = pd_demand.swap_dims({"run_time": "actual_run_time"}).drop(
            "run_time"
        )
        ## get P5 data
        p5_demand = compile_data(
            run_start,
            run_end,
            time,
```

(continues on next page)

(continued from previous page)

```
            time,
            "P5MIN",
            "REGIONSOLUTION",
            "nemseer_cache/",
            data_format="xr",
        )["REGIONSOLUTION"]["TOTALDEMAND"]
        ## calculate actual run time from run time, then swap out nominal for actual
        p5_demand = p5_demand.assign_coords(
            {"actual_run_time": p5_demand.coords["run_time"] - pd.Timedelta(5, "T")}
        )
        p5_demand = p5_demand.swap_dims({"run_time": "actual_run_time"}).drop(
            "run_time"
        )
        return pd_demand, p5_demand

    actual_demand = get_actual_data(time).query("REGIONID==@regionid")
    pd_demand, p5_demand = get_forecast_data(time)
    pd_demand = pd_demand.sel(REGIONID=regionid)
    p5_demand = p5_demand.sel(REGIONID=regionid)
    # create plotly figure
    fig = go.Figure(
        layout=dict(
            title=f"Demand Comparison: {regionid} {time} <br><sup>Actual run times for␣
↪forecasts</sup>",
                    template=nemseer_template
        ),

    )
    fig.add_traces(
        [
            go.Scatter(
                x=pd_demand.isel(forecasted_time=0).to_dataframe().index,
                y=pd_demand.isel(forecasted_time=0).to_dataframe()["TOTALDEMAND"],
                name="PD",
                mode="lines",
            ),
            go.Scatter(
                x=p5_demand.isel(forecasted_time=0).to_dataframe().index,
                y=p5_demand.isel(forecasted_time=0).to_dataframe()["TOTALDEMAND"],
                name="P5",
                mode="lines",
            ),
            go.Scatter(
                x=actual_demand["SETTLEMENTDATE"],
                y=actual_demand["TOTALDEMAND"],
                mode="lines",
                name="Actual",
            ),
        ]
    )
    return fig
```

Example: Plotting price and demand convergence on a summer's evening

Let's plot an example - a summer's evening ramp event

```
time = "2021/12/30 18:00:00"
price_comp = plot_price_comparison(time, "NSW1")
demand_comp = plot_demand_comparison(time, "NSW1")
```

Looking at price convergence over the year

To try and look at convergence a bit more systematically, we'll compute the *"price error"* across 2021.

The code below obtains `P5MIN` and `PREDISPATCH` price forecasts, removes overlapping forecasted periods and calculates a *"price error"*.

- The last two `PREDISPATCH` forecasts overlap with `P5MIN`
  - These are removed from `PREDISPATCH`

```
def calculate_price_error(analysis_start: str, analysis_end: str) -> pd.DataFrame:
    """
    Calculates price error in PREDISPATCH and P5MIN forecasts for periods between
    analysis_start and analysis_end.

    Args:
        analysis_start: Start datetime, YYYY/mm/dd HH:MM:SS
        analysis_end: End datetime, YYYY/mm/dd HH:MM:SS
    Returns:
        DataFrame with computed price error mapped to the ahead time of the
        forecast and the forecasted time.
    """

    def get_actual_price_data() -> pd.DataFrame:
        """
        Gets actual price data
        """
        # get actual demand data for forecasted_time
        # nemosis start time must precede end of interval of interest by 5 minutes
        nemosis_window = (
            (
                datetime.strptime(analysis_start, "%Y/%m/%d %H:%M:%S")
                - timedelta(minutes=5)
            ).strftime("%Y/%m/%d %H:%M:%S"),
            analysis_end,
        )
        nemosis_price = nemosis.dynamic_data_compiler(
            nemosis_window[0],
            nemosis_window[1],
            "DISPATCHPRICE",
            nemosis_cache,
            filter_cols=["INTERVENTION"],
            filter_values=([0],),
        )
        actual_price = nemosis_price[["SETTLEMENTDATE", "REGIONID", "RRP"]]
```

(continues on next page)

(continued from previous page)

```
        actual_price = actual_price.rename(
            columns={"SETTLEMENTDATE": "forecasted_time"}
        )
        return actual_price

    def get_forecast_price_data(ftype: str) -> pd.DataFrame:
        """
        Get price forecast data for the analysis period given a particular forecast type

        Args:
            ftype: 'P5MIN' or 'PREDISPATCH'
        Returns:
            DataFrame with price forecast data
        """
        # ftype mappings
        table = {"PREDISPATCH": "PRICE", "P5MIN": "REGIONSOLUTION"}
        run_col = {"PREDISPATCH": "PREDISPATCH_RUN_DATETIME", "P5MIN": "RUN_DATETIME"}
        forecasted_col = {"PREDISPATCH": "DATETIME", "P5MIN": "INTERVAL_DATETIME"}
        # get run times
        forecasts_run_start, forecasts_run_end = generate_runtimes(
            analysis_start, analysis_end, ftype
        )
        df = compile_data(
            forecasts_run_start,
            forecasts_run_end,
            analysis_start,
            analysis_end,
            ftype,
            table[ftype],
            "nemseer_cache/",
        )[table[ftype]]
        # remove intervention periods
        df = df.query("INTERVENTION == 0")
        # rename run and forecasted time cols
        df = df.rename(
            columns={
                run_col[ftype]: "run_time",
                forecasted_col[ftype]: "forecasted_time",
            }
        )
        # ensure values are sorted by forecasted and run times for nth groupby operation
        return df[["run_time", "forecasted_time", "REGIONID", "RRP"]].sort_values(
            ["forecasted_time", "run_time"]
        )

    def combine_pd_p5_forecasts(
        p5_df: pd.DataFrame, pd_df: pd.DataFrame
    ) -> pd.DataFrame:
        """
        Combines P5 and PD forecasts, including removing PD overlap with P5
        """
        # remove PD overlap with P5MIN
```

(continues on next page)

(continued from previous page)

```
        pd_nooverlap = pd_df.groupby(
            ["forecasted_time", "REGIONID"], as_index=False
        ).nth(slice(None, -2))
        # concatenate and rename RRP to reflect that these are forecasted values
        forecast_prices = pd.concat([pd_nooverlap, p5_df], axis=0).sort_values(
            ["forecasted_time", "actual_run_time"]
        )
        forecast_prices = forecast_prices.rename(columns={"RRP": "FORECASTED_RRP"})
        return forecast_prices

    def process_price_error(
        forecast_prices: pd.DataFrame, actual_price: pd.DataFrame
    ) -> pd.DataFrame:
        """
        Merges actual and forecast prices and calculates ahead time and price error
        """
        # left merge to ensure each forecasted price is mapped to its corresponding␣
↪actual price
        all_prices = pd.merge(
            forecast_prices,
            actual_price,
            how="left",
            on=["forecasted_time", "REGIONID"],
        )
        all_prices["ahead_time"] = (
            all_prices["forecasted_time"] - all_prices["actual_run_time"]
        )
        all_prices["error"] = all_prices["RRP"] - all_prices["FORECASTED_RRP"]
        price_error = all_prices.drop(
            columns=["RRP", "FORECASTED_RRP", "actual_run_time"]
        )
        return price_error

    p5_df = get_forecast_price_data("P5MIN")
    pd_df = get_forecast_price_data("PREDISPATCH")
    # calulate actual run time for each forecast type
    p5_df["actual_run_time"] = p5_df["run_time"] - pd.Timedelta(minutes=5)
    pd_df["actual_run_time"] = pd_df["run_time"] - pd.Timedelta(minutes=30)
    p5_df = p5_df.drop(columns="run_time")
    pd_df = pd_df.drop(columns="run_time")
    # get forecast prices
    forecast_prices = combine_pd_p5_forecasts(p5_df, pd_df)

    # actual prices
    actual_price = get_actual_price_data()

    price_error = process_price_error(forecast_prices, actual_price)
    return price_error
```

```
price_error = calculate_price_error(analysis_start, analysis_end)
```

Absolute price error for each region, by dispatch interval and ahead time

Below we plot absolute price error (absolute value of price error) for each region across the year for forecasts:

- 5 minutes ahead
- 1 hour ahead
- 5 hours ahead
- 24 hours ahead

Since forecasts 1+ hour ahead are only produced by `PREDISPATCH`, and because `PREDISPATCH` is only run every half hour, we will only plot price errors for intervals that end on the half hour.

Extract ahead times of interest for each region

```
regions = ("QLD1", "NSW1", "VIC1", "TAS1", "SA1")
aheads = [dict(minutes=5), dict(hours=1), dict(hours=5), dict(days=1)]
region_price_errors = {}
for region in regions:
    region_df = price_error.query("REGIONID==@region").set_index("forecasted_time")
    # only keep select ahead times
    ahead_df = region_df[region_df["ahead_time"].isin(aheads)]
    ahead_errors = []
    # manual pivot to bring data for each ahead time into columns
    for ahead in reversed(aheads):
        time_unit = list(ahead.keys())[0]
        ahead_time = list(ahead.values())[0]
        desc = f"{ahead_time} {time_unit} ahead"
        ahead_errors.append(
            region_df[region_df["ahead_time"] == timedelta(**ahead)]["error"].rename(
                desc
            )
        )
    # create a new DataFrame from ahead time series
    # drop any rows with NAs. These will be intervals not on the half hour
    ahead_df = pd.concat(ahead_errors, axis=1).dropna(axis=0)
    region_price_errors[region] = ahead_df.abs()
```

```
abs_price_error_figs = []
for region in regions:
    fig = px.line(
        region_price_errors[region],
        title=(
            f"{region} Absolute Price Error "
            + "<br><sup>Half-hourly dispatch intervals only</sup>"
        ),
        template=nemseer_template
    )
    fig.update_layout(yaxis_title="Price Error ($/MW/h)", xaxis_title="Forecasted Time")
    abs_price_error_figs.append(fig)
```

Zoom in and see how price error changes across forecast ahead times when:

- Price errors are large
- Price errors are relatively small

Are price forecast "errors" correlated with demand forecast errors?

Below, we visualise demand forecast error against price forecast error to see if the two are correlated.

To calculate demand error, we draw on functions from the previous two examples.

```
def calculate_predispatch_regional_demand_forecast_error_vectorised(
    analysis_start: str, analysis_end: str
) -> pd.DataFrame:
    """
    Calculates PD demand forecast error (Actual - Forecast) for all forecasts
    that are run for a given forecasted_time in a vectorised fashion.

    Args:
        forecasted_time: Datetime string in the form YYYY/mm/dd HH:MM:SS
    Returns:
        pandas DataFrame with forecast error in `TOTALDEMAND` columns, the ahead time
        of the forecast run in `ahead_time`, and the forecasted time in
        `forecasted_time`.
    """

    def get_forecast_data(analysis_start: str, analysis_end: str) -> pd.DataFrame:
        """
        Use NEMSEER to get PD forecast data. Also omits any intervention periods.
        """
        # use NEMSEER functions to compile pre-cached data
        forecasts_run_start, forecasts_run_end = generate_runtimes(
            analysis_start, analysis_end, "PREDISPATCH"
        )
        forecast_df = compile_data(
            forecasts_run_start,
            forecasts_run_end,
            analysis_start,
            analysis_end,
            "PREDISPATCH",
            "REGIONSUM",
            "nemseer_cache/",
        )["REGIONSUM"]
        # remove intervention periods
        forecast_df = forecast_df.query("INTERVENTION == 0")
        return forecast_df

    def get_actual_data(analysis_start: str, analysis_end: str) -> pd.DataFrame:
        """
        Use NEMOSIS to get actual data. Also omits any intervention periods
        """
        # NEMOSIS start time must precede end of interval of interest by 5 minutes
        nemosis_start = (
            datetime.strptime(analysis_start, "%Y/%m/%d %H:%M:%S")
```

(continues on next page)

(continued from previous page)

```
            - timedelta(minutes=5)
        ).strftime("%Y/%m/%d %H:%M:%S")
        # use NEMOSIS to compile pre-cached data and filter out interventions
        actual_df = nemosis.dynamic_data_compiler(
            nemosis_start,
            analysis_end,
            "DISPATCHREGIONSUM",
            nemosis_cache,
            filter_cols=["INTERVENTION"],
            filter_values=([0],),
            fformat="parquet",
        )
        return actual_df

    def calculate_pd_forecast_demand_error(
        actual_demand: pd.DataFrame, forecast_demand: pd.DataFrame
    ) -> pd.DataFrame:
        """
        Calculate PD forecast demand error given actual and forecast demand

        Ahead time calculation reflects the fact that PD actual run time is
        30 minutes before the nominal run time.
        """
        # merge the two types of demand
        merged = pd.merge(
            forecast_demand,
            actual_demand,
            on=["forecasted_time", "REGIONID"],
            how="left",
        )
        if len(merged) > len(forecast_demand):
            raise ValueError(
                "Merge should return DataFrame with dimensions of forecast data"
            )
        # subtract 30 minutes from run time to get actual run time
        merged["ahead_time"] = merged["forecasted_time"] - (
            merged["run_time"] - timedelta(minutes=30)
        )
        # calculate forecast error
        forecast_error = (
            merged["TOTALDEMAND"] - merged["FORECAST_TOTALDEMAND"]
        ).rename("TOTALDEMAND")
        # create the forecast error DataFrame
        forecast_error = pd.concat(
            [forecast_error, merged["ahead_time"], merged["REGIONID"]], axis=1
        ).set_index(merged["forecasted_time"])
        return forecast_error

    # get forecast data
    forecast_df = get_forecast_data(analysis_start, analysis_end)
    # rename columns in preparation for merge
    forecast_df = forecast_df.rename(
```

(continues on next page)

(continued from previous page)

```
        columns={
            "TOTALDEMAND": "FORECAST_TOTALDEMAND",
            "DATETIME": "forecasted_time",
            "PREDISPATCH_RUN_DATETIME": "run_time",
        }
    )
    forecast_demand = forecast_df[
        ["run_time", "forecasted_time", "REGIONID", "FORECAST_TOTALDEMAND"]
    ]

    # get actual data
    actual_df = get_actual_data(analysis_start, analysis_end)
    # rename columns in preparation for merge
    actual_df = actual_df.rename(
        columns={
            "SETTLEMENTDATE": "forecasted_time",
            "TOTALDEMAND": "TOTALDEMAND",
        }
    )
    actual_demand = actual_df[["forecasted_time", "REGIONID", "TOTALDEMAND"]]

    forecast_error = calculate_pd_forecast_demand_error(actual_demand, forecast_demand)
    return forecast_error
```

```
def calculate_p5min_regional_demand_forecast_error_vectorised(
    analysis_start: str, analysis_end: str
) -> pd.DataFrame:
    """
    Calculates P5MIN demand forecast error (Actual - Forecast) for all forecasts
    that are run for a given forecasted_time in a vectorised fashion.

    Args:
        forecasted_time: Datetime string in the form YYYY/mm/dd HH:MM:SS
    Returns:
        pandas DataFrame with forecast error in `TOTALDEMAND` columns, the ahead time
        of the forecast run in `ahead_time`, and the forecasted time in
        `forecasted_time`.
    """

    def get_forecast_data(analysis_start: str, analysis_end: str) -> pd.DataFrame:
        """
        Use NEMSEER to get P5MIN forecast data. Also omits any intervention periods.
        """
        # use NEMSEER functions to compile pre-cached data
        forecasts_run_start, forecasts_run_end = generate_runtimes(
            analysis_start, analysis_end, "P5MIN"
        )
        forecast_df = compile_data(
            forecasts_run_start,
            forecasts_run_end,
            analysis_start,
            analysis_end,
```

(continues on next page)

(continued from previous page)

```
            "P5MIN",
            "REGIONSOLUTION",
            "nemseer_cache/",
        )["REGIONSOLUTION"]
        # remove intervention periods
        forecast_df = forecast_df.query("INTERVENTION == 0")
        return forecast_df

    def get_actual_data(analysis_start: str, analysis_end: str) -> pd.DataFrame:
        """
        Use NEMOSIS to get actual data. Also omits any intervention periods
        """
        # NEMOSIS start time must precede end of interval of interest by 5 minutes
        nemosis_start = (
            datetime.strptime(analysis_start, "%Y/%m/%d %H:%M:%S")
            - timedelta(minutes=5)
        ).strftime("%Y/%m/%d %H:%M:%S")
        # use NEMOSIS to compile pre-cached data and filter out interventions
        actual_df = nemosis.dynamic_data_compiler(
            nemosis_start,
            analysis_end,
            "DISPATCHREGIONSUM",
            nemosis_cache,
            filter_cols=["INTERVENTION"],
            filter_values=([0],),
            fformat="parquet",
        )
        return actual_df

    def calculate_p5min_forecast_demand_error(
        actual_demand: pd.DataFrame, forecast_demand: pd.DataFrame
    ) -> pd.DataFrame:
        """
        Calculate P5MIN forecast demand error given actual and forecast demand

        Ahead time calculation reflects the fact that P5MIN actual run time is
        5 minutes before the nominal run time.
        """
        # merge the two types of demand
        merged = pd.merge(
            forecast_demand,
            actual_demand,
            on=["forecasted_time", "REGIONID"],
            how="left",
        )
        if len(merged) > len(forecast_demand):
            raise ValueError(
                "Merge should return DataFrame with dimensions of forecast data"
            )
        # subtract 5 minutes from run time to get actual run time
        merged["ahead_time"] = merged["forecasted_time"] - (
            merged["run_time"] - timedelta(minutes=5)
```

(continues on next page)

(continued from previous page)

```
        )
        # calculate forecast error
        forecast_error = (
            merged["TOTALDEMAND"] - merged["FORECAST_TOTALDEMAND"]
        ).rename("TOTALDEMAND")
        # create the forecast error DataFrame
        forecast_error = pd.concat(
            [forecast_error, merged["ahead_time"], merged["REGIONID"]], axis=1
        ).set_index(merged["forecasted_time"])
        return forecast_error

    # get forecast data
    forecast_df = get_forecast_data(analysis_start, analysis_end)
    # rename columns in preparation for merge
    forecast_df = forecast_df.rename(
        columns={
            "TOTALDEMAND": "FORECAST_TOTALDEMAND",
            "INTERVAL_DATETIME": "forecasted_time",
            "RUN_DATETIME": "run_time",
        }
    )
    forecast_demand = forecast_df[
        ["run_time", "forecasted_time", "REGIONID", "FORECAST_TOTALDEMAND"]
    ]

    # get actual data
    actual_df = get_actual_data(analysis_start, analysis_end)
    # rename columns in preparation for merge
    actual_df = actual_df.rename(
        columns={
            "SETTLEMENTDATE": "forecasted_time",
            "TOTALDEMAND": "TOTALDEMAND",
        }
    )
    actual_demand = actual_df[["forecasted_time", "REGIONID", "TOTALDEMAND"]]

    forecast_error = calculate_p5min_forecast_demand_error(
        actual_demand, forecast_demand
    )
    return forecast_error
```

Calculating demand forecast errors

```
pd_demand_error = calculate_predispatch_regional_demand_forecast_error_vectorised(
    analysis_start, analysis_end
)
# remove periods in which PREDISPATCH overlaps with P5MIN
pd_demand_error = pd_demand_error[pd_demand_error["ahead_time"] > timedelta(hours=1)]
```

```
p5_demand_error = calculate_p5min_regional_demand_forecast_error_vectorised(
```

(continues on next page)

(continued from previous page)

```
    analysis_start, analysis_end
)
```

```
# combine PD and P5 forecasts
demand_error = pd.concat([pd_demand_error, p5_demand_error], axis=0)
if len(demand_error) != len(price_error):
    raise ValueError(
        "Price and demand error DataFrames do not have the same number of rows"
    )
```

Merging demand and price forecast errors

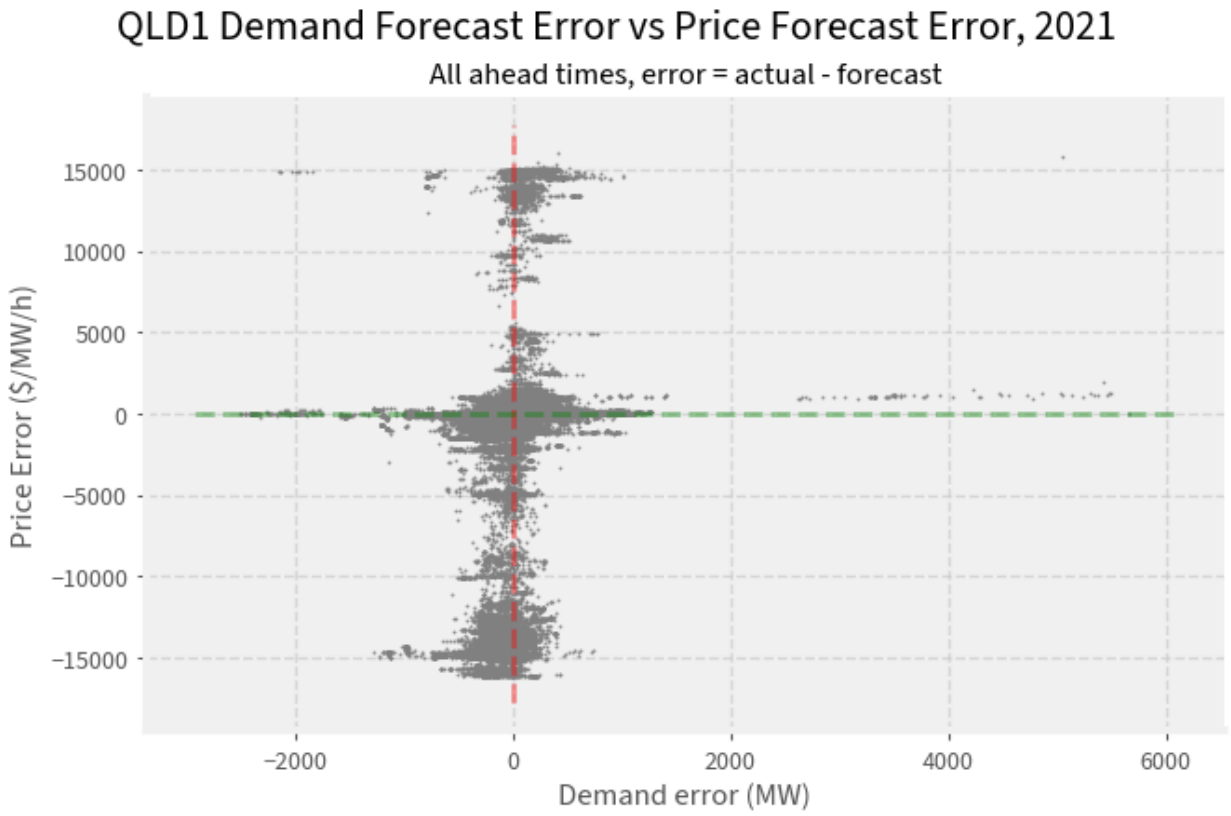
```
demand_price_error = pd.merge(
    demand_error.reset_index().rename(columns={"TOTALDEMAND": "demand_error"}),
    price_error.rename(columns={"error": "price_error"}),
    how="inner",
    on=["forecasted_time", "REGIONID", "ahead_time"],
)
if len(demand_price_error) != len(demand_error):
    raise ValueError("1:1 merge has not occurred")
```

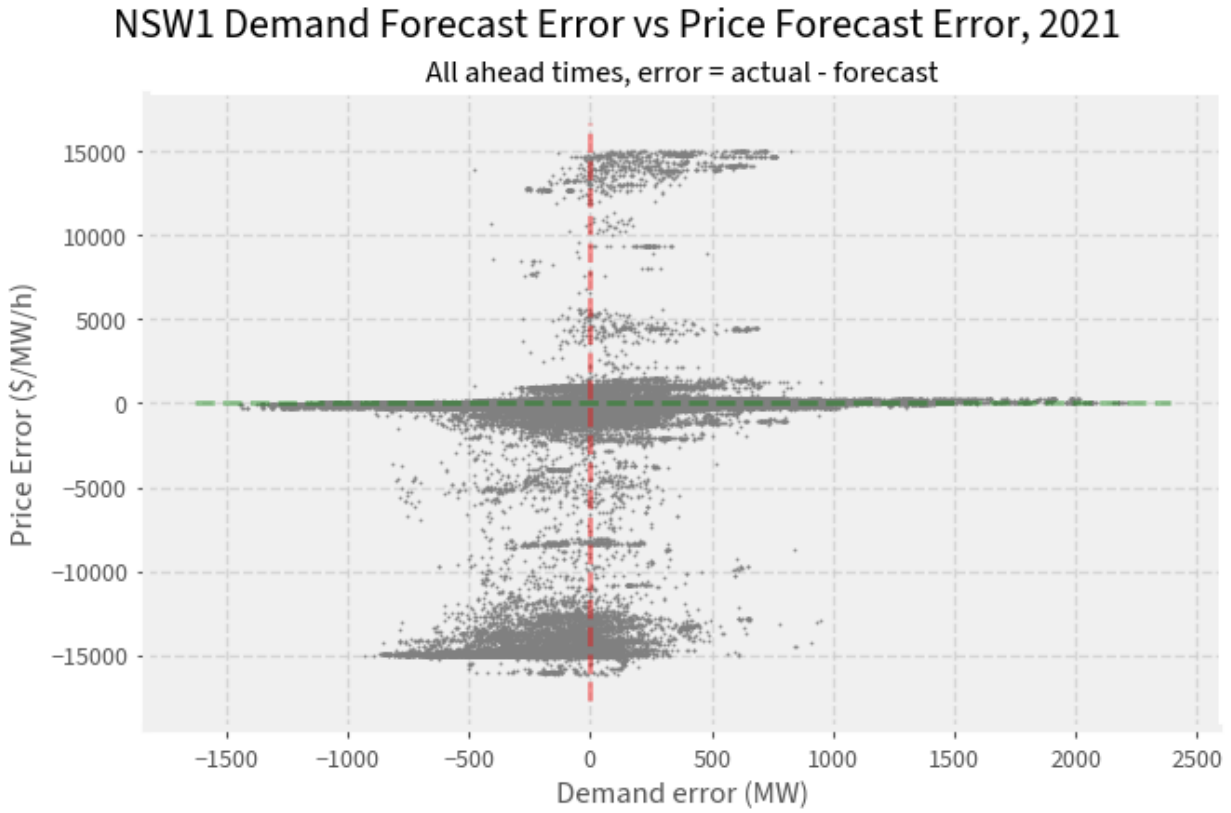
Regional scatter plots

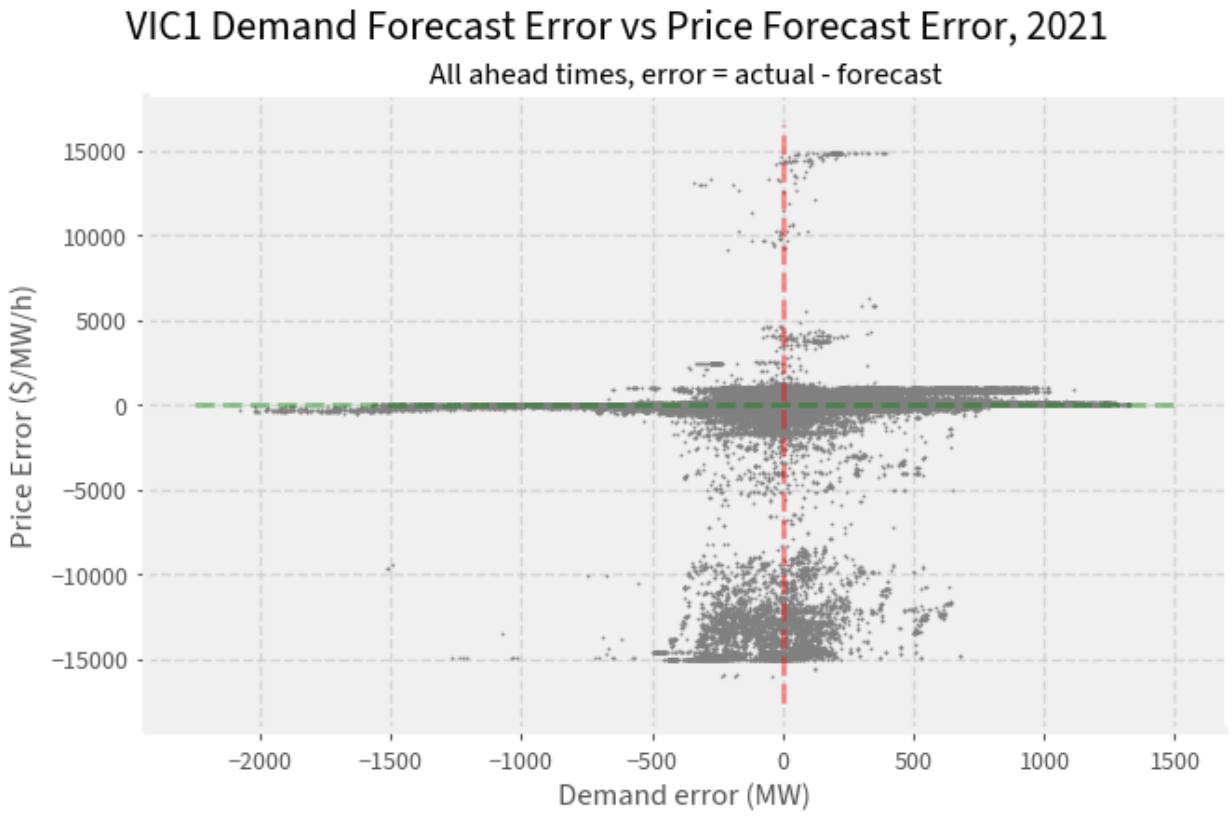
As we're plotting a lot of points, it can be faster to create static plots via `matplotlib` than interactive plots via `plotly`.

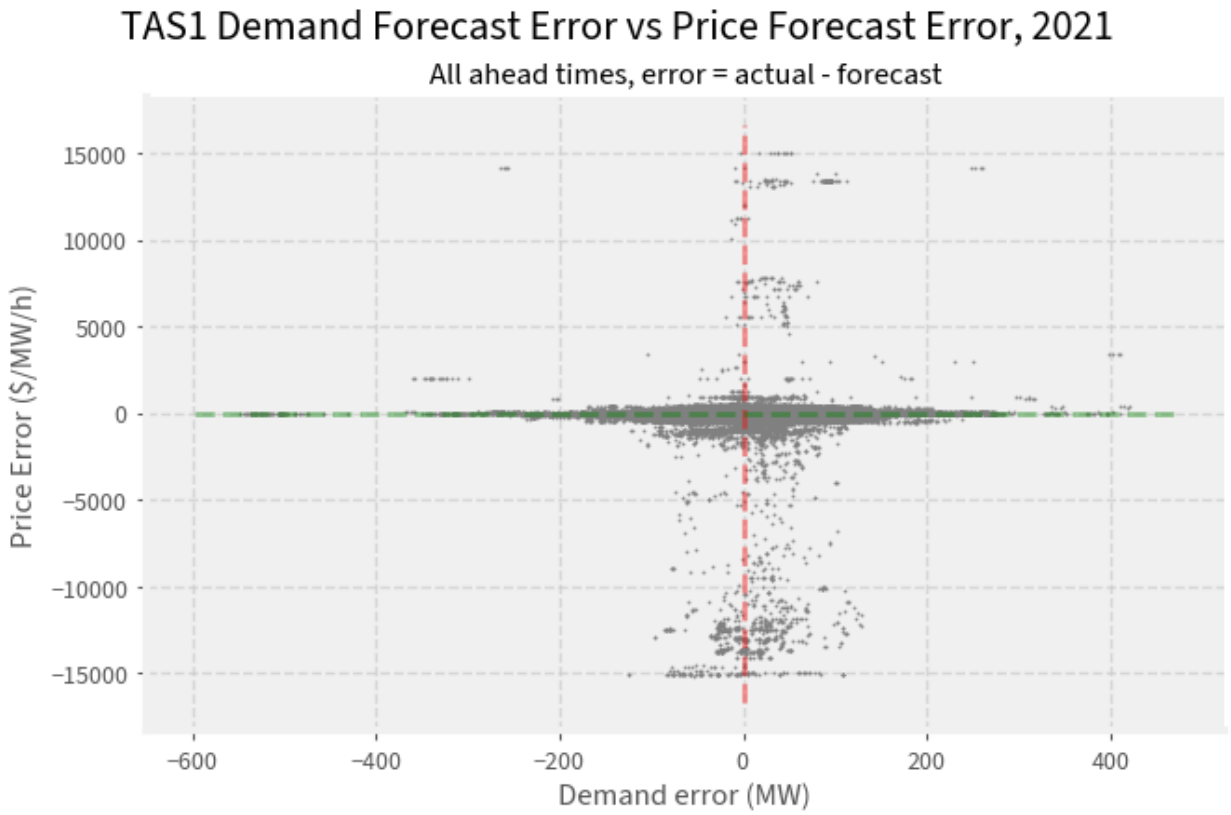
We can use the `.plot` attributes/methods offered by `pandas` to quickly generate static plots.

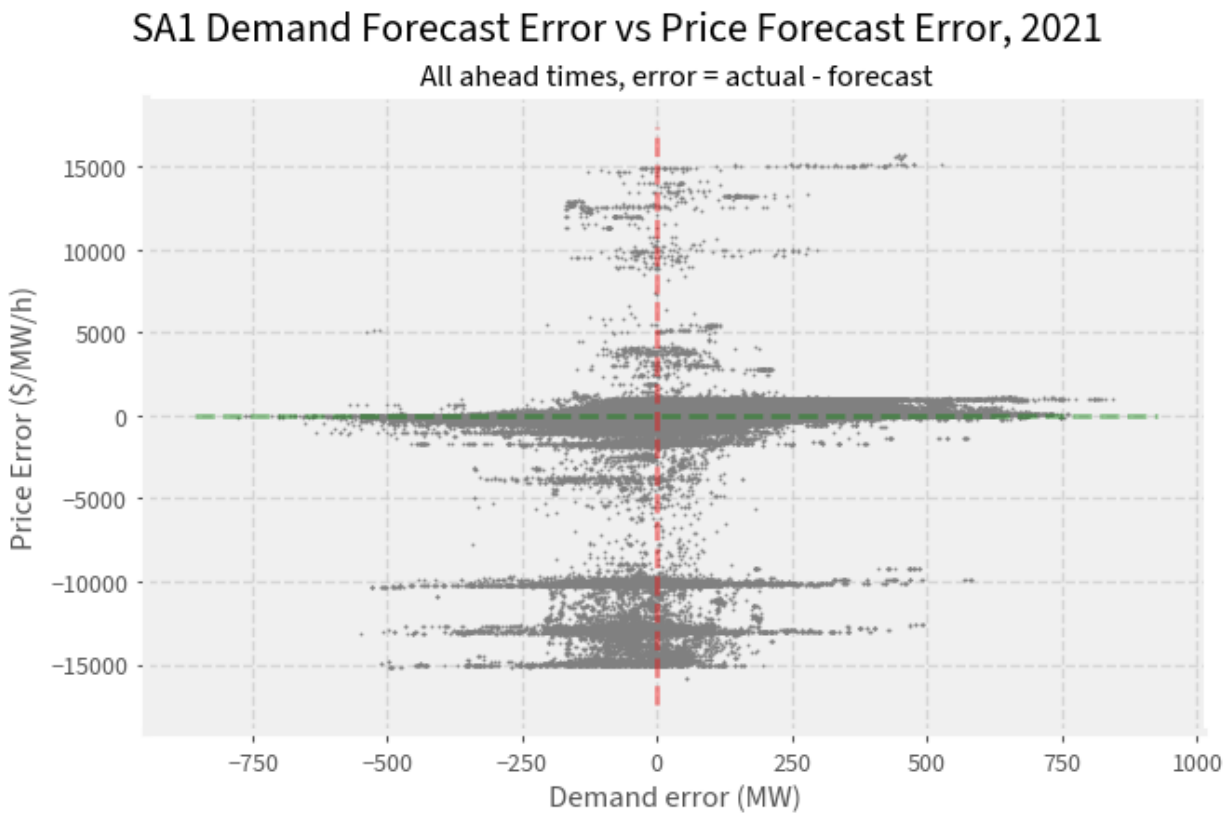
```
for region in regions:
    region_errors = demand_price_error.query("REGIONID==@region")
    ax = region_errors.plot.scatter(
        x="demand_error",
        y="price_error",
        s=0.75,
        xlabel="Demand error (MW)",
        ylabel="Price Error ($/MW/h)",
        c="grey",
        alpha=0.7
    )
    plt.suptitle(f"{region} Demand Forecast Error vs Price Forecast Error, 2021",␣
↪fontsize=16)
    plt.title("All ahead times, error = actual - forecast",fontsize=12)
    (ymin, ymax) = ax.get_ylim()
    (xmin, xmax) = ax.get_xlim()
    plt.vlines(0.0, ymin, ymax, ls="--", color="r", alpha=0.4)
    plt.hlines(0.0, xmin, xmax, ls="--", color="g", alpha=0.4)
```


QLD1 Demand Forecast Error vs Price Forecast Error, 2021
All ahead times, error = actual - forecast
Price Error ($/MW/h)
15000
10000
5000
0
−5000
−10000
−15000
−2000
0
2000
4000
6000
Demand error (MW)


NSW1 Demand Forecast Error vs Price Forecast Error, 2021
All ahead times, error = actual - forecast
Price Error ($/MW/h)
15000
10000
5000
0
−5000
−10000
−15000
−1500
−1000
−500
0
500
1000
1500
2000
2500
Demand error (MW)


VIC1 Demand Forecast Error vs Price Forecast Error, 2021
All ahead times, error = actual - forecast
Price Error ($/MW/h)
15000
10000
5000
0
−5000
−10000
−15000
−2000
−1500
−1000
−500
0
500
1000
1500
Demand error (MW)


TAS1 Demand Forecast Error vs Price Forecast Error, 2021
All ahead times, error = actual - forecast
Price Error ($/MW/h)
15000
10000
5000
0
−5000
−10000
−15000
−600
−400
−200
0
200
400
Demand error (MW)



Regional scatter plots, with ahead times > 1 hour

A large number of the points in the previous charts are for ahead times less than 1 hour (because each forecasted time will have 12 `P5MIN` forecasts that are within an hour).

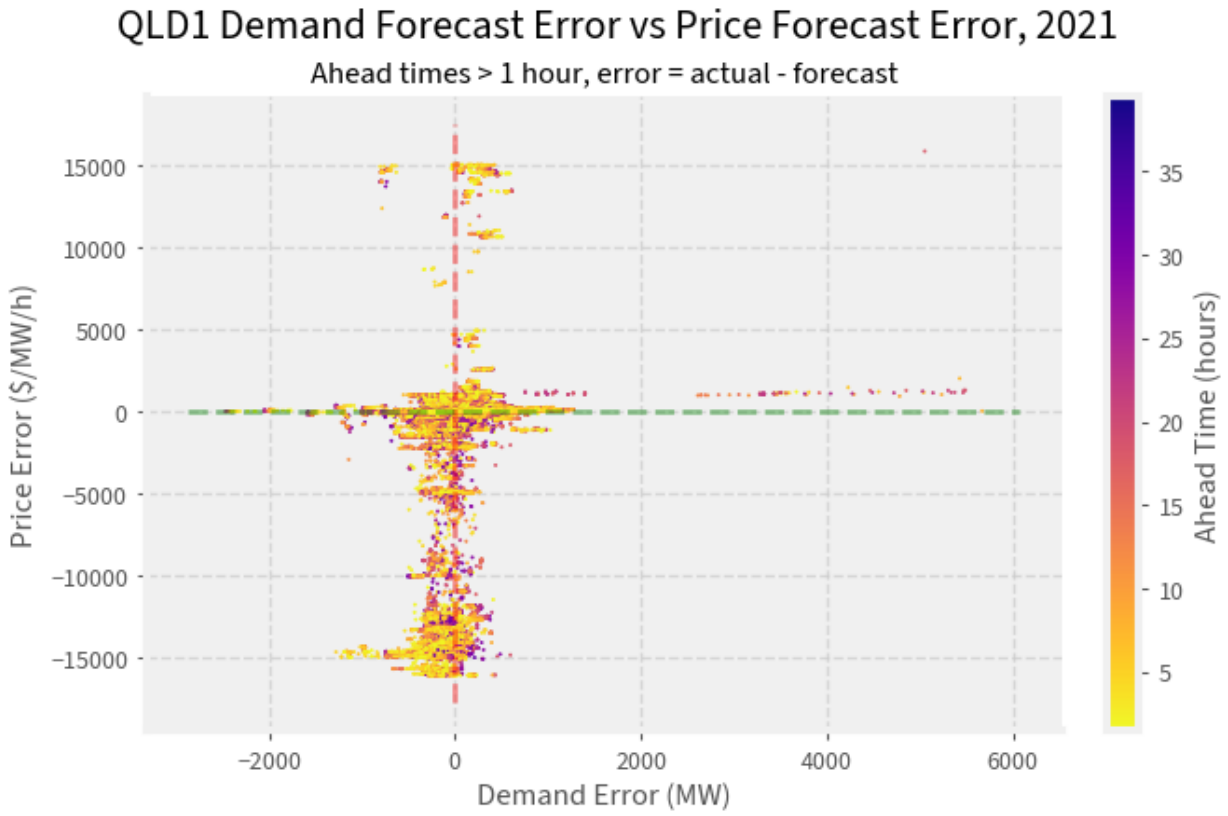
In the next few charts, we'll focus on ahead times > 1 hour (`PREDISPATCH`) and color each data point based on its ahead time.

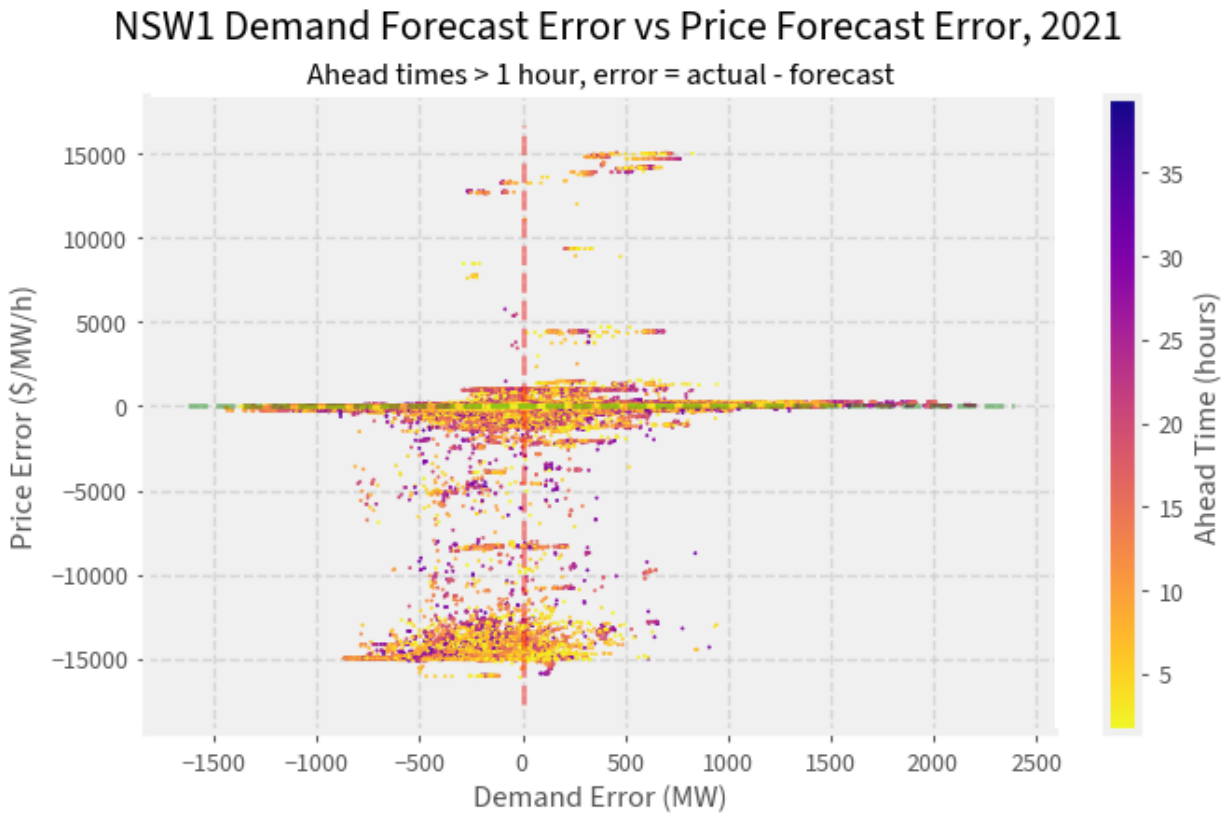
```
for region in regions:
    region_errors = demand_price_error.query("REGIONID==@region")
    # only retain ahead times > 1 hour
    region_errors = region_errors[region_errors["ahead_time"] > timedelta(hours=1)]
    # convert ahead time to hours
    region_errors["Ahead Time (hours)"] = (
        region_errors.loc[:, "ahead_time"].astype(int) * 10**-9 / 60.0 / 60.0
    )
    fig, ax = plt.subplots()
    im = ax.scatter(
        x=region_errors["demand_error"],
        y=region_errors["price_error"],
        s=0.75,
        c=region_errors["Ahead Time (hours)"],
        cmap=plt.get_cmap("plasma_r"),
    )
    (ymin, ymax) = ax.get_ylim()
```

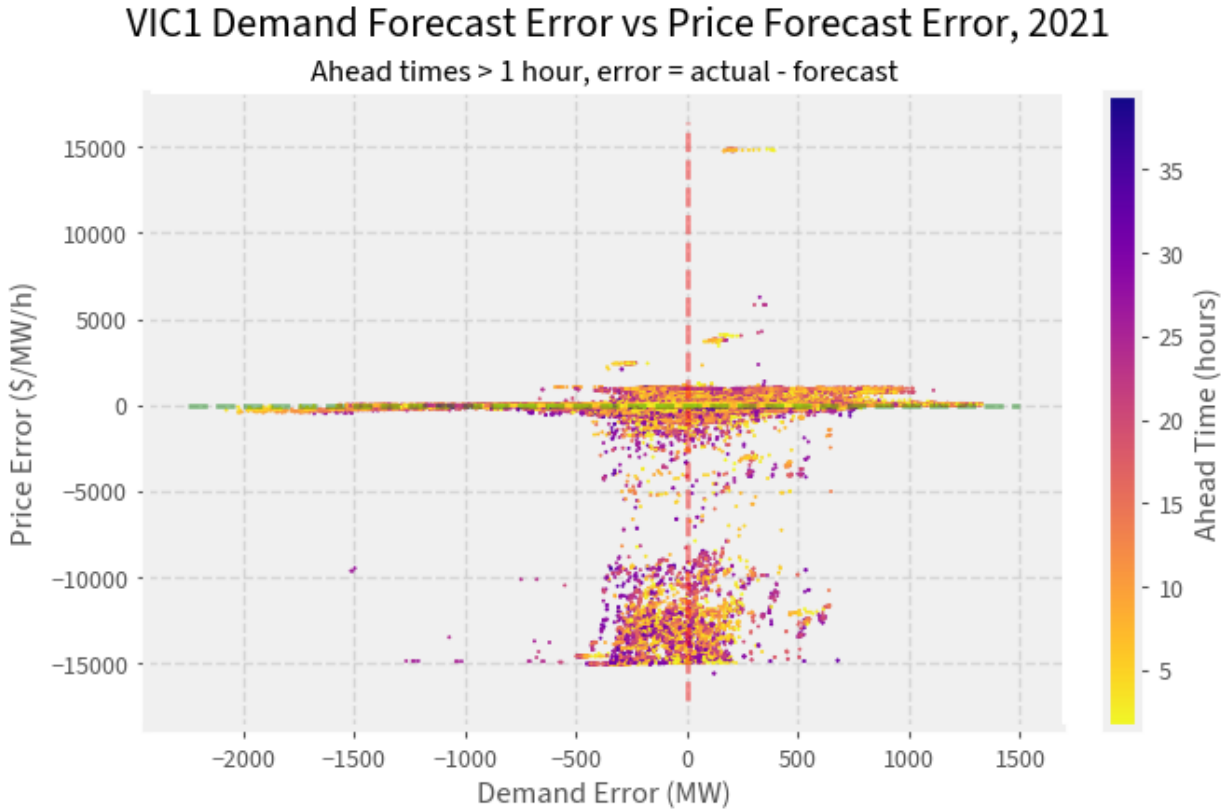
(continues on next page)

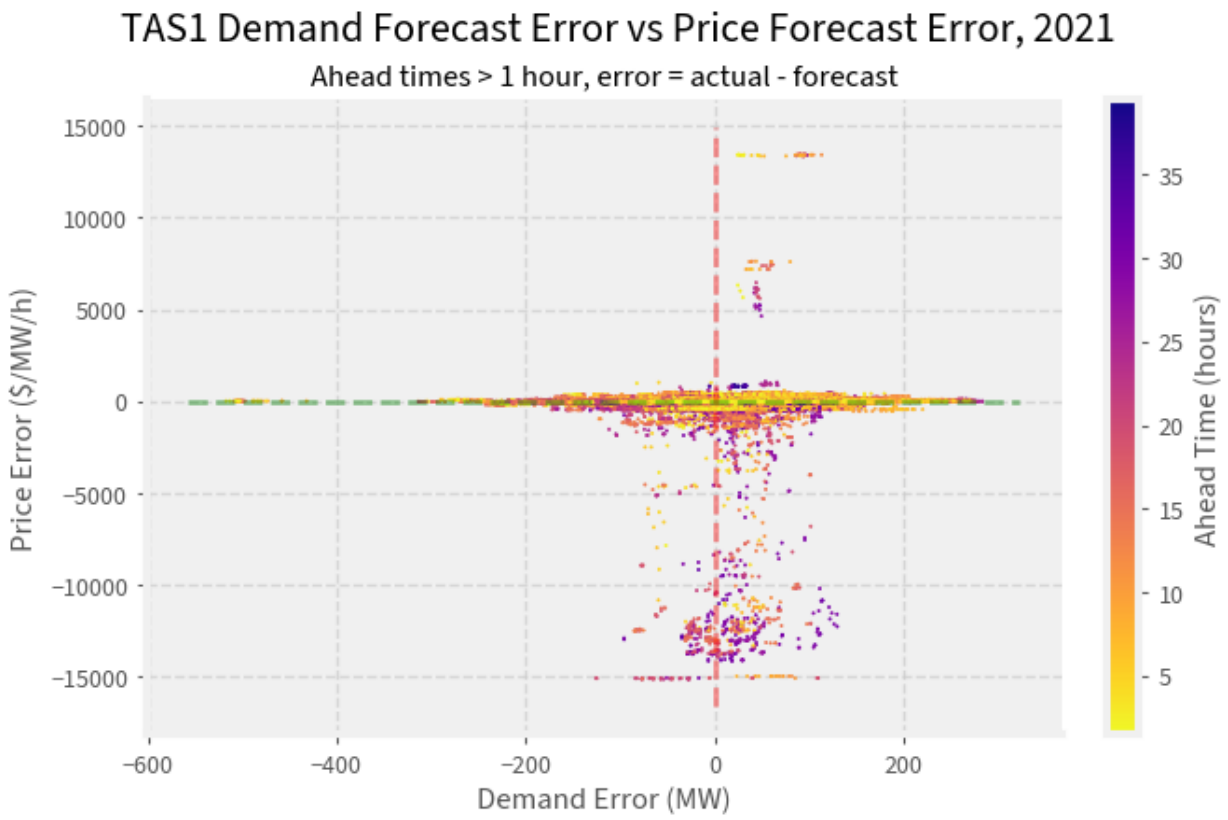
(continued from previous page)

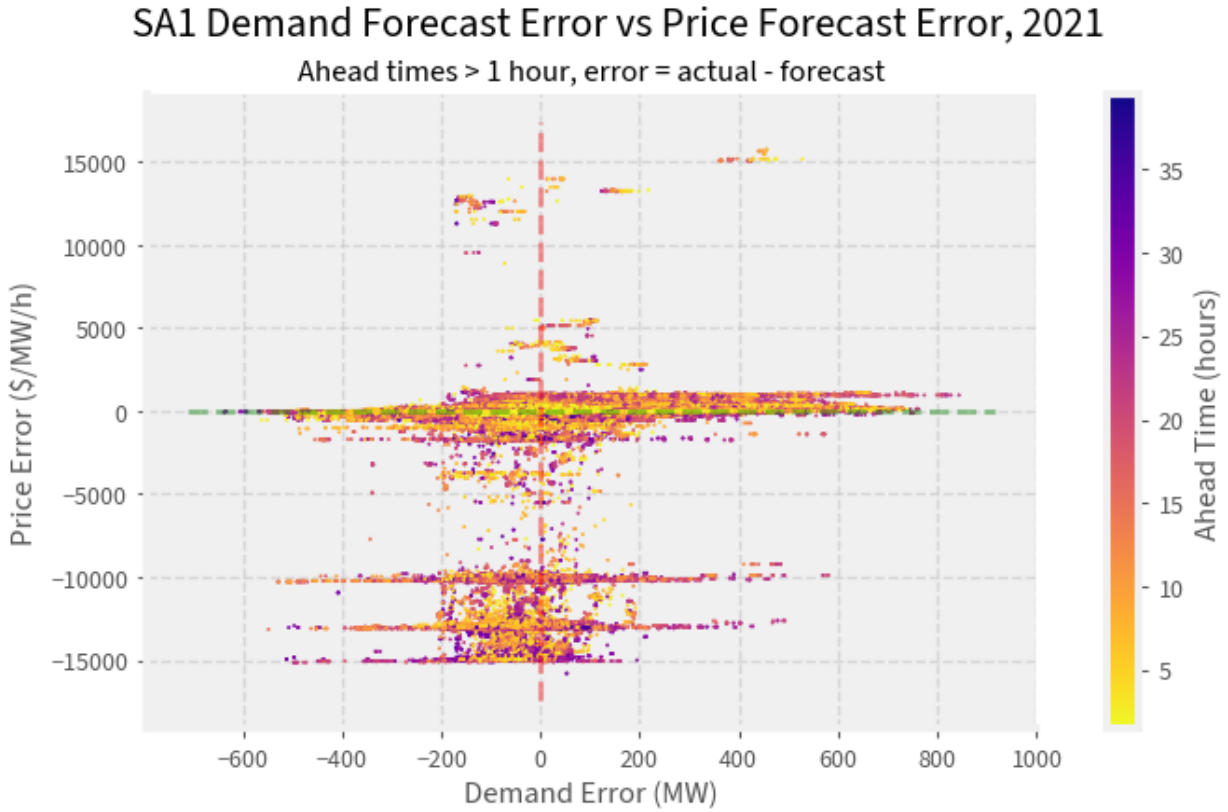
```
    (xmin, xmax) = ax.get_xlim()
    ax.set_xlabel("Demand Error (MW)")
    ax.set_ylabel("Price Error ($/MW/h)")
    plt.suptitle(f"{region} Demand Forecast Error vs Price Forecast Error, 2021",␣
↪fontsize=16)
    plt.title("Ahead times > 1 hour, error = actual - forecast",fontsize=12)
    plt.vlines(0.0, ymin, ymax, ls="--", color="r", alpha=0.4)
    plt.hlines(0.0, xmin, xmax, ls="--", color="g", alpha=0.4)
    fig.colorbar(im, ax=ax, label="Ahead Time (hours)")
```




NSW1 Demand Forecast Error vs Price Forecast Error, 2021
Ahead times > 1 hour, error = actual - forecast
Price Error ($/MW/h)
15000
10000
5000
0
−5000
−10000
−15000
−1500
−1000
−500
0
500
1000
1500
2000
2500
Demand Error (MW)
Ahead Time (hours)
35
30
25
20
15
10
5


VIC1 Demand Forecast Error vs Price Forecast Error, 2021
Ahead times > 1 hour, error = actual - forecast
Price Error ($/MW/h)
15000
10000
5000
0
−5000
−10000
−15000
−2000
−1500
−1000
−500
0
500
1000
1500
Demand Error (MW)
Ahead Time (hours)
35
30
25
20
15
10
5


TAS1 Demand Forecast Error vs Price Forecast Error, 2021
Ahead times > 1 hour, error = actual - forecast
Price Error ($/MW/h)
15000
10000
5000
0
−5000
−10000
−15000
−600
−400
−200
0
200
Demand Error (MW)
Ahead Time (hours)
35
30
25
20
15
10
5


SA1 Demand Forecast Error vs Price Forecast Error, 2021
Ahead times > 1 hour, error = actual - forecast
Price Error ($/MW/h)
15000
10000
5000
0
−5000
−10000
−15000
−600
−400
−200
0
200
400
600
800
1000
Demand Error (MW)
Ahead Time (hours)
35
30
25
20
15
10
5

What about closer to real-time?

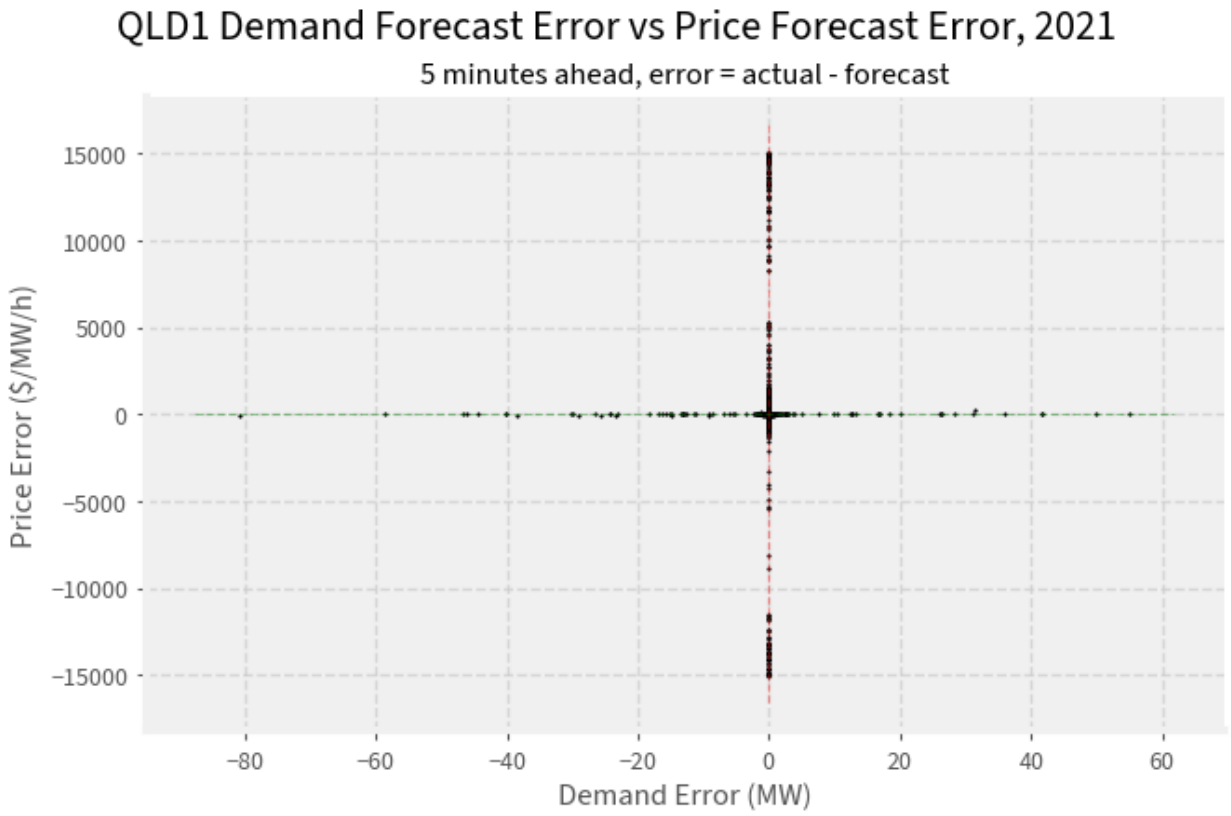
The previous charts include errors across all ahead times. **What about if we look at errors 5 minutes ahead of real time?**

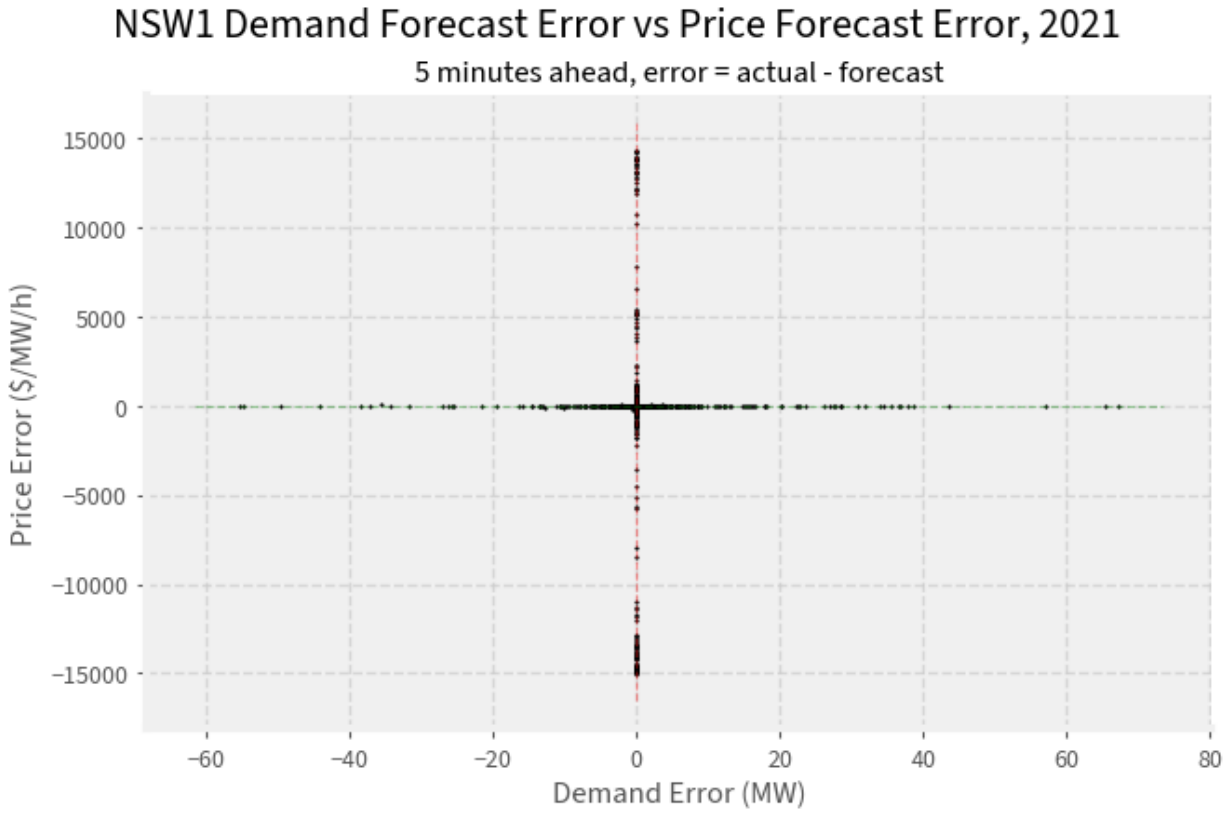
We can see that price errors can be significant 5 minutes ahead of dispatch, even if demand forecasts are roughly right. There are several factors that could account for this:

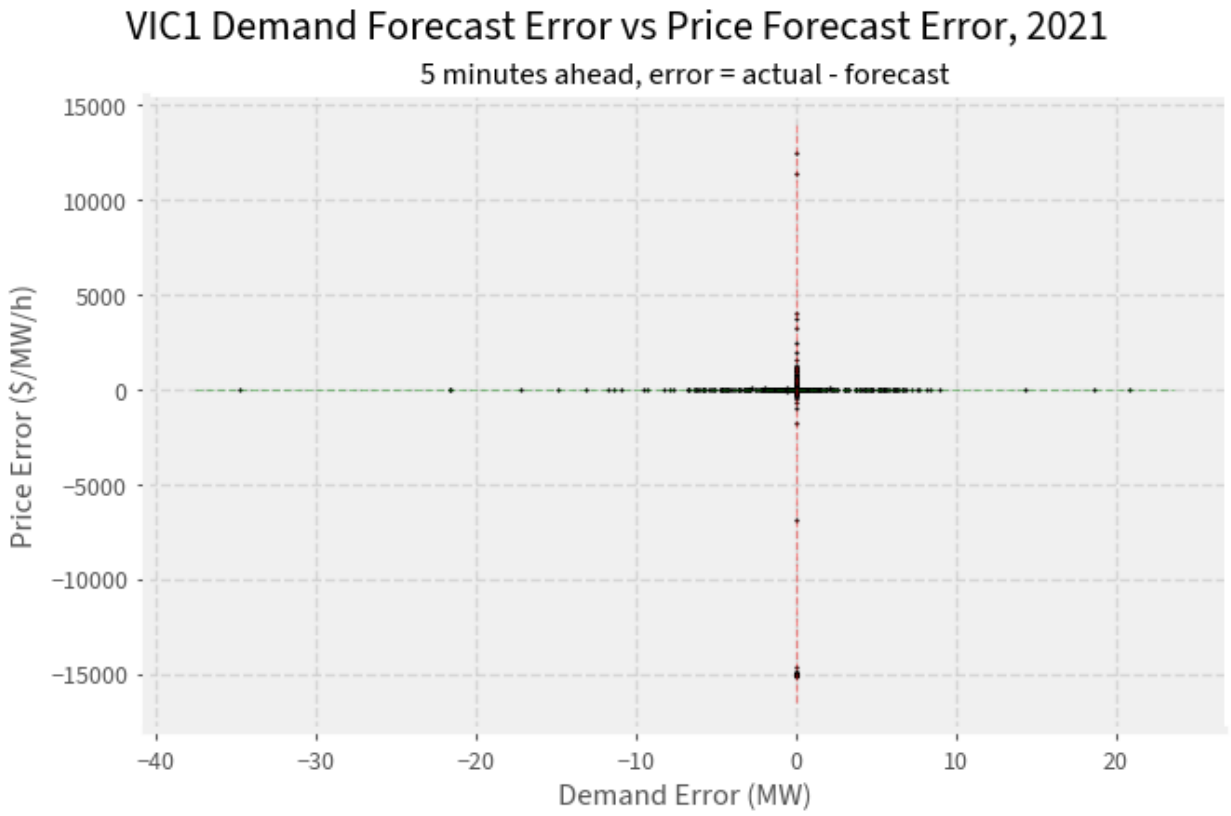
1. Renewable energy forecast accuracy
2. Dispatch vs. 5MPD constraint formulations
3. Sudden outages
4. Participant rebidding

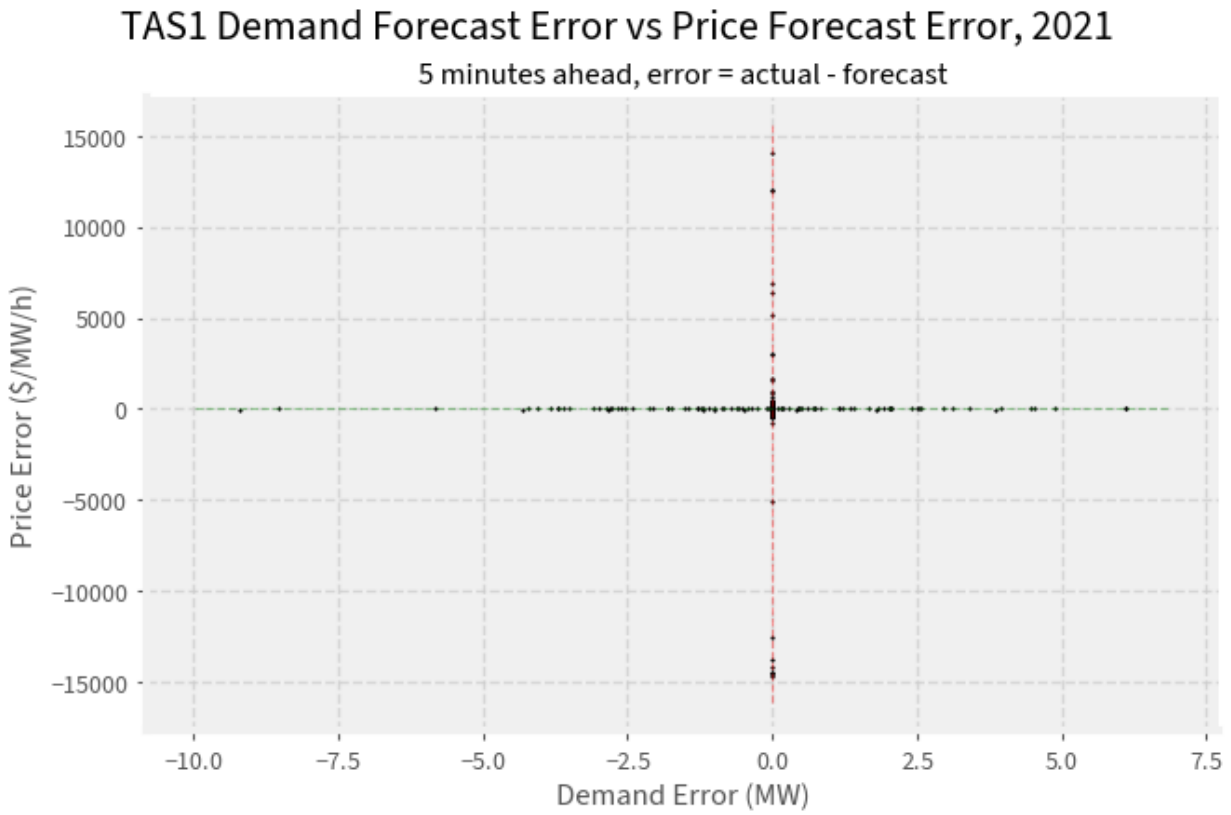
For more information, see the last page of the pre-dispatch standard operating procedure and page 33 of this paper.

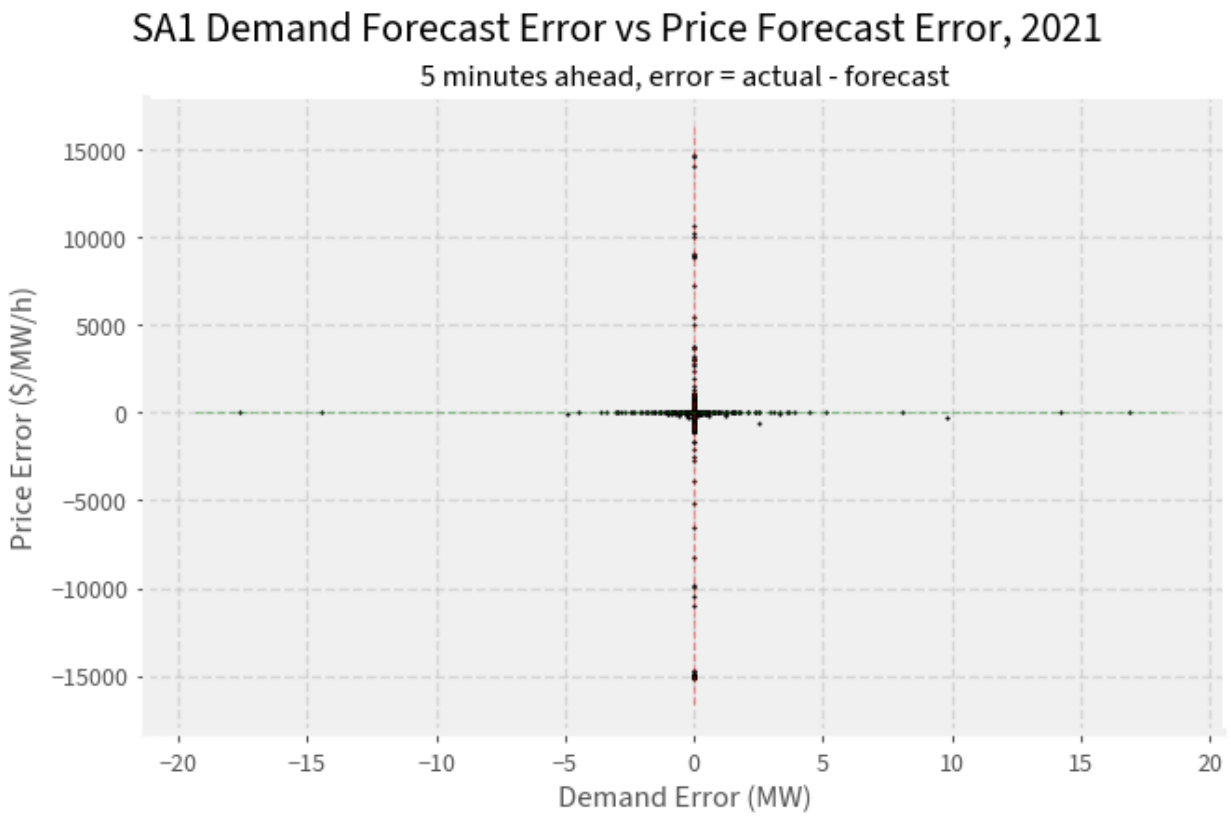
```
for region in regions:
    region_errors = demand_price_error.query("REGIONID==@region")
    region_errors = region_errors[region_errors["ahead_time"] == timedelta(minutes=5)]
    ax = region_errors.plot.scatter(
        x="demand_error",
        y="price_error",
        s=1,
        xlabel="Demand Error (MW)",
        ylabel="Price Error ($/MW/h)",
        color="black"
    )
    plt.suptitle(f"{region} Demand Forecast Error vs Price Forecast Error, 2021",␣
↪fontsize=16)
    plt.title("5 minutes ahead, error = actual - forecast",fontsize=12)
    (ymin, ymax) = ax.get_ylim()
    (xmin, xmax) = ax.get_xlim()
    plt.vlines(0.0, ymin, ymax, ls="--", color="r", alpha=0.4, lw=0.8)
    plt.hlines(0.0, xmin, xmax, ls="--", color="g", alpha=0.4, lw=0.8)
```


QLD1 Demand Forecast Error vs Price Forecast Error, 2021
5 minutes ahead, error = actual - forecast
Price Error ($/MW/h)
15000
10000
5000
0
−5000
−10000
−15000
−80
−60
−40
−20
0
20
40
60
Demand Error (MW)


NSW1 Demand Forecast Error vs Price Forecast Error, 2021
5 minutes ahead, error = actual - forecast
Price Error ($/MW/h)
15000
10000
5000
0
−5000
−10000
−15000
−60
−40
−20
0
20
40
60
80
Demand Error (MW)


VIC1 Demand Forecast Error vs Price Forecast Error, 2021
5 minutes ahead, error = actual - forecast
Price Error ($/MW/h)
15000
10000
5000
0
−5000
−10000
−15000
−40
−30
−20
−10
0
10
20
Demand Error (MW)


TAS1 Demand Forecast Error vs Price Forecast Error, 2021
5 minutes ahead, error = actual - forecast
Price Error ($/MW/h)
15000
10000
5000
0
−5000
−10000
−15000
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
Demand Error (MW)



9.5 API Reference

9.5.1 Classes

Query

Note: Use the `initialise()` class method to create an instance of the `Query` object, as this method assembles metadata relevant to NEMSEER cache searching.

class nemseer.query.**Query**(*run_start: str*, *run_end: str*, *forecasted_start: str*, *forecasted_end: str*, *forecast_type: str*, *tables: Union[str, List[str]]*, *metadata: Dict[str, str]*, *raw_cache*, *processed_cache=None*, *processed_queries: Optional[Union[Dict[str, Path], Dict]] = None*)

`Query` validates user inputs and dispatches data downloaders and compilers

Construct `Query` using the `Query.initialise()` constructor. This ensures query metadata is constructed approriately.

Query:

- **Validates user input data**
  - Checks datetimes fit `yyyy/mm/dd HH:MM` format
  - Checks datetime chronology (e.g. end is after start)

  - Checks requested datetimes are valid for each *forecast type*
  - Validates *forecast type*
  - Validates user-requested tables against what is available on NEMWeb
- Retains query metadata (via constructor class method *nemseer.query.Query.initialise()*)
- Can check *raw_cache* and *processed_cache* contents to streamline query compilation

Parameters

- **run_start** (*datetime*) –
- **run_end** (*datetime*) –
- **forecasted_start** (*datetime*) –
- **forecasted_end** (*datetime*) –
- **forecast_type** (*str*) –
- **tables** (*List[str]*) –
- **metadata** (*Dict[str, str]*) –
- **raw_cache** (*Path*) –
- **processed_cache** (*Optional[Path]*) –
- **processed_queries** (*Union[Dict[str, Path], Dict]*) –

run_start

Forecast runs at or after this datetime are queried.

Type
datetime.datetime

run_end

Forecast runs before or at this datetime are queried.

Type
datetime.datetime

forecasted_start

Forecasts pertaining to times at or after this datetime are retained.

Type
datetime.datetime

forecasted_end

Forecasts pertaining to times before or at this datetime are retaned.

Type
datetime.datetime

forecast_type

One of *nemseer.forecast_types*.

Type
str

tables

Table or tables required. A single table can be supplied as a string. Multiple tables can be supplied as a list of strings.

Type
List[str]

metadata

Metadata dictionary. Constructed by *Query.initialise()*.

Type
Dict[str, str]

raw_cache

Path to build or reuse *raw_cache*.

Type
pathlib.Path

processed_cache

Path to build or reuse *processed_cache*. Should be distinct from *raw_cache*

Type
optional

processed_queries

Defaults to *None* on initialisation. Populated once *Query.find_table_queries_in_processed_cache()* is called.

Type
Union[Dict[str, pathlib.Path], Dict]

check_all_raw_data_in_cache() → bool

Checks whether *all* requested data is already in the *raw_cache* as parquet

nemseer.downloader.ForecastTypeDownloader.download_csv() handles partial *raw_cache* completeness

If all requested data is already in the *raw_cache* as parquet, returns True. Otherwise returns False.

Return type
bool

find_table_queries_in_processed_cache(*data_format: str*) → None

Determines which tables already have queries saved in the *processed_cache*.

If data_format=df, this function will sieve through the metadata of all parquet files in the *processed_cache*. Note that parquet metadata is UTF-8 encoded. Similarly, data_format=xr will check the metadata of all netCDF files.

Modifies *Query.processed_queries* from `None` to a `dict`.

The `dict` is empty if:

1. *processed_cache* is `None`
2. No portion of the query has been saved in the *processed_cache*

If a portion of the queries are saved in the *processed_cache*, then *Query.processed_queries* will be equal to a `dict` that maps the saved query's table name to the saved query's filename.

Parameters
data_format (*str*) – As per *nemseer.compile_data()*

Return type
None

classmethod initialise(*run_start: str*, *run_end: str*, *forecasted_start: str*, *forecasted_end: str*, *forecast_type: str*, *tables: Union[str, List[str]]*, *raw_cache: str*, *processed_cache: Optional[str] = None*) → *Query*

Constructor method for *Query*. Assembles query metatdata.

Parameters

- **run_start** (*str*) –
- **run_end** (*str*) –
- **forecasted_start** (*str*) –
- **forecasted_end** (*str*) –
- **forecast_type** (*str*) –
- **tables** (*Union[str, List[str]]*) –
- **raw_cache** (*str*) –
- **processed_cache** (*Optional[str]*) –

Return type
Query

Downloader

Note: Use the *from_Query()* class method to create an instance of the *ForecastTypeLoader* object.

class nemseer.downloader.**ForecastTypeDownloader**(**, run_start: datetime*, *run_end: datetime*, *forecast_type: str*, *tables: List[str]*, *raw_cache: Path*)

ForecastTypeDownloader can initiate csv downloads and convert *raw_cache* csvs to the parquet format.

Parameters

- **run_start** (*datetime*) –
- **run_end** (*datetime*) –
- **forecast_type** (*str*) –
- **tables** (*List[str]*) –
- **raw_cache** (*Path*) –

run_start

Forecast runs at or after this datetime are queried.

Type
datetime.datetime

run_end

Forecast runs before or at this datetime are queried.

Type
datetime.datetime

forecast_type

One of *nemseer.forecast_types*

Type
str

tables

Table or tables required. A single table can be supplied as a string. Multiple tables can be supplied as a list of strings.

Type
List[str]

raw_cache

Path to download raw data to. Can reuse or build a new *raw_cache*.

Type
pathlib.Path

convert_to_parquet(*keep_csv=False*) → None

Converts all CSVs in the *raw_cache* to parquet

> **Warning:** A warning is printed if the filesize is greater than half of available memory as `pandas.DataFrame` consumes more than the file size in memory.

Return type
None

download_csv() → None

Downloads and unzips zip files given query loaded into *ForecastTypeDownloader*

This method will only download and unzip the relevant zip/csv if the corresponding *.parquet* file is not located in the specified *raw_cache*.

Return type
None

classmethod from_Query(*query:* Query) → *ForecastTypeDownloader*

Constructor method for *ForecastTypeDownloader* from *Query*

Parameters
query (Query) –

Return type
ForecastTypeDownloader

DataCompiler

Note: Use the *from_Query()* class method to create an instance of the *DataCompiler* object.

class nemseer.data_compilers.**DataCompiler**(*run_start: datetime*, *run_end: datetime*, *forecasted_start: datetime*, *forecasted_end: datetime*, *forecast_type: str*, *metadata: Dict[str, str]*, *raw_cache: Path*, *processed_cache: Union[None, Path]*, *processed_queries: Union[Dict[str, Path], Dict]*, *raw_tables: List[str]*, *compiled_data: Union[None, Dict[str, DataFrame], Dict[str, Dataset]] = None*)

DataCompiler compiles data from the *raw_cache* or *processed_cache*.

Parameters

- **run_start** (*datetime*) –
- **run_end** (*datetime*) –
- **forecasted_start** (*datetime*) –
- **forecasted_end** (*datetime*) –
- **forecast_type** (*str*) –
- **metadata** (*Dict[str, str]*) –
- **raw_cache** (*Path*) –
- **processed_cache** (*Union[None, Path]*) –
- **processed_queries** (*Union[Dict[str, Path], Dict]*) –
- **raw_tables** (*List[str]*) –
- **compiled_data** (*Union[None, Dict[str, DataFrame], Dict[str, Dataset]]*) –

run_start

Forecast runs at or after this datetime are queried.

Type
datetime.datetime

run_end

Forecast runs before or at this datetime are queried.

Type
datetime.datetime

forecasted_start

Forecasts pertaining to times at or after this datetime are retained.

Type
datetime.datetime

forecasted_end

Forecasts pertaining to times before or at this datetime are retaned.

Type
datetime.datetime

forecast_type

One of *nemseer.forecast_types*.

Type
str

tables

Table or tables required. A single table can be supplied as a string. Multiple tables can be supplied as a list of strings.

metadata

Metadata dictionary. Constructed by *Query.initialise()*.

Type
Dict[str, str]

raw_cache

Path to build or reuse *raw_cache*.

Type
optional

processed_cache

Path to build or reuse :term`processed cache`. Should be distinct from *raw_cache*

Type
optional

processed_queries

Defaults to None on initialisation.

Type
Union[Dict[str, pathlib.Path], Dict]

raw_table

Populated via *DataCompiler.from_Query()*

compiled_data

Defaults to None on initialisation. Populated once data is compiled by methods.

Type
Union[None, Dict[str, pandas.core.frame.DataFrame], Dict[str, xarray.core.dataset.Dataset]]

compile_processed_data(*data_format: str = 'df'*) → None

Compiles data from the *processed_cache*, as per entries in *processed_queries*, to a pandas.DataFrame (default) or to a xarray.Dataset.

This method will update *compiled_data*.

Parameters
data_format (*str*) – Default "df" (pandas.DataFrame). Other valid input is "xr", which compiles xarray.Dataset.

Return type
None

compile_raw_data(*data_format: str = 'df'*) → None

Compiles data from *raw_cache* to a pandas.DataFrame (default) or to a xarray.Dataset.

This compiler will:

- Skip invalid/corrupted files as recorded in *.invalid_aemo_files.txt*
- Read *raw_cache* parquet files and apply datetime filtering
- Convert DataFrame to xarray.Dataset (if data_format = "xr")
- Update *compiled_data*

Parameters
data_format (*str*) – Default "df" (pandas.DataFrame). Other valid input is "xr", which returns xarray.Dataset.

Return type
None

> **Warning:** Skips any files previously found to be invalid/corrupted and prints a warning

classmethod from_Query(*query:* Query) → *DataCompiler*

Constructor method for *DataCompiler* from *Query*.

Parameters
query (Query) –

Return type
DataCompiler

invalid_or_corrupted_files() → List[str]

A list of invalid/corrupted files as per files in *.invalid_aemo_files.txt*. Returns an empty list if the stubfile does not exist.

Return type
List[str]

write_to_processed_cache() → None

Writes netCDF4 for xarray.Dataset and parquet for pandas.DataFrame to the *processed_cache* with associated query metadata.

Note that parquet metadata needs to be UTF-8 encoded.

Raises

- **ValueError** – If *processed_cache* is None, or if *compiled_data* contains data that is neither all pandas.DataFrame or all xarray.Dataset
- **IOError** – If *compiled_data* is None

Return type
None

9.5.2 Functions

Query handlers

nemseer.query.**generate_sqlloader_filenames**(*run_start: datetime*, *run_end: datetime*, *forecast_type: str*, *tables: List[str]*) → Dict[Tuple[int, int, str], str]

Generates MMSDM Historical Data SQLLoader file names based on provided query data

Returns a tuple of query metadata (*table*, *year*, *month*) mapped to each filename

Parameters

- **run_start** (*datetime*) – Forecast runs at or after this datetime are queried.
- **run_end** (*datetime*) – Forecast runs before or at this datetime are queried.
- **forecast_type** (*str*) – One of *nemseer.forecast_types*.

- **tables** (*List[str]*) – Table or tables required, provided as a List.

Returns
A tuple of query metadata (*table*, *year*, *month*) mapped to each format-agnostic (*SQLLoader*) filename

Return type
Dict[*Tuple*[int, int, str], str]

Scrapers and downloaders

Scrapers: These functions scrape the *MMSDM Historical Data SQLLoader* repository to assist `nemseer` in validating inputs and providing feedback to users.

Downloaders: Used to download and unzip a `.zip` file.

`nemseer.downloader.`**`get_sqlloader_forecast_tables`**(*year: int*, *month: int*, *forecast_type: str*, *actual: bool = False*) → List[str]

Requestable tables of particular forecast type on MMSDM Historical Data SQLLoader

If `actual` = False, provides a list of tables that can be requested via *nemseer*.

If `actual` = True, returns actual tables available via NEMWeb, including all tables that are enumerated.

N.B.:

- Removes numbering from enumerated tables for *P5MIN* - e.g. *CONSTRAINTSOLUTION(x)* are all reduced to *CONSTRAINTSOLUTION*

Examples

See *querying table availability*

Parameters

- **year** (*int*) – Year
- **month** (*int*) – Month
- **forecast_type** (*str*) – One of *nemseer.forecast_types*
- **actual** (*bool*) –

Returns
List of tables associated with that forecast type for that period

Return type
List[str]

`nemseer.downloader.`**`get_sqlloader_years_and_months`**() → Dict[int, List[int]]

Years and months with data on NEMWeb MMSDM Historical Data SQLLoader .. rubric:: Examples

See *querying date ranges*

Returns
Months mapped to each year. Data is available for each of these months.

Return type
Dict[int, *List*[int]]

nemseer.downloader.**get_unzipped_csv**(*url: str*, *raw_cache: Path*) → None

Unzipped (single) csv file downloaded from *url* to *raw_cache*

This function:

1. Downloads zip file in chunks to limit memory use and enable progress bar
2. Validates that the zip contains a single file that has the same name as the zip
3. If the zip file is invalid, writes the file stub to *.invalid_aemo_files.txt*

Parameters

- **url** (*str*) – URL of zip
- **raw_cache** (*Path*) – Path to save zip. See *raw_cache*.

Returns
None. Extracts csvs to `raw_cache`.

Return type
None

Data handlers

Functions for handling various data states.

Valid inputs for *clean_forecast_csv* are the same as those for `pandas.read_csv`.

nemseer.data_handlers.**apply_run_and_forecasted_time_filters**(*df: DataFrame*, *run_start: datetime*, *run_end: datetime*, *forecasted_start: datetime*, *forecasted_end: datetime*, *forecast_type: str*) → DataFrame

Applies filtering for run times (i.e. *run_start* and *run_end*) and forecasted times (i.e. *forecasted_start* and *forecasted_end*).

Datetime filtering is applied to a column fetched from lookup tables that map the relevant column name to each *forecast type*. If the run time/forecasted column obtained from the lookup is not present in the DataFrame, the respective filter is not applied.

Parameters

- **run_start** (*datetime*) – Forecast runs at or after this datetime are queried.
- **run_end** (*datetime*) – Forecast runs before or at this datetime are queried.
- **forecasted_start** (*datetime*) – Forecasts pertaining to times at or after this datetime are retained.
- **forecasted_end** (*datetime*) – Forecasts pertaining to times before or at this datetime are retaned.
- **forecast_type** (*str*) – One of *nemseer.forecast_types*.
- **df** (*DataFrame*) –

Returns
DataFrame with appropriate datetime filtering applied.

Return type
DataFrame

nemseer.data_handlers.**clean_forecast_csv**(*filepath_or_buffer: Union[str, Path]*) → DataFrame

Given a forecast csv filepath or buffer, reads and cleans the forecast csv.

Cleans artefacts in the forecast csv files, including AEMO metadata at start of file and end of report line. Also removes any duplicate rows.

Parameters
filepath_or_buffer (*Union[str, Path]*) – As for `pandas.read_csv()`

Returns
Cleaned `pandas.DataFrame` with forecast data

Return type
DataFrame

Warning: Removes duplicate rows. Raises a warning when doing so.

nemseer.data_handlers.**to_xarray**(*df: DataFrame*, *forecast_type: str*)

Converts a `pandas.DataFrame` to a `xarray.Dataset` using nemseer definitions to determine Dataset dimensions.

The conversion is processed in chunks. If system memory usage exceeds 95%, the conversion is terminated with a MemoryError. This is more informative than the system killing the Python process.

Parameters

- **df** (*DataFrame*) – pandas.DataFrame to be converted.
- **forecast_type** (*str*) – One of *nemseer.forecast_types*.

Returns
`xarray.Dataset`.

Warning: Raises a warning when attempting to convert high-dimensional data.

9.5.3 Forecast-specific helpers

Datetime validators

These validators are specific to each *forecast type*. They are used prior to initiating data compilation, and check that user-supplied datetime inputs are valid for the relevant *forecast type*.

nemseer.forecast_type.validators.**validate_MTPASA_datetime_inputs**(*run_start: datetime*, *run_end: datetime*, *forecasted_start: datetime*, *forecasted_end: datetime*) → None

From AEMO PASA Outputs:

> [MT PASA] is produced weekly (on Tuesdays) and lists the medium-term supply/demand prospects for the period two years in advance. The information is provided for each day within the report period.

Noting that:

- *MT PASA* is actually run at half-hourly resolution - But results are aggregated and reported for each day

- Timing of "RUN_DATETIME" appears to be inconsistent on inspection - No validation on `run_start` and `run_end` - Compiler will instead collect all forecasts between provided forecast datetimes

Validation checks:

Check 1:
forecasted_start and *forecasted_end* are at 00:00 for each supplied date. This is because results are reported for a day.

Check 2:
forecasted_end is within 2 years and 16 days of *run_end*. A 16 day offset appears to be in older data.Newer data appears to have a 6 day offset.

Todo: Handle MTPASA DUID Availability

Parameters

- **run_start** (*datetime*) – Forecast runs at or after this datetime are queried.
- **run_end** (*datetime*) – Forecast runs before or at this datetime are queried.
- **forecasted_start** (*datetime*) – Forecasts pertaining to times at or after this datetime are retained.
- **forecasted_end** (*datetime*) – Forecasts pertaining to times before or at this datetime are retaned.

Raises
ValueError – If any validation conditions are failed.

Return type
None

nemseer.forecast_type.validators.**validate_P5MIN_datetime_inputs**(*run_start: datetime*, *run_end: datetime*, *forecasted_start: datetime*, *forecasted_end: datetime*) → None

Validates *P5MIN* forecast datetime inputs

From AEMO MMS Data Model Report:

> The 5-minute Predispatch cycle runs every 5-minutes to produce a dispatch and pricing schedule to a 5-minute resolution covering the next hour, a total of twelve periods.

Validation checks:

Check 1:
Minute component of datetime inputs is on a 5 minute basis

Check 2:
`forecasted_end` is not more than 55 minutes (12 cycles) from `run_end`

These 12 dispatch cycles include the immediate interval (i.e. where *RUN_DATETIME* = *INTERVAL_DATETIME*)

Parameters

- **run_start** (*datetime*) – Forecast runs at or after this datetime are queried.
- **run_end** (*datetime*) – Forecast runs before or at this datetime are queried.

- **forecasted_start** (*datetime*) – Forecasts pertaining to times at or after this datetime are retained.
- **forecasted_end** (*datetime*) – Forecasts pertaining to times before or at this datetime are retaned.

Raises
ValueError – If any validation conditions are failed.

Return type
None

nemseer.forecast_type.validators.**validate_PDPASA_datetime_inputs**(*run_start: datetime*, *run_end: datetime*, *forecasted_start: datetime*, *forecasted_end: datetime*) → None

Validates *PDPASA* forecast datetime inputs

Points to *validate_PREDISPATCH_datetime_inputs()* as validation for PREDISPATCH and PDPASA are the same.

Parameters

- **run_start** (*datetime*) – Forecast runs at or after this datetime are queried.
- **run_end** (*datetime*) – Forecast runs before or at this datetime are queried.
- **forecasted_start** (*datetime*) – Forecasts pertaining to times at or after this datetime are retained.
- **forecasted_end** (*datetime*) – Forecasts pertaining to times before or at this datetime are retaned.

Raises
ValueError – If any validation conditions are failed.

Return type
None

nemseer.forecast_type.validators.**validate_PREDISPATCH_datetime_inputs**(*run_start: datetime*, *run_end: datetime*, *forecasted_start: datetime*, *forecasted_end: datetime*) → None

Validates *PREDISPATCH* forecast datetime inputs

From AEMO Pre-dispatch SOP:

> Currently AEMO runs pre-dispatch every half hour, on the half hour for each 30-minute period up to and including the last 30-minute period of the last trading day for which bid band prices have closed. As changes to bid band prices for the next trading day close at 1230 hours EST, AEMO will at 1230 hours, publish pre-dispatch for all 30-minute periods up to the end of the next trading day.

Noting that:

- A market/trading day extends from 0400 to 0400 on the next day.
- Pre-dispatch executed at 1230 hours is associated with the 1300 hours run time. That is, *PREDISPATCHSEQ* corresponding to 13:00 contains bids for the next trading day.

Validation checks:

Check 1:
Minute component of datetime inputs is on a 30 minute basis

Check 2:
`forecasted_end` is no later than the end of the last trading day for which bid band prices have closed (the end of that day being 04:00) by `run_end`

Parameters

- **run_start** (*datetime*) – Forecast runs at or after this datetime are queried.
- **run_end** (*datetime*) – Forecast runs before or at this datetime are queried.
- **forecasted_start** (*datetime*) – Forecasts pertaining to times at or after this datetime are retained.
- **forecasted_end** (*datetime*) – Forecasts pertaining to times before or at this datetime are retaned.

Raises
ValueError – If any validation conditions are failed.

Return type
None

nemseer.forecast_type.validators.**validate_STPASA_datetime_inputs**(*run_start: datetime*, *run_end: datetime*, *forecasted_start: datetime*, *forecasted_end: datetime*) → None

Validates *STPASA* forecast datetime inputs

From AEMO PASA Outputs:

> [ST PASA] is published every 2 hours and provides detailed disclosure of short-term is published every 2 hours and provides detailed disclosure of short-term power-system supply/demand balance prospects for six days following the next trading day. The information is provided for each half-hour within the report period

Noting that:

- A market/trading day extends from 0400 to 0400 on the next day.
- *ST PASA* is the "reverse" of *PREDISPATCH - ST PASA* **starts** after the end of the next trading day for which bids have been submitted

The National Electricity Rules and some of AEMO's procedures state that ST PASA is run every two hours. The frequency was increased to hourly. See Spot Market Operations Timetable.

Validation checks:

Check 1:
Minute component of forecast datetimes is on an hourly basis (i.e. 0 minutes)

Check 2:
Minute component of forecasted datetimes is on a 30 minute basis

Check 3:
`forecasted_start` is not equal to or earlier than the end of the last trading day for which bid band prices have closed (the end of that day being 04:00) by `run_start`

Check 4:
`forecasted_end` is no later than 6 days from the end of the last trading day for which bid band prices have closed by `run_end`

Parameters

- **run_start** (*datetime*) – Forecast runs at or after this datetime are queried.
- **run_end** (*datetime*) – Forecast runs before or at this datetime are queried.
- **forecasted_start** (*datetime*) – Forecasts pertaining to times at or after this datetime are retained.
- **forecasted_end** (*datetime*) – Forecasts pertaining to times before or at this datetime are retaned.

Raises
ValueError – If any validation conditions are failed.

Return type
None

Run time generators

Run time generators produce the widest valid *run time* range for a particular *forecast type* given *forecasted_start* and *forecasted_end*.

nemseer.forecast_type.run_time_generators.**_generate_MTPASA_runtimes**(*forecasted_start: datetime*, *forecasted_end: datetime*) → Tuple[datetime, datetime]

Generates the earliest *run_start* and latest *run_end* for a set of user-supplied *forecasted_start* and *forecasted_end* times.

Calls validation function to ensure that user-supplied *forecasted* times are valid.

Parameters

- **forecasted_start** (*datetime*) – Forecasts pertaining to times at or after this datetime are retained.
- **forecasted_end** (*datetime*) – Forecasts pertaining to times before or at this datetime are retaned.

Returns
Tuple of datetimes containing the widest range of possible *forecasted* times

Return type
Tuple[*datetime*, *datetime*]

nemseer.forecast_type.run_time_generators.**_generate_P5MIN_runtimes**(*forecasted_start: datetime*, *forecasted_end: datetime*) → Tuple[datetime, datetime]

Generates the earliest *run_start* and latest *run_end* for a set of user-supplied *forecasted_start* and *forecasted_end* times.

Calls validation function to ensure that user-supplied *forecasted* times are valid.

Parameters

- **forecasted_start** (*datetime*) – Forecasts pertaining to times at or after this datetime are retained.
- **forecasted_end** (*datetime*) – Forecasts pertaining to times before or at this datetime are retaned.

Returns
Tuple of datetimes containing the widest range of possible *forecasted* times

Return type
Tuple[*datetime*, *datetime*]

nemseer.forecast_type.run_time_generators.**_generate_PDPASA_runtimes**(*forecasted_start: datetime*, *forecasted_end: datetime*) → Tuple[datetime, datetime]

Generates the earliest *run_start* and latest *run_end* for a set of user-supplied *forecasted_start* and *forecasted_end* times.

Calls validation function to ensure that user-supplied *forecasted* times are valid.

Parameters

- **forecasted_start** (*datetime*) – Forecasts pertaining to times at or after this datetime are retained.
- **forecasted_end** (*datetime*) – Forecasts pertaining to times before or at this datetime are retaned.

Returns
Tuple of datetimes containing the widest range of possible *forecasted* times

Return type
Tuple[*datetime*, *datetime*]

nemseer.forecast_type.run_time_generators.**_generate_PREDISPATCH_runtimes**(*forecasted_start: datetime*, *forecasted_end: datetime*) → Tuple[datetime, datetime]

Generates the earliest *run_start* and latest *run_end* for a set of user-supplied *forecasted_start* and *forecasted_end* times.

Calls validation function to ensure that user-supplied *forecasted* times are valid.

Parameters

- **forecasted_start** (*datetime*) – Forecasts pertaining to times at or after this datetime are retained.
- **forecasted_end** (*datetime*) – Forecasts pertaining to times before or at this datetime are retaned.

Returns
Tuple of datetimes containing the widest range of possible *forecasted* times

Return type
Tuple[*datetime*, *datetime*]

nemseer.forecast_type.run_time_generators.**_generate_STPASA_runtimes**(*forecasted_start: datetime*, *forecasted_end: datetime*) → Tuple[datetime, datetime]

Generates the earliest *run_start* and latest *run_end* for a set of user-supplied *forecasted_start* and *forecasted_end* times.

Calls validation function to ensure that user-supplied *forecasted* times are valid.

Parameters

- **forecasted_start** (*datetime*) – Forecasts pertaining to times at or after this datetime are retained.
- **forecasted_end** (*datetime*) – Forecasts pertaining to times before or at this datetime are retaned.

Returns
Tuple of datetimes containing the widest range of possible *forecasted* times

Return type
Tuple[*datetime*, *datetime*]

9.5.4 Data

nemseer.data.**DATETIME_FORMAT = '%Y/%m/%d %H:%M'**

nemseer date format

nemseer.data.**DEPRECATED_TABLES = {'MTPASA': ['CASESOLUTION']}**

Deprecated tables

nemseer.data.**ENUMERATED_TABLES = {'P5MIN': [('CONSTRAINTSOLUTION', 4)], 'PREDISPATCH': [('CONSTRAINT', 2), ('LOAD', 2)]}**

Enumerated tables for each forecast type First element of tuple is table name Second element of tuple is number which to enumerate table to

nemseer.data.**FORECASTED_COL = {'MTPASA': 'DAY', 'P5MIN': 'INTERVAL_DATETIME', 'PDPASA': 'INTERVAL_DATETIME', 'PREDISPATCH': 'DATETIME', 'STPASA': 'INTERVAL_DATETIME'}**

If it exists, *nemseer* uses the corresponding column for *forecasted* time filtering.

nemseer.data.**FORECAST_TYPES = ('P5MIN', 'PREDISPATCH', 'PDPASA', 'STPASA', 'MTPASA')**

Forecast types requestable through nemseer. See also *forecast types*, and *pre-dispatch* and *PASA*.

nemseer.data.**INVALID_STUBS_FILE = '.invalid_aemo_files.txt'**

File in *raw_cache* that contains invalid/corrupted AEMO files

nemseer.data.**MMSDM_ARCHIVE_URL = 'http://www.nemweb.com.au/Data_Archive/Wholesale_Electricity/MMSDM/'**

Wholesale electricity data archive base URL

nemseer.data.**MTPASA_DUID_URL = 'http://nemweb.com.au/Reports/Current/MTPASA_DUIDAvailability/'**

MTPASA DUID Availability

nemseer.data.**PREDISP_ALL_DATA = ('CONSTRAINT', 'INTERCONNECTORRES', 'PRICE', 'LOAD', 'REGIONSUM')**

Tables which should be directed to the PREDISP_ALL_DATA URL. The corresponding tables in the DATA folder (which end with "_D") only contain the latest forecasted value

nemseer.data.**RUNTIME_COL = {'MTPASA': 'RUN_DATETIME', 'P5MIN': 'RUN_DATETIME', 'PDPASA': 'RUN_DATETIME', 'PREDISPATCH': 'PREDISPATCH_RUN_DATETIME', 'STPASA': 'RUN_DATETIME'}**

If it exists, *nemseer* will use the corresponding column for *run* time filtering.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

ELEVEN

DEVELOPMENT

- Guide for contributing
- Our code of conduct
- List of to-dos
- Changelog

PYTHON MODULE INDEX

n

INDEX

Symbols

A

C

D

E

F

G

I

M

N

P

Q

R

S

T

V

W