User Manual for CoVEMDA Toolbox

Python Version 1.0 (Beta)

December 2021

Contents

Website

https://github.com/tamu-engineering-research/COVID-EMDA

The above website refers to a Github repository that releases the COVID-EMDA+ data hub and CoV-EMDA toolbox. Here, COVID-EMDA+ is the abbreviation for "Coronavirus Disease – Electricity Market Data Aggregation+", and CoVEMDA for "CoronaVirus – Electricity Market Data Analyzer".

Note that the Github repository provides a variety of useful resources, including the data release, the source codes, a few powerful parsers or supplementary resources.

1 Introduction

1.1 Background

The COVID-19 pandemic is an emerging respiratory disease with high contagiousness and long incubation period [1]. Over a year has passed, it still keeps tremendous pressure on the public health systems as well as the global economy. A few early publications have also found discernible changes in the operation of power system during COVID-19 [2–4]. This is the initial motivation for an international research team from Tsinghua University and Texas A&M University to develop a powerful data hub and toolbox to track the pandemic's impacts on the power sector.

One unique feature of our work is the public and free availability. This opens up a series of opportunities for diverse stakeholders to share opinions and learn through collaboration. Another unique feature is the cross-domain resources (data, methods, and toolbox) which could provide new insights into policy decisions [5]. In some aspect, the pandemic has created a special but informative situation to assess how power systems may respond to different human interventions [6, 7]. This is certainly a positive step forward for understanding the power system resilience.

Here, we establish a data hub (COVID-EMDA+) and a supporting toolbox (CoVEMDA) on our Github repository. A variety of practical functions are tailored in the toolbox to handle the applications of baseline estimation, regression analysis, and scientific visualization. Since the world is confronting a uncertain future induced by COVID-19, this work may hopefully advance our understanding of the ongoing situations, and guide the preparations through these difficult times.

1.2 Features

We summarize the main features of our work as follows:

- Data Resources: CoVEMDA originally supports reading data, online or offline (through data archive), from the COVID-EMDA+ data hub. The released data involve electricity market data, public health data, meteorological data, and mobility data for all the existing U.S. electricity marketplaces and some typical cities[1] in these markets. Strict quality control is conducted, and external data sources are also acceptable[2].

- Baseline Estimation: Baselines are critical in quantifying the impacts of COVID-19 by estimating the counterfactual situations (We assume all influencing factors except the pandemic are happening). A few classical and representative estimation methods are collected in CoVEMDA. Rigorous comparison between baselines are also allowed.

- Regression Analysis: Regression is widely used to estimate the effect of some explanatory variables on the dependent variable. This technique helps to explore and investigate the underlying impacts of COVID-19, and two popular regression techniques—ordinary least squares regression (OLS) and vector autoregression (VAR)—are developed in CoVEMDA with flexible extensions.

- Scientific Visualization: Visualization conveys intuitive messages to readers, and becomes extremely important in assessing the pandemic-induced impacts. A variety of visualization designs are implemented in CoVEMDA for different applications, including graphical and statistical outputs. Simplified but extensible syntax is applied to make it handy even for beginners.

[1] Specifically, Los Angeles, Boston, New York City, Philadelphia, Chicago, Kansas City, and Houston.
[2] This feature might be helpful for user-defined extensions.

1.3 Support Team

This project is a joint effort of the group members under the supervision of Prof. Le Xie at Texas A&M University and Prof. Haiwang Zhong at Tsinghua University. The support team keeps processing, correcting, and updating the data routinely.

Figure 1-1: Members in the Support Team

1.4 Suggested Citation

Publications that use the data resources, support toolbox, or supplementary materials should consider citing the following paper accordingly. A short introduction of each paper is given as follows:

- G. Ruan, D. Wu, X. Zheng, H. Zhong, C. Kang, M. A. Dahleh, S. Sivaranjani, and L. Xie, "A cross-domain approach to analyzing the short-run impact of COVID-19 on the U.S. electricity sector," *Joule*, 2020, 4(11), 2322-2337.

  This paper gives a general introduction of the data hub, and conducts a few cross-domain analysis on the eletricity consumption across the U.S. Available at: Joule and arXiv.

- G. Ruan, J. Wu, H. Zhong, Q. Xia, and L. Xie, "Quantitative assessment of U.S. bulk power systems and market operations during the COVID-19 pandemic," *Applied Energy*, 2021, 286: 116354.

  This paper substantiates the pandemic's impacts from the perspectives of power system security, electric power generation, electric power demand and electricity prices. Available at: Applied Energy and arXiv.

- H. Zhong, Z. Tan, Y. He, L. Xie, and C. Kang, "Implications of COVID-19 for the electricity industry: A comprehensive review," *CSEE Journal of Power and Energy Systems*, 2020, 6(3): 489-495.

  This paper provides an early review of the global impacts that COVID-19 has caused on the electricity industry. Available at: JPES.

- G. Ruan, Z. Yu, S. Pu, S. Zhou, H. Zhong, L. Xie, Q. Xia, and C. Kang, "Open-access data and toolbox for tracking COVID-19 impact on power systems," *arXiv*, 2021. (preprint, under review)

  This paper comprehensively introduces the upgrade of our data hub and toolbox. All technical details and extension settings are provided, and three empirical studies are thoroughly discussed as well. Available at: arXiv and TechRxiv.

1.5 License

The source codes of CoVEMDA are following the **MIT License**, and the full text can be found here. We also list it as follows for a quick view:

2 Getting Started

2.1 System Requirements

The dependencies of CoVEMDA toolbox include:

- `Python >= 3.6.0` (Compile System)
- `Pandas >= 1.2.4` (Data Processing)
- `SciPy >= 1.6.2` (Data Processing)
- `Statsmodels >= 0.12.2` (Regression)
- `Scikit-Learn >= 0.24.1` (Machine Learning)
- `Matplotlib >= 3.3.4` (Visualization)

Note that CoVEMDA toolbox is originally developed in a Windows 10 operating system, and it is unclear whether there might be any bugs in Unix-type systems.

2.2 Data and Toolbox Retrieval

COVID-EMDA+ data hub and CoVEMDA toolbox are both available on our Github repository. A binary distribution of CoVEMDA toolbox is also provided on PyPI. Users may select one of the following ways to retrieve the data and toolbox:

2.2.1 Download ZIP

Step 1: Visit the website of Github repository.
Step 2: Click the green `Code` button.
Step 3: Click the `Download ZIP` button.
Step 4: Uncompress the ZIP file to a certain directory.

2.2.2 Clone Github Repository

When the command-line tool git is available, make a clone by:

```
git clone https://github.com/tamu-engineering-research/COVID-EMDA.git
```

When the Github Desktop software is installed:
Step 1: Visit the website of Github repository.
Step 2: Click the green `Code` button.
Step 3: Click the `Open With Github Desktop` button.

2.2.3 Download from PyPI

Step 1: Visit the website of PyPI project.
Step 2: Click the `Download files` link on the left navigation bar.
Step 3: Download the wheel file (.whl) by clicking the file name link.
Note that this procedure does not include the data, but it is possible to make an automatic download with CoVEMDA toolbox later. Also, there is another way to install the toolbox without manual downloads.

2.3 Installation

Only the toolbox needs installation, and there are basically two modes to select from: a common mode, and a developer mode. As for the data, one can use the toolbox for online data entry (always updated but sometimes slow), automatic data download to local folder (slow for the first time), or manually download from the Github website (mentioned before). Below are the details of two installation modes:

2.3.1 Common Mode

Use `pip` package to install CoVEMDA by either of the following commands:

```
pip install covemda
pip install covemda-1.0-py3-none-any.whl
```

The first command will download the toolbox from PyPI, so the internet access is required. And the second command is installing the local wheel file that can be downloaded from PyPI manually.

You can check if the installation is successful with:

```
pip show covemda
```

The toolbox installation is complete if the above command gives the information of CoVEMDA package rather than a warning. Simply reinstall if something goes wrong. Also, it is convenient to uninstall the toolbox by calling:

```
pip uninstall covemda
```

2.3.2 Developer Mode

Another option is the developer mode, and this is much more flexible and powerful because any code modifications are completely allowed.

Step 1: Download the toolbox source codes from `toolbox/` folder on Github repository.

Step 2: Copy the source code folder `lib/covemda/` to the working directory.

Step 3: Run the `install.py` script to check the dependencies.

After above steps, users will be able to import the CoVEMDA toolbox as a user-defined package. Note that a copy will be preferred to avoid unexpected modifications of the source codes. As for the uninstallation, simply remove the relevant folders.

2.4 Quick Start Examples

2.4.1 Demand Baseline Estimation

CoVEMDA provides several estimators to calculate, analyze, and visualize demand baselines which are defined as the electricity consumption level without the impact of COVID-19.

Below is a quick start example demonstrating how to calculate the demand baselines in NYISO. Note that we only consider the default settings for simplification, and in fact, much more customized options are allowed. See Section 5 for more technical details in this topic.

At the very beginning, one should first import the toolbox and create a region object:

Console

```
>>> from covemda.integration import RTO
>>> nyiso = RTO('nyiso')
```

After that, use the following command to finalize the target calculation:

Console

```
>>> print(nyiso.cal_demand_baseline())

               00:00    01:00    02:00  ...    21:00    22:00    23:00
2020-01-01  15052.8  14481.0  13927.1  ...  17387.5  16399.2  15459.7
2020-01-02  14780.5  14410.4  14206.5  ...  18942.1  17616.3  16315.6
2020-01-03  15368.6  14858.0  14574.8  ...  18834.9  17619.5  16375.7
2020-01-04  15496.5  14960.4  14684.4  ...  18183.5  17165.8  16064.1
2020-01-05  15127.0  14518.8  14201.3  ...  17531.7  16720.0  15844.0
...             ...      ...      ...  ...      ...      ...      ...
2020-06-26  17737.0  16763.2  16172.0  ...  23399.9  21947.7  20170.2
2020-06-27  18625.2  17489.9  16780.0  ...  23928.0  22338.4  20479.4
2020-06-28  18898.6  17759.7  16914.2  ...  23653.6  22289.3  20694.8
2020-06-29  19371.8  18362.8  17613.6  ...  22289.7  21167.8  19883.9
2020-06-30  18661.4  17643.0  16924.3  ...  19816.1  18817.0  17458.6

[181 rows x 24 columns]
```

It is convenient to visualize the baseline series by the following single-line command. Here, both the baselines and observations are collected together for further analysis.

Console

```
>>> nyiso.plot_demand_baseline()
```

The result will be displayed in a pop-up window with the Figure 2-1. One may clearly find that a discernible difference can be found between the baseline and observation curves during April–May 2020, indicating a period under severe impacts.

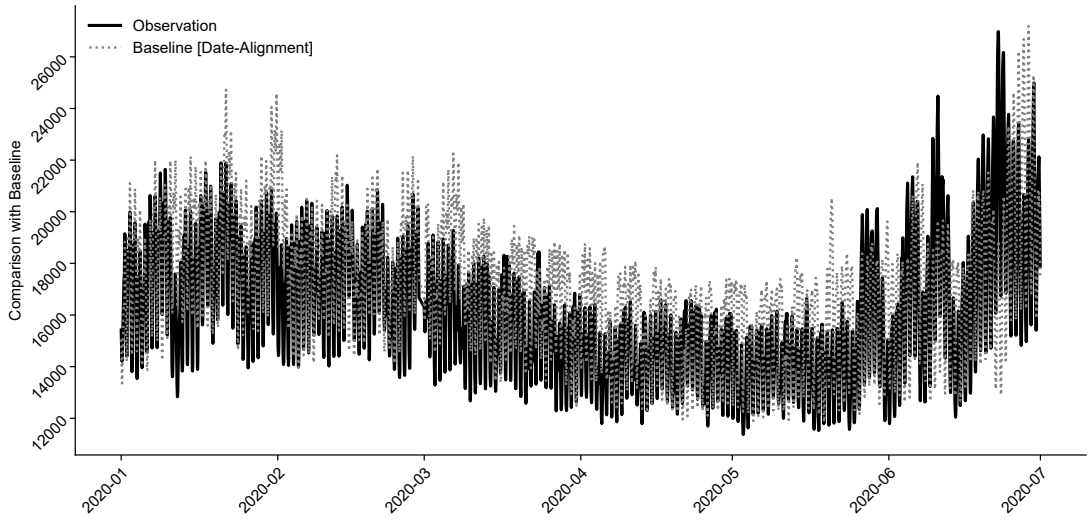

Figure 2-1: Observed and Baseline Demand in NYISO. By default, this estimation is implemented for the first half of 2020 with a baseline rule of date alignment.

2.4.2 Regression Analysis for Price

CoVEMDA also supports regression analysis such as linear regression. Such kind of analysis helps to explore some underlying correlations between different variables, and it often has flexible expressions to

effectively cover diverse use cases. See Section 6 for more technical details.

Below is a simple example to check the temporal dependencies between locational marginal prices in CAISO. Typically, we will check the correlation between prices during 00:00 – 02:00.

Console

```
>>> from covemda.integration import RTO
>>> caiso = RTO('caiso')
>>> caiso.run_general_ols(
        x=caiso.dfmt_price.select_by_column(('00:00', '01:00')),
        y=caiso.dfmt_price.select_by_column('02:00'),
    )
...
==============================================================================
            coef     std err       t       P>|t|       [0.025       0.975]
------------------------------------------------------------------------------
const    -0.4673      0.116   -4.028       0.000       -0.695       -0.240
00:00    -0.1648      0.022   -7.364       0.000       -0.209       -0.121
01:00     1.1465      0.023   48.935       0.000        1.100        1.192
==============================================================================
...

Testing OrdinaryLeastSquares:
RMSE = 1.4894, MAPE = 3.19 %
```

Please note that some outputs are folded above to keep clarification. Here the main results are the regression table with t test statistics, and this model is also simulated to check its regression accuracy. To be specific, one can check the p values for every regression coefficient, e.g., the price at 00:00. It is clear that this value is less than 0.001, suggesting a strong and robust correlation is found — this is not surprising as prices of adjacent periods often look similar.

3 Data Hub

CoVEMDA toolbox is designed to utilize the data sourced from COVID-EMDA+ to track the potential impacts of COVID-19 on the existing U.S. electricity markets. Many different data sources are merged and harmonized here in order to enable further interdisciplinary researches. Historical data dating back to 2017 are included as time-series benchmarks.

3.1 Data Category

Five major kinds of data are included in the data hub. For some categories, multiple data sources are carefully gathered to ensure accuracy through cross-checking or backup.

- Electricity market data. This part includes the generation mix, metered load profiles, day-ahead locational marginal prices data, as well as day-ahead load forecasting, congestion price, forced outage and renewable curtailment data.

  Source: CAISO, MISO, ISO-NE, NYISO, PJM, SPP, ERCOT, EIA, EnergyOnline.

- Public health data. This part includes the COVID-19 confirmed cases, deaths data, infection rate and fatal rate.

  Source: John Hopkins CSSE.

- Meteorological data. This part includes temperature, relative humidity, wind speed and dew point data. Typical weather stations are selected according to their geological locations and data quality.

  Source: Iowa State Univ IEM, NOAA.

- Mobile device location data. This part includes social-distancing data and patterns of visits to Point of Interests (POIs). These data are derived by aggregating and processing the real-time GPS location of cellphone users by Census Block Group.

  Source: Mobility Data From SafeGraph.

- Satellite images. This refers to the Night Time Light (NTL) Satellite data that include the raw satellite images taken at night time in each area. Due to unpredictive cloudy weather, the clear images are limited and only used for visualization.

  Source: NTL Images from NASA.

3.2 Folder Structure

Our data hub mainly contains five folders: source data, released data, supplementary resources, parser codes, and quick start tutorials. The following is a quick navigation:

All data source files are archived in folder `data_source/`, while the cleaned, processed data are stored in folder `data_release/`. Latest updates of the released data are properly classified by location. The file name convention is: `<market>_<area>_<category>.csv`, e.g., `nyiso_nyc_load.csv` is a dataset of the load profiles in New York City from 2017 to present.

Supplementary resources of several publication works and relevant projects including a series of extended data sources and analysis codes could be found in folder `supplementary/`.

Basic parser codes (written in Python) to handle the source data are implemented in folder `parser/`.

Some quick start examples for the data hub are also available in folder `startup/` and `supplementary/`.

3.3 Field Description

As mentioned before, `data_release/` folder contains all the cleaned and processed data. The file name follows the convention of `<market>_<area>_<category>.csv`. All the files are organized in a wide csv table. Here are some tips to further demonstrate the data.

3.3.1 Electricity Market Data

Field *date*: list all the date from 2017 to present. Format: *YYYYmmdd*.

Field *fuel*: only in generation mix dataset, represent the different fuel source used by generators. Possible values: *coal, gas, oil, nuclear, hydro, wind, solar, export, other.* Different definitions in different electricity markets are harmonized.

Field *kind*: only in meteorological dataset, represent the type of measurement that is recorded in each corresponding row. Possible value: *dwpc (dew point temperature), relh (relative humidity), sped (wind speed), tmpc (temperature).*

Field *HH:MM*: represent the hourly time slot. For example, column "08:00" records the data of period 8AM to 9AM.

3.3.2 Public Health Data

Field *date*: list all the date from Jan. 23, 2020 to present. Format: *YYYYmmdd*

Field *accum_confirm*: accmulated confirmed case number recorded for each date.

Field *new_confirm*: newly confirmed case number for each date.

Field *infect_rate*: infection ratio calculated for each date.

Field *accum_death*: accmulated confirmed death number recorded for each date.

Field *new_death*: newly confirmed death number for each date.

Field *fatal_rate*: fatal rate calculated for each date.

3.3.3 Visit Patterns to Point of Interests

Field *date*: list all the date from Dec.30, 2019 to present. Format: *YYYYmmdd*

Field *Restaurant_Recreaction*: daily total number of visits to Restaurant and Recreaction places.

Field *Grocery_Pharmacy*: daily total number of visits to Grocery and Pharmacy places.

Field *Retail*: daily total number of visits to Retail places.

3.3.4 Social Distancing Data

Field *date*: list all the date from Jan. 1, 2020 to present. Format: *YYYYmmdd*

Field *completely_home_device_count_percentage*: percentage of devices that stay at home 24 hours out of all devices.

Field *median_home_dwell_time_percentage*: median proportion of home dwell time in one day.

Field *part_time_work_behavior_devices_percentage*: percentage of devices that go to workplaces for 3–6 hours out of all devices.

Field *full_time_work_behavior_devices_percentage*: percentage of devices that go to workplaces for more than 6 hours out of all devices.

Field *completely_home_device_count*: count of devices that stay at home 24 hours.

Field *device_count*: total count of devices.

Field *part_time_work_behavior_devices_count*: count of devices that go to workplaces for 3–6 hours.

Field *full_time_work_behavior_devices_count*: count of devices that go to workplaces for more than 6 hours.

3.4 Quality Control

To construct a reliable and easy-to-use dataset, a series of data checking and cleaning procedures are implemented. Outlier detection and missing data recovery are two key points for the data quality control, and in most cases, outliers or missing data can be easily handled by considering the backup data or historical trend, and the whole procedure is shown in Figure 3-1.

Single missing data (most frequent) are filled by linear interpolation. For consecutive missing data (rare), data from the EIA or EnergyOnline are carefully supplemented.

Outlier data samples are automatically detected when they are beyond 5 times or below 20% of the associated daily average value. Exceptions such as price spikes and negative prices in price data are carefully handled to avoid unexpected distortions.

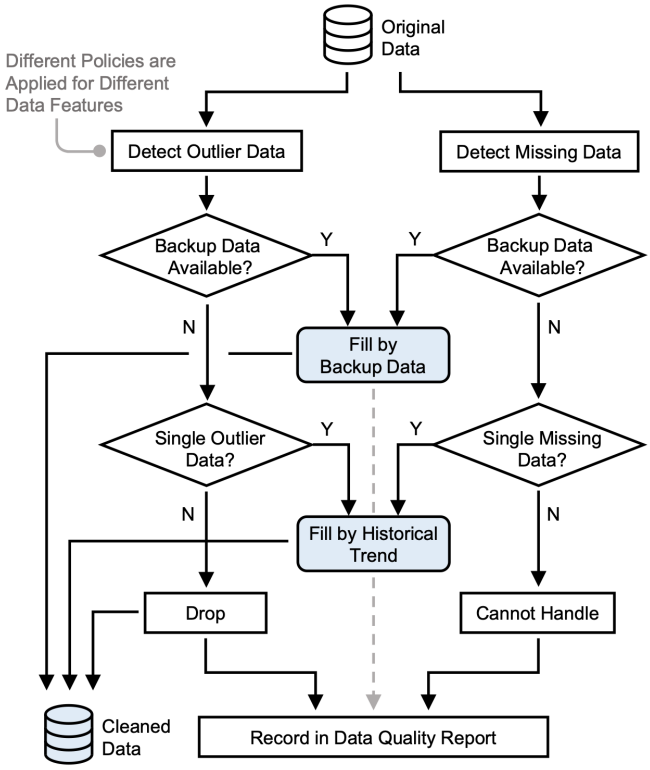

Figure 3-1: Procedure of Data Quality Control.

Duplicate data are dropped, only the first occurrence of each data sample is kept.

However, there are some cases that cannot be properly handled, e.g., no observations for an entire day due to device failures. We thus record these situations in a quality report on Github. Users are recommended to check the report to get full knowledge of the released data before any usage of the data hub. Table 3-1 shows a snapshot of this report on June 6, 2021.

Table 3-1: Online Quality Report (Snapshot on June 6, 2021).

Dataset Name	Market	Area	Category	Missing Dates
caiso_rto_genmix.csv	CAISO	RTO Level	Generation Mix	Sep 23 - Oct 3, 2019
caiso_rto_lmp.csv	CAISO	RTO Level	Day-Ahead LMP	Jan 1 - Jan 18, 2017; Oct 5, Oct 9, 2020
caiso_la_load.csv	Near CAISO	Los Angeles	Hourly Load	Nov 5, 2017; Jan 11 - 12, Jun 29 - 30, Jul 1 - 9, Sep 27, Nov 4, 2018; Mar 10, Nov 3, 2019; Mar 8, Nov 1, 2020; Mar 14, 2021
caiso_la_weather	CAISO	Los Angeles	Weather	Apr 7 - 8, 2021
ercot_rto_load.csv	ERCOT	RTO Level	Hourly Load	Dec 13, 2020
ercot_houston_load.csv	ERCOT	Houston City	Hourly Load	Dec 13, 2020
ercot_houston_weather.csv	ERCOT	Houston City	Weather	Feb 15 - 16, 2020
isone_rto_genmix.csv	ISO-NE	RTO Level	Generation Mix	Apr 30, 2018; Jun 21, Oct 30-31, Dec 06, 2020
isone_rto_load.csv	ISO-NE	RTO Level	Hourly Load	Dec 17, 2020
pjm_rto_genmix.csv	PJM	RTO Level	Generation Mix	Mar 29 - Apr 2, 2017
spp_rto_genmix.csv	SPP	RTO Level	Generation Mix	Mar 29, 2019
spp_kck_weather.csv	SPP	Kansas City	Weather	Sep 22 - 23

3.5 Data Structure

Almost all data records in the COVID-EMDA+ data hub can be expressed by X_{ymdt}. Here, X is a placeholder for some variable, and this value is taken at year y, month m, day d, and hour t. One can easily

use X_{ymd} or X_{ym} to represent a monthly or daily average respectively.

It has already been shown in Subsection 3.3 that these data are stored in a standardized form: they all have a row index of dates and a column index of hours. Such a data structure is specifically called a wide data frame, and we use it to standardize the inconsistent data structures from various sources.

Figure 3-2 shows the proposed data structure with details. Here, a wide data frame refers to a kind of unstacked table that has more columns than a long frame. This structure enables a more compact way to store data, and both the row-wise and column-wise operations have clear physical meanings. Besides, a variety of basic operations (e.g., filtering, resampling, and statistical computing) have been developed in Pandas package to handle such a matrix-like structure. Figure 3-2 also demonstrates how to finalize the released data after several preprocessing steps.

In fact, all the preprocessing steps have been automated by our backend system which consists of a few web crawlers, a set of automation and management modules, the workflow controller and quality controller, and a logging module. This backend system is scheduled to run periodically, and for each run, 31 raw data files from 25 sources will be extracted and cleaned to update 73 spreadsheets.

The CoVEMDA toolbox establishes a new DataFormatter class to realize the wide data frame structure mentioned above. This class wraps the popular DataFrame class from Pandas package, and extends the built-in function family with a lot of specialized features. See Appendix A for more technical details.

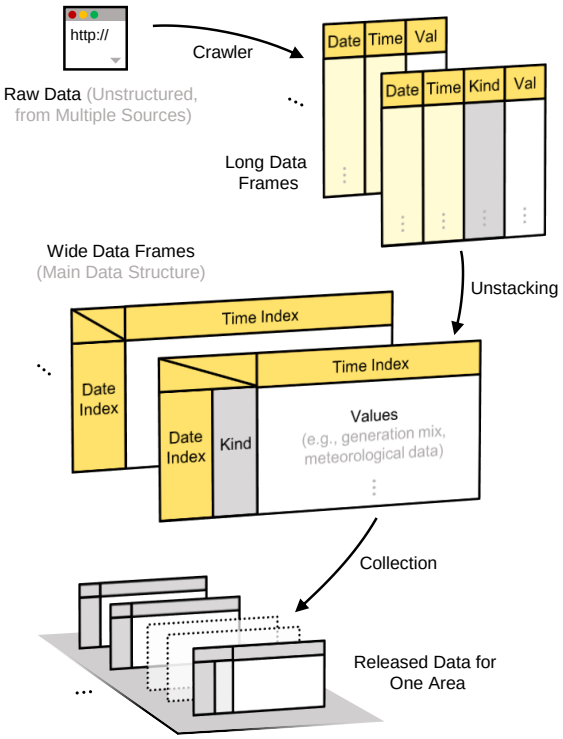

Figure 3-2: Demonstration of the wide data frames and preprocessing steps. This procedure is already automated and executed by a backend system.

4 Toolbox

CoVEMDA is an open-sourced toolbox for tracking COVID-19 impacts on power systems. It consists of a variety of specialized methods, models, and utilities to make it possible for even non-expert users to get data-driven insights by themselves. This section will provide a high-level summary of the architecture, workflow, and commonly-used interfaces of CoVEMDA. Users are suggested to carefully go through this section before diving into the subsequent technical sections.

4.1 Architecture

We first focus on the programming framework to organize the whole toolbox. Basically, we apply a three-layer structure to assign the functions from data side to user side—basic operations, low-level, and high-level. This architecture is visually depicted in Figure 4-1.

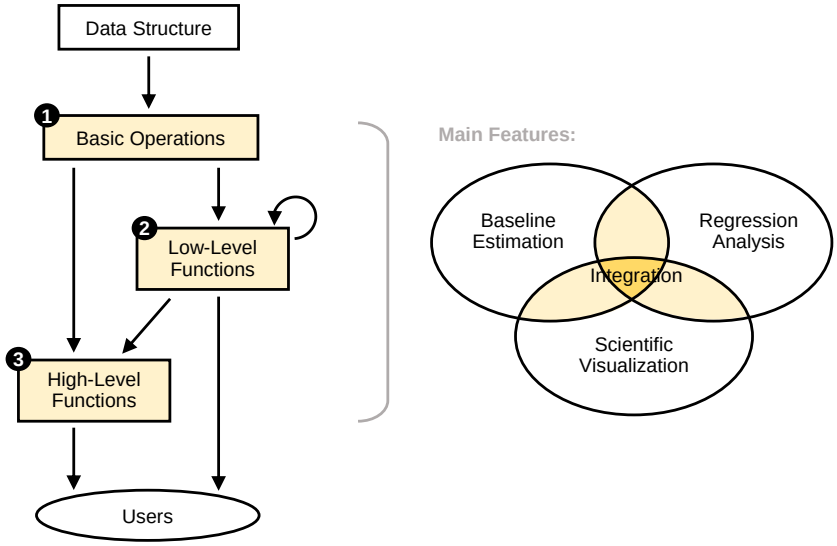

Figure 4-1: Programming architecture of CoVEMDA. This architecture is devided into three layers according to different levels of abstraction. Within this architecture, the main features are implemented accordingly with efficient integration designs.

An obvious advantage of such an architecture in Figure 4-1 is that it could help break down large tasks into small activities, and that makes the calling relationships and dependencies between different functions relatively clear to follow.

Within this architecture, high-level functions should be more user-friendly, and only require limited knowledge of the underlying data structure. On the opposite, basic operations are strongly connected with the raw data, and low-level functions are also strongly associated with data features. Although more complicated, the latter two perform better in the perspectives of flexibility and efficiency.

There are three major modules (or features) in CoVEMDA: baseline estimation, regression analysis, and scientific visualization. As shown in Figure 4-1, they are mutually related, and some widely-used cases are collected in another integration module. When organizing the toolbox scripts, each module will have the functional interfaces of all levels.

Discussing the overall architecture before diving into implementation details is critical and useful. The subsequent parts will introduce more about the details to fulfill this architecture.

4.2 Folder Structure

We will turn to introduce the folder structure of the whole toolbox. Figure 4-2 shows a concise folder structure (on the left) with an expanded view of the source code folder (on the right).

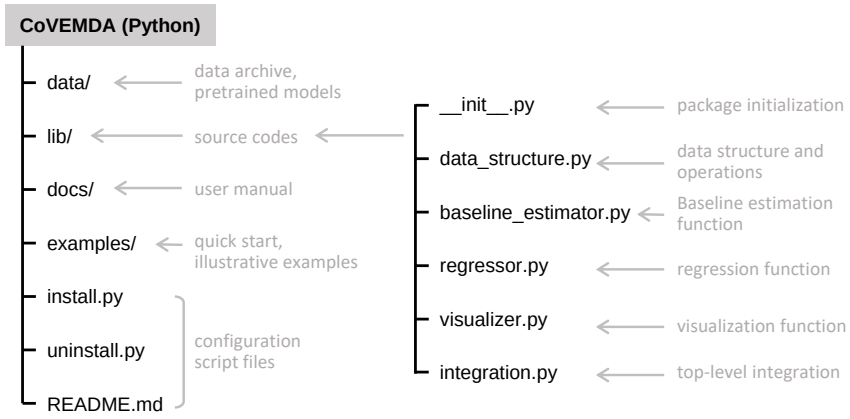

Figure 4-2: Folder structure of CoVEMDA. This folder contains all the source codes, tutorial examples, and documentations. Further instructions of the source code folder (lib/) are also given.

In CoVEMDA, the root folder has four major folders and several configuration files. The folders include data/, lib/, docs/, and examples/, while the configuration files are install.py, uninstall.py, and README.md.

Folder `data/`: This folder contains the archived data (updated to March 2021) and pretrained models, these resources may be very useful to accelerate the empirical analysis.

Folder `lib/`: This folder consists of all the source codes, and should be protected from unexpected modifications, especially when using this toolbox in a developer mode. A detailed view is also shown in the right of Figure 4-2. The relevant classes and functions are collected in the same file with increased readability. We make efforts to keep clean logic so that our codes can be easily reused or extended.

Folder `docs/`: This folder currently only has the user manual (this document).

Folder `examples/`: This folder involves all the quick-start and illustrative examples in the user manual, which are beneficial for beginners.

Configuration files: These files are mainly designed for automatic configurations, e.g., checking the package dependencies, and brief introductory documentations.

4.3 Object-Oriented Design

An important characteristic of a Python toolbox is the object-oriented design, which contributes to constructing an efficient class family. Specifically, Figure 4-3 elaborates how the CoVEMDA toolbox organizes the major classes and their inheritance relationship to realize the relevant modules.

There are five base classes in total—a data structure class, a baseline estimator class, a regressor class, a visualizer calss, and an area class—they mainly build up the fundamental properties and some key components. A few high-level classes are then established to specify the method details, and integrated to construct the RTO and City classes at last. These classes provide concise and powerful interfaces for ease of use.

As for extensions, users are allowed to develop their own class based on the predefined classes. External data sources, special parsers, and user-defined functions could be included in this new class to support further development.

4.4 Integration

In order to simplify the analysis workflow, we have summarized a few use cases in CoVEMDA, and developed an integration module to collect the built-in functions. As mentioned before, this module contains the commonly-used high=level functions (see Figure 4-1) which are collected in the integration.py (see Figure 4-2). From an objective-oriented design perspective, this module is the final step of all (see Figure 4-3), and tends to be useful in various typical use cases.

The significant advantage of integration is the ease of use. The built-in classes and functions allow handling quite a few complicated applications with only several lines of commands. This is extremely

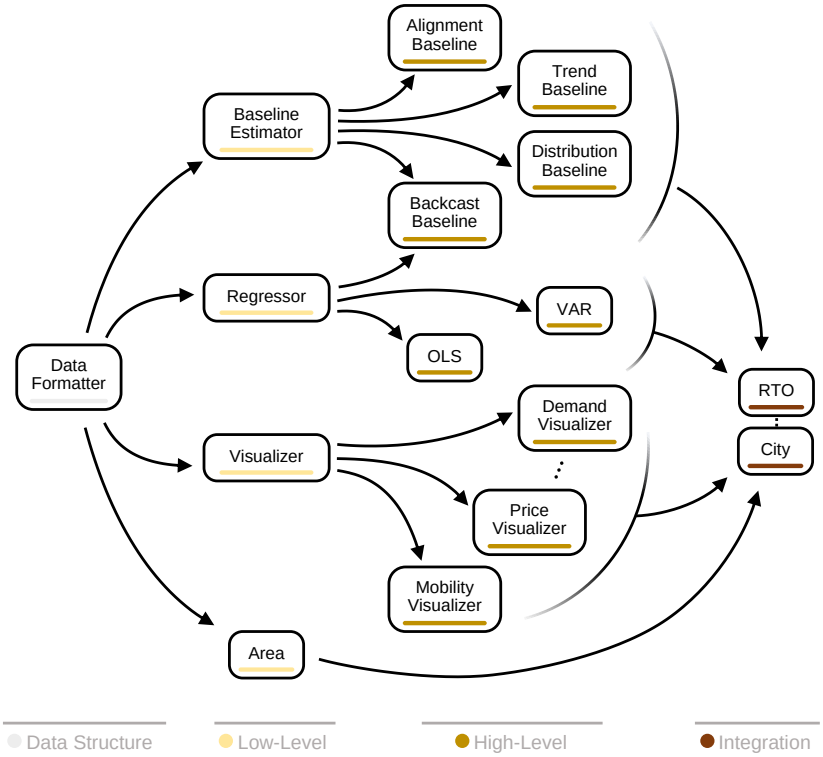

Figure 4-3: Object-oriented design scheme for CoVEMDA. This design is based on the proposed three-layer programming architecture, and all the classes are effectively organized, inherited, and coordinated to handle different use cases.

important for non-expert users who require a holistic and practical perspective.

It must be pointed out that conciseness may often contradict to flexibility. We preset most of additional configurations by default and leave as few required parameter inputs as possible. This may become a negative part in case users have their unusual needs. Another minor issue is the efficiency: some preparation steps might be repeated in similar tasks or functions, because we intend to keep every function stand-alone. This is neglectable in most cases, and one can optimize the workflow using high- or low-level functions.

Here are some best practices to use the integration module. If the objective is quickly verifying a claim within a predefined use case, it is highly recommended to use this module. But if user-defined extensions or comprehensive process control are in need, it would be better to work with high- or even low-level functions. Please note that more technical details are available in the subsequent sections and appendix.

4.5 Interface Overview

Following the above architecture, We have developed a set of interfaces to access the data or functions freely. We will now present a short summary on the interface before diving into any further details.

As shown in Figure 4-3, the top-level classes are RTO and City which have integrated all the major high-level features. These two classes are of critical importance, and users are recommended to consider using them as the first priority. In this aspect, the main text of this user manual will only focus on introducing the functional interfaces from RTO or City classes, and all the basic operations, low- or high-level interfaces are only briefly discussed in the Appendix.

It is easy to find that the RTO and City classes just look very similar. In fact, most of their attributes and methods are similar, but we keep them separated due to the different geographical scales. The biggest difference between them is that RTO class has no mobility data, while City class has no generation data. Some of the functions will be disabled accordingly.

From a writing aspect, we avoid repetitively proposing similar syntax definitions, so the main focus is on the methods of RTO class. One can readily extend the introductory content to City class if needed.

4.6 API Reference

4.6.1 `covemda.integration.RTO`

Function Definition

```
class covemda.integration.RTO(name, source='archive')
```

Overview

The standard integration class for RTO-level areas.
This class is the basic interface for different use cases within a specific RTO region, including baseline estimation, regression analysis, and visualization tasks.

Parameters

name: str{"caiso", "miso", "isone", "nyiso", "pjm", "spp", "ercot"}. [required]
Specify the RTO of interest. This name is case sensitive, and only seven lower-case options shown above are permitted, any other inputs will raise an exception.

source: str{"archive", "online"} or other folder path. [optional, default="archive"]
Specify the source for data loading. "archive" means using the local archive, automatic search will be conducted to find such a folder nearby. "online" means using online data retrieval, this will always keep the most updated data, but might not be fast due to the internet connection. A folder path is another option, and both relative or absolute path forms are supported.

Attributes

dfmt_demand: covemda.data_structure.DataFormatter
A wide data frame to collect the demand data. Data are automatically loaded during initialization. Support all basic operations for DataFormatters, so that users can conduct flexible and direct data manipulation for extension purposes.

dfmt_genmix: covemda.data_structure.DataFormatter
A wide data frame to collect the generation data. Support automatic data loading and other operations.

dfmt_price: covemda.data_structure.DataFormatter
A wide data frame to collect the price data. Support automatic data loading and other operations.

dfmt_weather: covemda.data_structure.DataFormatter
A wide data frame to collect the weather data. Support automatic data loading and other operations.

dfmt_mobility: covemda.data_structure.DataFormatter
A wide data frame to collect the mobility data, but this variable will remain empty because there is no mobility data in the RTO level.

dfmt_covid: covemda.data_structure.DataFormatter
A wide data frame to collect the public health data. Support automatic data loading and other operations.

big_event: dict
A dictionary to record the big events. Different sets of big events are prepared for different RTOs.

There are quite a few built-in methods in a RTO class, but we will leave it to the subsequent sections where those technical topics will be discussed more in details.

4.6.2 covemda.integration.City

Function Definition

```
class covemda.integration.City(name, source='archive')
```

Overview

The standard integration class for city-level regions.
This class is the basic interface for different use cases in a specific city region, including baseline estimation, regression analysis, and visualization tasks. This class inherits from the RTO class, so most definitions and methods of both classes are almost the same.

Parameters

name: str{"la", "boston", "nyc", "chicago", "phila", "kck", "houston"}. [required]
Specify the city of interest. This name is case sensitive, and only the above seven lower-case options are permitted. These cities are Los Angeles, Boston, New York City, Chicago, Philadelphia, Kansas City, and Houston respectively.

source: str{"archive", "online"} or other folder path. [optional, default="archive"]
Specify the source for data loading. "archive" means using the local archive. "online" means using online data retrieval. A folder path is another permitted option.

Attributes

dfmt_demand: DataFormatter
A wide data frame to collect the demand data. Support automatic data loading and other operations.

dfmt_genmix: DataFormatter
A wide data frame to collect the generation data, but this variable will remain empty because there is no generation data in the city level.

dfmt_price: DataFormatter
A wide data frame to collect the price data. Support automatic data loading and other operations.

dfmt_weather: DataFormatter
A wide data frame to collect the weather data. Support automatic data loading and other operations.

dfmt_mobility: DataFormatter
A wide data frame to collect the mobility data. Support automatic data loading and other operations.

dfmt_covid: DataFormatter
A wide data frame to collect the public health data. Support automatic data loading and other operations.

big_event: dict
A dictionary to record the big events. Different sets of big events are prepared for different Cities.

Similar, there are quite a few built-in methods in a City class, but we will stop here and leave it to the subsequent sections.

5 Baseline Estimation

Baselines are relatively important to quantify the existence and intensity of a potential impact. We can directly assess those impacts by comparing the actual observations with the estimated baselines because a reliable baseline is able to provide a control group that have not experienced COVID-19 [8, 9].

Estimating baselines is of great importance, but no estimation method is perfect, and different applications often calls for different methods. For instance, there are many baseline estimators proposed for the power consumption during COVID-19. One classical method is selecting similar dates in the history, and another advanced option is using the backcast model, which utilizes deep neural networks and multi-dimensional historical data for the estimation task.

In order to support reliable and ready-to-use baseline estimations, we collect and develop a few popular methods in CoVEMDA with adequate designs for further extensions. More technical details will be presented in the following subsections.

5.1 Date and Week Alignment

Date alignment refers to a method that calculates the baselines by searching for the data of the exact same date and month but in a previous year. This method is widely used in public media and for demand evaluation, but it can readily be extended to other variables. In mathematics, the baseline value of a variable X_{ymdt} is $X_{y-1,m,d,t}$.

A simple extension of date alignment is the month alignment, and the only difference is that we will calculate the monthly statistics (e.g., summation) before searching for the same month. So now, the baseline value of X_{ym} is $X_{y-1,m}$, and X_{ym} is the total amount in a month.

Week alignment refers to an improved method that searches for the same week indices. A week index is defined as the week number and weekday in a year. For example, June 1 2020 and June 3 2021 are both Mondays of the 22nd week of the corresponding year, so this pair is taken to calculate the baseline. Here, the baseline value of X_{ymdt} is $X_{y-1,m',d',t}$, and the associated two dates share the same week index.

5.2 Backcast Estimation

Backcast estimation is primarily used to estimate the electricity consumption profile in the absence of COVID-19 pandemic, which is useful to seriously quantify the impact of COVID-19.

A backcast model is expressed as a function that maps potential factors that may affect electricity consumption level, including weather variables (such as temperature, humidity and wind speed) and date of year, to the final electricity consumption. Take the daily demand backcast model as an example, the expression can be formulated as

$$\hat{D}_{md} = \hat{f}(C_{md}, T_{mdq}, H_{mdq}, S_{mdq}), \quad \forall m, d \tag{5-1}$$

where C_{md} is the calendar information including month, day, weekday, and holiday flag. $\hat{D}_{md}$ is the estimated daily average electricity consumption for month m and day d. $T_{mdq}, H_{mdq}, S_{mdq}$ are temperature, humidity and wind speed within the selected quantiles q. And $\hat{f}(\cdot)$ is the backcast function.

In the above case, the baseline value for D_{md} is $\hat{D}_{md}$. Also, it is easy to extend the backcast model to be an ensemble backcast model by integrating a few single models, and this will sometimes formulate a more comprehensive and robust estimator.

5.3 Abnormal Price Index

Locational marginal prices play a crucial role in electricity markets and contribute to balancing generation and demand efficiently. Since these prices are relatively stochastic in nature, it is unsatisfactory to analyze them in the same way as electricity demand, because the uncertainty intervals will be very wide. Thus, the abnormal price index is developed to provide a more reliable baseline for price analysis, which is defined as follows,

$$I(\lambda_{ymdt}) = |2\hat{F}_m(\lambda_{ymdt}) - 1|, \quad \forall y, m, d, t, \tag{5-2}$$

where $I(\cdot)$ is the proposed index that lies between zero and one, and λ_{ymdt} is the day-ahead locational marginal price of year y, month m, day d, and hour t. $\hat{F}_m(\cdot)$ is the cumulative distribution function for the prices in month m, which are more stable for analyzing these price changes [7]. The proposed index quantifies the abnormality of a typical price, and a larger index value represents a more unusual observation.

The statistical meaning of the above index is explained by the probability density function shown in Figure 5-1. For a given price, the index donates a possibility that measures how close this price is to the mean value. For example, if a price observation is located within the 25% and 75% quantiles, we obtain an index value below 0.5, which is considered normal.

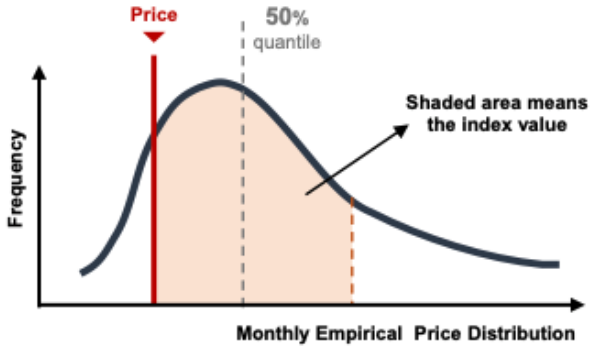

Figure 5-1: Statistical illustration of the proposed abnormal price index. The basic idea is to show the distance between a price observation and the mean value. This index can eliminate some stochastic factors when assessing price changes. The index value lies in [0,1], and a larger value means a higher possibility of abnormality.

5.4 Price Distribution Change

To conduct a cross-market comparison, a Wasserstein probability distance [11, 12] is used as metric to quantify the price change during the pandemic. A price change for year y can be formulated as follows:

$$s_y = \mathrm{WD}(\mathrm{Vec}(\lambda_{ymdt}) - \mathrm{Vec}(\lambda_{hist})), \quad \forall y \tag{5-3}$$

where s_y donates the price change for year y that is calculated by a Wasserstein distance function $\mathrm{WD}(\cdot)$. $\mathrm{Vec}(\cdot)$ is a vectorization function to place all the price data from March to mid-July in a one-dimensional array. λ_{hist} represents all the historical price data accordingly.

5.5 Excess Change Rate of Renewables

One simple way to quantify the changes of renewable share during COVID-19 is detrending the annual growth rate. One simple but acceptable statement assumes a linear growth rate, and based on that, the excess change rate is defined as the relative change rate between the observed renewable share and this estimation:

$$\eta = \left(\frac{r_{2020}}{2r_{2019} - r_{2018}} - 1 \right) \times 100\% \tag{5-4}$$

where η is the excess change rate, and $r_{2018} \sim r_{2020}$ are the market shares of renewable energy observed in 2018, 2019 and 2020.

It is clear that the above formula follows the idea of detrending, and helps us to get valuable insights of the changes beyond normal trends.

5.6 API Reference

5.6.1 covemda.integration.RTO.cal_demand_baseline

Function Definition

```
covemda.integration.RTO.cal_demand_baseline(date=pd.date_range('2020-01-01'
    , '2020-06-30'), method='date-alignment', shift_year=1, pickle_file=
    None)
```

Overview

Calculate the demand baselines during a specified period and with a specified method. Three built-in estimation methods are: date alignment, week alignment, and backcast estimation.

Note that both RTO and City class have this function with the same syntax definitions. Here we simply take the RTO's function as an example.

Parameters

date: str, pd.Timestamp, or pd.DatetimeIndex. [optional, default=Jan 1–June 30 2020]
Specify the date for baseline analysis. A single date or a date range are both allowed. This variable is very important, so users should be careful enough.

method: str{"date-alignment", "week-alignment", "backcast"}. [optional, default="date-alignment"]
Specify the baseline method, choose from the date alignment, week alignment, or backcast estimation.

shift_year: int. [optional, default=1]
Set the number of years to look back when estimating the baselines. For example, shift_year=1 means using some data in the last year as baselines. Note that this is an auxiliary variable associated with the date or week alignment methods, and it will be simply ignored when using the backcast method.

pickle_file: None or a file path. [optional, default=None]
Configure to load a trained backcast model. A previous model should be saved and prepared for use, and users may also turn to the pretrained model on Github (five-layer, ReLU-activated neural networks). When pickle_file=None, an automatic search engine will work to find the pretrained pickle files. Users can train and load their own models and give a file path here. Note that this is an auxiliary variable for backcast method only, and it makes no effects when using other two estimation methods.

Returns

baseline: pd.DataFrame
This function returns the baseline data which are organized with date-type row index and hour-type column index.

5.6.2 covemda.integration.RTO.eval_demand_baseline

Function Definition

```
covemda.integration.RTO.eval_demand_baseline(date=pd.date_range('2020-01-01
    ', '2020-06-30'), method='date-alignment', shift_year=1, pickle_file=
    None)
```

Overview

Evaluate the demand baselines by statistical analysis. Three statistical metrics (MAPE, sMAPE, and RMSE) are considered to compare the baselines with real observations. Often, a large difference represents a discernible impact. This function will print out and return the evaluation results.

Note that both RTO and City class have this function.

Parameters

Same as `covemda.integration.RTO.cal_demand_baseline` (above).

Returns

result: tuple(MAPE, sMAPE, RMSE)
This function returns a tuple collecting the MAPE, sMAPE, and RMSE statistics.

5.6.3 `covemda.integration.RTO.plot_demand_baseline`

Function Definition

```
covemda.integration.RTO.plot_demand_baseline(date=pd.date_range('2020-01-01
    ', '2020-06-30'), method='date-alignment', shift_year=1, pickle_file=
    None)
```

Overview

Plot the demand baselines along with the real observations in a line chart. Focusing on the pandemic period, this chart is useful to reflect the potential impacts by assuming the baselines as a counterfactual condition.

Note that both RTO and City class have this function.

Parameters

Same as `covemda.integration.RTO.cal_demand_baseline` (above).

Returns

bl: covemda.baseline_estimator.BaselineEstimator or other
This function returns an object of the BaselineEstimator type. Different estimators will export different kinds of objects. Often ignored in practical applications.

5.6.4 `covemda.integration.RTO.cal_genmix_baseline`

Function Definition

```
covemda.integration.RTO.cal_genmix_baseline(date=pd.date_range('2020-01-01'
    , '2020-06-30'), method='month-alignment', kind='renewable', shift_year
    =1, verbose=False)
```

Overview

Calculate the generation baselines. Since most generation data are best viewed with monthly statistics, this function implements two built-in methods: month alignment and monthly trend estimation. Here, we typically use ARIMA(2,0,1) as the monthly trend estimator.

Note that only RTO class has this function, because generation data is not available in the city level.

Parameters

date: str, pd.Timestamp, or pd.DatetimeIndex. [optional, default=Jan 1–June 30 2020]
Specify the date for baseline analysis.

method: str{"month-alignment", "monthly-trend"}. [optional, default="month-alignment"]
Specify the baseline method, either month alignment or monthly trend method.

kind: str{"coal", "gas", "oil", "nuclear", "hydro", "wind", "solar", "renewable"}. [optional, default="renewable"]
Specify which kind of generation data shoud be analyzed. The valid inputs involve all the generation kinds and a collection called "renewable" (including hydro, solar, and wind energy).

shift_year: int. [optional, default=1]
Set the number of years to look back when estimating the baselines. Only associated with month alignment method.

verbose: bool. [optional, default=False]
Whether to show the training details. Only associated with monthly trend method.

Returns

baseline: pd.DataFrame

5.6.5 `covemda.integration.RTO.eval_genmix_baseline`

Function Definition

```
covemda.integration.RTO.eval_genmix_baseline(date=pd.date_range('2020-01-01
    ', '2020-06-30'), method='month-alignment', kind='renewable',
    shift_year=1, verbose=False)
```

Overview

Evaluate the generation baselines with three statistical metrics (MAPE, sMAPE, and RMSE).
Note that only RTO class has this function.

Parameters

Same as `covemda.integration.RTO.cal_genmix_baseline` (above).

Returns

result: tuple(MAPE, sMAPE, RMSE)

5.6.6 `covemda.integration.RTO.plot_genmix_baseline`

Function Definition

```
covemda.integration.RTO.plot_genmix_baseline(date=pd.date_range('2020-01-01
    ', '2020-06-30'), method='month-alignment', kind='renewable',
    shift_year=1, verbose=False)
```

Overview

Plot the generation baselines. Note that only RTO class has this function.

Parameters

Same as `covemda.integration.RTO.cal_genmix_baseline` (above).

Returns

bl: covemda.baseline_estimator.BaselineEstimator or other

5.6.7 `covemda.integration.RTO.cal_price_baseline`

Function Definition

```
covemda.integration.RTO.cal_price_baseline(date=pd.date_range('2020-01-01',
    '2020-06-30'), method='distrib-index', hist_date_range=pd.date_range('
    2017-01-01', '2019-12-31'))
```

Overview

Calculate the price baselines. Since prices are always fluctuating, a distribution perspective is thus needed. Two built-in functions are: distribution index and distributional distance estimation.

Note that both RTO and City class have this function.

Parameters

date: str, pd.Timestamp, or pd.DatetimeIndex. [optional, default=Jan 1–June 30 2020]
Specify the date for baseline analysis.

method: str{"distrib-index", "distrib-dist"}. [optional, default="distrib-index"]
Specify the baseline method, either distribution index or distributional distance method.

hist_date_range: pd.DatetimeIndex. [optional, default=Jan 1 2017–Dec 31 2019]
Specify the historical date range to derive the price distrition. This is an important setting, and avoid using any biased date ranges.

Returns

baseline: pd.DataFrame

5.6.8 `covemda.integration.RTO.eval_price_baseline`

Function Definition

```
covemda.integration.RTO.eval_price_baseline(date=pd.date_range('2020-01-01'
    , '2020-06-30'), method='distrib-index', hist_date_range=pd.date_range(
    '2017-01-01', '2019-12-31'))
```

Overview

Evaluate the price baselines with three statistical metrics (MAPE, sMAPE, and RMSE).
Note that both RTO and City class have this function.

Parameters

Same as `covemda.integration.RTO.cal_price_baseline` (above).

Returns

result: tuple(MAPE, sMAPE, RMSE)

5.6.9 covemda.integration.RTO.plot_price_baseline

Function Definition

```
covemda.integration.RTO.plot_price_baseline(date=pd.date_range('2020-01-01'
    , '2020-06-30'), method='distrib-index', hist_date_range=pd.date_range(
    '2017-01-01', '2019-12-31'))
```

Overview

Plot the price baselines. When using distributional distance estimation, this function will use a histogram chart instead of a line chart.

Note that both RTO and City class have this function.

Parameters

Same as `covemda.integration.RTO.cal_price_baseline` (above).

Returns

bl: covemda.baseline_estimator.BaselineEstimator or other

5.6.10 covemda.integration.City.cal_mobility_baseline

Function Definition

```
covemda.integration.City.cal_mobility_baseline(date=pd.date_range('
    2020-01-01', '2020-06-30'), method='date-alignment', kind='retail',
    shift_year=1)
```

Overview

Calculate the mobility baselines with either date alignment or week alignment method.
Note that only City has this function, because mobility data is not available in the RTO level.

Parameters

date: str, pd.Timestamp, or pd.DatetimeIndex. [optional, default=Jan 1–June 30 2020]
Specify the date for baseline analysis.

method: str{"date-alignment", "week-alignment"}. [optional, default="date-alignment"]
Specify the baseline method, either date alignment or week alignment method.

kind: str{"retail", "grocery", "restaurant", "completely_home", "median_home", "part_time_work", "full_time_work"}. [optional, default="retail"]
Specify which kind of mobility data should be analyzed.

shift_year: int. [optional, default=1]
Set the number of years to look back when estimating the baselines.

Returns

baseline: pd.DataFrame

5.6.11 covemda.integration.City.eval_mobility_baseline

Function Definition

```
covemda.integration.City.eval_mobility_baseline(date=pd.date_range('
    2020-01-01', '2020-06-30'), method='date-alignment', kind='retail',
    shift_year=1)
```

Overview

Evaluate the mobility baselines. Note that only City has this function.

Parameters

Same as `covemda.integration.City.cal_mobility_baseline` (above).

Returns

result: tuple(MAPE, sMAPE, RMSE)

5.6.12 covemda.integration.City.plot_mobility_baseline

Function Definition

```
covemda.integration.City.plot_mobility_baseline(date=pd.date_range('
    2020-01-01', '2020-06-30'), method='date-alignment', kind='retail',
    shift_year=1)
```

Overview

Plot the mobility baselines. Note that only City has this function.

Parameters

Same as `covemda.integration.City.cal_mobility_baseline` (above).

Returns

bl: covemda.baseline_estimator.BaselineEstimator or other

5.7 Examples

This section will introduce two real-world examples to show how to estimate and compare different baselines with CoVEMDA. This is one of the most basic but important requirements for many use cases.

5.7.1 Demand Baselines Comparison

For the first case, we analyze the demand baselines on April 27 2020 in NYISO market. Three available estimation methods are all considered: date alignment, week alignment, and backcast.

Every baseline method is associated with a simplified plotter, which is showing the baseline and observation at the same chart. The following codes will run these plotters to give an overall impression of different methods:

Console

```
>>> from covemda.integration import RTO
>>> nyiso = RTO('nyiso')
>>> for method in ('date-alignment', 'week-alignment', 'backcast'):
        nyiso.plot_demand_baseline('2020-04-27', method)
```

The above code will generate three figures one by one, and we collect them together in Figure 5-2. It is interesting to find a huge difference between different estimation results, especially for backcast method. The message we get from Figure 5-2 is that the backcast estimator indicates that NYISO has experienced a large demand drop on April 27 2020, which might be largely underestimated when using other traditional methods such as date or week alignment.

Selecting a proper estimation method is of critical importance, and should raise the attention from the power community. Beyond a visual result, one may require some statistics before diving into the detailed numbers. This can be readily done by using the built-in evaluation function:

Console

```
>>> for method in ('date-alignment', 'week-alignment', 'backcast'):
        print(f"\nMethod:{method}")
        nyiso.eval_demand_baseline('2020-04-27', method)

Method: date-alignment
Statistics of Difference from Baseline:
[MAPE]  = 2.35 %
[sMAPE] = 2.39 %
[RMSE]  = 418.78

Method: week-alignment
Statistics of Difference from Baseline:
[MAPE]  = 4.73 %
[sMAPE] = 4.57 %
[RMSE]  = 872.24

Method: backcast
Statistics of Difference from Baseline:
[MAPE]  = 8.81 %
[sMAPE] = 8.37 %
[RMSE]  = 1479.06
```

In addition, all the baselines are accessible, and the user can simply derive them with a single-line command. For example, let us extract and print out the date-aligned baseline:

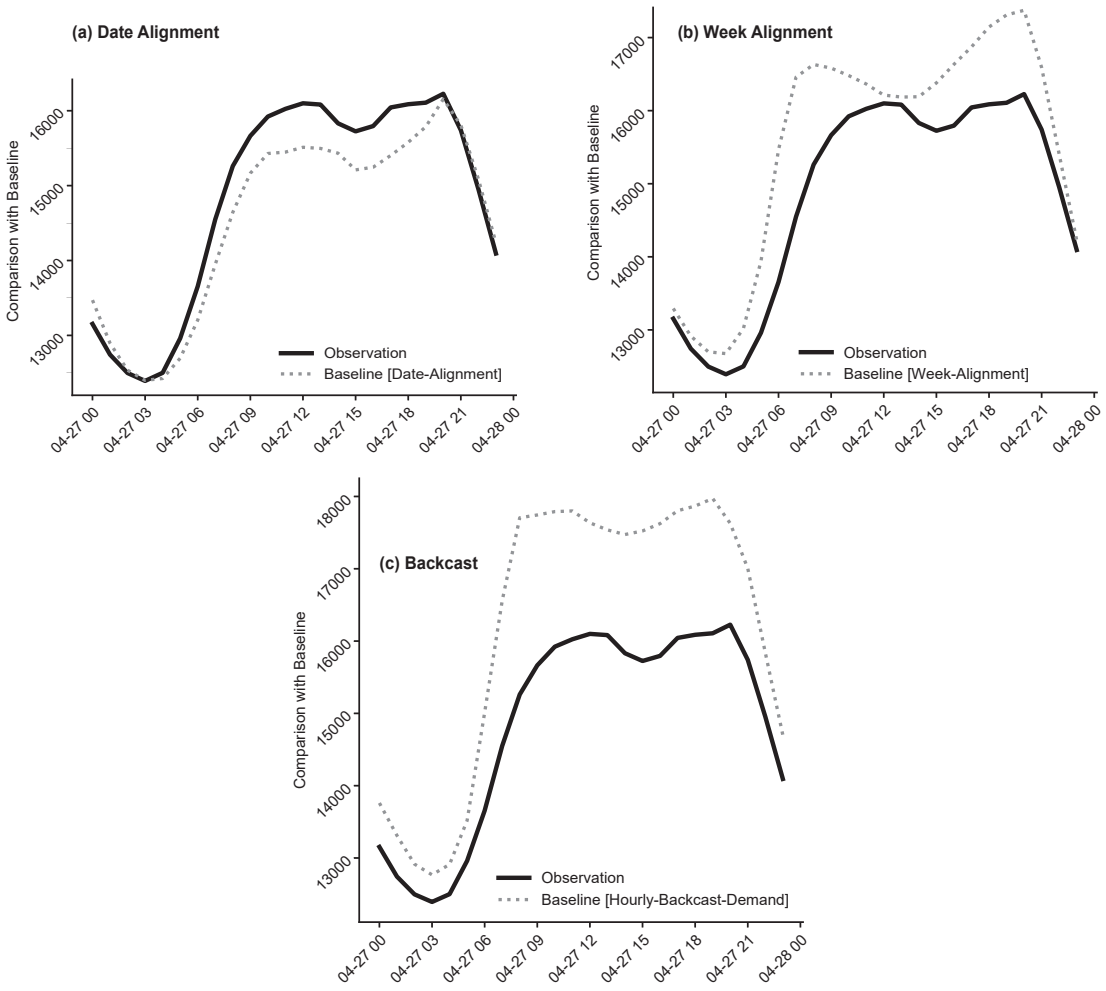

Figure 5-2: Baseline and observed Demand in NYISO on April 27 2020. Different baseline estimation methods are also compared. This figure is combined and labeled for better illustration.

Console

```
>>> print(nyiso.cal_demand_baseline('2020-04-27', method='date-alignment'))

               00:00     01:00     02:00  ...     21:00     22:00     23:00
2020-04-27  13471.1  12896.6  12535.1  ...  15808.4  15086.8  14235.1
[1 rows x 24 columns]
```

Same as the data structure in CoVEMDA, the baseline result is also structured as a wide data frame, with the row showing dates and the column showing hours.

To go a little bit further, date alignment may not seem to be a suitable method in this case (although simple and informative), because the target date April 27 2020 is a Monday (weekday), but the same day in 2019 is a Saturday (weekend). This statement can be easily checked with some low-level functions (see Appendix A to get more knowledge about them), shown as follows:

Console

```
>>> for date in ('2020-04-27', '2019-04-27'):
>>> dfmt = nyiso.dfmt_demand.select_by_date(after=date, before=date)
        print(dfmt.gen_calendar_data().data)

            month  day  weekday  holiday
date
2020-04-27      4   27        0        0

            month  day  weekday  holiday
date
2019-04-27      4   27        5        0
```

Focus on the above weekday columns, number 0 means Monday, and 5 means Saturday. Likewise, users are able to deepen the analysis to cover more aspects.

5.7.2 Renewable Generation Baselines

The next example shows how the monthly renewable generation has changed during COVID-19, and this may lead us to an empirical assessment of any additional gain or loss for renewable energy.

Below we will demonstrate two kinds of baselines: month alignment, and monthly trend. Run the following codes and get the result shown in Figure 5-3.

Console

```
>>> for method in ('month-alignment', 'monthly-trend'):
        nyiso.plot_genmix_baseline(
            pd.date_range('2019-06-30', '2020-06-30'), method)
```

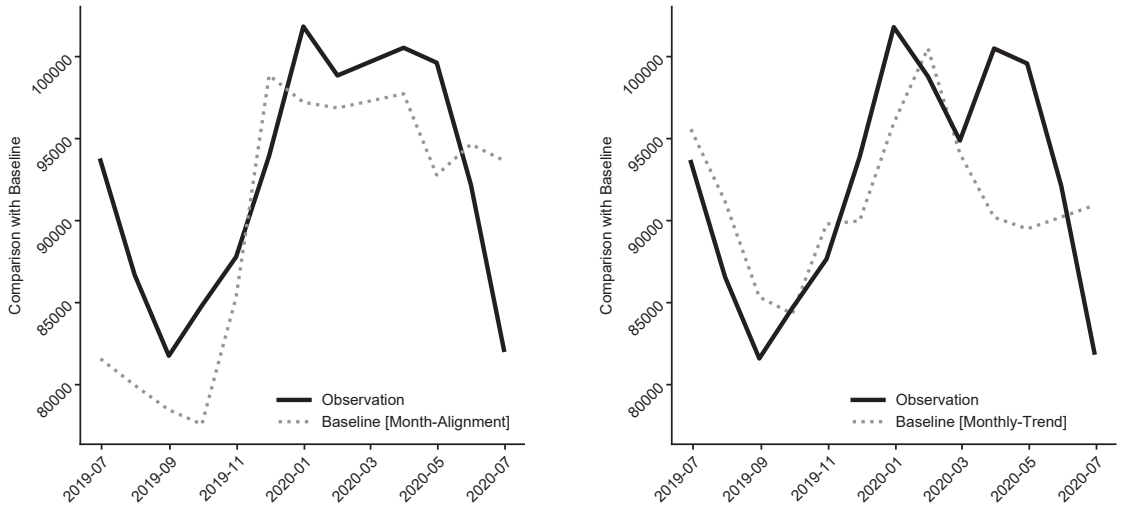

Figure 5-3: Baseline and observed renewable generation in NYISO from June 2019 to June 2020. Two different estimation methods, based on monthly statistics, are compared.

An intuition derived from Figure 5-3 is that a little increment could be found during March-June 2020 in NYISO. Note that this statement may not be robust anyway, and more efforts should be taken for further assessments. In fact, the power community has not come to a consensus on whether the renewable energy suffered more or less during COVID-19, but the proposed data and toolbox have provided a starting point, and may hopefully help increase our knowledge in this topic.

6 Regression Analysis

Regression is powerful to answer a variety of questions regarding the correlation and causality. It is possible, for example, to explore the underlying relationship between demand drop during COVID-19 and other cross-domain factors of public health, social distancing, and retail mobility. In this section, we hereby introduce two popular regression methods to study the correlated and causal relationships during COVID-19.

It should be noted that the toolbox supports a flexible syntax for regression, and users may choose from a built-in collection of linear, interaction, quadratic, or pure-quadratic methods. But external data inputs and user-defined functions are allowed for extensions.

6.1 Ordinary Least Squares Regression

Ordinary least squares (OLS) models [13] cover a wide range of linear or non-linear regression expressions, offering multiple choices for diverse use cases of model fitting. Besides, CoVEMDA also supports correlation analysis for variables either from the proposed data hub or from external sources.

In theory, an OLS model is calibrating by minimizing the regression residuals, shown as follows:

$$minf(x) = \sum_{i=1}^{m} L_i^2(x) = \sum_{i=1}^{m} [y_i - f(x_i, \omega_i)]^2, \tag{6-1}$$

where $L_i(x)$ is the residual function. If $L_i(x)$ is a linear function about x, the optimization problem is called 'linear least squares problem', while nonlinear ones are addressed as 'nonlinear least squares problems'.

6.2 Vector Autoregression

Vector autoregression (VAR) is a stochastic process model that can be used to capture the linear correlation between multiple time-series data [14]. This model can be extended to be restricted vector autoregression when some parameters are set to be zeros. Both models are powerful tools for multivariate time series analysis and have been widely adopted in economics [15] and electricity markets [16].

A typical expression of a p-order VAR model is shown as follows:

$$X_t = C + A_1 X_{t-1} + \cdots + A_p X_{t-p} + E_t, \tag{6-2}$$

where

$$A_i = \begin{bmatrix} a_{1,1}^i & a_{1,2}^i & \cdots & a_{1,n}^i \\ a_{2,1}^i & a_{2,2}^i & \cdots & a_{2,n}^i \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1}^i & a_{n,2}^i & \cdots & a_{n,n}^i \end{bmatrix}, X_t = \begin{bmatrix} x_t^1 \\ x_t^2 \\ \vdots \\ x_t^n \end{bmatrix}, C = \begin{bmatrix} c^1 \\ c^2 \\ \vdots \\ c^n \end{bmatrix}, E_t = \begin{bmatrix} e_t^1 \\ e_t^2 \\ \vdots \\ e_t^n \end{bmatrix}, \tag{6-3}$$

in which A_i is the regression matrix, x_t^1 represents the target output variable at time t, namely the reduction in electricity consumption we wish to model, $x_t^2, \cdots, x_t^n$ represent the selected $n-1$ parameter variables including confirmed case numbers, stay-at-home population, median home dwell time rate, population of on-site workers, mobility in the retail sector and etc., C and E_t are respectively column vectors of intercept and random errors, and the time notation $t-p$ represents the p-th lag of the variables.

The full procedure of building a VAR model mainly contains four steps as follows, including pre-estimation preparation, VAR model estimation, VAR model verification, and post-estimation analysis.

6.2.1 Pre-estimation Preparation

Data Preprocessing: Several datasets are collected to calculate the inputs of a VAR model, including electricity market data, meteorological data, number of COVID-19 cases, and mobile device location data. We take logarithms of several variables, including electricity consumption reduction, new daily confirmed cases, stay-at-home population, population of full-time on-site workers, population of part-time on-site workers, and mobility in the retail sector, while only keeping the original value of the median home dwell time rate.

Augmented Dickey-Fuller (ADF) Test [17]: Test whether a time-series variable is non-stationary and possesses a unit root.

Cointegration Test: Test the long-term correlation between multiple non-stationary time-series.
Granger Causality Wald Test: Estimate the possible causality relationship among two variables that are represented as time-series.

6.2.2 VAR Model Estimation

Ordinary Least Square: We apply the most typical and popular method for model estimation, and one could impose extra constraints or regularization terms to eliminate any undesirable correlations between variables.

6.2.3 VAR Model Verification

ADF Test: Verify if the residual time-series are non-stationary and possess a unit root.
Ljung-Box Test: Verify the endogeneity of residual data that may render the result untrustworthy.
Durbin-Watson Test: Detect the presence of autocorrelation at one-step delay lag in the residual series.
Robustness Test: Test the robustness of the Restricted VAR model against parameter perturbations.

6.2.4 Post-estimation Analysis

Impulse Response Analysis: Describe the evolution of all model variables in response to a shock in one or more variables.
Forecast Error Variance Decomposition: Interpret the VAR models by determining the proportion of each variable's forecast variance that is contributed by shocks to the other variables.

6.3 API Reference

6.3.1 covemda.integration.RTO.test_demand_correlation

Function Definition

```
covemda.integration.RTO.test_demand_correlation(date=pd.date_range('
    2020-01-01', '2020-06-30')
```

Overview

Test how demand is correlated with temperature. This function will run an OLS regression: demand = $k_1 + k_2$ tmpc + k_3 tmpc^2, and print out all the statistical test details.

Note that both RTO and City class have this function.

Parameters

date: str, pd.Timestamp, or pd.DatetimeIndex. [optional, default=Jan 1–June 30 2020]
Specify the date for correlation analysis on demand.

Returns

ols: covemda.regressor.OrdinaryLeastSquares
This function returns an object with the specified model and regression results. This object could be useful to extract coefficients or other regression information afterward.

6.3.2 covemda.integration.RTO.test_price_correlation

Function Definition

```
covemda.integration.RTO.test_price_correlation(date=pd.date_range('
    2020-01-01', '2020-06-30')
```

Overview

Test how price is correlated with gas-fired generation, demand, and temperature. This function will run an OLS regression: price = k_1 + k_2 gas + k_3 demand + k_4 tmpc, and print out all the statistical details.

Note that both RTO and City class have this function, but the formula in the city class should drop the generation term due to data inavailability.

Parameters

date: str, pd.Timestamp, or pd.DatetimeIndex. [optional, default=Jan 1–June 30 2020]
Specify the date for correlation analysis on price.

Returns

ols: covemda.regressor.OrdinaryLeastSquares

6.3.3 `covemda.integration.RTO.cal_correlation_coeff`

Function Definition

```
covemda.integration.RTO.cal_correlation_coeff(x, y)
```

Overview

Calculate the Pearson coefficient between two or more specified variables. This function will output a correlation matrix showing the correlation values for all variable pairs.

Note that both RTO and City class have this function.

Parameters

x: list, dict, np.ndarray, pd.DataFrame, or covemda.data_structure.DataFormatter. [required]
The explanatory variable(s) of interest. We do not assume independence among different explanatory variables, and their correlations will also be calculated.

y: list, dict, np.ndarray, pd.DataFrame, or covemda.data_structure.DataFormatter. [required]
The response variable of interest, only one variable is allowed. Users may start by randomly selecting one among all variables if there is not a clear response variable at the very beginning.

Returns

corr: pd.DataFrame
This function returns a DataFrame to show the correlation matrix. It is a N*N symmetric matrix, where N is the number of variables.

6.3.4 `covemda.integration.RTO.run_general_ols`

Function Definition

```
covemda.integration.RTO.run_general_ols(x, y, frac_train=0.7)
```

Overview

Formulate and run a general OLS regression. This is extremely flexible to cover a large number of possible regression applications, and can be extended to create more complicated regressors.

Note that both RTO and City class have this function.

Parameters

x: list, dict, np.ndarray, pd.DataFrame, or covemda.data_structure.DataFormatter. [required]
The explanatory variable(s) of interest.

y: list, dict, np.ndarray, pd.DataFrame, or covemda.data_structure.DataFormatter. [required]
The response variable of interest, only one variable is allowed.

ratio_train: float or int, within [0, 1]. [optional, default=0.7]
The proportion of training data volume among all records. The default setting is partitioning the entire dataset at a 70:30 ratio for training and test. Note that this function simply take the first 70% data as the training set, and the left 30% for test. Users may need to do shuffling before data feed-in.

Returns

ols: covemda.regressor.OrdinaryLeastSquares

6.3.5 `covemda.integration.RTO.run_general_var`

Function Definition

```
covemda.integration.RTO.run_general_var(x, y, frac_train=0.7)
```

Overview

Formulate and run a general VAR regression. This is flexible to cover a large number of time-series applications. A full workflow is implemented, including: training, testing, model verification, impulse response analysis, and variance decomposition.

Note that both RTO and City class have this function.

Parameters

x: list, dict, np.ndarray, pd.DataFrame, or covemda.data_structure.DataFormatter. [required]
The time-series variable(s) of interest. This function will combine x and y (below), then run a VAR regression. Here we keep x and y apart to make the interface same as run_general_ols.

y: list, dict, np.ndarray, pd.DataFrame, or covemda.data_structure.DataFormatter. [required]
The time-series variable of interest. This function will combine x (above) and y, then run a VAR regression. Here we keep x and y apart to make the interface same as run_general_ols.

ratio_train: float or int, within [0, 1]. [optional, default=0.7]
The proportion of training data volume among all records. Users should do shuffling before data feed-in if randomization is needed.

Returns

var: covemda.regressor.VectorAutoregression
This function returns an object with the specified model and regression resutls. All the coefficients or statistical tests are available within this object.

6.4 Examples

This section will introduce two examples to show how to conduct regression analysis with CoVEMDA. Considering that there are a variety of possible regression expressions, we put more focus on the overall workflow as well as syntax description.

6.4.1 Price Correlation Analysis

The first use case is the price correlation analysis in Philadelphia. The built-in function will test how demand and temperature may influence the prices. We begin by creating a new City object:

Console

```
>>> from covemda.integration import City
>>> phila = City('phila')
```

Simply run the following command, and get a table with all the estimation and statistical tests results, shown as follows:

Console

```
>>> phila.test_price_correlation()
...
==============================================================================
            coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const     0.2259      0.222      1.016      0.310      -0.210       0.662
demand    0.0045   5.29e-05     85.599      0.000       0.004       0.005
tmpc     -0.1803      0.005    -38.889      0.000      -0.189      -0.171
==============================================================================
...
```

Some outputs are ignored above for simplification, and we can learn here that the demand and temperature (denoted as tmpc) have obvious impacts on the prices, because of their tiny p-values. But their impacts are different in many aspects: the demand series has a small positive correlation, while the temperature series shows a large negative correlation.

Another way to evaluate the relationship between different variables is calculating the so-called correlation coefficients. Below is a command to test the temperature-price correlation:

Console

```
>>> phila.cal_correlation_coeff(
        x=phila.dfmt_weather.select_by_kind('tmpc'),
        y=phila.dfmt_price,
    )
...
            tmpc     price
tmpc    1.000000 -0.224133
price  -0.224133  1.000000
```

Here we get a correlation matrix, and the cross-diagonal element -0.224 means a negative correlation.

Please note that the toolbox also supports any general form of OLS regression by calling the associated function. The syntax is just the same with VAR regression, and we will leave this part to the next example.

6.4.2 Mobility Dynamics

The second use case is analyzing the mobility dynamics in Los Angeles, and special focus is on the complicated time-series correlation. This will give us an overall impression on how different mobility data may be related or connected.

Create a City object, then call the general VAR regression command as follows. Note that this command is considering the mobility data to restaurants, groceries, and retail places. By default, the regression is ran throughout Jan 1–June 30, 2020.

Console

```
>>> from covemda.integration import City
>>> la = City('la')
>>> la.run_general_var(
        x=phila.dfmt_mobility.select_by_column(('grocery', 'restaurant')),
        y=phila.dfmt_mobility.select_by_column('retail'),
    )
...
Results for equation retail
===============================================================================
                  coefficient       std. error      t-stat        prob
-------------------------------------------------------------------------------
const            52534.551038     14332.160563       3.666       0.000
L1.restaurant       -0.116486         0.427663      -0.272       0.785
L1.grocery          -2.139474         0.900197      -2.377       0.017
L1.retail            1.296968         0.346707       3.741       0.000
===============================================================================
...
```

We will get a bunch of results after running the above command, including the statistical charts (one of them is shown above) and some visual results (Figure 6-1). There are quite a lot of work to make further demonstrations, but all these results could be useful for various kinds of empirical studies.

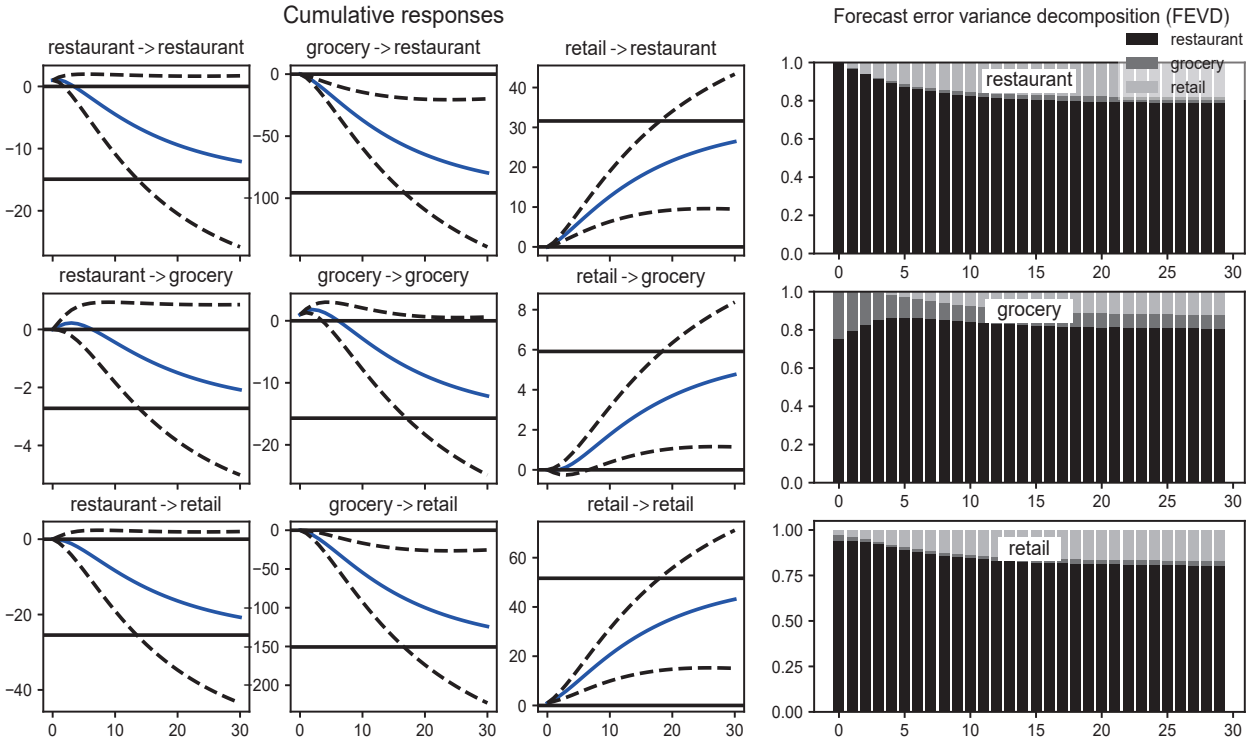

Figure 6-1: Cumulative impulse response and forecast error variance decomposition results for mobility dynamics in Los Angeles.

7 Scientific Visualization

Visualizing the raw data or statistical measurements is an intuitive way to convey messages. This is also useful to explain the baselines or regression results obtained from above sections. The CoVEMDA toolbox collects a variety of classical visualization use cases, and designs a few specific methodologies accordingly. We carefully balance the requirements of ease-of-use and flexibility by providing various functions and interfaces for different requirements.

7.1 Demand Visualizer

Electric demand (or electricity consumption) is a basic bu important component of a whole power system, and an unexpected demand change may always lead to systematic impacts. One important finding from existing publications is that the demand side may be severely impacted during COVID-19. Scientific visualization helps to show such changes in a clear way.

In particular, CoVEMDA provides two kinds of plotters to show the demand series and daily demand profile respectively. Multiple types of date slicing and statistics calculation could be embedded to create more complicated figures.

7.2 Generation Visualizer

After the COVID-19 outbreak, the total generation and generation mix of each U.S. power systems appeared to be quite varied. In order to capture this difference, we try visualizing the generation data from different analysis perspectives to clearly find out the structural changes induced by COVID-19.

In particular, CoVEMDA provides three kinds of plotters to visualize the generation mix, renewable share, and duck curve. Considering that the changes of generation data are best detectable in a month duration, all the calculations are based on monthly statistics by default.

7.3 Price Visualizer

In the U.S. electricity markets, locational marginal prices are often stochastic due to so many possible influencing factors. People have also recognized that electricity prices may have experienced a double impact of COVID-19 and gas price collapse, so analyzing the price change becomes a bit complex. Here, we will visualize the prices from a distributional perspective, and that will help stabilize the data and largely eliminate the potential disturbances. This may offer some new insights about the structural changes induced by COVID-19.

In particular, CoVEMDA provides two kinds of plotters to show the price series and price distribution respectively. Users can advance their analysis results by combining the figures with the analysis methods developed in the previous sections.

7.4 Mobility Visualizer

Mobility is an important variable to show how active a society or an economy is operating. To be specific, mobility means the number of visits to business premises, indicating the level of commercial activities. During COVID-19, the mobility in various sectors have experienced dramatic reduction, which is one of the first signals of the outbreak.

The mobility for three sectors are provided: `Restaurant_Recreation`, `Grocery_Pharmacy` and `Retail`. Unlike other functions in this section, these data are exclusively focused on seven major cities: Los Angeles(`la`), Houston(`houston`), Boston(`boston`), New York City(`nyc`), Chicago(`chicago`), Philadelphia(`phila`) and Kansas City(`kck`).

Although the mobility data is beyond the scope of traditional power system topics, these data become much more valuable when our objective is to track the pandemic's impacts. The proposed data hub and toolbox, in this aspect, open up a new opportunity to derive some cross-domain insights by collectively considering both power system and mobility data.

In particular, CoVEMDA provides a plotter to show different kinds of mobility series in a standard way. Tracking the turning points could be beneficial and informative for empirical studies.

7.5 API Reference

7.5.1 covemda.integration.RTO.plot_demand_series

Function Definition

```
covemda.integration.RTO.plot_demand_series(date=pd.date_range('2020-01-01',
    '2020-06-30'), mark_big_event=False, config=None)
```

Overview

Plot the demand series in a line chart. Users can enable the big event markers on the chart for emphasis. Note that both RTO and City class have this function.

Parameters

date: str, pd.Timestamp, or pd.DatetimeIndex. [optional, default=Jan 1–June 30 2020]
Specify the date to show.

mark_big_event: bool. [optional, default=False]
Whether mark the preset big events in the chart. If True, several vertical lines with labels will be added automatically.

config: None or dict. [optional, default=None]
Customization settings of the chart. None means keeping all the default settings. This parameter follows the syntax of matplotlib, and users are recommended to follow the example codes for more details.

Returns

vs: covemda.visualizer.DemandVisualizer
This function returns a visualizer object that generates the demand series chart. In practical applications, this object is often ignored.

7.5.2 covemda.integration.RTO.plot_daily_demand_profile

Function Definition

```
covemda.integration.RTO.plot_daily_demand_profile(date=pd.date_range('
    2020-01-01', '2020-06-30'), config=None)
```

Overview

Plot the daily demand profiles in a line chart with quantile bands. This chart is similar as a box chart, but the quantiles at different hours are connected to generate the so-called quantile bands.

Note that both RTO and City class have this function.

Parameters

date: str, pd.Timestamp, or pd.DatetimeIndex. [optional, default=Jan 1–June 30 2020]
Specify the date to show.

config: None or dict. [optional, default=None]
Customization settings of the chart.

Returns

vs: covemda.visualizer.DemandVisualizer

7.5.3 covemda.integration.RTO.plot_generation_mix

Function Definition

```
covemda.integration.RTO.plot_generation_mix(self, date=pd.date_range('
    2017-01-01', '2020-06-30'), duration='1M', config=None)
```

Overview

Plot the generation mix in a stacked area chart. This is often used to explore the mid- to long-term structual changes in generation side.

Note that only RTO class has this function.

Parameters

date: str, pd.Timestamp, or pd.DatetimeIndex. [optional, default=Jan 1 2017–June 30 2020]
Specify the date to show.

duration: str. [optional, default="1M"]
Specify the resample duration for statistical calculation. The default setting means calculating the total generation for each month, and these monthly data will be displayed later.

config: None or dict. [optional, default=None]
Customization settings of the chart.

Returns

vs: covemda.visualizer.GenerationVisualizer

7.5.4 covemda.integration.RTO.plot_renewable_share

Function Definition

```
covemda.integration.RTO.plot_renewable_share(date=pd.date_range('2017-01-01
    ', '2020-06-30'), kind=('hydro', 'solar', 'wind'), duration='1M',
    config=None)
```

Overview

Plot the renewable share in a line chart. Hydro, solar, and wind generation are all considered and displayed.
Note that only RTO class has this function.

Parameters

date: str, pd.Timestamp, or pd.DatetimeIndex. [optional, default=Jan 1 2017–June 30 2020]
Specify the date to show.

kind: str{"hydro", "solar", "wind"}, tuple, or list. [optional, default=tuple("hydro", "solar", "wind")]
Specify which renewable source(s) are considered. This parameter can be any single one or any combination of hydro, solar, and wind.

duration: str. [optional, default="1M"]
Specify the resample duration for statistical calculation. The default setting means calculating the total generation for each month, and these monthly data will later be displayed.

config: None or dict. [optional, default=None]
Customization settings of the chart.

Returns

vs: covemda.visualizer.GenerationVisualizer

7.5.5 covemda.integration.RTO.plot_duck_curve

Function Definition

```
covemda.integration.RTO.plot_duck_curve(date=pd.date_range('2017-01-01', '
    2020-06-30'), config=None)
```

Overview

Plot the duck curves in a line chart with quantile bands. A duck curve can be calculated by summing up all kinds of generation except solar energy.

Note that only RTO class has this function.

Parameters

date: str, pd.Timestamp, or pd.DatetimeIndex. [optional, default=Jan 1–June 30 2020]
Specify the date to show.

config: None or dict. [optional, default=None]
Customization settings of the chart.

Returns

vs: covemda.visualizer.GenerationVisualizer

7.5.6 covemda.integration.RTO.plot_price_series

Function Definition

```
covemda.integration.RTO.plot_price_series(date=pd.date_range('2020-01-01',
    '2020-06-30'), mark_big_event=False, config=None)
```

Overview

Plot the price series in a line chart. Users can enable the big event markers on the chart for emphasis. Note that both RTO and City class have this function.

Parameters

date: str, pd.Timestamp, or pd.DatetimeIndex. [optional, default=Jan 1–June 30 2020]
Specify the date to show.

mark_big_event: bool. [optional, default=False]
Whether mark the preset big events in the chart.

config: None or dict. [optional, default=None]
Customization settings of the chart.

Returns

vs: covemda.visualizer.PriceVisualizer

7.5.7 covemda.integration.RTO.plot_price_distribution

Function Definition

```
covemda.integration.RTO.plot_price_distribution(date=pd.date_range('
    2020-01-01', '2020-06-30'), config=None)
```

Overview

Plot the price distribution in a histogram chart. Note that both RTO and City class have this function.

Parameters

date: str, pd.Timestamp, or pd.DatetimeIndex. [optional, default=Jan 1–June 30 2020]
Specify the date to show.

config: None or dict. [optional, default=None]
Customization settings of the chart.

Returns

vs: covemda.visualizer.PriceVisualizer

7.5.8 covemda.integration.RTO.plot_weather_series

Function Definition

```
covemda.integration.RTO.plot_weather_series(date=pd.date_range('2020-01-01'
    , '2020-06-30'), kind='tmpc', config=None)
```

Overview

Plot the weather series in a line chart. Note that both RTO and City class have this function.

Parameters

date: str, pd.Timestamp, or pd.DatetimeIndex. [optional, default=Jan 1–June 30 2020]
Specify the date to show.

kind: str{“tmpc”, “relh”, “sped”}. [optional, default=“tmpc”]
Specify which kind of weather data to show.

config: None or dict. [optional, default=None]
Customization settings of the chart.

Returns

vs: covemda.visualizer.WeatherVisualizer

7.5.9 covemda.integration.RTO.plot_covid_series

Function Definition

```
covemda.integration.RTO.plot_covid_series(date=pd.date_range('2020-01-01',
    '2020-06-30'), kind='new_confirm', config=None)
```

Overview

Plot the public health data series in a line chart. Note that both RTO and City class have this function.

Parameters

date: str, pd.Timestamp, or pd.DatetimeIndex. [optional, default=Jan 1–June 30 2020]
Specify the date to show.

kind: str{"accum_confirm", "new_confirm", "infect_rate", "accum_death", "new_death", "fatal_rate"}. [optional, default="new_confirm"]
Specify which kind of public health data to show.

config: None or dict. [optional, default=None]
Customization settings of the chart.

Returns

vs: covemda.visualizer.CovidVisualizer

7.5.10 covemda.integration.City.plot_mobility_series

Function Definition

```
covemda.integration.City.plot(date=pd.date_range('2020-01-01', '2020-06-30'
    ), kind='retail', config=None)
```

Overview

Plot the mobility series in a line chart. Note that only City class has this function.

Parameters

date: str, pd.Timestamp, or pd.DatetimeIndex. [optional, default=Jan 1–June 30 2020]
Specify the date to show.

kind: str{"retail", "grocery", "restaurant", "completely_home", "median_home", "part_time_work", "full_time_work"}. [optional, default="retail"]
Specify which kind of mobility data to show.

config: None or dict. [optional, default=None]
Customization settings of the chart.

Returns

vs: covemda.visualizer.MobilityVisualizer

7.6 Examples

This section will introduce a few examples of scientific visualization with CoVEMDA. We will first show the basic usage by some typical use cases, then turn to the flexible customization of the chart exports.

7.6.1 Typical Use Cases

The first use case is visualizing the price distribution in ISO-NE market. One need to import the packages and generate the RTO objects at the preparation stage:

```
Console
>>> import pandas as pd
>>> from covemda.integration import RTO
>>> isone = RTO('isone')
>>> nyiso = RTO('nyiso')
```

Then the distribution histogram can be made by calling the built-in plotter as follows, and that gives us a chart like Figure 7-1.

```
Console
>>> isone.plot_price_distribution()
```

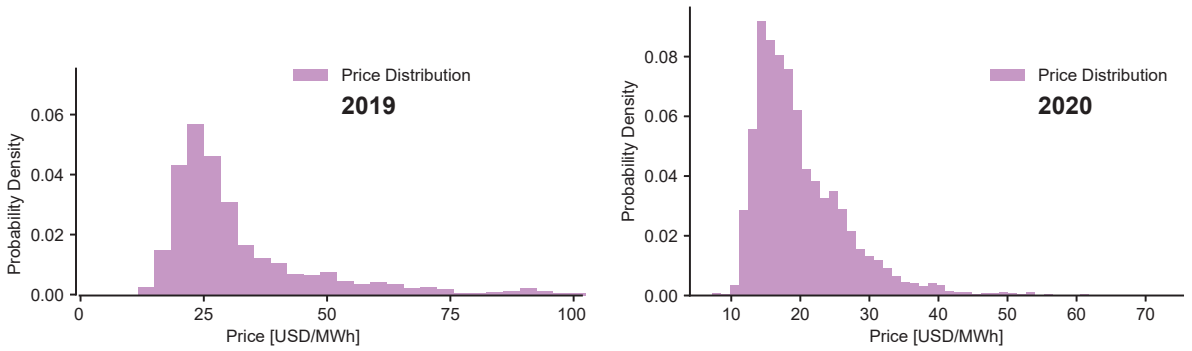

Figure 7-1: Price Distribution Comparison in ISO-NE. The data for the first half year in 2019 and 2020 are collected for visualization.

By default, the above histogram takes the data in first half of 2020. The date range can be specified when calling the plotter.

It may be surprising to find in Figure 7-1 that the prices in 2020 seem to shrink down significantly, with the average price dropping from around 30 USD/MWh in 2019 to around 20 USD/MWh in 2020.

More data details can be visualized by directly plotting the raw time-series data. For example, the below command is intended to observe the price series in 2019, and the result is displayed in Figure 7-2.

```
Console
>>> nyiso.plot_price_series(date=pd.date_range('2019-01-01', '2019-12-31'))
```

Despite the price data, CoVEMDA is embedded with a series of useful plotters for diverse kinds of data. To visualize the demand series, one may simply use the following command, and this will generate Figure 7-3.

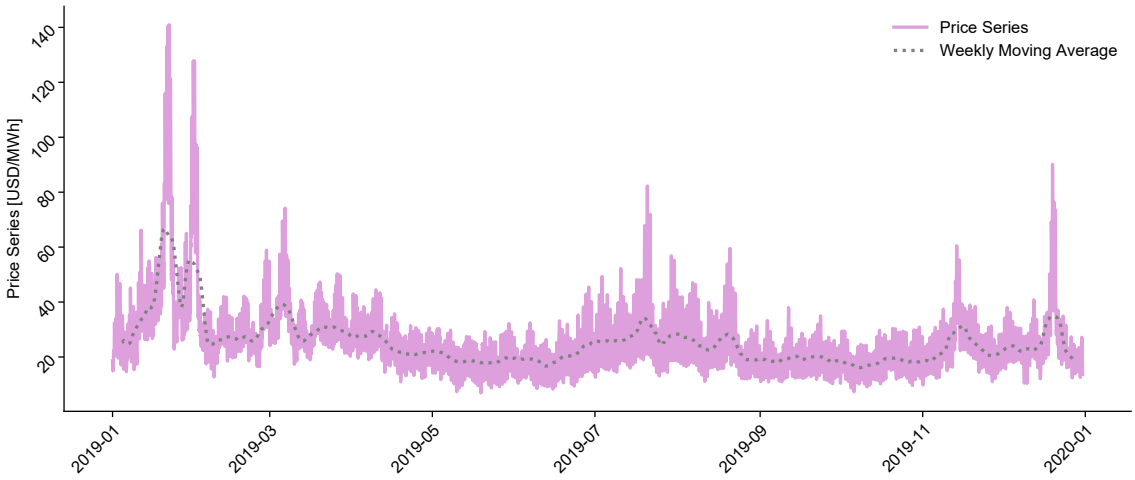

Figure 7-2: Price series throughout 2019 in NYISO. The weekly moving average curve (dotted grey) is also provided to show the changing trend.

```
Console
>>> nyiso.plot_demand_series(
        date=pd.date_range('2020-02-01', '2020-04-30'))
>>> nyiso.plot_demand_series(
        date=pd.date_range('2019-02-01', '2019-04-30'))
```

Users may find some pattern changes in Figure 7-3, which is a very good starting point to initiate an in-depth analysis. This is definitely the power of visualization: scientific visualization results are easy to understand, great to share insights, and helpful to accelerate the knowledge mining.

7.6.2 Chart Customization

Beyond the basic usage, CoVEMDA actually provides many flexible options to allow users customizing their own charts. Since CoVEMDA is using the matplotlib package, the additional configurations just share the same syntax. This is handy to give large flexibility to control the figure elements, and below we will discuss some possible ways for chart customization.

Let us generate two RTO objects first:

```
Console
>>> import pandas as pd
>>> from covemda.integration import RTO
>>> nyiso = RTO('nyiso')
>>> spp = RTO('spp')
```

We are interested to add markers on the data points. This can be done by using a configuration parameter when calling the plotter, shown as follows:

```
Console
>>> nyiso.plot_price_series(
        date=pd.date_range('2020-03-01', '2020-03-07'),
        config={'price_series': dict(marker='o')},
    )
```

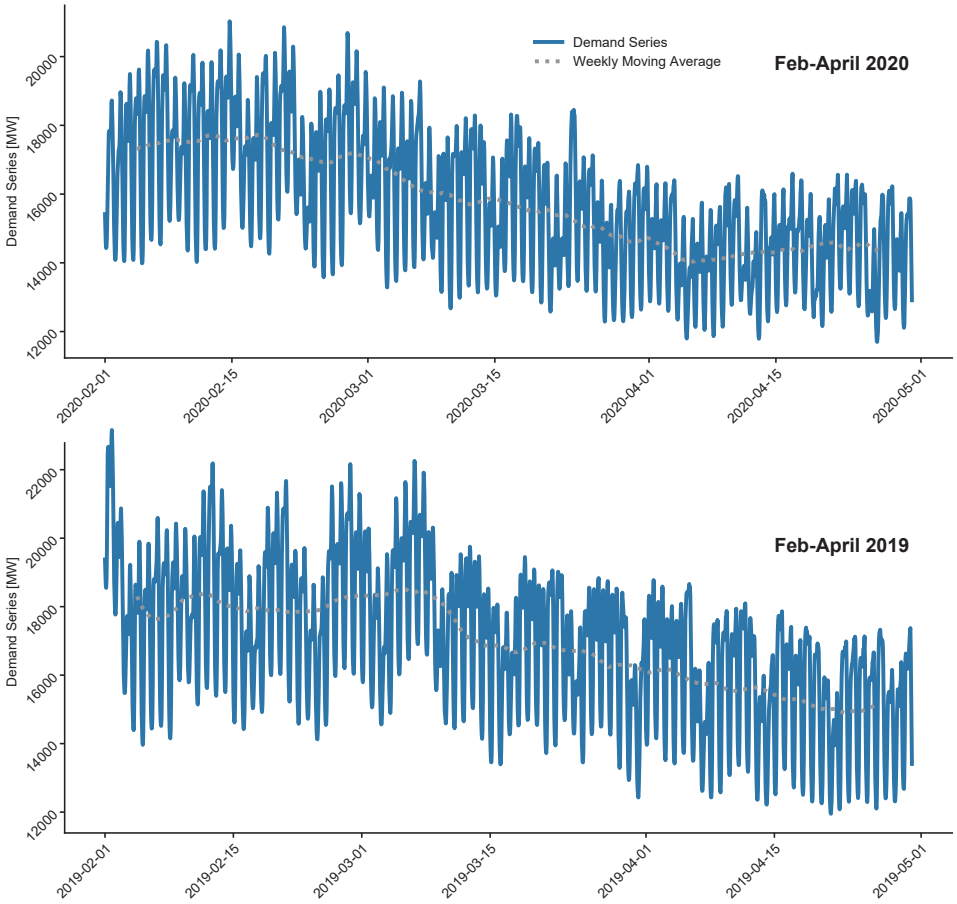

Figure 7-3: Demand series comparison in NYISO. The data for February–April in 2019 and 2020 are collected for visualization.

Now the markers are displayed in Figure 7-4. Users may largely extend the configurations to control almost everything, but remember to follow the property guide of matplotlib.

In addition, applications that require highlighting some elements are very common, and we will next focus on such a case to customize generation mix chart. The following command exports two visual results while the colors of the latter one are changed accordingly.

Console

```
>>> new_color = {
        'coal':    'dimgray',   # changed
        'gas':     'gray',      # changed
        'oil':     'darkgray',  # changed
        'nuclear': 'silver',    # changed
        'hydro':   '#a0d8f1',
        'wind':    '#73c698',
        'solar':   '#ffbd4a',
        'other':   '#b4b4b4',
        'import':  'white',
    }
>>> spp.plot_generation_mix()
>>> spp.plot_generation_mix(
        config={'generation_mix': dict(colormap=new_color)})
```

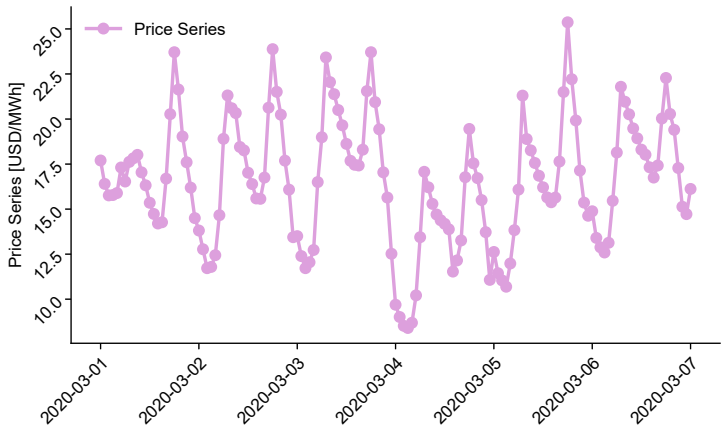

Figure 7-4: Price series in NYISO (with markers)

The results are compared and shown in Figure 7-5.

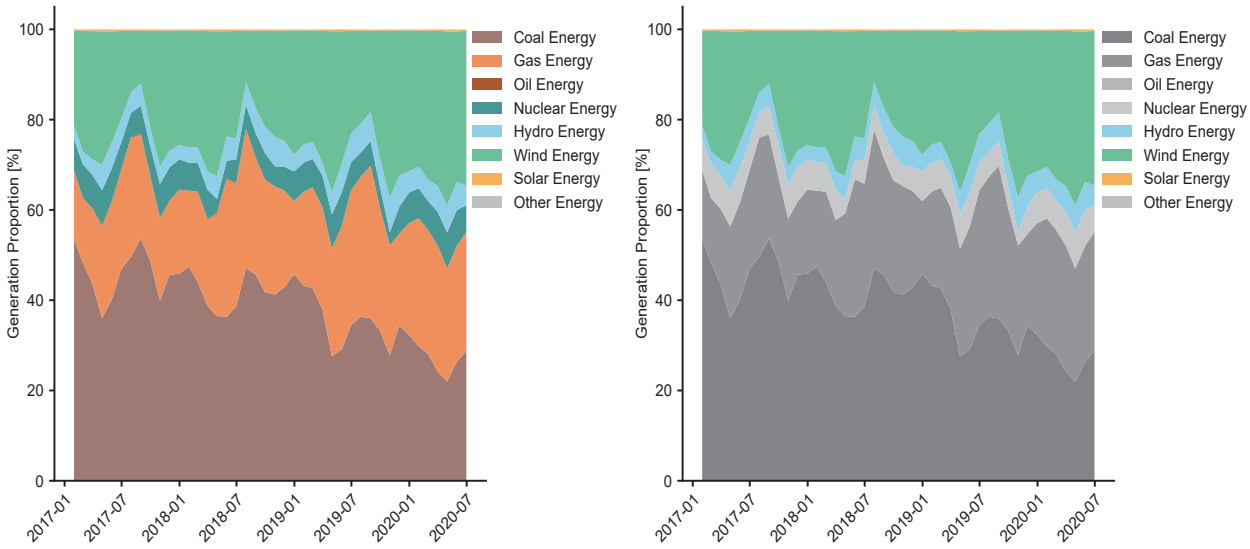

Figure 7-5: Generation mix in SPP with different color configurations. The right panel has only remained bright colors for renewable energy.

8 Acknowledgments

The CoVEMDA Project is supported in part by the Student Research Training Program of Tsinghua University. We would like to acknowledge all experts for their suggestions during Tsinghua University Challenge Cup Competition. We are also grateful for the support of EILAB from Department of Electrical Engineering, Tsinghua University.

Appendix A Extended API List

The main body of this manual only introduces the usage of integration module. Although this module has successfully handled many typical use cases, it may still lack enough flexibility for user-defined extensions. This is the major motivation that we should turn to low- or high-level functions, and customize new types of approaches that have not been covered by the existing integration module.

In fact, the integration module is also formulated by calling those low- or high-level functions, so there will be no compatible issues about mixing up the codes within different modules. At the same time, the integration functions may provide a good starting point to initialize any new features.

High flexibility and efficiency could be expected when using those low- or high-level functions, but users are required to get familiar with the toolbox workflow. Below is a full list of low- and high-level functions or classes:

`data_structure.py`

- `covemda.data_structure.DataFormatter`
  The basic data format in CoVEMDA. This class defines and implements the so-called wide data frame, which is an advanced version of pd.DataFrame. There are over 20 specialized methods developed in this class.
- `covemda.data_structure.merge_by_date`
  Merge several DataFormatters by their row indices (date), and returns a new wider DataFormatter of the merging result.
- `covemda.data_structure.merge_by_column`
  Merge several DataFormatters by their column indices, and returns a new longer DataFormatter of the merging result.

`baseline_estimator.py`

- `covemda.baseline_estimator.BaselineEstimator`
  The basic estimator class for baseline analysis. A standardized estimation framework is constructed with target dates, observations, and baselines.
- `covemda.baseline_estimator.DateAlignmentBaseline`
  The date alignment estimator class for baseline analysis.
- `covemda.baseline_estimator.WeekAlignmentBaseline`
  The week alignment estimator class for baseline analysis.
- `covemda.baseline_estimator.MonthAlignmentBaseline`
  The month alignment estimator class for baseline analysis.
- `covemda.baseline_estimator.WeeklyTrendBaseline`
  The weekly trend estimator class for baseline analysis. The baseline is selected to be the weekly trend which is derived by calculating the weekly moving average values.
- `covemda.baseline_estimator.MonthlyTrendBaseline`
  The weekly trend estimator class for baseline analysis. This class is working on monthly statistics data, and uses ARIMA(2, 0, 1) to estimate the monthly trend.
- `covemda.baseline_estimator.TempSensitiveTrendBaseline`
  The temperature-sensitive trend estimator class for baseline analysis. This class is useful to eliminate the potential impacts from temperature. It uses a quadratic regression formula (data = k_1 + k_2 tmpc + k_3 tmpc^2) to detrend the raw data.

- `covemda.baseline_estimator.BackcastBaseline`
  The basic backcast estimator class for baseline analysis. This class is more advanced and complicated as it uses a deep learning model to eliminate a group of influencing factors (specified by users). Allow using pretrained models for a quick start.

- `covemda.baseline_estimator.BackcastDemandBaseline`
  The demand backcast estimator class for baseline analysis. This class is specifically focused on estimating the demand baselines, and the deep learning model is instantiated as a multiple-level, ReLU-activated neural network.

- `covemda.baseline_estimator.DistributionIndexBaseline`
  The distribution index estimator class for baseline analysis. Often, calculating or updating the distribution indices may be a bit slow.

- `covemda.baseline_estimator.DistributionDistanceBaseline`
  The distributional distance estimator class for baseline analysis. The baselines in this class are displayed by histogram charts, which is different from other classes.

`regressor.py`

- `covemda.regressor.Regressor`
  The basic regressor class for regression analysis. A standardized regression framework is constructed with train/test data and models.

- `covemda.regressor.LearningRegressor`
  The basic regressor class for machine learning methods. This class inherits from Regressor and adds the input/output scalars to enable feature scaling.

- `covemda.regressor.OrdinaryLeastSquares`
  The OLS class to conduct general (multivariate) linear regressions. A popular trick can let users consider nonlinear terms in their formulas by using nonlinear transformation on data while keeping the regression formula linear.

- `covemda.regressor.VectorAutoregression`
  The VAR class to conduct general (multivariate) time-series regressions. A series of model verification and post-estimation analysis features are included.

- `covemda.regressor.DataPrecheck`
  The time-series data pre-check class. This class involves a few pre-estimation preparation tests for VAR regression.

- `covemda.regressor.AutoregressiveIntegratedMovingAverage`
  The ARIMA class to conduct general (univariate) time-series regressions.

`visualizer.py`

- `covemda.visualizer.Visualizer`
  The basic visualizer class to automate plotter configurations. A standardized plotter framework is constructed with adaptive and concise settings.

- `covemda.visualizer.VisualizerPlus`
  The improved visualizer class with big events and configuration recorders. Most high-level visualizers are based on this class.

- `covemda.visualizer.DemandVisualizer`
  The demand visualizer class to display the demand series and daily demand profiles.

- `covemda.visualizer.GenerationVisualizer`
  The generation visualizer class to display the generation mix, renewable shares, and duck curves.

- `covemda.visualizer.PriceVisualizer`
  The price visualizer class to display the price series and price distributions.
- `covemda.visualizer.WeatherVisualizer`
  The weather visualizer class to display different weather observation series.
- `covemda.visualizer.MobilityVisualizer`
  The mobility visualizer class to display different mobility series.
- `covemda.visualizer.CovidVisualizer`
  The public health visualizer class to display different pandemic-related statistics.

References

[1] W. H. Organization, "Coronavirus disease (covid-19) pandemic," https://www.who.int/emergencies/diseases/novel-coronavirus-2019, 2021, retrieved June 8, 2021.

[2] K. T. Gillingham, C. R. Knittel, J. Li, M. Ovaere, and M. Reguant, "The short-run and long-run effects of covid-19 on energy and the environment," *Joule*, vol. 4, no. 7, pp. 1337–1341, 2020.

[3] H. Zhong, Z. Tan, Y. He, L. Xie, and C. Kang, "Implications of covid-19 for the electricity industry: A comprehensive review," *CSEE Journal of Power and Energy Systems*, vol. 6, no. 3, pp. 489–495, 2020.

[4] R. M. Elavarasan, G. Shafiullah, K. Raju, V. Mudgal, M. T. Arif, T. Jamal, S. Subramanian, V. S. Balaguru, K. Reddy, and U. Subramaniam, "Covid-19: Impact analysis and recommendations for power sector operation," *Applied energy*, vol. 279, p. 115739, 2020.

[5] X. Xu and C.-f. Chen, "Energy efficiency and energy justice for us low-income households: An analysis of multifaceted challenges and potential," *Energy Policy*, vol. 128, pp. 763–774, 2019.

[6] G. Ruan, D. Wu, X. Zheng, H. Zhong, C. Kang, M. A. Dahleh, S. Sivaranjani, and L. Xie, "A cross-domain approach to analyzing the short-run impact of covid-19 on the us electricity sector," *Joule*, vol. 4, no. 11, pp. 2322–2337, 2020.

[7] G. Ruan, J. Wu, H. Zhong, Q. Xia, and L. Xie, "Quantitative assessment of us bulk power systems and market operations during the covid-19 pandemic," *Applied Energy*, vol. 286, p. 116354, 2021.

[8] T. K. Wijaya, M. Vasirani, and K. Aberer, "When bias matters: An economic assessment of demand response baselines for residential customers," *IEEE Transactions on Smart Grid*, vol. 5, no. 4, pp. 1755–1763, 2014.

[9] S. Mohajeryami, M. Doostan, A. Asadinejad, and P. Schwarz, "Error analysis of customer baseline load (cbl) calculation methods for residential customers," *IEEE Transactions on Industry Applications*, vol. 53, no. 1, pp. 5–14, 2016.

[10] L. Gosink, K. Bensema, T. Pulsipher, H. Obermaier, M. Henry, H. Childs, and K. I. Joy, "Characterizing and visualizing predictive uncertainty in numerical ensembles through bayesian model averaging," *IEEE transactions on visualization and computer graphics*, vol. 19, no. 12, pp. 2703–2712, 2013.

[11] J. A. Carrillo and G. Toscani, "Wasserstein metric and large–time asymptotics of nonlinear diffusion equations," in *New Trends in Mathematical Physics: In Honour of the Salvatore Rionero 70th Birthday*. World Scientific, 2004, pp. 234–244.

[12] L. V. Kantorovich, "On the translocation of masses," *Journal of mathematical sciences*, vol. 133, no. 4, pp. 1381–1382, 2006.

[13] A. Rzhetsky and M. Nei, "Statistical properties of the ordinary least-squares, generalized least-squares, and minimum-evolution methods of phylogenetic inference," *Journal of molecular evolution*, vol. 35, no. 4, pp. 367–375, 1992.

[14] C. A. Sims, "Macroeconomics and reality," *Econometrica: journal of the Econometric Society*, pp. 1–48, 1980.

[15] J. H. Stock and M. W. Watson, "Vector autoregressions," *Journal of Economic perspectives*, vol. 15, no. 4, pp. 101–115, 2001.

[16] Y. Liu, M. C. Roberts, and R. Sioshansi, "A vector autoregression weather model for electricity supply and demand modeling," *Journal of Modern Power Systems and Clean Energy*, vol. 6, no. 4, pp. 763–776, 2018.

[17] Y.-W. Cheung and K. S. Lai, "Practitioners corner: Lag order and critical values of a modified dickey-fuller test," *Oxford Bulletin of Economics and Statistics*, vol. 57, no. 3, pp. 411–419, 1995.

[18] D. B. Rubin, "Basic concepts of statistical inference for causal effects in experiments and observational studies," *Course material in Quantitative Reasoning*, vol. 33, 2003.

[19] "Deep learning toolbox user's guide," *The Mathworks, Inc.*, retrieved June 23, 2021 from https://www.mathworks.com/help/pdf_doc/deeplearning/nnet_ug.pdf.

[20] "Statistics and machine learning toolbox user's guide," *The MathWorks, Inc.*, retrieved June 23, 2021 from https://www.mathworks.com/help/pdf_doc/stats/stats.pdf.