orbit

Release 1.1.4.4

Edwin Ng, Steve Yang, Zhishi Wang, Yifeng Wu, Jing Pan

Mar 15, 2024
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 About Orbit</td>
<td>1</td>
</tr>
<tr>
<td>2 Installation</td>
<td>3</td>
</tr>
<tr>
<td>3 Quick Start</td>
<td>5</td>
</tr>
<tr>
<td>4 Methods of Estimations and Predictions</td>
<td>9</td>
</tr>
<tr>
<td>5 Randomness Control and Reproducible Results</td>
<td>17</td>
</tr>
<tr>
<td>6 Using Pyro for Estimation</td>
<td>21</td>
</tr>
<tr>
<td>7 Damped Local Trend (DLT)</td>
<td>25</td>
</tr>
<tr>
<td>8 Local Global Trend (LGT)</td>
<td>33</td>
</tr>
<tr>
<td>9 Regression Priors in DLT</td>
<td>39</td>
</tr>
<tr>
<td>10 Regression Penalties in DLT</td>
<td>45</td>
</tr>
<tr>
<td>11 Handling Missing Response</td>
<td>55</td>
</tr>
<tr>
<td>12 Kernel-based Time-varying Regression - Part I</td>
<td>61</td>
</tr>
<tr>
<td>13 Kernel-based Time-varying Regression - Part II</td>
<td>69</td>
</tr>
<tr>
<td>14 Kernel-based Time-varying Regression - Part III</td>
<td>81</td>
</tr>
<tr>
<td>15 Kernel-based Time-varying Regression - Part IV</td>
<td>89</td>
</tr>
<tr>
<td>16 Prediction Decomposition</td>
<td>99</td>
</tr>
<tr>
<td>17 Model Diagnostics</td>
<td>103</td>
</tr>
<tr>
<td>18 Backtest</td>
<td>109</td>
</tr>
<tr>
<td>19 WBIC/BIC</td>
<td>119</td>
</tr>
<tr>
<td>20 EDA Utilities</td>
<td>129</td>
</tr>
<tr>
<td>21 Simulation Data</td>
<td>133</td>
</tr>
<tr>
<td>22 Other Utilities</td>
<td>139</td>
</tr>
</tbody>
</table>
ABOUT ORBIT

Orbit is a Python package for Bayesian time series modeling and inference. It provides a familiar and intuitive initialize-fit-predict interface for working with time series tasks, while utilizing probabilistic programing languages under the hood.

Currently, it supports the following models:

- Damped Local Trend (DLT)
- Exponential Smoothing (ETS)
- Local Global Trend (LGT)
- Kernel-based Time-varying Regression (KTR)

It also supports the following sampling methods for model estimation:

- Markov-Chain Monte Carlo (MCMC) as a full sampling method
- Maximum a Posteriori (MAP) as a point estimate method
- Stochastic Variational Inference (SVI) as a hybrid-sampling method on approximate distribution

Under the hood, the package is leveraging probabilistic program such as pyro and cmdstanpy.

1.1 Citation

To cite Orbit in publications, refer to the following whitepaper:

Orbit: Probabilistic Forecast with Exponential Smoothing

Bibtex:

```bibtex
@misc{ng2020orbit,
    title={Orbit: Probabilistic Forecast with Exponential Smoothing},
    author={Edwin Ng, Zhishi Wang, Huigang Chen, Steve Yang, Slawek Smyl},
    year={2020},
    eprint={2004.08492},
    archivePrefix={arXiv},
    primaryClass={stat.CO}
}
```
1.2 Blog Post

1. Introducing Orbit, An Open Source Package for Time Series Inference and Forecasting [Link] 2. The New Version of Orbit (v1.1) is Released: The Improvements, Design Changes, and Exciting Collaborations [Link]
Install from PyPi:

```
pip install orbit-ml
```

Install from GitHub:

```
$ git clone https://github.com/uber/orbit.git
$ cd orbit
$ pip install -r requirements.txt
$ pip install .
```
This session covers topics:

- a forecast task on iclaims dataset
- a simple Bayesian ETS Model using CmdStanPy
- posterior distribution extraction
- tools to visualize the forecast

### 3.1 Load Library

```
[1]: %matplotlib inline
    import matplotlib.pyplot as plt

    import orbit
    from orbit.utils.dataset import load_iclaims
    from orbit.models import ETS
    from orbit.diagnostics.plot import plot_predicted_data
```

```
[2]: print(orbit.__version__)
1.1.4.4
```

### 3.2 Data

The *iclaims* data contains the weekly initial claims for US unemployment (obtained from Federal Reserve Bank of St. Louis) benefits against a few related Google trend queries (unemploy, filling and job) from Jan 2010 - June 2018. This aims to demo a similar dataset from the Bayesian Structural Time Series (BSTS) model (Scott and Varian 2014).

Note that the numbers are log-log transformed for fitting purpose and the discussion of using the regressors can be found in later chapters with the Damped Local Trend (DLT) model.

```
[3]: # load data
df = load_iclaims()
date_col = 'week'
response_col = 'claims'
df.dtypes
```
3.3 Forecasting Using Orbit

Orbit aims to provide an intuitive `initialize-fit-predict` interface for working with forecasting tasks. Under the hood, it utilizes probabilistic modeling API such as CmdStanPy and Pyro. We first illustrate a Bayesian implementation of Rob Hyndman’s ETS (which stands for Error, Trend, and Seasonality) Model (Hyndman et. al, 2008) using CmdStanPy.

```python
ets = ETS(  
    response_col=response_col,  
    date_col=date_col,  
    seasonality=52,  
    seed=2024,  
    estimator="stan-mcmc",  
    stan_mcmc_args={'show_progress': False},
)
```

```bash
%%time
ets.fit(df=train_df)
```

3.4 Extract and Analyze Posterior Samples

Users can use .get_posterior_samples() to extract posterior samples in an OrderedDict format.

```python
[10]: posterior_samples = ets.get_posterior_samples()
persistent_samples.keys()
```

The extracted parameters posteriors are pretty much compatible diagnostic with arviz. To do that, users can set permute=False to preserve chain information.

```python
[11]: import arviz as az

posterior_samples = ets.get_posterior_samples(permute=False)

# example from https://arviz-devs.github.io/arviz/index.html
az.style.use("arviz-darkgrid")
az.plot_pair(
    posterior_samples,
```

(continues on next page)
For more details in model diagnostics visualization, there is a subsequent section dedicated to it.
There are three methods supported in Orbit model parameters estimation (a.k.a posteriors in Bayesian).

1. Maximum a Posteriori (MAP)
2. Markov Chain Monte Carlo (MCMC)
3. Stochastic Variational Inference (SVI)

This session will cover the first two: MAP and MCMC which mainly uses CmdStanPy at the back end. Users can simply can leverage the args estimator to pick the method (stan-map and stan-mcmc). The details will be covered by the sections below. The SVI method is calling Pyro by specifying estimator='pyro-svi'. However, it is covered by a separate session.

### 4.1 Data and Libraries

```python
[1]: %matplotlib inline
import matplotlib.pyplot as plt
import orbit
from orbit.utils.dataset import load_iclaims
from orbit.models import ETS
from orbit.diagnostics.plot import plot_predicted_data, plot_predicted_components

[2]: print(orbit.__version__)
1.1.4.4

[3]: # load data
df = load_iclaims()
test_size = 52
train_df = df[:-test_size]
test_df = df[-test_size:]
response_col = 'claims'
date_col = 'week'
```
4.2 Maximum a Posteriori (MAP)

To use MAP method, one can simply specify `estimator='stan-map'` when instantiating a model. The advantage of MAP estimation is a faster computational speed. In MAP, the uncertainty is mainly generated the noise process with bootstrapping. However, the uncertainty would not cover parameters variance as well as the credible interval from seasonality or other components.

```python
[4]: %time
ets = ETS(
    response_col=response_col,
    date_col=date_col,
    estimator='stan-map',
    seasonality=52,
    seed=8888,
)
ets.fit(df=train_df)
predicted_df = ets.predict(df=test_df)
```

2024-03-13 21:57:02 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.

CPU times: user 8.66 ms, sys: 16.4 ms, total: 25.1 ms
Wall time: 349 ms

```python
[5]: _ = plot_predicted_data(train_df, predicted_df, date_col, response_col, title='Prediction with ETSMAP')
```

To have the uncertainty from MAP, one can specify `n_bootstrap_draws`. The default is set to be -1 which mutes the bootstrap process. Users can also specify a particular percentiles to report prediction intervals by passing list of percentiles with args `prediction_percentiles`.

```python
[6]: # default: [10, 90]
prediction_percentiles=[10, 90]
```

(continues on next page)
ets = ETS(
    response_col=response_col,
    date_col=date_col,
    estimator='stan-map',
    seasonality=52,
    seed=8888,
    n_bootstrap_draws=1e4,
    prediction_percentiles=prediction_percentiles,
)

ets.fit(df=train_df)
predicted_df = ets.predict(df=test_df)

_ = plot_predicted_data(train_df, predicted_df, date_col, response_col,
    prediction_percentiles=prediction_percentiles,
    title='Prediction with ETS-MAP')

One can access the posterior estimated by calling the .get_point_posteriors(). The outcome from this function is a dict of dict where the top layer stores the type of point estimate while the second layer stores the parameters labels and values.

[7]: pt_posteriors = ets.get_point_posteriors()['map']
    pt_posteriors.keys()

[7]: dict_keys(['l', 'lev_sm', 'obs_sigma', 's', 'sea_sm'])

In general, the first dimension is just 1 as a point estimate for each parameter. The rest of the dimension will depend on the dimension of parameter itself.

[8]: lev = pt_posteriors['l']
    lev.shape

4.2. Maximum a Posteriori (MAP)
4.3 MCMC

To use MCMC method, one can specify `estimator='stan-mcmc'` (the default) when instantiating a model. Compared to MAP, it usually takes longer time to fit. As the model now fitted as a full Bayesian model where No-U-Turn Sampler (NUTS) (Hoffman and Gelman 2011) is carried out under the hood. By default, a full sampling on posteriors distribution is conducted. Hence, full distribution of the predictions are always provided.

4.3.1 MCMC - Full Bayesian Sampling

```python
%time
et = ETS(
    response_col=response_col,
    date_col=date_col,
    estimator='stan-mcmc',
    seasonality=52,
    seed=8888,
    num_warmup=400,
    num_sample=400,
    stan_mcmc_args={'show_progress': False},
)
et.fit(df=train_df)
predicted_df = et.predict(df=test_df)
```

```bash
2024-03-13 21:57:03 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8, →
˓temperature: 1.000, warmups (per chain): 100 and samples(per chain): 100.
˓CPU times: user 141 ms, sys: 72.5 ms, total: 213 ms
˓Wall time: 593 ms
```

```python
_ = plot_predicted_data(train_df, predicted_df, date_col, response_col, title=˓'
˓'Prediction with ETS-Full Bayesian')
```
Also, users can request prediction with credible intervals of each component.

```python
predicted_df = ets.predict(df=df, decompose=True)
plot_predicted_components(predicted_df, date_col=date_col,
                          plot_components=['prediction', 'trend', 'seasonality'])
```

Just like the MAPForecaster, one can also access the posterior samples by calling the function `get_posterior_samples()`.
As mentioned, in MCMC (Full Bayesian) models, the first dimension reflects the sample size.

```
lev = posterior_samples['l']
lev.shape
```

(400, 391)

### 4.3.2 MCMC - Point Estimation

Users can also choose to derive point estimates via MCMC by specifying point_method as mean or median via the call of `.fit`. In that case, posteriors samples are first aggregated by mean or median and store as a point estimate for final prediction. Just like other point estimate, users can specify n_bootstrap_draws to report uncertainties.

```
%%time
ets = ETS(
    response_col=response_col,
    date_col=date_col,
    estimator='stan-mcmc',
    seasonality=52,
    seed=8888,
    n_bootstrap_draws=1e4,
    stan_mcmc_args={'show_progress': False},
)
```

# specify point_method e.g. 'mean', 'median'
```
ets.fit(df=train_df, point_method='mean')
predicted_df = ets.predict(df=test_df)
```

```
```

CPU times: user 202 ms, sys: 88.6 ms, total: 290 ms
Wall time: 764 ms

```
_= plot_predicted_data(train_df, predicted_df, date_col, response_col,
    title='Prediction with Point(Mean) Estimation')
```
One can always access the point estimated posteriors by `.get_point_posteriors()` (including the cases fitting the parameters through MCMC).

```python
[16]: ets.get_point_posteriors()['mean'].keys()
[16]: dict_keys(['l', 'lev_sm', 'obs_sigma', 's', 'sea_sm'])

[17]: ets.get_point_posteriors()['median'].keys()
[17]: dict_keys(['l', 'lev_sm', 'obs_sigma', 's', 'sea_sm'])
```
CHAPTER FIVE

RANDOMNESS CONTROL AND REPRODUCIBLE RESULTS

There are randomness involved in the random initialization, sampling and bootstrapping process. Some of them with sufficient condition such as converging status and large amount of samples, can be fixed even without a fixed seed. However, they are not guaranteed. Two settings needed to allow fully reproducible results and will be demoed from this session:

1. fix the seed on fitting
2. fix the seed on prediction

5.1 Data and Libraries

```python
[1]: import numpy as np

import orbit
from orbit.models import DLT
from orbit.utils.dataset import load_iclaims

[2]: print(orbit.__version__)
1.1.4.4

[3]: df = load_iclaims()
df.head(5)

[3]:
<table>
<thead>
<tr>
<th>week</th>
<th>claims</th>
<th>trend.unemploy</th>
<th>trend.filling</th>
<th>trend.job</th>
<th>sp500</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2010-01-03</td>
<td>13.386595</td>
<td>0.219882</td>
<td>-0.318452</td>
<td>0.117500</td>
<td>-0.417633</td>
</tr>
<tr>
<td>1 2010-01-10</td>
<td>13.624218</td>
<td>0.219882</td>
<td>-0.194838</td>
<td>0.168794</td>
<td>-0.425480</td>
</tr>
<tr>
<td>2 2010-01-17</td>
<td>13.398741</td>
<td>0.236143</td>
<td>-0.292477</td>
<td>0.117500</td>
<td>-0.465229</td>
</tr>
<tr>
<td>3 2010-01-24</td>
<td>13.137549</td>
<td>0.203353</td>
<td>-0.194838</td>
<td>0.106918</td>
<td>-0.481751</td>
</tr>
<tr>
<td>4 2010-01-31</td>
<td>13.196760</td>
<td>0.134360</td>
<td>-0.242466</td>
<td>0.074483</td>
<td>-0.488929</td>
</tr>
</tbody>
</table>

vix
| 0 | 0.122654 |
| 1 | 0.110445 |
| 2 | 0.532339 |
| 3 | 0.428645 |
| 4 | 0.487404 |
```
5.2 Fixing Seed in Fitting

By default, the seed supplied during the initialization step is fixed. This allows fully reproducible posteriors samples by default. For other purpose, users can randomize the seed. Nonetheless, this process usually assumes stable result with or without a fixed seed. Otherwise, convergence alert should be raised.

With different seeds, results should be closed but not identical:

```
[4]: dlt1 = DLT(response_col='claims', date_col='week', seed=2021, estimator='stan-map', n_˓→bootstrap_draws=1e3)
   dlt2 = DLT(response_col='claims', date_col='week', seed=2020, estimator='stan-map', n_˓→bootstrap_draws=1e3)
   
   dlt1.fit(df);
   dlt2.fit(df);
   
   lev1 = dlt1.get_point_posteriors()['map']['l']
   lev2 = dlt2.get_point_posteriors()['map']['l']

   2024-03-13 22:01:13 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
   2024-03-13 22:01:13 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.

[5]: np.all(lev1 == lev2)
[5]: False

[6]: np.allclose(lev1, lev2, rtol=1e-3)
[6]: True
```

With fixed seeds, results should be identical:

```
[7]: dlt1 = DLT(response_col='claims', date_col='week', seed=2020, estimator='stan-map', n_˓→bootstrap_draws=1e3)
   dlt2 = DLT(response_col='claims', date_col='week', seed=2020, estimator='stan-map', n_˓→bootstrap_draws=1e3)
   
   dlt1.fit(df);
   dlt2.fit(df);
   
   lev1 = dlt1.get_point_posteriors()['map']['l']
   lev2 = dlt2.get_point_posteriors()['map']['l']

   2024-03-13 22:01:13 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
   2024-03-13 22:01:13 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.

[8]: np.all(lev1 == lev2)
[8]: True
```

In sampling algorithm, users should expect identical posteriors with fixed seed:

```
[9]: dlt_mcmc1 = DLT(response_col='claims', date_col='week', seed=2020, estimator='stan-mcmc', ˓→stan_mcmc_args={'show_progress': False})
   dlt_mcmc2 = DLT(response_col='claims', date_col='week', seed=2020, estimator='stan-mcmc',
   (continues on next page)
stan_mcmc_args={"show_progress": False})

dlt_mcmc1.fit(df);
dlt_mcmc2.fit(df);

lev_mcmc1 = dlt_mcmc1.get_posterior_samples()['l'
lev_mcmc2 = dlt_mcmc2.get_posterior_samples()['l'

2024-03-13 22:01:13 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,
temperature: 1.000, warmups (per chain): 225 and samples(per chain): 25.
2024-03-13 22:01:19 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,
temperature: 1.000, warmups (per chain): 225 and samples(per chain): 25.

[10]: print(lev_mcmc1.shape)
print(lev_mcmc2.shape)
np.all(lev1 == lev2)

(100, 443)
(100, 443)

[10]: True

5.3 Fixing Seed in Prediction

Unlike the fitting process, the seed in prediction is set to be random by default unless users provided a fixed seed. Once a fixed seed provided through the args in .predict(). Users should expect identical result.

[11]: # check with MAP estimator
pred1 = dlt1.predict(df, seed=2020)['prediction'].values
pred2 = dlt2.predict(df, seed=2020)['prediction'].values
np.all(pred1 == pred2)

[11]: True

[12]: # check with MCMC estimator
pred1 = dlt_mcmc1.predict(df, seed=2020)['prediction'].values
pred2 = dlt_mcmc2.predict(df, seed=2020)['prediction'].values
np.all(pred1 == pred2)

[12]: True
Currently we are still experimenting with Pyro and support Pyro only in LGT and KTR models.

Pyro is a flexible, scalable deep probabilistic programming library built on PyTorch. Pyro was originally developed at Uber AI and is now actively maintained by community contributors, including a dedicated team at the Broad Institute.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import orbit
from orbit.models import LGT
from orbit.diagnostics.plot import plot_predicted_data
from orbit.diagnostics.plot import plot_predicted_components
from orbit.utils.dataset import load_iclaims
from orbit.constants.palette import OrbitPalette

warnings.filterwarnings('ignore')

print(orbit.__version__)
1.1.4.4

df = load_iclaims()

test_size=52
train_df=df[:-test_size]
test_df=df[-test_size:]
```
6.1 VI Fit and Predict

Although Pyro provides a variety of ways to optimize/sample posteriors. Currently, we only support Stochastic Variational Inference (SVI). For details, please refer to this doc.

To use SVI for LGT, specify estimator as `pyro-svi`.

```python
5: lgt_vi = LGT(
    response_col='claims',
    date_col='week',
    seasonality=52,
    seed=8888,
    estimator='pyro-svi',
    num_steps=101,
    num_sample=300,
    # trigger message per 50 steps
    message=50,
    learning_rate=0.1,
)
```

```python
6: %time
    lgt_vi.fit(df=train_df)
    2024-03-13 21:59:52 - orbit - INFO - Using SVI (Pyro) with steps: 101, samples: 300,...
    → learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
    2024-03-13 21:59:53 - orbit - INFO - step 0 loss = 658.91, scale = 0.11635
    INFO:orbit:step 0 loss = 658.91, scale = 0.11635
    2024-03-13 21:59:56 - orbit - INFO - step 50 loss = -432, scale = 0.48623
    INFO:orbit:step 50 loss = -432, scale = 0.48623
    2024-03-13 21:59:58 - orbit - INFO - step 100 loss = -444.07, scale = 0.34976
    INFO:orbit:step 100 loss = -444.07, scale = 0.34976
    CPU times: user 6.13 s, sys: 561 ms, total: 6.69 s
    Wall time: 6.24 s
6: <orbit.forecaster.svi.SVIForecaster at 0x2921fa850>
```

```python
7: predicted_df = lgt_vi.predict(df=test_df)
8: _ = plot_predicted_data(training_actual_df=train_df, predicted_df=predicted_df,
                          date_col=lgt_vi.date_col, actual_col=lgt_vi.response_col,
                          test_actual_df=test_df)
```
We can also extract the ELBO loss from the training metrics.

[9]: loss_elbo = lgt_vi.get_training_metrics()['loss_elbo']

[10]: steps = np.arange(len(loss_elbo))
plt.subplots(1, 1, figsize=(8, 4))
plt.plot(steps, loss_elbo, color=OrbitPalette.BLUE.value)
plt.title('ELBO Loss per Step')

[10]: Text(0.5, 1.0, 'ELBO Loss per Step')
DAMPED LOCAL TREND (DLT)

This section covers topics including:

- DLT model structure
- DLT global trend configurations
- Adding regressors in DLT
- Other configurations

1: %matplotlib inline
import matplotlib.pyplot as plt
import orbit
from orbit.models import DLT
from orbit.diagnostics.plot import plot_predicted_data, plot_predicted_components
from orbit.utils.dataset import load_iclaims

2: print(orbit.__version__)
    1.1.4.4

7.1 Model Structure

DLT is one of the main exponential smoothing models we support in orbit. Performance is benchmarked with M3 monthly, M4 weekly dataset and some Uber internal dataset (Ng and Wang et al., 2020). The model is a fusion between the classical ETS (Hyndman et. al., 2008) with some refinement leveraging ideas from Rltg (Smyl et al., 2019). The model has a structural forecast equations

\[
\begin{align*}
    y_t &= \mu_t + s_t + r_t + \epsilon_t \\
    \mu_t &= g_t + l_{t-1} + \theta b_{t-1} \\
    \epsilon_t &\sim \text{Student}(\nu, 0, \sigma) \\
    \sigma &\sim \text{HalfCauchy}(0, \gamma_0)
\end{align*}
\]

with the update process

\[
\begin{align*}
    g_t &= D(t) \\
    l_t &= \rho_l(y_t - g_t - s_t - r_t) + (1 - \rho_l)(l_{t-1} + \theta b_{t-1}) \\
    b_t &= \rho_b(l_t - l_{t-1}) + (1 - \rho_b)\theta b_{t-1} \\
    s_{t+m} &= \rho_s(y_t - l_t - r_t) + (1 - \rho_s)s_t \\
    r_t &= \sum_j \beta_j x_{jt}
\end{align*}
\]
One important point is that using $y_t$ as a log-transformed response usually yield better result, especially we can interpret such log-transformed model as a *multiplicative form* of the original model. Besides, there are two new additional components compared to the classical damped ETS model:

1. $D(t)$ as the deterministic trend process
2. $r$ as the regression component with $x$ as the regressors

```python
# load log-transformed data
df = load_iclaims()
response_col = 'claims'
date_col = 'week'
```

**Note**

Just like LGT model, we also provide MAP and MCMC (full Bayesian) methods for DLT model (by specifying `estimator='stan-map'` or `estimator='stan-mcmc'` when instantiating a model).

MCMC is usually more robust but may take longer time to train. In this notebook, we will use the MAP method for illustration purpose.

### 7.2 Global Trend Configurations

There are a few choices of $D(t)$ configured by `global_trend_option`:

1. linear (default)
2. loglinear
3. flat
4. logistic

Mathematically, they are expressed as such,

**1. Linear:**

$$D(t) = \delta_{\text{intercept}} + \delta_{\text{slope}} \cdot t$$

**2. Log-linear:**

$$D(t) = \delta_{\text{intercept}} + \ln(\delta_{\text{slope}} \cdot t)$$

**3. Logistic:**

$$D(t) = L + \frac{U-L}{1+e^{-\delta_{\text{slope}} \cdot t}}$$

**4. Flat:**

$$D(t) = \delta_{\text{intercept}}$$

where $\delta_{\text{intercept}}$ and $\delta_{\text{slope}}$ are fitted parameters and $t$ is rescaled time-step between 0 and $T$ (=number of time steps).

To show the difference among these options, their predictions are projected in the charts below. Note that the default is set to `linear` which is also used in the benchmarking process mentioned previously. During prediction, a convenient function `make_future_df()` is called to generate future data frame (ONLY applied when you don’t have any regressors!).
7.2.1 linear global trend

```python
[dlt := DLT(
    response_col=response_col,
    date_col=date_col,
    estimator='stan-map',
    seasonality=52,
    seed=8888,
    global_trend_option='linear',
    # for prediction uncertainty
    n_bootstrap_draws=1000,
)

dlt.fit(df)

test_df = dlt.make_future_df(periods=52 * 10)
predicted_df = dlt.predict(test_df)
_= plot_predicted_data(df, predicted_df, date_col, response_col, title='DLT Linear→Global Trend')
```

2024-03-13 23:36:44 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.

![DLT Linear Global Trend](image)

CPU times: user 1.64 s, sys: 389 ms, total: 2.03 s
Wall time: 871 ms

```python
[dlt := DLT(
    response_col=response_col,
    date_col=date_col,
    estimator='stan-mcmc',
    seasonality=52,
    seed=8888,
) (continues on next page)
```
global_trend_option='linear',
# for prediction uncertainty
n_bootstrap_draws=1000,
stan_mcmc_args={'show_progress': False},
)

dlt.fit(df, point_method="mean")


[5]: <orbit.forecaster.full_bayes.FullBayesianForecaster at 0x29d052b50>

One can use .get_posterior_samples() to extract the samples for all sampling parameters.

[6]: dlt.get_posterior_samples().keys()

[6]: dict_keys(['l', 'b', 'lev_sm', 'slp_sm', 'obs_sigma', 'nu', 'lt_sum', 's', 'sea_sm', 'gt_sum', 'gb', 'gl', 'loglk'])

[7]: %%time
# log-linear global trend
dlt = DLT(
    response_col=response_col,
    date_col=date_col,
    seasonality=52,
    estimator='stan-map',
    seed=8888,
    global_trend_option='loglinear',
    # for prediction uncertainty
    n_bootstrap_draws=1000,
)

dlt.fit(df)
# re-use the test_df generated above
predicted_df = dlt.predict(test_df)
_= plot_predicted_data(df, predicted_df, date_col, response_col, title='DLT Log-Linear → Global Trend')

2024-03-13 23:36:47 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
In logistic trend, users need to specify the args `global_floor` and `global_cap`. These args are with default 0 and 1.

### 7.2.2 logistic global trend

```
[8]: %%time

dlt = DLT(
    response_col=response_col,
    date_col=date_col,
    estimator='stan-map',
    seasonality=52,
    seed=8888,
    global_trend_option='logistic',
    global_cap=9999,
    global_floor=11.75,
    damped_factor=0.1,
    # for prediction uncertainty
    n_bootstrap_draws=1000,
)

dlt.fit(df)
predicted_df = dlt.predict(test_df)
ax = plot_predicted_data(df, predicted_df, date_col, response_col,
    title='DLT Logistic Global Trend', is_visible=False);
ax.axhline(y=11.75, linestyle='--', color='orange')
ax.figure
```
Note: Theoretically, the trend is bounded by the `global_floor` and `global_cap`. However, because of seasonality and regression, the predictions can still be slightly lower than the floor or higher than the cap.

### 7.2.3 flat trend

```python
%%time
dlt = DLT(
    response_col=response_col,
    date_col=date_col,
    estimator='stan-map',
    seasonality=52,
    seed=8888,
    global_trend_option='flat',
    # for prediction uncertainty
    n_bootstrap_draws=1000,
)

dlt.fit(df)
predicted_df = dlt.predict(test_df)
_ = plot_predicted_data(df, predicted_df, date_col, response_col, title='DLT Flat → Global Trend')
```

2024-03-13 23:36:48 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
7.3 Regression

You can also add regressors into the model by specifying `regressor_col`. This serves the purpose of nowcasting or forecasting when exogenous regressors are known such as events and holidays. Without losing generality, the interface is set to be

$$\beta_j \sim \mathcal{N}(\mu_j, \sigma_j^2)$$

where $\mu_j = 0$ and $\sigma_j = 1$ by default as a non-informative prior. These two parameters are set by the arguments `regressor_beta_prior` and `regressor_sigma_prior` as a list. For example,

```python
[10]: dlt = DLT(
    response_col=response_col,
    date_col=date_col,
    estimator='stan-mcmc',
    seed=8888,
    seasonality=52,
    regressor_col=['trend.unemploy', 'trend.filling'],
    regressor_beta_prior=[0.1, 0.3],
    regressor_sigma_prior=[0.5, 2.0],
    stan_mcmc_args={'show_progress': False},
)

dlt.fit(df)
predicted_df = dlt.predict(df, decompose=True)
plot_predicted_components(predicted_df, date_col);
```
One can also use \texttt{.get\_regression\_coefs} to extract the regression coefficients along with the confidence interval when posterior samples are available. The default lower and upper limits are set to be .05 and .95.

\begin{verbatim}
[11]: dlt.get_regression_coefs()

<table>
<thead>
<tr>
<th>regressor</th>
<th>regressor_sign</th>
<th>coefficient</th>
<th>coefficient_lower</th>
<th>coefficient_upper</th>
<th>Pr(coef &gt;= 0)</th>
<th>Pr(coef &lt; 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>trend.unemploy</td>
<td>Regular</td>
<td>0.052448</td>
<td>0.020843</td>
<td>0.076498</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>trend.filling</td>
<td>Regular</td>
<td>0.084483</td>
<td>0.012782</td>
<td>0.139546</td>
<td>0.99</td>
<td>0.01</td>
</tr>
</tbody>
</table>
\end{verbatim}

There are much more configurations on regression such as the regressors sign and penalty type. They will be discussed in subsequent sections.

### 7.3.1 High Dimensional and Fourier Series Regression

In case of high dimensional regression, users can consider fixing the smoothness with a relatively small levels smoothing values e.g. setting \texttt{level_sm_input=0.01}. This is particularly useful in modeling higher frequency time-series such as daily and hourly data using regression on Fourier series. Check out the \texttt{examples/} folder for the details.
LOCAL GLOBAL TREND (LGT)

In this section, we will cover:

- LGT model structure
- difference between DLT and LGT
- syntax to call LGT classes with different estimation methods

LGT stands for Local and Global Trend and is a refined model from Rlgt (Smyl et al., 2019). The main difference is that LGT is an additive form taking log-transformation response as the modeling response. This essentially converts the model into a multiplicative with some advantages (Ng and Wang et al., 2020). However, one drawback of this approach is that negative response values are not allowed due to the existence of the global trend term and because of that we start to deprecate the support of regression of this model.

```
[1]: %matplotlib inline

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import orbit
from orbit.models import LGT
from orbit.diagnostics.plot import plot_predicted_data
from orbit.diagnostics.plot import plot_predicted_components
from orbit.utils.dataset import load_iclaims

[2]: print(orbit.__version__)

1.1.4.4
```

### 8.1 Model Structure

\[ y_t = \mu_t + s_t + \epsilon_t \]

\[ \mu_t = t_{t-1} + \xi_1 b_{t-1} + \xi_2 t_{t-1} \]

\[ \epsilon_t \sim \text{Student}(\nu, 0, \sigma) \]

\[ \sigma \sim \text{HalfCauchy}(0, \gamma_0) \]
with the update process,

\[ l_t = \rho_l (y_t - s_t) + (1 - \rho_l)l_{t-1} \]

\[ b_t = \rho_b (l_t - l_{t-1}) + (1 - \rho_b)b_{t-1} \]

\[ s_{t+m} = \rho_s (y_t - l_t) + (1 - \rho_s)s_t \]

Unlike DLT model which has a deterministic trend, LGT introduces a hybrid trend where it consists of

- local trend takes on a fraction \( \xi_1 \) rather than a damped factor
- global trend is with a auto-regressive term \( \xi_2 \) and a power term \( \lambda \)

We will continue to use the iclaims data with 52 weeks train-test split.

```python
# load data
df = load_iclaims()
# define date and response column
date_col = 'week'
response_col = 'claims'
df.dtypes
test_size = 52
train_df = df[:-test_size]
test_df = df[-test_size:]
```

### 8.2 LGT Model

In orbit, we provide three methods for LGT model estimation and inferences, which are *MAP* *MCMC* (also providing the point estimate method, mean or median), which is also the default *SVI*

Orbit follows a sklearn style model API. We can create an instance of the Orbit class and then call its fit and predict methods.

In this notebook, we will only cover MAP and MCMC methods. Refer to this notebook for the pyro estimation.

#### 8.2.1 LGT - MAP

To use MAP, specify the estimator as `stan-map`.

```python
lgt = LGT(
    response_col=response_col,
date_col=date_col,
estimator='stan-map',
seasonality=seasonality,
seed=seed,
)
```

```
```

```python
%%time
lgt.fit(df=train_df)
```

```
CPU times: user 7.84 ms, sys: 9.53 ms, total: 17.4 ms
Wall time: 148 ms
```
8.2.2 LGT - MCMC

To use MCMC sampling, specify the estimator as `stan-mcmc` (the default).

- By default, full Bayesian samples will be used for the predictions: for each set of parameter posterior samples, the prediction will be conducted once and the final predictions are aggregated over all the results. To be specific, the final predictions will be the median (aka 50th percentile) along with any additional percentiles provided. One can use `.get_posterior_samples()` to extract the samples for all sampling parameters.

- One can also specify `point_method` (either `mean` or `median`) via `.fit` to have the point estimate: the parameter posterior samples are aggregated first (mean or median) then conduct the prediction once.

LGT - full

```python
[8]: lgt = LGT(
    response_col=response_col,
    date_col=date_col,
    seasonality=52,
    seed=2024,
    stan_mcmc_args={'show_progress': False},
)```
%time
lgt.fit(df=train_df)


CPU times: user 82.2 ms, sys: 94.3 ms, total: 177 ms
Wall time: 4.18 s

<orbit.forecaster.full_bayes.FullBayesianForecaster at 0x2a5dc7690>

predicted_df = lgt.predict(df=test_df)

predicted_df.tail(5)

<table>
<thead>
<tr>
<th>week</th>
<th>prediction_5</th>
<th>prediction</th>
<th>prediction_95</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>12.114602</td>
<td>12.250131</td>
<td>12.382320</td>
</tr>
<tr>
<td>48</td>
<td>12.058250</td>
<td>12.173431</td>
<td>12.272940</td>
</tr>
<tr>
<td>49</td>
<td>12.164898</td>
<td>12.253941</td>
<td>12.387880</td>
</tr>
<tr>
<td>50</td>
<td>12.138711</td>
<td>12.241891</td>
<td>12.362063</td>
</tr>
<tr>
<td>51</td>
<td>12.182641</td>
<td>12.284261</td>
<td>12.397172</td>
</tr>
</tbody>
</table>

lgt.get_posterior_samples().keys()

dict_keys(['l', 'b', 'lev_sm', 'slp_sm', 'obs_sigma', 'nu', 'lgt_sum', 'gt_pow', 'lt_coef', 'gt_coef', 's', 'sea_sm', 'loglk'])

_ = plot_predicted_data(training_actual_df=train_df, predicted_df=predicted_df, date_col=lgt.date_col, actual_col=lgt.response_col, test_actual_df=test_df, title='Prediction with LGTFull Model')
LGT - point estimate

```python
[14]: lgt = LGT(
    response_col=response_col,
    date_col=date_col,
    seasonality=52,
    seed=2024,
    stan_mcmc_args={'show_progress': False},
)

[15]: %time
    lgt.fit(df=train_df, point_method='mean')

2024-03-13 21:55:23 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,
→ temperature: 1.000, warmups (per chain): 225 and samples(per chain): 25.
CPU times: user 116 ms, sys: 66.6 ms, total: 182 ms
Wall time: 4.28 s

[15]: <orbit.forecaster.full_bayes.FullBayesianForecaster at 0x2a5e22e90>

[16]: predicted_df = lgt.predict(df=test_df)

[17]: predicted_df.tail(5)

<table>
<thead>
<tr>
<th>week</th>
<th>prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>12.210257</td>
</tr>
<tr>
<td>48</td>
<td>12.145213</td>
</tr>
<tr>
<td>49</td>
<td>12.239412</td>
</tr>
<tr>
<td>50</td>
<td>12.207138</td>
</tr>
<tr>
<td>51</td>
<td>12.253422</td>
</tr>
</tbody>
</table>

[18]: _ = plot_predicted_data(training_actual_df=train_df, predicted_df=predicted_df,
    date_col=lgt.date_col, actual_col=lgt.response_col,
    test_actual_df=test_df, title='Prediction with LGTAggregated Model')

8.2. LGT Model
Chapter 8. Local Global Trend (LGT)
This notebook demonstrates usage of priors in the regression analysis. The *iclaims* data will be used in demo purpose. Examples include

1. regression with default setting
2. regression with bounded priors for regression coefficients

Generally speaking, regression coefficients are more robust under full Bayesian sampling and estimation. The default setting `estimator='stan-mcmc'` will be used in this tutorial.

```python
[1]: %matplotlib inline

import matplotlib.pyplot as plt
import numpy as np

import orbit
from orbit.utils.dataset import load_iclaims
from orbit.models import DLT
from orbit.diagnostics.plot import plot_predicted_data

[2]: print(orbit.__version__)
1.1.4.4

### 9.1 US Weekly Initial Claims

Recall the *iclaims* dataset by previous section. In order to use this data to nowcast the US unemployment claims during COVID-19 period, the dataset is extended to Jan 2021 and the S&P 500 (^GSPC) and VIX Index historical data are attached for the same period.

The data is standardized and log-transformed for the model fitting purpose.

```python
[3]: # load data
df = load_iclaims(end_date='2021-01-03')
df = df[['week', 'claims', 'trend.unemploy', 'trend.job', 'sp500', 'vix']]
df = df[1:].reset_index(drop=True)

date_col = 'week'
response_col = 'claims'
df.dtypes
We can see from the plot below, there are seasonality, trend, and as well as a huge change point due the impact of COVID-19.

# using relatively updated data
df[['sp500']] = df[['sp500']].diff()
df = df[1:].reset_index(drop=True)
test_size = 12
test_df = df[:-test_size]
train_df = df[-test_size:]

9.1.1 Naive Model

Here we will use DLT models to compare the model performance with vs. without regression.

[7]: %%time
dlt = DLT(
    response_col=response_col,
    date_col=date_col,
    seasonality=52,
    seed=8888,
    num_warmup=4000,
    stan_mcmc_args={'show_progress': False}
)
dlt.fit(df=train_df)
predicted_df = dlt.predict(df=test_df)
2024-03-13 22:00:52 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
˓→temperature: 1.000, warmups (per chain): 1000 and samples(per chain): 25.
CPU times: user 114 ms, sys: 29.9 ms, total: 144 ms
Wall time: 5.93 s

9.1.2 DLT With Regression

The regressor columns can be supplied via argument regressor_col. Recall the regression formula in DLT:

\[ \hat{y}_t = \mu_t + s_t + r_t \]

\[ r_t = \sum_j \beta_j x_{jt} \]

\[ \beta_j \sim \mathcal{N}(\mu_j, \sigma_j^2) \]

By default, \( \mu_j = 0 \) and \( \sigma_j = 1 \). In addition, we can set a sign constraint for each coefficient \( \beta_j \). This is can be done by supplying the regressor_sign as a list where elements are in one of followings:

- ‘=’: \( \beta_j \sim \mathcal{N}(0, \sigma_j^2) \) i.e. \( \beta_j \in (\text{- inf, inf}) \)
- ‘+’: \( \beta_j \sim \mathcal{N}^+(0, \sigma_j^2) \) i.e. \( \beta_j \in [0, \text{inf}) \)
- ‘-’: \( \beta_j \sim \mathcal{N}^-(0, \sigma_j^2) \) i.e. \( \beta_j \in (-\text{inf}, 0] \)

Based on some intuition, it’s reasonable to assume search terms such as “unemployment”, “filling” and VIX index to be positively correlated and stock index such as SP500 to be negatively correlated to the outcome. Then we will leave whatever unspecified as a regular regressor.

[8]: %%time
dlt_reg = DLT(
    response_col=response_col,
    date_col=date_col,
    regressor_col=[...]
)
(continues on next page)
regressor_col=['trend.unemploy', 'trend.job', 'sp500', 'vix'],
seasonality=52,
seed=8888,
um_warmup=4000,
stan_mcmc_args={"show_progress": False}
)
dlt_reg.fit(df=train_df)
predicted_df_reg = dlt_reg.predict(test_df)

2024-03-13 22:00:57 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,
→ temperature: 1.000, warmups (per chain): 1000 and samples(per chain): 25.

CPU times: user 117 ms, sys: 25.8 ms, total: 142 ms
Wall time: 6.98 s

The estimated regressor coefficients can be retrieved via .get_regression_coefs().

<table>
<thead>
<tr>
<th>regressor</th>
<th>regressor_sign</th>
<th>coefficient</th>
<th>coefficient_lower</th>
<th>coefficient_upper</th>
<th>Pr(coef &gt;= 0)</th>
<th>Pr(coef &lt; 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>trend.unemploy</td>
<td>Regular</td>
<td>0.076715</td>
<td>0.042656</td>
<td>0.105937</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>trend.job</td>
<td>Regular</td>
<td>-0.038551</td>
<td>-0.081939</td>
<td>0.019654</td>
<td>0.13</td>
<td>0.87</td>
</tr>
<tr>
<td>sp500</td>
<td>Regular</td>
<td>-0.001387</td>
<td>-0.201171</td>
<td>0.195193</td>
<td>0.49</td>
<td>0.51</td>
</tr>
<tr>
<td>vix</td>
<td>Regular</td>
<td>0.011533</td>
<td>-0.013659</td>
<td>0.035988</td>
<td>0.73</td>
<td>0.27</td>
</tr>
</tbody>
</table>

9.1.3 Regression with Informative Priors

Due to various reasons, users may obtain further knowledge on some of the regressors or they want to propose different regularization on different regressors. These informative priors basically means to replace the defaults ($\mu, \sigma$) mentioned previously. In orbit, this process is done via the arguments regressor_beta_prior and regressor_sigma_prior. These two lists should be of the same length as regressor_col.

In addition, we can set a sign constraint for each coefficient $\beta_j$. This is can be done by supplying the regressor_sign as a list where elements are in one of followings:

- ‘=’: $\beta_j \sim \mathcal{N}(0, \sigma_j^2)$ i.e. $\beta_j \in (-\infty, \infty)$
- ‘+’: $\beta_j \sim \mathcal{N}^+(0, \sigma_j^2)$ i.e. $\beta_j \in [0, \infty)$
- ‘-’: $\beta_j \sim \mathcal{N}^-(0, \sigma_j^2)$ i.e. $\beta_j \in (-\infty, 0]$ Based on intuition, it’s reasonable to assume search terms such as “unemployment”, “filling” and VIX index to be positively correlated (+ sign is used in this case) and upward shock of SP500 (- sign) to be negatively correlated to the outcome. Otherwise, an unbounded coefficient can be used (= sign).

Furthermore, regressors such as search queries may have more direct impact than stock marker indices. Hence, a smaller $\sigma$ is considered.
[10]: dlt_reg_adjust = DLT(
    response_col=response_col,
    date_col=date_col,
    regressor_col=[‘trend.unemploy’, ‘trend.job’, ’sp500’, ’vix’],
    regressor_sign=['+', '=','-', '+'],
    regressor_sigma_prior=[0.3, 0.1, 0.05, 0.1],
    num_warmup=4000,
    num_sample=1000,
    estimator='stan-mcmc',
    seed=2022,
    stan_mcmc_args={'show_progress': False}
)
dlt_reg_adjust.fit(df=train_df)
predicted_df_reg_adjust = dlt_reg_adjust.predict(test_df)

2024-03-13 22:01:04 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
˓→temperature: 1.000, warmups (per chain): 1000 and samples(per chain): 250.

[11]: dlt_reg_adjust.get_regression_coefs()

<table>
<thead>
<tr>
<th>regressor</th>
<th>regressor_sign</th>
<th>coefficient</th>
<th>coefficient_lower</th>
<th>coefficient_upper</th>
<th>Pr(coef &gt;= 0)</th>
<th>Pr(coef &lt; 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>trend.unemploy</td>
<td>Positive</td>
<td>0.126584</td>
<td>0.075630</td>
<td>0.198016</td>
<td>1.000</td>
<td>0.000</td>
</tr>
<tr>
<td>vix</td>
<td>Positive</td>
<td>0.019553</td>
<td>0.002202</td>
<td>0.054368</td>
<td>1.000</td>
<td>0.000</td>
</tr>
<tr>
<td>sp500</td>
<td>Negative</td>
<td>-0.032251</td>
<td>-0.087838</td>
<td>-0.002386</td>
<td>0.000</td>
<td>1.000</td>
</tr>
<tr>
<td>trend.job</td>
<td>Regular</td>
<td>-0.011294</td>
<td>-0.086100</td>
<td>0.058422</td>
<td>0.394</td>
<td>0.606</td>
</tr>
</tbody>
</table>

Let’s compare the holdout performance by using the built-in function smape().

[12]: def mae(x, y):
    return np.mean(np.abs(x - y))

naive_mae = mae(predicted_df[‘prediction’].values, test_df[‘claims’].values)
reg_mae = mae(predicted_df_reg[‘prediction’].values, test_df[‘claims’].values)
reg_adjust_mae = mae(predicted_df_reg_adjust[‘prediction’].values, test_df[‘claims’].values)

print("----------------Mean Absolute Error Summary----------------")
print("Naive Model: {:.3f}\nRegression Model: {:.3f}\nRefined Regression Model: {:.3f}".
˓→format(naive_mae, reg_mae, reg_adjust_mae))
----------------Mean Absolute Error Summary----------------
Naive Model: 0.255
Regression Model: 0.242
Refined Regression Model: 0.096

9.1. US Weekly Initial Claims
9.2 Summary

This demo showcases a use case in nowcasting. Although this may not be applicable in real-time forecasting, it mainly introduces the regression analysis with time-series modeling in Orbit. For people who have concerns on the forecastability, one can consider introducing lag on regressors.

Also, Orbit allows informative priors where sometime can be useful in combining multiple source of insights together.
This notebook continues to discuss regression problems with DLT and covers various penalties:

1. fixed-ridge
2. auto-ridge
3. lasso

Generally speaking, regression coefficients are more robust under full Bayesian sampling and estimation. The default setting `estimator='stan-mcmc'` will be used in this tutorial. Besides, a fixed and small smoothing parameters are used such as `level_sm_input=0.01` and `slope_sm_input=0.01` to facilitate high dimensional regression.

```python
[1]: %matplotlib inline

import matplotlib.pyplot as plt
import numpy as np

import orbit
from orbit.utils.dataset import load_iclaims
from orbit.models import DLT
from orbit.diagnostics.plot import plot_predicted_data
from orbit.constants.palette import OrbitPalette

[2]: print(orbit.__version__)

1.1.4.4

10.1 Regression on Simulated Dataset

A simulated dataset is used to demonstrate sparse regression.

```python
[3]: import pandas as pd
from orbit.utils.simulation import make_trend, make_regression
from orbit.diagnostics.metrics import mse

A few utilities from the package is used to generate simulated data. For details, please refer to the API doc. In brief, the process generates observations $y$ such that
\[ y_t = l_t + \sum_{p}^{P} \beta_p x_{t,p} \]

for \( t = 1, 2, \cdots, T \)

where

\[ l_t = l_{t-1} + \delta_t \]
\[ \delta_t \sim N(0, \sigma_\delta) \]

### 10.1.1 Regular Regression

To begin with, the setting \( P = 10 \) and \( T = 100 \) is used.

```python
NUM_OF_REGRESSORS = 10
SERIES_LEN = 50
SEED = 20210101
# sample some coefficients
COEFS = np.random.default_rng(SEED).uniform(-1, 1, NUM_OF_REGRESSORS)
trend = make_trend(SERIES_LEN, rw_loc=0.01, rw_scale=0.1)
x, regression, coefs = make_regression(series_len=SERIES_LEN, coefs=COEFS)
print(regression.shape, x.shape)
(50,) (50, 10)
```

```python
# combine trend and the regression
y = trend + regression
```

```python
x_cols = [f"x{x}" for x in range(1, NUM_OF_REGRESSORS + 1)]
response_col = "y"
dt_col = "date"
obs_matrix = np.concatenate([y.reshape(-1, 1), x], axis=1)
# make a data frame for orbit inputs
df = pd.DataFrame(obs_matrix, columns=[response_col] + x_cols)
# make some dummy date stamp
dt = pd.date_range(start='2016-01-04', periods=SERIES_LEN, freq="1W")
df["date"] = dt
df.shape
(50, 12)
```

Here is a peek on the coefficients.

```python
coefs
```
By default, `regression_penalty` is set as fixed-ridge i.e.

$$
\beta_j \sim N(\mu_j, \sigma^2_j)
$$

with a default $\mu_j = 0$ and $\sigma_j = 1$

**Fixed Ridge Penalty**

```python
# this is default
regression_penalty='fixed_ridge',
# fixing the smoothing parameters to learn regression coefficients more effectively
level_sm_input=0.01,
slope_sm_input=0.01,
num_warmup=4000,
)
```

```python
 coef_fridge = np.quantile(dlt_fridge._posterior_samples['beta'], q=[0.05, 0.5, 0.95], axis=0 )
```

```python
plt.figure(figsize=(20, 8))
plt.title("Weights of the model", fontsize=20)
plt.plot(idx, coef_fridge[1], color=OrbitPalette.GREEN.value, linewidth=lw, drawstyle='steps', label="Fixed-Ridge", alpha=0.5, linestyle='--')
```
Auto-Ridge Penalty

Users can also set the `regression_penalty` to be `auto-ridge` in case users are not sure what to set for the `regressor_sigma_prior`.

Instead of using fixed scale in the coefficients prior, a prior can be assigned to them, i.e.

$$\sigma_j \sim \text{Cauchy}^+(0, \alpha)$$

This can be done by setting `regression_penalty="auto_ridge"` with the argument `auto_ridge_scale` (default of 0.5) set the prior $\alpha$. A higher `adapt_delta` is recommend to reduce divergence. Check here for details of `adapt_delta`.

```
[10]: %%time
dlt_auto_ridge = DLT(
    response_col=response_col,
    date_col=dt_col,
    regressor_col=x_cols,
    seed=SEED,
    # this is default
    regression_penalty='auto_ridge',
    # fixing the smoothing parameters to learn regression coefficients more effectively
    level_sm_input=0.01,
    slope_sm_input=0.01,
    num_warmup=4000,
)
dlt_auto_ridge.fit(df=df)
```

2024-03-13 22:00:39 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8, temperature: 1.000, warmups (per chain): 1000 and samples(per chain): 25.

chain 1 |   | 00:00 Status
chain 2 | | 00:00 Status
chain 3 | | 00:00 Status
chain 4 | | 00:00 Status

CPU times: user 73 ms, sys: 213 ms, total: 286 ms
Wall time: 1.7 s

[10]: <orbit.forecaster.full_bayes.FullBayesianForecaster at 0x2a76c5810>

[11]:
coef_auto_ridge = np.quantile(dlt_auto_ridge._posterior_samples['beta'], q=[0.05, 0.5, 0.95], axis=0 )
lw=3
idx = np.arange(NUM_OF_REGRESSORS)
plt.figure(figsize=(20, 8))
plt.title("Weights of the model", fontsize=24)
plt.plot(idx, coef_auto_ridge[1], color=OrbitPalette.GREEN.value, linewidth=lw, drawstyle='steps', label='Auto-Ridge', alpha=0.5, linestyle='--', color=OrbitPalette.GREEN.value)
plt.fill_between(idx, coef_auto_ridge[0], coef_auto_ridge[2], step='pre', alpha=0.3, color=OrbitPalette.GREEN.value)
plt.plot(coefs, color="black", linewidth=lw, drawstyle='steps', label="Ground truth")
plt.ylim(1, -1)
plt.legend(prop={'size': 20})
plt.grid();

[12]:
print('Fixed Ridge MSE:{:.3f}\nAuto Ridge MSE:{:.3f}'.format(mse(coef_fridge[1], coefs), mse(coef_auto_ridge[1], coefs))

Fixed Ridge MSE:0.091
Auto Ridge MSE:0.075

10.1. Regression on Simulated Dataset
10.1.2 Sparse Regression

In reality, users usually face a more challenging problem with a much higher $P$ to $N$ ratio with a sparsity specified by the parameter relevance=0.5 under the simulation process.

```python
[13]: NUM_OF_REGRESSORS = 50
SERIES_LEN = 50
SEED = 20210101
COEFS = np.random.default_rng(SEED).uniform(0.3, 0.5, NUM_OF_REGRESSORS)
SIGNS = np.random.default_rng(SEED).choice([1, -1], NUM_OF_REGRESSORS)
# to mimic a either zero or relative observable coefficients
COEFS = COEFS * SIGNS
trend = make_trend(SERIES_LEN, rw_loc=0.01, rw_scale=0.1)
x, regression, coefs = make_regression(series_len=SERIES_LEN, coefs=COEFS, relevance=0.5)
print(regression.shape, x.shape)
(50,) (50, 50)

[14]: # generated sparsed coefficients
coops
array([[0. , 0. , -0.45404565, 0.37813559, 0. , 0. , 0. , 0.48036792, -0.32535635, -0.37337302, -0.42474576, 0. , -0.37000755, 0.44887456, 0.47082836, 0. , 0. , 0.32678039, 0.37436121, 0.38932392, 0.40216056, 0. , 0. , -0.3076828 , -0.35036047, 0. , 0. , 0. , 0. , 0. , 0.45838674, 0.3171478 , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.41599814, 0. , -0.30964341, -0.42072894, 0.36255583, 0. , -0.39326337, 0.44455655, 0. , 0. , 0. , 0.30064161, -0.46083203, 0. ]])

[15]: # combine trend and the regression
y = trend + regression

[16]: x_cols = [f"x(x)" for x in range(1, NUM_OF_REGRESSORS + 1)]
response_col = "y"
dt_col = "date"
obs_matrix = np.concatenate([y.reshape(-1, 1), x], axis=1)
# make a data frame for orbit inputs
df = pd.DataFrame(obs_matrix, columns=[response_col] + x_cols)
# make some dummy date stamp
dt = pd.date_range(start='2016-01-04', periods=SERIES_LEN, freq="1W")
df['date'] = dt
df.shape
(50, 52)
```
10.1.3 Fixed Ridge Penalty

```python
[17]:
    dlt_fridge = DLT(
        response_col=response_col,
        date_col=dt_col,
        regressor_col=x_cols,
        seed=SEED,
        level_sm_input=0.01,
        slope_sm_input=0.01,
        num_warmup=8000,
    )
    dlt_fridge.fit(df=df)


<table>
<thead>
<tr>
<th>chain</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>chain 1</td>
<td>00:00 Status</td>
</tr>
<tr>
<td>chain 2</td>
<td>00:00 Status</td>
</tr>
<tr>
<td>chain 3</td>
<td>00:00 Status</td>
</tr>
<tr>
<td>chain 4</td>
<td>00:00 Status</td>
</tr>
</tbody>
</table>

[17]: <orbit.forecaster.full_bayes.FullBayesianForecaster at 0x2a8838150>

[18]:
    coef_fridge = np.quantile(dlt_fridge._posterior_samples['beta'], q=[0.05, 0.5, 0.95], axis=0)
    lw=3
    idx = np.arange(NUM_OF_REGRESSORS)
    plt.figure(figsize=(20, 8))
    plt.title("Weights of the model", fontsize=24)
    plt.plot(coef_fridge[1], color=OrbitPalette.GREEN.value, linewidth=lw, drawstyle='steps-', label="Ridge", alpha=0.5, linestyle='--')
    plt.fill_between(idx, coef_fridge[0], coef_fridge[2], step='pre', alpha=0.3, color=OrbitPalette.GREEN.value)
    plt.plot(coefs, color="black", linewidth=lw, drawstyle='steps', label="Ground truth")
    plt.legend(prop={'size': 20})
    plt.grid();
```

10.1. Regression on Simulated Dataset 51
LASSO Penalty

In high $P$ to $N$ problems, LASSO penalty usually shines compared to Ridge penalty.

```python
[19]: dlt_lasso = DLT(
    response_col=response_col,
    date_col=dt_col,
    regressor_col=x_cols,
    seed=SEED,
    regression_penalty='lasso',
    level_sm_input=0.01,
    slope_sm_input=0.01,
    num_warmup=8000,
)

# dlt_lasso.fit(df=df)

2024-03-13 22:00:46 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
˓→temperature: 1.000, warmups (per chain): 2000 and samples(per chain): 25.

chain 1 | | 00:00 Status
chain 2 | | 00:00 Status
chain 3 | | 00:00 Status
chain 4 | | 00:00 Status

[19]: <orbit.forecaster.full_bayes.FullBayesianForecaster at 0x2a886f850>

[20]: coef_lasso = np.quantile(dlt_lasso._posterior_samples['beta'], q=[0.05, 0.5, 0.95],␣
˓→axis=0 )

lw=3
idx = np.arange(NUM_OF_REGRESSORS)
plt.figure(figsize=(20, 8))
plt.title("Weights of the model", fontsize=24)
```

(continues on next page)
10.2 Summary

This notebook covers a few choices of penalty in regression regularization. A lasso and auto-ridge can be considered in highly sparse data.
Because of the generative nature of the exponential smoothing models, they can naturally deal with missing response during the training process. It simply replaces observations by prediction during the 1-step ahead generating process. Below users can find the simple examples of how those exponential smoothing models handle missing responses.

```python
[1]: import pandas as pd
    import numpy as np
    import orbit
    import matplotlib.pyplot as plt
    from orbit.utils.dataset import load_iclaims
    from orbit.diagnostics.plot import plot_predicted_data, plot_predicted_components
    from orbit.utils.plot import get_orbit_style
    from orbit.models import ETS, LGT, DLT
    from orbit.diagnostics.metrics import smape
    plt.style.use(get_orbit_style())
    %load_ext autoreload
    %autoreload 2
    %matplotlib inline

[2]: orbit.__version__
[2]: '1.1.4.4'

11.1 Data

[3]: # can also consider transform=False
    raw_df = load_iclaims(transform=True)
    raw_df.dtypes
    df = raw_df.copy()
    df.head()
[3]:     week    claims  trend.unemploy  trend.filling  trend.job  sp500
 0  2010-01-03  13.386595        0.219882       -0.318452     0.117500  -0.417633
 1  2010-01-10  13.624218        0.219882       -0.194838     0.168794  -0.425480
 2  2010-01-17  13.398741        0.236143       -0.292477     0.117500  -0.465229
```

(continues on next page)
11.1 Define Missing Data

Now, we manually created a dataset with a few missing values in the response variable.

```python
[n]: test_size=52
train_df=df[:-test_size]
test_df=df[-test_size:]
```

### 11.2 Exponential Smoothing Examples

#### 11.2.1 ETS

```python
[7]: ets = ETS(
    response_col='claims',
    date_col='week',
    seasonality=52,
    seed=2022,
    estimator='stan-mcmc'
    )

ets.fit(train_df_na)
ets.predicted = ets.predict(df=train_df_na)
```

```
2024-03-13 21:51:58 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8, ⨪
...temperature: 1.000, warmups (per chain): 225 and samples(per chain): 25.
```

```
chain 1 | | 00:00 Status
chain 2 | | 00:00 Status
```
11.2.2 LGT

```python
[8]: lgt = LGT(
    response_col='claims',
    date_col='week',
    estimator='stan-mcmc',
    seasonality=52,
    seed=2022
)
lgt.fit(df=train_df_na)
lgt_predicted = lgt.predict(df=train_df_na)
```


chain 1 | | 00:00 Status
chain 2 | | 00:00 Status
chain 3 | | 00:00 Status
chain 4 | | 00:00 Status

11.2.3 DLT

```python
[9]: dlt = DLT(
    response_col='claims',
    date_col='week',
    estimator='stan-mcmc',
    seasonality=52,
    seed=2022
)
dlt.fit(df=train_df_na)
dlt_predicted = dlt.predict(df=train_df_na)
```


chain 1 | | 00:00 Status
chain 2 | | 00:00 Status
chain 3 | | 00:00 Status
chain 4 | | 00:00 Status

11.2. Exponential Smoothing Examples
11.2.4 Summary

Users can verify this behavior with a table and visualization of the actuals and predicted.

```
[10]: train_df_na['ets-predict'] = ets_predicted['prediction']
    train_df_na['lgt-predict'] = lgt_predicted['prediction']
    train_df_na['dlt-predict'] = dlt_predicted['prediction']

[11]: # table summary of prediction during absence of observations
    train_df_na.iloc[na_idx, :].head(5)

        week  claims trend.unemploy trend.filling trend.job  sp500  \
    53 2011-01-09  NaN 0.152060     -0.127397   0.085412 -0.295869
    54 2011-01-16  NaN 0.186546     -0.044015   0.074483 -0.303546
    55 2011-01-23  NaN 0.169451     -0.004795   0.074483 -0.309024
    56 2011-01-30  NaN 0.079300     -0.032946 -0.005560 -0.282329
    57 2011-02-06  NaN 0.060252     -0.024213   0.06275   -0.268480

    vix  ets-predict  lgt-predict  dlt-predict
    53  -0.036658  13.519096  13.512083     13.512583
    54  0.141233  13.281033  13.279732     13.278579
    55  0.222816  13.011531  13.010502     13.013743
    56  0.006710  13.056016  13.068143     13.061067
    57  0.021891  12.992839  13.015295     13.007281
```

```
[12]: from orbit.constants.palette import OrbitPalette
    # just to get some color from orbit palette
    orbit_palette = [
        OrbitPalette.BLACK.value,
        OrbitPalette.BLUE.value,
        OrbitPalette.GREEN.value,
        OrbitPalette.YELLOW.value,
    ]

[13]: pred_list = ['ets-predict', 'lgt-predict', 'dlt-predict']
    fig, axes = plt.subplots(len(pred_list), 1, figsize=(16, 16))
    for idx, p in enumerate(pred_list):
        axes[idx].scatter(train_df_na['week'], train_df_na['claims'].values,
                          label='actuals' if idx == 0 else '', color=orbit_palette[0],
                          alpha=0.5)
        axes[idx].plot(train_df_na['week'], train_df_na[p].values,
                        label=p, color=orbit_palette[idx + 1], lw=2.5)
    fig.legend()
    fig.tight_layout()
11.3 First Observation Exception

It is worth pointing out that the very first value of the response variable cannot be missing since this is the starting point of the time series fitting. An error message will be raised when the first observation in response is missing.

```python
# DO NOT RUN
# na_idx2 = list(na_idx) + [0]
# train_df_na2 = train_df.copy()
# train_df_na2.iloc[na_idx2, 1] = np.nan
# ets.fit(train_df_na2)
```
Kernel-based time-varying regression (KTR) is a time series model to address

1. time-varying regression coefficients
2. complex seasonality pattern

The full details of the model structure with an application in marketing media mix modeling can be found in Ng, Wang and Dai (2021). The core of KTR is the use of latent variables to define a smooth time varying representation of model coefficients, which bears a similar ideas in Kernel Smoothing. The KTR approach sharply reduces the number of parameters compared to typical dynamic linear models such as Harvey (1989), and Durbin and Koopman (2002). The reduced number of parameters improves the computation speed, and allows for handling of high dimensional data and detecting small variances.

To topics covered here in Part I, are

1. KTR model structure
2. syntax to initialize, fit and predict a model
3. fit a model with complex seasonality
4. visualization of prediction and decomposed components

[1]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import orbit
from orbit.models import KTR
from orbit.diagnostics.plot import plot_predicted_data, plot_predicted_components
from orbit.utils.dataset import load_electricity_demand

%matplotlib inline
pd.set_option('display.float_format', lambda x: '%.5f' % x)
```

[2]:
```python
print(orbit.__version__)
```

1.1.4.4
12.1 Model Structure

This section gives the mathematical structure of the KTR model. In short, it considers a time-series \( y_t \) as the linear combination of three parts which are the local-trend \( l_t \), seasonality \( s_t \), and regression \( r_t \) terms. Mathematically,

\[
y_t = l_t + s_t + r_t + \epsilon_t, \quad t = 1, \cdots, T,
\]

where the \( \epsilon_t \) comprise a stationary random error process.

In KTR, the distinction between the local-trend, seasonality, and regressors while useful is semi-arbitrary and the time-series can also be considered as

\[
y_t = X_t^T \beta_t + \epsilon_t, \quad t = 1, \cdots, T,
\]

where \( \beta_t \) is a \( P \)-dimensional vector of coefficients that vary over time (i.e., \( \beta_i \) is almost certainly different from \( \beta_j \) for \( i \neq j \)) and \( X_t \) \( P \)-dimensional covariate vector (i.e., the \( t \)th row of \( X \), the design matrix).

To reduce the total number of parameters in the model (potentially \( P \times T \)) the \( \beta_t \) are parameterized with a weighted sum of \( J \) local latent variables \( (b_1, \ldots, b_J) \). That is

\[
B = Kb^T
\]

where - coefficient matrix \( B \) has size \( T \times P \) with rows equal to the \( \beta_t \), - knot matrix \( b \) with size \( P \times J \); each entry is a latent variable \( b_{p,j} \). The \( b_{j} \) can be viewed as the “knots” from the perspective of spline regression and \( j \) is a time index such that \( t_j \in [1, \cdots, T] \). - kernel matrix \( K \) with size \( T \times J \) where the \( i \)th row and \( j \)th element can be viewed as the normalized weight \( k(t_j, t) / \sum_{j=1}^{J} k(t_j, t) \)

For the level/trend,

\[
l_t = \beta_{t,lev}
\]

It can also be viewed as a dynamic intercept (where the regressor is a vector of ones).

For the seasonality,

\[
B_{seas} = K_{seas}b_{seas}^T
\]

\[
s_t = X_{t,seas}\beta_{t,seas}
\]

We use Fourier series to handle the seasonality; i.e., sin waves with varying periods are used for the columns of \( X_{seas} \).

The details for the additional regressors are given in Part II, as they are not used in this tutorial. Note this includes different choices of kernel function (which determines the kernel matrix \( K \)) and prior for matrix \( b \).

12.2 Data

To illustrate the usage of KTR, consider the daily series of electricity demand in Turkey from the 2000 - 2008.

```python
# from 2000-01-01 to 2008-12-31
df = load_electricity_demand()
date_col = 'date'
response_col = 'electricity'
df[response_col] = np.log(df[response_col])
print(df.shape)
df.head()
```

[3]
(3288, 2)

<table>
<thead>
<tr>
<th>date</th>
<th>electricity</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>9.43760</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>9.50130</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>9.63565</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>9.65392</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>9.66089</td>
</tr>
</tbody>
</table>

```python
print(f'starts with {df[date_col].min()}
ends with {df[date_col].max()}
shape: {df.shape}')
```

starts with 2000-01-01 00:00:00
ends with 2008-12-31 00:00:00
shape: (3288, 2)

### 12.2.1 Train / Test Split

Split the data into a training set and test set for model validation.

```python
test_size=365
train_df=df[:-test_size]
test_df=df[-test_size:]
```

### 12.3 A Quick Start on KTR

Here the Similar to other model types in Orbit, KTR follows sklearn model API style. First an instance of the Orbit class KTR is created. Second fit and predict methods are called for that instance. Note that unlike version <=1.0.15, the fitting API arg are within the function; thus, KTR is called directly.

```python
ktr = KTR(
    response_col=response_col,
    date_col=date_col,
    seed=2021,
    estimator='pyro-svi',
    # bootstrap sampling to capture uncertainties
    n_bootstrap_draws=1e4,
    # pyro training config
    num_steps=301,
    message=100,
)
```

```python
ktr.fit(train_df)
```

(continues on next page)
We can take a look how the level is fitted with the data.

```python
predicted_df = ktr.predict(df=df)
predicted_df.head()
```

<table>
<thead>
<tr>
<th>date</th>
<th>prediction_5</th>
<th>prediction</th>
<th>prediction_95</th>
</tr>
</thead>
</table>

One can use `.get_posterior_samples()` to extract the samples for all sampling parameters.

```python
ktr.get_posterior_samples().keys()
```

```
dict_keys(['lev_knot', 'lev', 'yhat', 'obs_scale'])
```

```python
_ = plot_predicted_data(training_actual_df=train_df, predicted_df=predicted_df, date_col=date_col, actual_col=response_col, test_actual_df=test_df, markersize=20, lw=.5)
```
It can also be helpful to see the trend knot locations and levels. This is done with the `plot_lev_knots` function.

```
[11]: _ = ktr.plot_lev_knots()
```
12.4 Fitting with Complex Seasonality

The previous model fit is not satisfactory as there is clear seasonality in the electrical demand time-series that is not accounted for. In this modelling example the electrical demand data is fit with a dual seasonality for weekly and yearly patterns. Since the data is daily, the seasonality periods are 7 and 365.25. These are added into the KTR object as a list through the `seasonality` arg. Otherwise the process is the same as the previous example.

```python
[12]: ktr_with_seas = KTR(
response_col=response_col,
date_col=date_col,
seed=2021,
seasonality=[7, 365.25],
estimator='pyro-svi',
n_bootstrap_draws=1e4,
# pyro training config
num_steps=301,
message=100,
)
```

```python
[13]: ktr_with_seas.fit(train_df)
```

```
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-13 21:53:23 - orbit - INFO - Using SVI (Pyro) with steps: 301, samples: 100,␣
˓→learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
INFO:orbit:Using SVI (Pyro) with steps: 301, samples: 100, learning rate: 0.1, learning␣
˓→rate_total_decay: 1.0 and particles: 100.
2024-03-13 21:53:25 - orbit - INFO - step 0 loss = -2190.8, scale = 0.093667
INFO:orbit:step 0 loss = -2190.8, scale = 0.093667
2024-03-13 21:53:28 - orbit - INFO - step 100 loss = -4356.8, scale = 0.0069845
INFO:orbit:step 100 loss = -4356.8, scale = 0.0069845
2024-03-13 21:53:30 - orbit - INFO - step 200 loss = -4301.3, scale = 0.0071019
INFO:orbit:step 200 loss = -4301.3, scale = 0.0071019
2024-03-13 21:53:32 - orbit - INFO - step 300 loss = -4362, scale = 0.0072349
INFO:orbit:step 300 loss = -4362, scale = 0.0072349
```

```python
[13]: <orbit.forecaster.svi.SVIForecaster at 0x2abff0c10>
```

```python
[14]: predicted_df = ktr_with_seas.predict(df=df, decompose=True)
```

```python
[15]: predicted_df.head(5)
```

<table>
<thead>
<tr>
<th>date</th>
<th>prediction_5</th>
<th>prediction</th>
<th>prediction_95</th>
<th>trend_5</th>
<th>trend</th>
<th>trend_95</th>
<th>regression_5</th>
<th>regression</th>
<th>regression_95</th>
<th>seasonality_7_5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2000-01-01</td>
<td>9.53352</td>
<td>9.61277</td>
<td>9.69350</td>
<td>9.50462</td>
<td>9.58387</td>
<td></td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>-0.02784</td>
</tr>
<tr>
<td>1 2000-01-02</td>
<td>9.48100</td>
<td>9.56119</td>
<td>9.64121</td>
<td>9.50211</td>
<td>9.58229</td>
<td></td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>-0.07872</td>
</tr>
<tr>
<td>2 2000-01-03</td>
<td>9.54230</td>
<td>9.62288</td>
<td>9.70269</td>
<td>9.50224</td>
<td>9.58282</td>
<td></td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>-0.07872</td>
</tr>
<tr>
<td>3 2000-01-04</td>
<td>9.60186</td>
<td>9.68151</td>
<td>9.76075</td>
<td>9.50299</td>
<td>9.58265</td>
<td></td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>-0.01838</td>
</tr>
<tr>
<td>4 2000-01-05</td>
<td>9.58472</td>
<td>9.66577</td>
<td>9.74660</td>
<td>9.50168</td>
<td>9.58273</td>
<td></td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>-0.02784</td>
</tr>
</tbody>
</table>
```

(continues on next page)
Tips: there is an additional argument `seasonality_fs_order` to control the number of orders in Fourier series terms we want to approximate the seasonality. In general, they cannot violate the condition that $2 \times$ Fourier series order $< \text{seasonality}$ since each order represents adding a pair of sine and cosine regressors.

### 12.5 More Diagnostic and Visualization

Here are a few more diagnostic and visualization. The fit is decomposed into components, the local trend and both periods of seasonality.

```python
[16]: _ = plot_predicted_data(training_actual_df=train_df, predicted_df=predicted_df, date_col=date_col, actual_col=response_col, test_actual_df=test_df, markersize=10, lw=.5)
```
12.6 References

The previous tutorial covered the basic syntax and structure of KTR (or so called BTVC); time-series data was fitted with a KTR model accounting for trend and seasonality. In this tutorial a KTR model is fit with trend, seasonality, and additional regressors. To summarize part 1, KTR considers a time-series as an additive combination of local-trend, seasonality, and additional regressors. The coefficients for all three components are allowed to vary over time. The time-varying of the coefficients is modeled using kernel smoothing of latent variables. This can also be an advantage of picking this model over other static regression coefficients models.

This tutorial covers:

1. KTR model structure with regression
2. syntax to initialize, fit and predict a model with regressors
3. visualization of regression coefficients

```
[1]: import pandas as pd
    import numpy as np
    from math import pi
    import matplotlib.pyplot as plt
    import orbit
    from orbit.models import KTR
    from orbit.diagnostics.plot import plot_predicted_components
    from orbit.utils.plot import get_orbit_style
    from orbit.constants.palette import OrbitPalette

    %matplotlib inline
    pd.set_option('display.float_format', lambda x: '%.5f' % x)
    orbit_style = get_orbit_style()
    plt.style.use(orbit_style);

[2]: print(orbit.__version__)

1.1.4.4
```
13.1 Model Structure

This section gives the mathematical structure of the KTR model. In short, it considers a time-series \( (y_t) \) as the linear combination of three parts. These are the local-trend \( (l_t) \), seasonality \( (s_t) \), and regression \( (r_t) \) terms at time \( t \). That is

\[
y_t = l_t + s_t + r_t + \epsilon_t, \quad t = 1, \cdots, T,
\]

where

- \( \epsilon_t \)'s comprise a stationary random error process.
- \( r_t \) is the regression component which can be further expressed as \( \sum_{i=1}^{J} x_{i,t} \beta_{i,t} \) with covariate \( x \) and coefficient \( \beta \) on indexes \( i, t \)

For details of how on \( l_t \) and \( s_t \), please refer to Part I.

Recall in KTR, we express coefficients as

\[
B = Kb^T
\]

where
- coefficient matrix \( B \) has size \( t \times P \) with rows equal to the \( \beta_t \) - knot matrix \( b \) with size \( P \times J \); each entry is a latent variable \( b_{p,j} \). The \( b_j \) can be viewed as the “knots” from the perspective of spline regression and \( j \) is a time index such that \( t_j \in [1, \cdots, T] \).
- kernel matrix \( K \) with size \( T \times J \) where the \( i \)th row and \( j \)th element can be viewed as the normalized weight \( k(t_j, t)/\sum_{j=1}^{J} k(t_j, t) \)

In regression, we generate the matrix \( K \) with Gaussian kernel \( k_{\text{reg}} \) as such:

\[
k_{\text{reg}}(t, t_j; \rho) = \exp\left(\frac{-(t-t_j)^2}{2\rho^2}\right),
\]

where \( \rho \) is the scale hyper-parameter.

13.2 Data Simulation Module

In this example, we will use simulated data in order to have true regression coefficients for comparison. We propose two set of simulation data with three predictors each:

The two data sets are: - random walk - sine-cosine like

Note the data are random so it may be worthwhile to repeat the next few sets a few times to see how different data sets work.

13.2.1 Random Walk Simulated Dataset

[3]:

```python
def sim_data_seasonal(n, RS):
    """
    coefficients curve are sine-cosine like
    """
    np.random.seed(RS)
    # make the time varying coefs
tau = np.arange(1, n+1)/n
data = pd.DataFrame(
    {'tau': tau,
     'date': pd.date_range(start='1/1/2018', periods=n),
     'beta1': 2 * tau,
     'beta2': 1.01 + np.sin(2*pi*tau),
    }
)
```

(continues on next page)
\[
\text{'beta3': } 1.01 + \sin(4\pi(\tau-1/8)), \\
\text{'x1': } \text{np.random.normal(0, 10, size=n),} \\
\text{'x2': } \text{np.random.normal(0, 10, size=n),} \\
\text{'x3': } \text{np.random.normal(0, 10, size=n),} \\
\text{\'trend\': } \text{np.cumsum(np.concatenate([np.array([1]), \text{np.random.normal(0, 0.1, n-1)}])}, \\
\text{\'error\': } \text{np.random.normal(0, 1, size=n)}
\]
### 13.2.2 Sine-Cosine Like Simulated Dataset

```python
[6]: sc_data = sim_data_seasonal(n=80, RS=2021)
sc_data.head(10)
```

<table>
<thead>
<tr>
<th>tau</th>
<th>date</th>
<th>beta1</th>
<th>beta2</th>
<th>beta3</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01250</td>
<td>2018-01-01</td>
<td>0.02500</td>
<td>1.08846</td>
<td>0.02231</td>
<td>14.88609</td>
<td>1.56556</td>
<td>-14.69399</td>
</tr>
<tr>
<td>0.02500</td>
<td>2018-01-02</td>
<td>0.05000</td>
<td>1.16643</td>
<td>0.05894</td>
<td>6.76011</td>
<td>-0.56861</td>
<td>4.93157</td>
</tr>
<tr>
<td>0.03750</td>
<td>2018-01-03</td>
<td>0.07500</td>
<td>1.24345</td>
<td>0.11899</td>
<td>-4.18451</td>
<td>-5.38234</td>
<td>-13.90578</td>
</tr>
<tr>
<td>0.05000</td>
<td>2018-01-04</td>
<td>0.10000</td>
<td>1.31992</td>
<td>0.20098</td>
<td>-8.06521</td>
<td>9.01387</td>
<td>-0.75244</td>
</tr>
<tr>
<td>0.06250</td>
<td>2018-01-05</td>
<td>0.12500</td>
<td>1.39268</td>
<td>0.30289</td>
<td>5.55876</td>
<td>2.24944</td>
<td>-2.53510</td>
</tr>
<tr>
<td>0.07500</td>
<td>2018-01-06</td>
<td>0.15000</td>
<td>1.46939</td>
<td>0.42221</td>
<td>7.05504</td>
<td>12.77788</td>
<td>14.25841</td>
</tr>
<tr>
<td>0.08750</td>
<td>2018-01-07</td>
<td>0.17500</td>
<td>1.53250</td>
<td>0.55601</td>
<td>11.30858</td>
<td>6.29269</td>
<td>7.82098</td>
</tr>
<tr>
<td>0.10000</td>
<td>2018-01-08</td>
<td>0.20000</td>
<td>1.59779</td>
<td>0.70098</td>
<td>6.45002</td>
<td>3.61891</td>
<td>16.28098</td>
</tr>
<tr>
<td>0.11250</td>
<td>2018-01-09</td>
<td>0.22500</td>
<td>1.65945</td>
<td>0.85357</td>
<td>1.06414</td>
<td>36.38726</td>
<td>8.80457</td>
</tr>
<tr>
<td>0.12500</td>
<td>2018-01-10</td>
<td>0.25000</td>
<td>1.71711</td>
<td>1.01000</td>
<td>4.22155</td>
<td>-12.01221</td>
<td>8.43176</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>trend</th>
<th>error</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00000</td>
<td>-0.73476</td>
<td>1.01359</td>
</tr>
<tr>
<td>1.07746</td>
<td>-0.97007</td>
<td>-1.00463</td>
</tr>
<tr>
<td>1.19201</td>
<td>-0.13891</td>
<td>-8.80009</td>
</tr>
<tr>
<td>1.22883</td>
<td>0.66550</td>
<td>11.59721</td>
</tr>
<tr>
<td>1.31341</td>
<td>-1.58259</td>
<td>1.47715</td>
</tr>
<tr>
<td>1.25911</td>
<td>-0.98049</td>
<td>22.68806</td>
</tr>
<tr>
<td>1.23484</td>
<td>-0.53751</td>
<td>15.43357</td>
</tr>
<tr>
<td>1.13237</td>
<td>-1.32858</td>
<td>17.15636</td>
</tr>
<tr>
<td>1.02834</td>
<td>0.87859</td>
<td>69.01607</td>
</tr>
<tr>
<td>1.00649</td>
<td>-0.22055</td>
<td>-11.27534</td>
</tr>
</tbody>
</table>

### 13.3 Fitting a Model with Regressors

The metadata for simulated data sets.

```python
[7]: # num of predictors
    p = 3
    regressor_col = ['x{0}'.format(pp) for pp in range(1, p + 1)]
    response_col = 'y'
    date_col='date'
```

As in Part I KTR follows sklearn model API style. First an instance of the Orbit class KTR is created. Second fit and predict methods are called for that instance. Besides providing meta data such as `response_col`, `date_col` and `regressor_col`, there are additional args to provide to specify the estimator and the setting of the estimator. For details, please refer to other tutorials of the Orbit site.
Here `predict` has the additional argument `decompose=True`. This returns the components \((l_t, s_t, r_t)\) of the regression along with the prediction.

```python
ktr = KTR(
    response_col=response_col,
    date_col=date_col,
    regressor_col=regressor_col,
    prediction_percentiles=[2.5, 97.5],
    seed=2021,
    estimator='pyro-svi',
)
```

```python
ktr.fit(df=rw_data)
ktr.predict(df=rw_data, decompose=True).head(5)
```

```
2024-03-13 21:54:02 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-13 21:54:02 - orbit - INFO - Using SVI (Pyro) with steps: 301, samples: 100,
    learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
    _C._set_default_tensor_type(t)
2024-03-13 21:54:02 - orbit - INFO - step 0 loss = 3107.8, scale = 0.091353
INFO:orbit:step 0 loss = 3107.8, scale = 0.091353
2024-03-13 21:54:03 - orbit - INFO - step 100 loss = 307.18, scale = 0.04889
INFO:orbit:step 100 loss = 307.18, scale = 0.04889
2024-03-13 21:54:03 - orbit - INFO - step 200 loss = 299.24, scale = 0.052646
INFO:orbit:step 200 loss = 299.24, scale = 0.052646
2024-03-13 21:54:04 - orbit - INFO - step 300 loss = 314.51, scale = 0.05106
INFO:orbit:step 300 loss = 314.51, scale = 0.05106
```

```
<table>
<thead>
<tr>
<th>date</th>
<th>prediction_2.5</th>
<th>prediction</th>
<th>prediction_97.5</th>
<th>trend_2.5</th>
<th>trend_97.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-01</td>
<td>5.18593</td>
<td>6.31800</td>
<td>7.44407</td>
<td>4.01896</td>
<td>5.17107</td>
</tr>
<tr>
<td>2018-01-02</td>
<td>3.28170</td>
<td>4.32130</td>
<td>5.31726</td>
<td>4.10800</td>
<td>5.12472</td>
</tr>
<tr>
<td>2018-01-03</td>
<td>3.39199</td>
<td>4.60176</td>
<td>5.90753</td>
<td>4.06752</td>
<td>5.24010</td>
</tr>
<tr>
<td>2018-01-04</td>
<td>2.05339</td>
<td>3.21789</td>
<td>4.37279</td>
<td>3.99131</td>
<td>5.11851</td>
</tr>
<tr>
<td>2018-01-05</td>
<td>4.73718</td>
<td>5.65588</td>
<td>6.60197</td>
<td>4.14160</td>
<td>5.13381</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>trend_97.5</th>
<th>regression_2.5</th>
<th>regression</th>
<th>regression_97.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.39624</td>
<td>0.83381</td>
<td>1.13830</td>
</tr>
<tr>
<td>1</td>
<td>6.16573</td>
<td>-1.01967</td>
<td>-0.81734</td>
</tr>
<tr>
<td>2</td>
<td>6.53175</td>
<td>-0.91607</td>
<td>-0.61901</td>
</tr>
<tr>
<td>3</td>
<td>6.26958</td>
<td>-2.30892</td>
<td>-1.89482</td>
</tr>
<tr>
<td>4</td>
<td>6.08087</td>
<td>0.31028</td>
<td>0.55018</td>
</tr>
</tbody>
</table>
```

13.3. Fitting a Model with Regressors 73
13.4 Visualization of Regression Coefficient Curves

The function `get_regression_coefs` to extract coefficients (they will have central credibility intervals if the argument `include_ci=True` is used).

```
[10]: coef_mid, coef_lower, coef_upper = ktr.get_regression_coefs(include_ci=True)
[11]: coef_mid.head(5)

<table>
<thead>
<tr>
<th></th>
<th>date</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2018-01-01</td>
<td>0.03861</td>
<td>0.06444</td>
<td>0.15967</td>
</tr>
<tr>
<td>1</td>
<td>2018-01-02</td>
<td>0.03766</td>
<td>0.06358</td>
<td>0.15637</td>
</tr>
<tr>
<td>2</td>
<td>2018-01-03</td>
<td>0.03670</td>
<td>0.06271</td>
<td>0.15305</td>
</tr>
<tr>
<td>3</td>
<td>2018-01-04</td>
<td>0.03573</td>
<td>0.06182</td>
<td>0.14970</td>
</tr>
<tr>
<td>4</td>
<td>2018-01-05</td>
<td>0.03475</td>
<td>0.06092</td>
<td>0.14633</td>
</tr>
</tbody>
</table>
```

Because this is simulated data it is possible to overlay the estimate with the true coefficients.

```
[12]: fig, axes = plt.subplots(p, 1, figsize=(12, 12), sharex=True)

    x = np.arange(coef_mid.shape[0])
    for idx in range(p):
        axes[idx].plot(x, coef_mid['x{}'].format(idx + 1)), label='est' if idx == 0 else '',
                        color=OrbitPalette.BLUE.value)
        axes[idx].fill_between(x, coef_lower['x{}'].format(idx + 1)], coef_upper['x{}'].
                        format(idx + 1)], alpha=0.2, color=OrbitPalette.BLUE.value)
        axes[idx].scatter(x, rw_data['beta{}'].format(idx + 1)], label='truth' if idx == 0,
                        else '', s=10, alpha=0.6, color=OrbitPalette.BLACK.value)
        axes[idx].set_title('beta{}'.format(idx + 1))

    fig.legend(bbox_to_anchor = (1,0.5));
```
To plot coefficients use the function `plot_regression_coefs` from the KTR class.

```python
[13]: ktr.plot_regression_coefs(figsize=(10, 5), include_ci=True);
```
These type of time-varying coefficients detection problems are not new. Bayesian approach such as the R packages Bayesian Structural Time Series (a.k.a BSTS) by Scott and Varian (2014) and tvReg by Isabel Casas and Ruben Fernandez-Casal (2021). Other frequentist approach such as Wu and Chiang (2000).

For further studies on benchmarking coefficients detection, Ng, Wang and Dai (2021) provides a detailed comparison of KTR with other popular time-varying coefficients methods; KTR demonstrates superior performance in the random walk data simulation.

### 13.5 Customizing Priors and Number of Knot Segments

To demonstrate how to specify the number of knots and priors consider the sine-cosine like simulated dataset. In this dataset, the fitting is more tricky since there could be some better way to define the number and position of the knots. There are obvious “change points” within the sine-cosine like curves. In KTR there are a few arguments that can leveraged to assign a priori knot attributes:

1. `regressor_init_knot_loc` is used to define the prior mean of the knot value. e.g. in this case, there is not a lot of prior knowledge so zeros are used.

2. The `regressor_init_knot_scale` and `regressor_knot_scale` are used to tune the prior sd of the global mean of the knot and the sd of each knot from the global mean respectively. These create a plausible range for the knot values.

3. The `regression_segments` defines the number of between knot segments (the number of knots - 1). The higher the number of segments the more change points are possible.

```python
[14]: ktr = KTR(
    response_col=response_col,
    date_col=date_col,
    regressor_col=regressor_col,
    regressor_init_knot_loc=[0] * len(regressor_col),
    regressor_init_knot_scale=[1.0] * len(regressor_col),
    regressor_knot_scale=[2.0] * len(regressor_col),
)```

(continues on next page)
regression_segments=6,
    prediction_percentiles=[2.5, 97.5],
    seed=2021,
    estimator='pyro-svi',
)

ktr.fit(df=sc_data)

2024-03-13 21:54:05 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-13 21:54:05 - orbit - INFO - Using SVI (Pyro) with steps: 301, samples: 100, learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
INFO:orbit:Using SVI (Pyro) with steps: 301, samples: 100, learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
2024-03-13 21:54:05 - orbit - INFO - step 0 loss = 828.02, scale = 0.10882
INFO:orbit:step 0 loss = 828.02, scale = 0.10882
2024-03-13 21:54:05 - orbit - INFO - step 100 loss = 340.58, scale = 0.87797
INFO:orbit:step 100 loss = 340.58, scale = 0.87797
2024-03-13 21:54:06 - orbit - INFO - step 200 loss = 266.67, scale = 0.37411
INFO:orbit:step 200 loss = 266.67, scale = 0.37411
2024-03-13 21:54:06 - orbit - INFO - step 300 loss = 261.21, scale = 0.43775
INFO:orbit:step 300 loss = 261.21, scale = 0.43775

[14]: <orbit.forecaster.svi.SVIForecaster at 0x2aa70a410>

[15]: coef_mid, coef_lower, coef_upper = ktr.get_regression_coefs(include_ci=True)
    fig, axes = plt.subplots(p, 1, figsize=(12, 12), sharex=True)
    x = np.arange(coef_mid.shape[0])
    for idx in range(p):
        axes[idx].plot(x, coef_mid['x{idx}'].format(idx + 1), label='est' if idx == 0 else '', color=OrbitPalette.BLUE.value)
        axes[idx].fill_between(x, coef_lower['x{idx}'].format(idx + 1), coef_upper['x{idx}'].format(idx + 1), alpha=0.2, color=OrbitPalette.BLUE.value)
        axes[idx].scatter(x, sc_data['beta{idx}'].format(idx + 1), label='truth' if idx == 0 else '', s=10, alpha=0.6, color=OrbitPalette.BLACK.value)
    axes[idx].set_title('beta{idx}'.format(idx + 1))

fig.legend(bbox_to_anchor=(1, 0.5));
Visualize the knots using the `plot_regression_coefs` function with `with_knot=True`.

```
[16]: ktr.plot_regression_coefs(with_knot=True, figsize=(10, 5), include_ci=True);
```
There are more ways to define knots for regression as well as seasonality and trend (a.k.a levels). These are described in Part III.

13.6 References


CHAPTER FOURTEEN

KERNEL-BASED TIME-VARYING REGRESSION - PART III

The tutorials I and II described the KTR model, its fitting procedure, visualizations and diagnostics / validation methods. This tutorial covers more KTR configurations for advanced users. In particular, it describes how to use knots to model change points in the seasonality and regression coefficients.

For more detail on this see Ng, Wang and Dai (2021), which describes how KTR knots can be thought of as change points. This highlights a similarity between KTR and Facebook’s Prophet package which introduces the change point detection on levels.

Part III covers different KTR arguments to specify knots position:

• level_segment
• level_knot_distance
• level_knot_dates

[1]:
import pandas as pd
import numpy as np
from math import pi
import matplotlib.pyplot as plt
import orbit
from orbit.models import KTR
from orbit.diagnostics.plot import plot_predicted_data
from orbit.utils.plot import get_orbit_style
from orbit.utils.dataset import load_iclaims

%matplotlib inline
pd.set_option('display.float_format', lambda x: '%.5f' % x)

[2]:
print(orbit.__version__)
1.1.4.4
14.1 Fitting with iClaims Data

The iClaims data set gives the weekly log number of claims and several regressors.

```python
# without the endate, we would get end date='2018-06-24' to make our tutorial consistent with the older version
df = load_iclaims(end_date='2020-11-29')

DATE_COL = 'week'
RESPONSE_COL = 'claims'

print(df.shape)
df.head()
```

```
(570, 7)
```

14.1.1 Specifying Levels Segments

The first way to specify the knot locations and number is the level_segments argument. This gives the number of between knot segments; since there is a knot on each end of each the total number of knots would be the number of segments plus one. To illustrate that, try level_segments=10 (line 5).

```python
response_col = 'claims'
date_col='week'

ktr = KTR(
    response_col=response_col,
    date_col=date_col,
    level_segments=10,
    prediction_percentiles=[2.5, 97.5],
    seed=2020,
    estimator='pyro-svi'
)

ktr.fit(df=df)
_ = ktr.plot_lev_knots()
```
Note that there are precisely there are 11 knots (triangles) evenly spaced in the above chart.

### 14.1.2 Specifying Knots Distance

An alternative way of specifying the number of knots is the `level_knot_distance` argument. This argument gives the distance between knots. It can be useful as number of knots grows with the length of the time-series. Note that if the total length of the time-series is not a multiple of `level_knot_distance` the first segment will have a different length. For example, in a weekly data, by putting `level_knot_distance=104` roughly means putting a knot once in two years.

```python
[ktr = KTR(  
  response_col=response_col,  
  date_col=date_col,  
  level_knot_distance=104,
```

(continues on next page)
# fit a weekly seasonality
seasonality=52,
# high order for sharp turns on each week
seasonality_fs_order=12,
prediction_percentiles=[2.5, 97.5],
seed=2020,
estimator='pyro-svi'
)

```
[8]: ktr.fit(df=df)
  _ = ktr.plot_lev_knots()
```

In the above chart, the knots are located about every 2-years.

To highlight the value of the next method of configuring knot position, consider the prediction for this model show below.
As the knots are placed evenly the model cannot adequately describe the change point in early 2020. The model fit can potentially be improved by inserting knots around the sharp change points (e.g., 2020-03-15). This insertion can be done with the `level_knot_dates` argument described below.

### 14.1.3 Specifying Knots Dates

The `level_knot_dates` argument allows for the explicit placement of knots. It needs a string of dates; see line 4.

```python
[10]: ktr = KTR(
    response_col=response_col,
    date_col=date_col,
    level_knot_dates = ['2010-01-03', '2020-03-15', '2020-03-22', '2020-11-29'],
    # fit a weekly seasonality
    seasonality=52,
    # high order for sharp turns on each week
    seasonality_fs_order=12,
    prediction_percentiles=[2.5, 97.5],
    seed=2020,
    estimator='pyro-svi'
)
```

```python
[11]: ktr.fit(df=df)
2024-03-13 21:54:29 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
```

(continues on next page)
INFO:orbit: Using SVI (Pyro) with steps: 301, samples: 100, learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.

INFO:orbit: step 0 loss = 99.354, scale = 0.096314
INFO:orbit: step 100 loss = -440.9, scale = 0.027049
INFO:orbit: step 200 loss = -446.03, scale = 0.028019
INFO:orbit: step 300 loss = -445.62, scale = 0.029141
INFO:orbit: step 301 loss = -445.62, scale = 0.029141

[11]: <orbit.forecaster.svi.SVIForecaster at 0x2ad2bdb50>

[12]: _ = ktr.plot_lev_knots()

[13]: predicted_df = ktr.predict(df=df)
_ = plot_predicted_data(training_actual_df=df, predicted_df=predicted_df, prediction_percentiles=[2.5, 97.5],
    date_col=date_col, actual_col=response_col)
Note this fit is even better than the previous one using less knots. Of course, the case here is trivial because the pandemic onset is treated as known. In other cases, there may not be an obvious way to find the optimal knots dates.

### 14.2 Conclusion

This tutorial demonstrates multiple ways to customize the knots location for levels. In KTR, there are similar arguments for seasonality and regression such as `seasonality_segments` and `regression_knot_dates` and `regression_segments`. Due to their similarities with their knots location equivalent arguments they are not demonstrated here. However it is encouraged for KTR users to explore them.

### 14.3 References


CHAPTER
FIFTEEN

KERNEL-BASED TIME-VARYING REGRESSION - PART IV

This is final tutorial on KTR. It continues from Part III with additional details on some of the advanced arguments. For other details on KTR see either the previous three tutorials or the original paper Ng, Wang and Dai (2021).

In Part IV covers advance inputs for regression including

- regressors signs
- time-point coefficients priors

```
[1]: import pandas as pd
    import numpy as np
    from math import pi
    import matplotlib.pyplot as plt

    import orbit
    from orbit.models import KTR
    from orbit.diagnostics.plot import plot_predicted_components
    from orbit.utils.plot import get_orbit_style
    from orbit.utils.kernels import gauss_kernel
    from orbit.constants.palette import OrbitPalette

    %matplotlib inline
    pd.set_option('display.float_format', lambda x: '%.5f % x)
    orbit_style = get_orbit_style()
    plt.style.use(orbit_style);

[2]: print(orbit.__version__)
    1.1.4.4
```

15.1 Data

To demonstrate the effect of specifying regressors coefficients sign, it is helpful to modify the data simulation code from part II. The simulation is altered to impose strictly positive regression coefficients.

In the KTR model below, the coefficient curves are approximated with Gaussian kernels having positive values of knots. The levels are also included in the process with vector of ones as the covariates.

The parameters used to setup the data simulation are:

- n : number of time steps
- p : number of predictors
```python
[3]: np.random.seed(2021)
    n = 300
    p = 2
    tp = np.arange(1, 301) / 300
    knot_tp = np.array([1, 100, 200, 300]) / 300
    beta_knot = np.array(
        [[1.0, 0.1, 0.15],
         [3.0, 0.01, 0.05],
         [3.0, 0.01, 0.05],
         [2.0, 0.05, 0.02]]
    )
    gk = gauss_kernel(tp, knot_tp, rho=0.2)
    beta = np.matmul(gk, beta_knot)
    covar_lev = np.ones((n, 1))
    covar = np.concatenate((covar_lev, np.random.normal(0, 1.0, (n, p))), axis=1)
    # observation with noise
    y = (covar * beta).sum(-1) + np.random.normal(0, 0.1, n)
    regressor_col = ['x{}'.format(pp) for pp in range(1, p+1)]
    data = pd.DataFrame(covar[:, 1:], columns=regressor_col)
    data['y'] = y
    data['date'] = pd.date_range(start='1/1/2018', periods=len(y))
    data = data[['date', 'y'] + regressor_col]
    beta_data = pd.DataFrame(beta[:, 1:], columns=beta_col)
    data = pd.concat([data, beta_data], axis=1)
[4]: data.tail(10)

<table>
<thead>
<tr>
<th>date</th>
<th>y</th>
<th>x1</th>
<th>x2</th>
<th>beta1</th>
<th>beta2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-10-18</td>
<td>2.15947</td>
<td>-0.62762</td>
<td>0.17840</td>
<td>0.04015</td>
<td>0.02739</td>
</tr>
<tr>
<td>2018-10-19</td>
<td>2.25871</td>
<td>-0.92975</td>
<td>0.81415</td>
<td>0.04036</td>
<td>0.02723</td>
</tr>
<tr>
<td>2018-10-20</td>
<td>2.24356</td>
<td>0.82438</td>
<td>-0.92705</td>
<td>0.04057</td>
<td>0.02707</td>
</tr>
<tr>
<td>2018-10-21</td>
<td>2.26948</td>
<td>1.57181</td>
<td>-0.78098</td>
<td>0.04077</td>
<td>0.02692</td>
</tr>
<tr>
<td>2018-10-22</td>
<td>2.26375</td>
<td>-1.07504</td>
<td>-0.86523</td>
<td>0.04097</td>
<td>0.02677</td>
</tr>
<tr>
<td>2018-10-23</td>
<td>2.21349</td>
<td>0.24637</td>
<td>-0.98398</td>
<td>0.04117</td>
<td>0.02663</td>
</tr>
<tr>
<td>2018-10-24</td>
<td>2.13297</td>
<td>-0.58716</td>
<td>0.59911</td>
<td>0.04136</td>
<td>0.02648</td>
</tr>
<tr>
<td>2018-10-25</td>
<td>2.09049</td>
<td>-2.01610</td>
<td>0.08618</td>
<td>0.04155</td>
<td>0.02634</td>
</tr>
<tr>
<td>2018-10-26</td>
<td>2.14302</td>
<td>0.33863</td>
<td>-0.37912</td>
<td>0.04173</td>
<td>0.02620</td>
</tr>
<tr>
<td>2018-10-27</td>
<td>2.10795</td>
<td>-0.96160</td>
<td>-0.42383</td>
<td>0.04192</td>
<td>0.02606</td>
</tr>
</tbody>
</table>
```

Just like previous tutorials in regression, some additional args are used to describe the regressors and the scale parameters for the knots.

```python
[5]: ktr = KTR(
        response_col='y',
        date_col='date',
        regressor_col=regressor_col,
```

(continues on next page)
regressor_init_knot_scale=\[0.1\] * p,
regressor_knot_scale=\[0.1\] * p,
prediction_percentiles=\[2.5, 97.5\],
seed=2021,
estimator='pyro-svi',
)
ktr.fit(df=data)

2024-03-13 21:55:03 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-13 21:55:03 - orbit - INFO - Using SVI (Pyro) with steps: 301, samples: 100,␣
˓→learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
˓→torch/__init__.py:690: UserWarning: torch.set_default_tensor_type() is deprecated as␣
˓→of PyTorch 2.1, please use torch.set_default_dtype() and torch.set_default_device() as␣
˓→alternatives. (Triggered internally at /Users/runner/work/pytorch/pytorch/pytorch/
˓→torch/csrc/tensor/python_tensor.cpp:453.)
  C._set_default_tensor_type(t)
2024-03-13 21:55:04 - orbit - INFO - step 0 loss = -3.5592, scale = 0.085307
2024-03-13 21:55:04 - orbit - INFO - step 100 loss = -228.48, scale = 0.036575
2024-03-13 21:55:05 - orbit - INFO - step 200 loss = -230.1, scale = 0.038104
2024-03-13 21:55:05 - orbit - INFO - step 300 loss = -229.33, scale = 0.037629
INFO:orbit:step 0 loss = -3.5592, scale = 0.085307
INFO:orbit:step 100 loss = -228.48, scale = 0.036575
INFO:orbit:step 200 loss = -230.1, scale = 0.038104
INFO:orbit:step 300 loss = -229.33, scale = 0.037629

The get_regression_coefs argument is used to extract coefficients with intervals by supplying the argument
include_ci=True.

coef_mid, coef_lower, coef_upper = ktr.get_regression_coefs(include_ci=True)

The next figure shows the overlay of the estimate on the true coefficients. Since the lower bound is below zero some of
the coefficient posterior samples are negative.

fig, axes = plt.subplots(p, 1, figsize=(12, 12), sharex=True)
x = np.arange(coef_mid.shape[0])
for idx in range(p):
    axes[idx].plot(x, coef_mid['x{idx}'.format(idx + 1)], label='est' if idx == 0 else '',␣
˓→alpha=0.8, color=OrbitPalette.BLUE.value)
    axes[idx].fill_between(x, coef_lower['x{idx}'.format(idx + 1)], coef_upper['x{idx}'.
˓→format(idx + 1)], alpha=0.15, color=OrbitPalette.BLUE.value)
    axes[idx].plot(x, data['beta{idx}'.format(idx + 1)], label='truth' if idx == 0 else '',␣
ｒ˓→alpha=0.6, color = OrbitPalette.BLACK.value)
    axes[idx].set_title('beta{idx}'.format(idx + 1))
fig.legend(bbox_to_anchor = (1,0.5));
15.3 Regressor Sign

Strictly positive coefficients can be imposed by using the `regressor_sign` arg. It can have values “=”, “-”, or “+” which denote no restriction, strictly negative, strictly positive. Note that it is possible to have a mixture by providing a list of strings one for each regressor.

```
[8]: ktr = KTR(
    response_col='y',
    date_col='date',
    regressor_col=regressor_col,
    regressor_init_knot_scale=[0.1] * p,
    regressor_knot_scale=[0.1] * p,
    regressor_sign=['+'] * p,
    prediction_percentiles=[2.5, 97.5],
```

(continues on next page)
seed=2021,
estimator='pyro-svi',
)
ktr.fit(df=data)

2024-03-13 21:55:06 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-13 21:55:06 - orbit - INFO - Using SVI (Pyro) with steps: 301, samples: 100, learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
INFO:orbit:Using SVI (Pyro) with steps: 301, samples: 100, learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
2024-03-13 21:55:06 - orbit - INFO - step 0 loss = 9.7371, scale = 0.10482
INFO:orbit:step 0 loss = 9.7371, scale = 0.10482
2024-03-13 21:55:06 - orbit - INFO - step 100 loss = -231.22, scale = 0.41649
INFO:orbit:step 100 loss = -231.22, scale = 0.41649
2024-03-13 21:55:06 - orbit - INFO - step 200 loss = -230.94, scale = 0.42589
INFO:orbit:step 200 loss = -230.94, scale = 0.42589
2024-03-13 21:55:06 - orbit - INFO - step 300 loss = -230.26, scale = 0.41749
INFO:orbit:step 300 loss = -230.26, scale = 0.41749

[8]: <orbit.forecaster.svi.SVIForecaster at 0x2aea5a690>

[9]: coef_mid, coef_lower, coef_upper = ktr.get_regression_coefs(include_ci=True)

[10]: fig, axes = plt.subplots(p, 1, figsize=(12, 12), sharex=True)
x = np.arange(coef_mid.shape[0])
for idx in range(p):
    axes[idx].plot(x, coef_mid['x{}'.format(idx + 1)], label='est' if idx == 0 else '', alpha=0.8, color=OrbitPalette.BLUE.value)
    axes[idx].fill_between(x, coef_lower['x{}'.format(idx + 1)], coef_upper['x{}'.format(idx + 1)], alpha=0.15, color=OrbitPalette.BLUE.value)
    axes[idx].plot(x, data['beta{}'.format(idx + 1)], label='truth' if idx == 0 else '', alpha=0.6, color = OrbitPalette.BLACK.value)
    axes[idx].set_title('beta{}'.format(idx + 1))

fig.legend(bbox_to_anchor = (1,0.5));
Observe the curves lie in the positive range with a slightly improved fit relative to the last model.

To conclude, it is useful to have a strictly positive range of regression coefficients if that range is known a priori. KTR allows these priors to be specified. For regression scenarios where there is no a priori knowledge of the coefficient sign it is recommended to use the default which contains both sides of the range.
15.4 Time-point coefficient priors

Users can incorporate coefficient priors for any regressor and any time period. This feature is quite useful when users have some prior knowledge or beliefs on regressor coefficients. For example, if an A/B test is conducted for a certain regressor over a specific time range, then users can ingest the priors derived from such A/B test.

This can be done by supplying a list of dictionaries via `coef_prior_list`. Each dict in the list should have keys as `name`, `prior_start_tp_idx` (inclusive), `prior_end_tp_idx` (not inclusive), `prior_mean`, `prior_sd`, and `prior_regressor_col`.

Below is an illustrative example by using the simulated data above.

```
[11]: from copy import deepcopy

[12]: prior_duration = 50
       coef_list_dict = []
       prior_idx=[
                   np.arange(150, 150 + prior_duration),
                   np.arange(200, 200 + prior_duration),
                   ]
       regressor_idx = range(1, p + 1)
       plot_dict = {}
       for i in regressor_idx:
                   plot_dict[i] = {'idx': [], 'val': []}

[13]: for idx, idx2, regressor in zip(prior_idx, regressor_idx, regressor_col):
                   prior_dict = {}
                   prior_dict['name'] = f'prior_{regressor}''
                   prior_dict['prior_start_tp_idx'] = idx[0]
                   prior_dict['prior_end_tp_idx'] = idx[-1] + 1
                   prior_dict['prior_mean'] = beta[idx, idx2]
                   prior_dict['prior_sd'] = [0.1] * len(idx)
                   prior_dict['prior_regressor_col'] = [regressor] * len(idx)

                   plot_dict[idx2]['idx'].extend(idx)
                   plot_dict[idx2]['val'].extend(beta[idx, idx2])

                   coef_list_dict.append(deepcopy(prior_dict))

[14]: ktr = KTR(
               response_col='y',
               date_col='date',
               regressor_col=regressor_col,
               regressor_init_knot_scale=[0.1] * p,
               regressor_knot_scale=[0.1] * p,
               regressor_sign=['+'] * p,
               coef_prior_list=coef_list_dict,
               prediction_percentiles=[2.5, 97.5],
               seed=2021,
               estimator='pyro-svi',
           )
       ktr.fit(df=data)
```
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Using SVI (Pyro) with steps: 301, samples: 100, learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
INFO:orbit:step 0 loss = -5741.9, scale = 0.094521
INFO:orbit:step 100 loss = -7140.3, scale = 0.31416
INFO:orbit:step 200 loss = -7139.1, scale = 0.31712
INFO:orbit:step 300 loss = -7139.5, scale = 0.33039
INFO:orbit:step 300 loss = -7139.5, scale = 0.33039

[14]: <orbit.forecaster.svi.SVIForecaster at 0x2ae79a690>

[15]: coef_mid, coef_lower, coef_upper = ktr.get_regression_coefs(include_ci=True)

[16]: fig, axes = plt.subplots(p, 1, figsize=(10, 8), sharex=True)

    x = np.arange(coef_mid.shape[0])
    for idx in range(p):
        axes[idx].plot(x, coef_mid['x{0}'.format(idx + 1)], label='est', alpha=0.8, color=OrbitPalette.BLUE.value)
        axes[idx].fill_between(x, coef_lower['x{0}'.format(idx + 1)], coef_upper['x{0}'.format(idx + 1)], alpha=0.15, color=OrbitPalette.BLUE.value)
        axes[idx].plot(x, data['beta{0}'.format(idx + 1)], label='truth', alpha=0.6, color=OrbitPalette.BLACK.value)
        axes[idx].set_title('beta{0}'.format(idx + 1))
        axes[idx].scatter(plot_dict[idx + 1]['idx'], plot_dict[idx + 1]['val'], s=5, color=OrbitPalette.RED.value, alpha=.6, label='ingested priors')
        handles, labels = axes[0].get_legend_handles_labels()
    fig.legend(handles, labels, loc='upper center', ncol=3, bbox_to_anchor=(.5, 1.05))
    plt.tight_layout()
As seen above, for the ingested prior time window, the estimation is aligned better with the truth and the resulting confidence interval also becomes narrower compared to other periods.

15.5 References

In this section, we will demonstrate how to visualize

- time series forecasting
- predicted components

by using the plotting utilities that come with the Orbit package.

[1]: %matplotlib inline

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import orbit
from orbit.models import DLT
from orbit.diagnostics.plot import plot_predicted_data, plot_predicted_components
from orbit.utils.dataset import load_iclaims
```

[2]:

```python
print(orbit.__version__)
```

```
1.1.4.4
```

[3]:

```python
# load log-transformed data
df = load_iclaims()
train_df = df[df['week'] < '2017-01-01']
test_df = df[df['week'] >= '2017-01-01']

response_col = 'claims'
date_col = 'week'
regressor_col = ['trend.unemploy', 'trend.filling', 'trend.job']
```
16.1 Fit a model

Here we use the DLTFull model as example.

```python
[dlt = DLT(
    response_col=response_col,
    regressor_col=regressor_col,
    date_col=date_col,
    seasonality=52,
    prediction_percentiles=[5, 95],
    stan_mcmc_args={'show_progress': False},
)

dlt.fit(train_df)
```

2024-03-13 21:44:09 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8, →
˓temperature: 1.000, warmups (per chain): 225 and samples(per chain): 25.

16.2 Plot Predictions

First, we do the prediction on the training data before the year 2017.

```python
[predicted_df = dlt.predict(df=train_df, decompose=True)
_ = plot_predicted_data(train_df, predicted_df,
    date_col=dlt.date_col, actual_col=dlt.response_col)
```

Next, we do the predictions on the test data after the year 2017. This plot is useful to help check the overall model performance on the out-of-sample period.
16.3 Plot Predicted Components

plot_predicted_components is the utility to plot each component separately. This is useful when one wants to look into the model prediction results and inspect each component separately.

[6]:
```python
predicted_df = dlt.predict(df=test_df, decompose=True)
```

```python
_ = plot_predicted_data(training_actual_df=train_df, predicted_df=predicted_df,
                        date_col=dlt.date_col, actual_col=dlt.response_col,
                        test_actual_df=test_df)
```

[7]:
```python
predicted_df = dlt.predict(df=train_df, decompose=True)
```

```python
_ = plot_predicted_components(predicted_df, date_col)
```
One can use `plot_components` to have more components to be plotted if they are available in the supplied `predicted_df`.

```
[8]: _ = plot_predicted_components(predicted_df, date_col,
    plot_components=['prediction', 'trend', 'seasonality',
    'regression'])
```
In this section, we introduce a few recommended diagnostic plots to diagnostic Orbit models. The posterior samples in SVI and Full Bayesian i.e. FullBayesianForecaster and SVIForecaster.

The plots are created by arviz for the plots. ArviZ is a Python package for exploratory analysis of Bayesian models, includes functions for posterior analysis, data storage, model checking, comparison and diagnostics.

- Trace plot
- Pair plot
- Density plot

```
[1]: import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt
    import arviz as az
    import seaborn as sns

    %matplotlib inline

    import orbit
    from orbit.models import LGT, DLT
    from orbit.utils.dataset import load_iclaims

    import warnings
    warnings.filterwarnings('ignore')

    from orbit.diagnostics.plot import params_comparison_boxplot
    from orbit.constants import palette

[2]: print(orbit.__version__)
    1.1.4.4
```
17.1 Load data

```python
[3]: df = load_iclaims()
df.dtypes

[3]:
<table>
<thead>
<tr>
<th></th>
<th>dtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>week</td>
<td>datetime64[ns]</td>
</tr>
<tr>
<td>claims</td>
<td>float64</td>
</tr>
<tr>
<td>trend.unemploy</td>
<td>float64</td>
</tr>
<tr>
<td>trend.filling</td>
<td>float64</td>
</tr>
<tr>
<td>trend.job</td>
<td>float64</td>
</tr>
<tr>
<td>sp500</td>
<td>float64</td>
</tr>
<tr>
<td>vix</td>
<td>float64</td>
</tr>
<tr>
<td>dtype: object</td>
<td></td>
</tr>
</tbody>
</table>

[4]: df.head(5)

[4]:
<table>
<thead>
<tr>
<th>week</th>
<th>claims</th>
<th>trend.unemploy</th>
<th>trend.filling</th>
<th>trend.job</th>
<th>sp500</th>
<th>vix</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010-01-03</td>
<td>13.386595</td>
<td>0.219882</td>
<td>-0.318452</td>
<td>0.117500</td>
<td>-0.417633</td>
<td></td>
</tr>
<tr>
<td>2010-01-03</td>
<td>13.624218</td>
<td>0.219882</td>
<td>-0.194838</td>
<td>0.168794</td>
<td>-0.425480</td>
<td></td>
</tr>
<tr>
<td>2010-01-17</td>
<td>13.398741</td>
<td>0.236143</td>
<td>-0.292477</td>
<td>0.117500</td>
<td>-0.465229</td>
<td></td>
</tr>
<tr>
<td>2010-01-24</td>
<td>13.137549</td>
<td>0.203353</td>
<td>-0.194838</td>
<td>0.106918</td>
<td>-0.481751</td>
<td></td>
</tr>
<tr>
<td>2010-01-31</td>
<td>13.196760</td>
<td>0.134360</td>
<td>-0.242466</td>
<td>0.074483</td>
<td>-0.488929</td>
<td></td>
</tr>
<tr>
<td>vix</td>
<td>0.122654</td>
<td>0.110445</td>
<td>0.532339</td>
<td>0.428645</td>
<td>0.487404</td>
<td></td>
</tr>
</tbody>
</table>

17.2 Fit a Model

[5]: DATE_COL = 'week'
RESPONSE_COL = 'claims'
REGRESSOR_COL = ['trend.unemploy', 'trend.filling', 'trend.job']

[6]: dlt = DLT(  
    response_col=RESPONSE_COL,  
    date_col=DATE_COL,  
    regressor_col=REGRESSOR_COL,  
    seasonality=52,  
    num_warmup=2000,  
    num_sample=2000,  
    chains=4,  
    stan_mcmc_args={'show_progress': False},
)

[7]: dlt.fit(df=df)

2024-03-13 21:56:41 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,→
→temperature: 1.000, warmups (per chain): 500 and samples(per chain): 500.
RuntimeError

---

Cell In[7], line 1
----> 1 dlt.fit(df=df)

File ~/opt/miniconda3/envs/orbit-install-test/lib/python3.11/site-packages/orbit/forecaster/full_bayes.py:36, in FullBayesianForecaster.fit(self, df, point_method, keep_samples, sampling_temperature, **kwargs)
    def fit(self, df, **kwargs):
    super().fit(df, sampling_temperature=sampling_temperature, **kwargs)
    self._point_method = point_method
    if point_method is not None:

    _posterior_samples, training_metrics = estimator.fit(
        model_name=model_name,
        model_param_names=model_param_names,
        data_input=data_input,
        fitter=self._model.get_fitter(),
        init_values=init_values,
        **kwargs,
    )
    self._posterior_samples = _posterior_samples
    self._training_metrics = training_metrics

    compiled_mod = get_compiled_stan_model(stan_model_name=model_name)
    # check https://mc-stan.org/cmdstanpy/api.html#cmdstanpy.CmdStanModel.sample
    # for additional args
    stan_mcmc_fit = compiled_mod.sample(
        data=data_input,
        iter_sampling=self._num_sample_per_chain,
        iter_warmup=self._num_warmup_per_chain,
        chains=self.chains,
        parallel_chains=self.cores,
        init_values=init_values,
        seed=self.seed,
        **self._stan_mcmc_args,
    )
    stan_extract = stan_mcmc_fit.stan_variables()
    posteriors = {
        param: stan_extract[param] for param in model_param_names + ["loglk"]
    }

17.2. Fit a Model
We can use .get_posterior_samples() to extract posteriors. Note that we need permute=False to retrieve additional information such as chains when we extract posterior samples for posteriors plotting. For regression, in order to collapse and relabel regression from parameters (usually called as beta), we use relabel=True.

```python
[ ]: ps = dlt.get_posterior_samples(relabel=True, permute=False)
ps.keys()
```
17.3 Diagnostics Visualization

In the following section, we are going to use the regression coefficients as an example. In practice, you could check other parameters extracted from the model. For now, it only supports 1-D parameter which in generally capture the most important parameters of the model (e.g. obs_sigma, lev_sm etc.)

17.3.1 Convergence Status

Trace plots help us verify the convergence of model. In general, a largely overlapped distribution across samples from different chains indicates the convergence.

```py
[ ]: az.style.use('arviz-darkgrid')
az.plot_trace(
    ps,
    var_names=['trend.unemploy', 'trend.filling', 'trend.job'],
    chain_prop={'color': ['r', 'b', 'g', 'y']},
    figsize=(10, 8),
)
```

Note that this is only applicable for FullBayesianForecaster using sampling method such as MCMC.

17.3.2 Samples Density

We can also check the density of samples by pair plot.

```py
[ ]: az.plot_pair(
    ps,
    var_names=['trend.unemploy', 'trend.filling', 'trend.job'],
    kind=['scatter', 'kde'],
    marginals=True,
    point_estimate='median',
    textsize=18.5,
)
```

17.3.3 Compare Models

You can also compare posteriors across different models with the same parameters. You can use plots such as density plot and forest plot to do so.

```py
[ ]: dlt_smaller_prior = DLT(
    response_col=RESPONSE_COL,
    date_col=DATE_COL,
    regressor_col=REGRESSOR_COL,
    regressor_sigma_prior=[0.05, 0.05, 0.05],
    seasonality=52,
    num_warmup=2000,
    num_sample=2000,
    chains=4,
    stan_mcmc_args={'show_progress': False},
)
```

(continues on next page)
dlt_smaller_prior.fit(df=df)
ps_smaller_prior = dlt_smaller_prior.get_posterior_samples(relabel=True, permute=False)

[ ]: az.plot_density(
    [ps, ps_smaller_prior],
    var_names=['trend.unemploy', 'trend.filling', 'trend.job'],
    data_labels=['Default', "Smaller Sigma"],
    shade=0.1,
    textsize=18.5,
);

[ ]: az.plot_forest(
    [ps, ps_smaller_prior],
    var_names=['trend.unemploy'],
    model_names=['Default', "Smaller Sigma"],
);

[ ]: params_comparison_boxplot(
    [ps, ps_smaller_prior],
    var_names=['trend.unemploy', 'trend.filling', 'trend.job'],
    model_names=['Default', "Smaller Sigma"],
    box_width = .1, box_distance=0.1,
    showfliers=True
);

17.4 Conclusion

Orbit models allow multiple visualization to diagnostics models and compare different models. We briefly introduce some basic syntax and usage of arviz. There is an example gallery built by the original team. Users can learn the details and more advance usage there. Meanwhile, the Orbit team aims to continue expand the scope to leverage more work done from the arviz project.
This section will cover following topics:

- How to create a TimeSeriesSplitter
- How to create a BackTester and retrieve the backtesting results
- How to leverage the backtesting to tune the hyper-parameters for orbit models

```python
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import orbit
from orbit.models import LGT, DLT
from orbit.diagnostics.backtest import BackTester, TimeSeriesSplitter
from orbit.diagnostics.plot import plot_bt_predictions
from orbit.diagnostics.metrics import smape, wmape
from orbit.utils.dataset import load_iclaims

import warnings
warnings.filterwarnings('ignore')

print(orbit.__version__)
1.1.4.4

# load log-transformed data
data = load_iclaims()
data.shape
(443, 7)
```

The way to gauge the performance of a time-series model is through re-training models with different historic periods and check their forecast within certain steps. This is similar to a time-based style cross-validation. More often, it is called backtest in time-series modeling.

The purpose of this notebook is to illustrate how to backtest a single model using BackTester. BackTester will compose a TimeSeriesSplitter within it, but TimeSeriesSplitter is useful as a standalone, in case there are other tasks to perform that requires splitting but not backtesting.
each ‘slices’ as generator, i.e it can be used in a for loop. You can also retrieve the composed TimeSeriesSplitter object from BackTester to utilize the additional methods in TimeSeriesSplitter.

Currently, there are two schemes supported for the back-testing engine: expanding window and rolling window.

- **expanding window**: for each back-testing model training, the train start date is fixed, while the train end date is extended forward.
- **rolling window**: for each back-testing model training, the training window length is fixed but the window is moving forward.

### 18.1 Create a TimeSeriesSplitter

There are two main ways to splitting a time series: expanding and rolling. Expanding window has a fixed starting point, and the window length grows as users move forward in time series. It is useful when users want to incorporate all historical information. On the other hand, rolling window has a fixed window length, and the starting point of the window moves forward as users move forward in time series. Below section illustrates how users can use TimeSeriesSplitter to split the claims time series.

#### 18.1.1 Expanding window

```python
# configs
min_train_len = 380 # minimal length of window length
forecast_len = 20 # length forecast window
incremental_len = 20 # step length for moving forward

ex_splitter = TimeSeriesSplitter(df=data,
                               min_train_len=min_train_len,
                               incremental_len=incremental_len,
                               forecast_len=forecast_len,
                               window_type='expanding',
                               date_col='week')

print(ex_splitter)
```

```
------------ Fold: (1 / 3)------------
Train start date: 2010-01-03 00:00:00 Train end date: 2017-04-09 00:00:00
Test start date: 2017-04-16 00:00:00 Test end date: 2017-08-27 00:00:00

------------ Fold: (2 / 3)------------
Train start date: 2010-01-03 00:00:00 Train end date: 2017-08-27 00:00:00
Test start date: 2017-09-03 00:00:00 Test end date: 2018-01-14 00:00:00

------------ Fold: (3 / 3)------------
Train start date: 2010-01-03 00:00:00 Train end date: 2018-01-14 00:00:00
Test start date: 2018-01-21 00:00:00 Test end date: 2018-06-03 00:00:00
```

Users can visualize the splits using the internal plot() function. One may notice that the last few data points may not be included in the last split, which is expected when min_train_len is specified.
18.1.2 Rolling window

```
# configs
min_train_len = 380  # in case of rolling window, this specify the length of window length
forecast_len = 20  # length forecast window
incremental_len = 20  # step length for moving forward

roll_splitter = TimeSeriesSplitter(data,
    min_train_len=min_train_len,
    incremental_len=incremental_len,
    forecast_len=forecast_len,
    window_type='rolling', date_col='week')
```

Users can visualize the splits, green is training window and yellow it the forecasting window. The window length is always 380, while the starting point moves forward 20 weeks each steps.

```
_ = roll_splitter.plot()
```
18.1.3 Specifying number of splits

User can also define number of splits using `n_splits` instead of specifying minimum training length. That way, minimum training length will be automatically calculated.

```
[13]: ex_splitter2 = TimeSeriesSplitter(data,
         min_train_len=min_train_len,
         incremental_len=incremental_len,
         forecast_len=forecast_len,
         n_splits=5,
         window_type='expanding', date_col='week')
```

```
[14]: _ = ex_splitter2.plot()
```

---

### Train/Test Split Scheme

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

18.1.4 TimeSeriesSplitter as generator

TimeSeriesSplitter is implemented as a generator, therefore users can call `split()` to loop through it. It comes handy even for tasks other than backtest.

```
[15]: for train_df, test_df, scheme, key in roll_splitter.split():
       print('Initial Claim slice {} rolling mean: {:.3f}'.format(key, train_df['claims'].mean()))
```

Initial Claim slice 0 rolling mean: 12.712
Initial Claim slice 1 rolling mean: 12.671
Initial Claim slice 2 rolling mean: 12.647

---

18.2 Create a BackTester

To actually run backtest, first let's initialize a DLT model and a BackTester. You pass in `TimeSeriesSplitter` parameters to BackTester.

```
[16]: # instantiate a model
      dlt = DLT(
              date_col='week',
              response_col='claims',
              regressor_col=['trend.unemploy', 'trend.filling', 'trend.job'],
              seasonality=52,
```

(continues on next page)
estimator='stan-map',
# reduce number of messages
verbose=False,
)

[17]: # configs
min_train_len = 100
forecast_len = 20
incremental_len = 100
window_type = 'expanding'

bt = BackTester(
    model=dlt,
    df=data,
    min_train_len=min_train_len,
    incremental_len=incremental_len,
    forecast_len=forecast_len,
    window_type=window_type,
)

**18.3 Backtest fit and predict**

The most expensive portion of backtesting is fitting the model iteratively. Thus, users can separate the API calls for `fit_predict` and `score` to avoid redundant computation for multiple metrics or scoring methods

[18]: bt.fit_predict()

Once `fit_predict()` is called, the fitted models and predictions can be easily retrieved from `BackTester`. Here the data is grouped by the date, split_key, and whether or not that observation is part of the training or test data

[19]:
predicted_df = bt.get_predicted_df()
predicted_df.head()

<table>
<thead>
<tr>
<th>date</th>
<th>actual</th>
<th>prediction</th>
<th>training_data</th>
<th>split_key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2010-01-03 13.386595</td>
<td>13.386576</td>
<td>True</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1 2010-01-10 13.624218</td>
<td>13.649070</td>
<td>True</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2 2010-01-17 13.398741</td>
<td>13.373163</td>
<td>True</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3 2010-01-24 13.137549</td>
<td>13.151905</td>
<td>True</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4 2010-01-31 13.196760</td>
<td>13.187853</td>
<td>True</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

A plotting utility is also provided to visualize the predictions against the actuals for each split.

[20]: plot_bt_predictions(predicted_df, metrics=smape, ncol=2, include_vline=True);
Users might find this useful for any custom computations that may need to be performed on the set of predicted data. Note that the columns are renamed to generic and consistent names.

Sometimes, it might be useful to match the data back to the original dataset for ad-hoc diagnostics. This can easily be done by merging back to the original dataset:

```
[predicted_df.merge(data, left_on='date', right_on='week')]
```

<table>
<thead>
<tr>
<th>date</th>
<th>actual</th>
<th>prediction</th>
<th>training_data</th>
<th>split_key</th>
<th>week</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010-01-03</td>
<td>13.386595</td>
<td>13.386576</td>
<td>True</td>
<td>0</td>
<td>2010-01-03</td>
</tr>
<tr>
<td>2010-01-10</td>
<td>13.624218</td>
<td>13.649070</td>
<td>True</td>
<td>0</td>
<td>2010-01-10</td>
</tr>
<tr>
<td>2010-01-17</td>
<td>13.398741</td>
<td>13.373163</td>
<td>True</td>
<td>0</td>
<td>2010-01-17</td>
</tr>
<tr>
<td>2010-01-24</td>
<td>13.137549</td>
<td>13.151905</td>
<td>True</td>
<td>0</td>
<td>2010-01-24</td>
</tr>
<tr>
<td>2010-01-31</td>
<td>13.196760</td>
<td>13.187853</td>
<td>True</td>
<td>0</td>
<td>2010-01-31</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2017-12-17</td>
<td>12.566428</td>
<td>12.566428</td>
<td>False</td>
<td>3</td>
<td>2017-12-17</td>
</tr>
<tr>
<td>2017-12-24</td>
<td>12.675789</td>
<td>12.675789</td>
<td>False</td>
<td>3</td>
<td>2017-12-24</td>
</tr>
<tr>
<td>2017-12-31</td>
<td>12.783320</td>
<td>12.783320</td>
<td>False</td>
<td>3</td>
<td>2017-12-31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>claims</th>
<th>trend.unemploy</th>
<th>trend.filling</th>
<th>trend.job</th>
<th>sp500</th>
<th>vix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13.386595</td>
<td>0.219882</td>
<td>-0.318452</td>
<td>0.117500</td>
<td>-0.417633</td>
</tr>
<tr>
<td>1</td>
<td>13.624218</td>
<td>0.219882</td>
<td>-0.194838</td>
<td>0.168794</td>
<td>-0.425480</td>
</tr>
<tr>
<td>2</td>
<td>13.398741</td>
<td>0.236143</td>
<td>-0.292477</td>
<td>0.117500</td>
<td>-0.465229</td>
</tr>
<tr>
<td>3</td>
<td>13.137549</td>
<td>0.203353</td>
<td>-0.194838</td>
<td>0.106918</td>
<td>-0.481751</td>
</tr>
<tr>
<td>4</td>
<td>13.196760</td>
<td>0.134360</td>
<td>-0.242466</td>
<td>0.074483</td>
<td>-0.488929</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1075</td>
<td>12.568616</td>
<td>0.298863</td>
<td>0.248654</td>
<td>-0.216869</td>
<td>0.434042</td>
</tr>
<tr>
<td>1076</td>
<td>12.691451</td>
<td>0.328516</td>
<td>0.233616</td>
<td>-0.358839</td>
<td>0.430410</td>
</tr>
<tr>
<td>1077</td>
<td>12.769532</td>
<td>0.503457</td>
<td>0.069313</td>
<td>-0.092571</td>
<td>0.456087</td>
</tr>
<tr>
<td>1078</td>
<td>12.908227</td>
<td>0.527849</td>
<td>0.051295</td>
<td>0.029532</td>
<td>0.471673</td>
</tr>
<tr>
<td>1079</td>
<td>12.777193</td>
<td>0.465717</td>
<td>0.032946</td>
<td>0.006275</td>
<td>0.480271</td>
</tr>
</tbody>
</table>

(continues on next page)
18.4 Backtest Scoring

The main purpose of BackTester are the evaluation metrics. Some of the most widely used metrics are implemented and built into the BackTester API.

The default metric list is smape, wmape, mape, mse, mae, rmsse.

<table>
<thead>
<tr>
<th>metric_name</th>
<th>metric_values</th>
<th>is_training_metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>smape</td>
<td>0.006649</td>
<td>False</td>
</tr>
<tr>
<td>wmape</td>
<td>0.006645</td>
<td>False</td>
</tr>
<tr>
<td>mape</td>
<td>0.006631</td>
<td>False</td>
</tr>
<tr>
<td>mse</td>
<td>0.012889</td>
<td>False</td>
</tr>
<tr>
<td>mae</td>
<td>0.084415</td>
<td>False</td>
</tr>
<tr>
<td>rmsse</td>
<td>0.810353</td>
<td>False</td>
</tr>
</tbody>
</table>

It is possible to filter for only specific metrics of interest, or even implement your own callable and pass into the score() method. For example, see this function that uses last observed value as a predictor and computes the mse. Or naive_error which computes the error as the delta between predicted values and the training period mean.

Note these are not really useful error metrics, just showing some examples of callables you can use ;)

```python
def mse_naive(test_actual):
    actual = test_actual[1:]
    prediction = test_actual[:-1]
    return np.mean(np.square(actual - prediction))

def naive_error(train_actual, test_prediction):
    train_mean = np.mean(train_actual)
    return np.mean(np.abs(test_prediction - train_mean))
```

It doesn’t take additional time to re-fit and predict the model, since the results are stored when fit_predict() is called. Check docstrings for function criteria that is required for it to be supported with this api.

In some cases, users may want to evaluate our metrics on both train and test data. To do this you can call score again with the following indicator

```python
bt.score(include_training_metrics=True)
```

<table>
<thead>
<tr>
<th>metric_name</th>
<th>metric_values</th>
<th>is_training_metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>smape</td>
<td>0.006649</td>
<td>False</td>
</tr>
<tr>
<td>wmape</td>
<td>0.006645</td>
<td>False</td>
</tr>
<tr>
<td>mape</td>
<td>0.006631</td>
<td>False</td>
</tr>
<tr>
<td>mse</td>
<td>0.012889</td>
<td>False</td>
</tr>
</tbody>
</table>

(continues on next page)
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>mae</td>
<td>0.084415</td>
<td>False</td>
</tr>
<tr>
<td>5</td>
<td>rmsse</td>
<td>0.810353</td>
<td>False</td>
</tr>
<tr>
<td>6</td>
<td>smape</td>
<td>0.002738</td>
<td>True</td>
</tr>
<tr>
<td>7</td>
<td>wmape</td>
<td>0.002742</td>
<td>True</td>
</tr>
<tr>
<td>8</td>
<td>mape</td>
<td>0.002738</td>
<td>True</td>
</tr>
<tr>
<td>9</td>
<td>mse</td>
<td>0.003118</td>
<td>True</td>
</tr>
<tr>
<td>10</td>
<td>mae</td>
<td>0.035037</td>
<td>True</td>
</tr>
</tbody>
</table>

### 18.5 Backtest Get Models

In cases where BackTester doesn't cut it or for more custom use-cases, there’s an interface to export the TimeSeriesSplitter and predicted data, as shown earlier. It’s also possible to get each of the fitted models for deeper diving

```python
[26]: fitted_models = bt.get_fitted_models()
```

```python
[27]:
    model_1 = fitted_models[0]
    model_1.get_regression_coefs()
```

<table>
<thead>
<tr>
<th>regressor</th>
<th>regressor_sign</th>
<th>coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>trend.unemploy</td>
<td>Regular</td>
<td>-0.048327</td>
</tr>
<tr>
<td>trend.filling</td>
<td>Regular</td>
<td>-0.120384</td>
</tr>
<tr>
<td>trend.job</td>
<td>Regular</td>
<td>0.027867</td>
</tr>
</tbody>
</table>

BackTester composes a TimeSeriesSplitter within it, but TimeSeriesSplitter can also be created on its own as a standalone object. See section below on TimeSeriesSplitter for more details on how to use the splitter.

All of the additional TimeSeriesSplitter args can also be passed into BackTester on instantiation

```python
[28]: ts_splitter = bt.get_splitter()
    _ = ts_splitter.plot()
```

![Train/Test Split Scheme](image)
18.6 Hyperparameter Tuning

After seeing the results from the backtest, users may wish to fine tune the hyperparameters. Orbit also provide a `grid_search_orbit` utilities for parameter searching. It uses Backtester under the hood so users can compare backtest metrics for different parameters combination.

```
[29]: from orbit.utils.params_tuning import grid_search_orbit

[30]: # defining the search space for level smoothing paramter and seasonality smooth paramter
param_grid = {
    'level_sm_input': [0.3, 0.5, 0.8],
    'seasonality_sm_input': [0.3, 0.5, 0.8],
}

[31]: # configs
min_train_len = 380  # in case of rolling window, this specify the length of window length
forecast_len = 20  # length forecast window
incremental_len = 20  # step length for moving forward
best_params, tuned_df = grid_search_orbit(
    param_grid,
    model=dlt,
    df=data,
    min_train_len=min_train_len,
    incremental_len=incremental_len,
    forecast_len=forecast_len,
    metrics=None,
    criteria="min",
    verbose=False,
)

0% | 0/9 [00:00<?, ?it/s]

[32]: tuned_df.head()  # backtest output for each parameter searched

<table>
<thead>
<tr>
<th>level_sm_input</th>
<th>seasonality_sm_input</th>
<th>metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3</td>
<td>0.3</td>
<td>0.004908</td>
</tr>
<tr>
<td>0.3</td>
<td>0.5</td>
<td>0.004058</td>
</tr>
<tr>
<td>0.3</td>
<td>0.8</td>
<td>0.003608</td>
</tr>
<tr>
<td>0.5</td>
<td>0.3</td>
<td>0.007907</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
<td>0.006306</td>
</tr>
</tbody>
</table>

[33]: best_params  # output best parameters
[33]: [{'level_sm_input': 0.3, 'seasonality_sm_input': 0.8}]
This notebook gives a tutorial on how to use Watanabe-Bayesian information criterion (WBIC) and Bayesian information criterion (BIC) for feature selection (Watanabe[2010], McElreath[2015], and Vehtari[2016]). The WBIC or BIC is an information criterion. Similar to other criteria (AIC, DIC), the WBIC/BIC endeavors to find the most parsimonious model, i.e., the model that balances fit with complexity. In other words a model (or set of features) that optimizes WBIC/BIC should neither over nor under fit the available data.

In this tutorial a data set is simulated using the damped linear trend (DLT) model. This data set is then used to fit DLT models with varying number of features as well as a global local trend model (GLT), and a Error-Trend-Seasonal (ETS) model. The WBIC/BIC criteria is then show to find the true model.

Note that we recommend the use of WBIC for full Bayesian and SVI estimators and BIC for MAP estimator.

19.1 Data Simulation

This block of code creates random data set (365 observations with 10 features) assuming a DLT model. Of the 10 features 5 are effective regressors; i.e., they are used in the true model to create the data.

As an exercise left to the user once you have run the code once try changing the NUM_OF_EFFECTIVE_REGRESSORS (line 2), the SERIES_LEN (line 3), and the SEED (line 4) to see how it effects the results.
```python
[3]:
NUM_OF_REGRESSORS = 10
NUM_OF_EFFECTIVE_REGRESSORS = 4
SERIES_LEN = 365
SEED = 1

# sample some coefficients
COEFS = np.random.default_rng(SEED).uniform(-1, 1, NUM_OF_EFFECTIVE_REGRESSORS)
trend = make_trend(SERIES_LEN, rw_loc=0.01, rw_scale=0.1)
x, regression, coefs = make_regression(series_len=SERIES_LEN, coefs=COEFS)

# combine trend and the regression
y = trend + regression
y = y - y.min()

x_extra = np.random.normal(0, 1, (SERIES_LEN, NUM_OF_REGRESSORS - NUM_OF_EFFECTIVE_REGRESSORS))
x = np.concatenate([x, x_extra], axis=-1)

x_cols = [f"x{x}" for x in range(1, NUM_OF_REGRESSORS + 1)]
response_col = "y"
dt_col = "date"
obs_matrix = np.concatenate([y.reshape(-1, 1), x], axis=1)

# make a data frame for orbit inputs
df = pd.DataFrame(obs_matrix, columns=[response_col] + x_cols)
# make some dummy date stamp
dt = pd.date_range(start='2016-01-04', periods=SERIES_LEN, freq="1W")
df[dt_col] = dt

[4]:
print(df.shape)
print(df.head())
```

```
(365, 12)

y    x1       x2         x3        x4    x5       x6
0  4.426242  0.172792  0.000000   0.165219 -0.000000 -0.039392 -1.212797
1  5.580432  0.452678  0.223187  -0.000000  0.290559  0.082625 -1.670856
2  5.031773  0.182286  0.147066   0.014211  0.273356  0.973938  1.171322
3  3.264027 -0.368227  -0.081455  -0.241060  0.299423 -0.548116  0.568035
4  5.246511  0.019861  -0.146228  -0.390954 -0.128596  0.292448  1.683389

   x7         x8        x9       x10      date
0  -0.939325 -0.917972  0.858560  0.520975  2016-01-10
1  -1.911240 -0.558409 -1.915537  0.282090  2016-01-17
2   0.132576 -0.352861  1.278877  1.074302  2016-01-24
3   0.066535  0.610858  0.670429  0.318656  2016-01-31
4  -0.150782  0.906211  1.241767 -1.708807  2016-02-07
```
19.2 WBIC

In this section, we use DLT model as an example. Different DLT models (the number of features used changes) are fitted and their WBIC values are calculated respectively.

```python
WBIC_ls = []
for k in range(1, NUM_OF_REGRESSORS + 1):
    regressor_col = x_cols[:k]
    dlt_mod = DLT(
        response_col=response_col,
        date_col=dt_col,
        regressor_col=regressor_col,
        seed=2022,
        level_sm_input=0.01,
        slope_sm_input=0.01,
        num_warmup=4000,
        num_sample=4000,
        stan_mcmc_args={
            'show_progress': False,
        },
    )
    WBIC_temp = dlt_mod.fit_wbic(df=df)
    print("WBIC value with {:d} regressors: {:.3f}".format(k, WBIC_temp))
    print('----------------------------------------------')
    WBIC_ls.append(WBIC_temp)
```

WBIC value with 1 regressors: 1202.020
----------------------------------------------

WBIC value with 2 regressors: 1149.747
----------------------------------------------

WBIC value with 3 regressors: 1103.782
----------------------------------------------

WBIC value with 4 regressors: 1054.616
----------------------------------------------
WBIC value with 5 regressors: 1060.230


WBIC value with 6 regressors: 1065.537


WBIC value with 7 regressors: 1072.151


WBIC value with 8 regressors: 1076.994


WBIC value with 9 regressors: 1085.112

WBIC value with 10 regressors: 1091.461

CPU times: user 18.8 s, sys: 873 ms, total: 19.7 s
Wall time: 1min 56s

It is also interesting to see if WBIC can distinguish between model types; not just do feature selection for a given type of model. To that end the next block fits an LGT and ETS model to the data; the WBIC values for both models are then calculated.

Note that WBIC is supported for both the ‘stan-mcmc’ and ‘pyro-svi’ estimators. Currently only the LGT model has both. Thus WBIC is calculated for LGT for both estimators.

[6]: %%time
lgt = LGT(response_col=response_col,
    date_col=dt_col,
    regressor_col=regressor_col,
    seasonality=52,
    estimator='stan-mcmc',
    seed=8888)
WBIC_lgt_mcmc = lgt.fit_wbic(df=df)
print("WBIC value for LGT model (stan MCMC): {:.3f}".format(WBIC_lgt_mcmc))

lgt = LGT(response_col=response_col,
    date_col=dt_col,
    regressor_col=regressor_col,
    seasonality=52,
    estimator='pyro-svi',
    seed=8888)
WBIC_lgt_pyro = lgt.fit_wbic(df=df)
print("WBIC value for LGT model (pyro SVI): {:.3f}".format(WBIC_lgt_pyro))

(continues on next page)
ets = ETS(
    response_col=response_col,
    date_col=dt_col,
    seed=2020,
    # fixing the smoothing parameters to learn regression coefficients more effectively
    level_sm_input=0.01,
)

WBIC_ets = ets.fit_wbic(df=df)
print("WBIC value for ETS model: {:.3f}".format(WBIC_ets))

WBIC_ls.append(WBIC_lgt_mcmc)
WBIC_ls.append(WBIC_lgt_pyro)
WBIC_ls.append(WBIC_ets)


chain 1 | | 00:00 Status
chain 2 | | 00:00 Status
chain 3 | | 00:00 Status
chain 4 | | 00:00 Status

2024-03-13 23:44:02 - orbit - INFO - Using SVI (Pyro) with steps: 301, samples: 100, learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.

WBIC value for LGT model (stan MCMC): 1148.217

2024-03-13 23:44:02 - orbit - INFO - step 0 loss = 320.06, scale = 0.11505
INFO:orbit:step 0 loss = 320.06, scale = 0.11505
2024-03-13 23:44:10 - orbit - INFO - step 100 loss = 116.69, scale = 0.5124
INFO:orbit:step 100 loss = 116.69, scale = 0.5124
2024-03-13 23:44:17 - orbit - INFO - step 200 loss = 116.64, scale = 0.4936
INFO:orbit:step 200 loss = 116.64, scale = 0.4936
2024-03-13 23:44:25 - orbit - INFO - step 300 loss = 117.04, scale = 0.51977
INFO:orbit:step 300 loss = 117.04, scale = 0.51977

WBIC value for LGT model (pyro SVI): 1128.337

chain 1 | | 00:00 Status
chain 2 | | 00:00 Status
chain 3 | | 00:00 Status
chain 4 | | 00:00 Status
WBIC value for ETS model: 1197.855
CPU times: user 22.9 s, sys: 2.81 s, total: 25.7 s
Wall time: 24.9 s

The plot below shows the WBIC vs the number of features / model type (blue line). The true model is indicated by the vertical red line. The horizontal gray line shows the minimum (optimal) value. The minimum is at the true value.

```
[7]: labels = ["DLT_{\text{}}".format(x) for x in range(1, NUM_OF_REGRESSORS + 1)] + ['LGT_MCMC', ...
               'LGT_SVI', 'ETS']

fig, ax = plt.subplots(1, 1, figsize=(12, 6), dpi=80)
markerline, stemlines, baseline = ax.stem(
    np.arange(len(labels)), np.array(WBIC_ls), label='WBIC', markerfmt='D')
baseline.set_color('none')
markerline.set_markersize(12)
x.set_ylim(1020, )
x.set_xticks(np.arange(len(labels)))
x.set_xticklabels(labels)
# because list type is mixed index from 1;
x.axvline(x=NUM_OF_EFFECTIVE_REGRESSORS - 1, color='red', linewidth=3, alpha=0.5,
          linestyle='-', label='truth')
x.set_ylabel("WBIC")
x.set_xlabel("# of Features / Model Type")
x.legend();
```
19.3 BIC

In this section, we use DLT model as an example. Different DLT models (the number of features used changes) are fitted and their BIC values are calculated respectively.

```python
[8]: %%time
BIC_ls = []
for k in range(0, NUM_OF_REGRESSORS):
    regressor_col = x_cols[:k + 1]
    dlt_mod = DLT(
        estimator='stan-map',
        response_col=response_col,
        date_col=dt_col,
        regressor_col=regressor_col,
        seed=2022,
        # fixing the smoothing parameters to learn regression coefficients more effectively
        level_sm_input=0.01,
        slope_sm_input=0.01,
    )
    dlt_mod.fit(df=df)
    BIC_temp = dlt_mod.get_bic()
    print("BIC value with {:d} regressors: {:.3f}".format(k + 1, BIC_temp))
    print('--------------------------------------------------')
    BIC_ls.append(BIC_temp)
```

BIC value with 1 regressors: 1247.444
--------------------------------------------------
BIC value with 2 regressors: 1191.892
--------------------------------------------------
BIC value with 3 regressors: 1139.408
--------------------------------------------------
BIC value with 4 regressors: 1081.432
--------------------------------------------------
BIC value with 5 regressors: 1082.559
--------------------------------------------------
The plot below shows the BIC vs the number of features (blue line). The true model is indicated by the vertical red line. The horizontal gray line shows the minimum (optimal) value. The minimum is at the true value.

```python
labels = ["DLT_\{\}".format(x) for x in range(1, NUM_OF_REGRESSORS + 1)]
fig, ax = plt.subplots(1, 1, figsize=(12, 6), dpi=80)
markerline, stemlines, baseline = ax.stem(
    np.arange(len(labels)), np.array(BIC_ls), label='BIC', markerfmt='D')
baseline.set_color('none')
markerline.set_markersize(12)
ax.set_ylim(1020, )
ax.set_xticks(np.arange(len(labels)))
ax.set_xticklabels(labels)
# because list type is mixed index from 1;
ax.axvline(x=NUM_OF_EFFECTIVE_REGRESSORS - 1, color='red', linewidth=3, alpha=0.5,
    linestyle='-', label='truth')
ax.set_ylabel("BIC")
ax.set_xlabel("# of Features")
ax.legend();
```
19.4 References


3. Vehtari Aki, Gelman Andrew, Gabry Jonah (2016) “Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC”
In this section, we will introduce a rich set of plotting functions in orbit for the EDA (exploratory data analysis) purpose. The plots include:

- Time series heatmap
- Correlation heatmap
- Dual axis time series plot
- Wrap plot

```
[1]: import seaborn as sns
    from matplotlib import pyplot as plt
    import pandas as pd
    import numpy as np

    import orbit
    from orbit.utils.dataset import load_iclaims
    from orbit.eda import eda_plot

[2]: print(orbit.__version__)
1.1.4.4

[3]: df = load_iclaims()
    df['week'] = pd.to_datetime(df['week'])

[4]: df.head()
```

```
   week    claims  trend.unemploy  trend.filling  trend.job   sp500  vix
0 2010-01-03  13.386595   0.219882       -0.318452  0.117500  -0.417633
1 2010-01-10  13.624218   0.219882       -0.194838  0.168794  -0.425480
2 2010-01-17  13.98741    0.236143       -0.292477  0.117500  -0.465229
3 2010-01-24  13.137549   0.203353       -0.194838  0.106918  -0.481751
4 2010-01-31  13.196760   0.134360       -0.242466  0.074483  -0.488929
   vix
0 0.122654
1 0.119445
2 0.532339
3 0.428645
4 0.487404
```
20.1 Time series heat map

[5]: _ = eda_plot.ts_heatmap(df = df, date_col = 'week', seasonal_interval=52, value_col= 'claims')

[6]: _ = eda_plot.ts_heatmap(df = df, date_col = 'week', seasonal_interval=52, value_col= 'claims', normalization=True)
20.2 Correlation heatmap

```python
[7]: var_list = ['trend.unemploy', 'trend.filling', 'trend.job', 'sp500', 'vix']
    _ = eda_plot.correlation_heatmap(df, var_list = var_list,
                                    fig_width=10, fig_height=6)
```

20.3 Dual axis time series plot

```python
[8]: _ = eda_plot.dual_axis_ts_plot(df=df, var1='trend.unemploy', var2='claims', date_col='week')
```

20.4 Wrap plots for quick glance of data patterns

```python
[9]: var_list=['week', 'trend.unemploy', 'trend.filling', 'trend.job', 'sp500', 'vix']
    df[var_list].melt(id_vars = ['week'])

[9]:

<table>
<thead>
<tr>
<th>week</th>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2010-01-03</td>
<td>trend.unemploy</td>
<td>0.219882</td>
</tr>
<tr>
<td>1 2010-01-10</td>
<td>trend.unemploy</td>
<td>0.219882</td>
</tr>
<tr>
<td>2 2010-01-17</td>
<td>trend.unemploy</td>
<td>0.236143</td>
</tr>
<tr>
<td>3 2010-01-24</td>
<td>trend.unemploy</td>
<td>0.203353</td>
</tr>
<tr>
<td>4 2010-01-31</td>
<td>trend.unemploy</td>
<td>0.134360</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2210 2018-05-27</td>
<td>vix</td>
<td>-0.175192</td>
</tr>
<tr>
<td>2211 2018-06-03</td>
<td>vix</td>
<td>-0.275119</td>
</tr>
<tr>
<td>2212 2018-06-10</td>
<td>vix</td>
<td>-0.291676</td>
</tr>
<tr>
<td>2213 2018-06-17</td>
<td>vix</td>
<td>-0.152422</td>
</tr>
<tr>
<td>2214 2018-06-24</td>
<td>vix</td>
<td>0.003284</td>
</tr>
</tbody>
</table>

[2215 rows x 3 columns]

```python
[10]: _ = eda_plot.wrap_plot_ts(df, 'week', var_list)
```
CHAPTER TWENTYONE

SIMULATION DATA

Orbit provide the functions to generate the simulation data including:

1. Generate the data with time-series trend:
   • random walk
   • arima

2. Generate the data with seasonality
   • discrete
   • fourier series

3. Generate regression data

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

import orbit
from orbit.utils.simulation import make_trend, make_seasonality, make_regression
from orbit.utils.plot import get_orbit_style
plt.style.use(get_orbit_style())
from orbit.constants.palette import OrbitPalette
%

[2]: print(orbit.__version__)
1.1.4.4
```

**21.1 Trend**

**21.1.1 Random Walk**

```
[3]: rw = make_trend(200, rw_loc=0.02, rw_scale=0.1, seed=2020)
    _ = plt.plot(rw, color = OrbitPalette.BLUE.value)
```
21.1.2 ARMA


[4]: arma_trend = make_trend(200, method='arma', arma=[.8, -.1], seed=2020)
_ = plt.plot(arma_trend, color = OrbitPalette.BLUE.value)
21.2 Seasonality

21.2.1 Discrete

generating a weekly seasonality(=7) where seasonality within a day is constant(duration=24) on an hourly time-series

[5]:

```python
ds = make_seasonality(500, seasonality=7, duration=24, method='discrete', seed=2020)
_ = plt.plot(ds, color = OrbitPalette.BLUE.value)
```
21.2.2 Fourier

generating a sine-cosine wave seasonality for an annual seasonality (365) using Fourier series

```python
[6]: fs = make_seasonality(365 * 3, seasonality=365, method='fourier', order=5, seed=2020) _ = plt.plot(fs, color = OrbitPalette.BLUE.value)
```
21.3 Regression

generating multiplicative time-series with trend, seasonality and regression components

```python
coefs = [0.1, -0.33, 0.8]
x, y, coefs = make_regression(200, coefs, scale=2.0, seed=2020)
_ = plt.plot(y, color = OrbitPalette.BLUE.value)
```
For next step, need to install scikit-learn

[11]: # !python -m pip install scikit-learn

[12]: from sklearn.linear_model import LinearRegression

# check if get the coefficients as set up
reg = LinearRegression().fit(x, y)
print(reg.coef_)

[ 0.1586677 -0.33126796 0.7974205 ]
CHAPTER TWENTYTWO

OTHER UTILITIES

22.1 Generating Full Span of multiple time-series

```python
import pandas as pd
import numpy as np
from orbit.utils.general import expand_grid, regenerate_base_df
import warnings
warnings.filterwarnings('ignore')

Define the series keys and datetime array.
```
[2]: dt = pd.date_range('2020-01-31', '2022-12-31', freq='M')
keys = ['x' + str(x) for x in range(10)]
print(keys)
print(dt)
['x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9']
DatetimeIndex(['2020-01-31', '2020-02-29', '2020-03-31', '2020-04-30',
               '2020-09-30', '2020-10-31', '2020-11-30', '2020-12-31',
               '2021-01-31', '2021-02-28', '2021-03-31', '2021-04-30',
               '2021-05-31', '2021-06-30', '2021-07-31', '2021-08-31',
               '2021-09-30', '2021-10-31', '2021-11-30', '2021-12-31',
               '2022-09-30', '2022-10-31', '2022-11-30', '2022-12-31'],
dtype='datetime64[ns]', freq='M')
```

Users can use `expand_grid` to generate dataframe with observations in `key` and `dt` levels.
```
[3]:
   df_base = expand_grid({
     'key': keys,
     'dt': dt,
   })
x = np.random.normal(0, 1, 10 * 36)
df_base['x'] = x
print(df_base.shape)
print(df_base.head(5))
(360, 3)
```
22.2 Regenerate Multiple Timeseries with Missing rows

Create missing rows.

```python
np.random.seed(2022)
drop_idx = np.random.choice(df_base.index, 5, replace=False)
df_missing = df_base.drop(drop_idx).reset_index(drop=True)
print(df_missing.shape)
df_missing.head(5)
```

(355, 3)

Use `regenerate_base_df` to regenerate the base dataframe.

```python
time_col = "dt"
key_col = "key"
new_df_base = regenerate_base_df(df_missing, time_col, key_col, val_cols=['x'])
```

By default, the missing entries regenerated come with a null value.

```python
new_df_base.iloc[drop_idx]
```

Users can also use `fill_na` option to fill the missing values.

```python
new_df_base = regenerate_base_df(df_missing, time_col, key_col, val_cols=['x'], fill_na=0)
```

```python
new_df_base.iloc[drop_idx]
```
<table>
<thead>
<tr>
<th></th>
<th>Date</th>
<th>Value</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>75</td>
<td>2020-04-30</td>
<td>x2</td>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td>135</td>
<td>2022-04-30</td>
<td>x3</td>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td>43</td>
<td>2020-08-31</td>
<td>x1</td>
<td></td>
<td>0.0</td>
</tr>
</tbody>
</table>

22.2. Regenerate Multiple Timeseries with Missing rows
One important feature of `orbit` is to allow developers to build their own models in a relatively flexible manner to serve their own purpose. This tutorial will go over a demo on how to build up a simple Bayesian linear regression model using Pyro API in the backend with orbit interface.

### 23.1 Orbit Class Design

In version 1.1.0, the classes within Orbit are re-designed as such:

1. Forecaster
2. Model
3. Estimator

#### 23.1.1 Forecaster

`Forecaster` provides general interface for users to perform `fit` and `predict` task. It is further inherited to provide different types of forecasting methodology:

1. Maximum a posterior (MAP)
2. [Stochastic Variational Inference (SVI)]
3. Full Bayesian

The discrepancy on these three methods mainly lie on the posteriors estimation where MAP will yield point posterior estimate and can be extracted through the method `get_point_posterior()`. Meanwhile, SVI and Full Bayesian allow posterior sample extraction through the method `get_posteriors()`. Alternatively, you can also approximate point estimate by passing through additional arg such as `point_method='median'` in the `.fit()` process.

To make use of a `Forecaster`, one must provide these two objects:

1. Model
2. Estimator

Theses two objects are prototyped as abstract and next subsections will cover how they work.
23.1.2 Model

Model is an object defined by a class inherited from BaseTemplate a.k.a Model Template in the diagram below. It mainly turns the logic of \texttt{fit()} and \texttt{predict()} concrete by supplying the fitter as a file (\texttt{CmdStanPy}) or a callable class (\texttt{Pyro}) and the internal \texttt{predict()} method. This object defines the overall inputs, model structure, parameters and likelihoods.

23.1.3 Estimator

Meanwhile, there are different APIs implement slightly different ways of sampling and optimization (for MAP). \texttt{orbit} is designed to support various APIs such as \texttt{CmdStanPy} and \texttt{Pyro} (hopefully PyMC3, Numpyro in the future!). The logic separating the call of different APIs with different interface is done by the \texttt{Estimator} class which is further inherited in \texttt{PyroEstimator} and \texttt{StanEstimator}.

Diagram above shows the interaction across classes under the Orbit package design.
23.2 Creating a Bayesian Linear Regression Model

The plan here is to build a classical regression model with the formula below:

\[ y = \alpha + X\beta + \epsilon \]

where \( \alpha \) is the intercept, \( \beta \) is the coefficients matrix and \( \epsilon \) is the random noise.

To start with let’s load the libraries.

```
[1]: import pandas as pd
    import numpy as np
    import torch
    import pyro
    import pyro.distributions as dist
    from copy import deepcopy
    import matplotlib.pyplot as plt
    import orbit
    from orbit.template.model_template import ModelTemplate
    from orbit.forecaster import SVIForecaster
    from orbit.estimators.pyro_estimator import PyroEstimatorSVI
    from orbit.utils.simulation import make_regression
    from orbit.diagnostics.plot import plot_predicted_data
    from orbit.utils.plot import get_orbit_style

    plt.style.use(get_orbit_style())
    %matplotlib inline

[2]: print(orbit.__version__)
    1.1.4.4
```

Since the Forecaster and Estimator are already built inside orbit, the rest of the ingredients to construct a new model will be a Model object that contains the follow:

- a callable class as a fitter
- a predict method

### 23.2.1 Define a Fitter

For Pyro users, you should find the code below familiar. All it does is to put a Bayesian linear regression (BLR) model code in a callable class. Details of BLR will not be covered here. Note that the parameters here need to be consistent.

```
[3]: class MyFitter:
    max_plate_nesting = 1  # max number of plates nested in model

    def __init__(self, data):
        for key, value in data.items():
            key = key.lower()
            if isinstance(value, (list, np.ndarray)):
                value = torch.tensor(value, dtype=torch.float)
```

(continues on next page)
```python
self.__dict__[key] = value

def __call__(self):
    extra_out = {}

    p = self.regressor.shape[1]
    bias = pyro.sample("bias", dist.Normal(0, 1))
    weight = pyro.sample("weight", dist.Normal(0, 1).expand([p]).to_event(1))
    yhat = bias + weight @ self.regressor.transpose(-1, -2)
    obs_sigma = pyro.sample("obs_sigma", dist.HalfCauchy(self.response_sd))

    with pyro.plate("response_plate", self.num_of_obs):
        pyro.sample("response", dist.Normal(yhat, obs_sigma), obs=self.response)

    log_prob = dist.Normal(yhat[..., 1:], obs_sigma).log_prob(self.response[1:]).
    extra_out.update({"log_prob": log_prob})

    return extra_out
```

### 23.2.2 Define the Model Class

This is the part requires the knowledge of orbit most. First we construct a class by plugging in the fitter callable. Users need to let the orbit estimators know the required input in addition to the defaults (e.g. `response`, `response_sd` etc.). In this case, it takes `regressor` as the matrix input from the data frame. That is why there are lines of code to provide this information in

1. `_data_input_mapper` - a list or Enum to let estimator keep tracking required data input
2. `set_dynamic_attributes` - the logic define the actual inputs i.e. `regressor` from the data frame. This is a reserved function being called inside `Forecaster`.

Finally, we code the logic in `predict()` to define how we utilize posteriors to perform in-sample / out-of-sample prediction. Note that the output needs to be a dictionary where it supports components decomposition.

```python
[4]: class BayesLinearRegression(ModelTemplate):
    _fitter = MyFitter
    _data_input_mapper = ['regressor']
    _supported_estimator_types = [PyroEstimatorSVI]

    def __init__(self, regressor_col, **kwargs):
        super().__init__(**kwargs)
        self.regressor_col = regressor_col
        self.regressor = None
        self._model_param_names = ['bias', 'weight', 'obs_sigma']

    def set_dynamic_attributes(self, df, training_meta):
        self.regressor = df[self.regressor_col].values

    def predict(self, posterior_estimates, df, training_meta, prediction_meta, include_
_ERROR=False, **kwargs):
```

(continues on next page)
model = deepcopy(posterior_estimates)
new_regressor = df[self.regressor_col].values.T
bias = np.expand_dims(model.get('bias'), -1)
obs_sigma = np.expand_dims(model.get('obs_sigma'), -1)
weight = model.get('weight')
pred_len = df.shape[0]
batch_size = weight.shape[0]
prediction = bias + np.matmul(weight, new_regressor) +
np.random.normal(0, obs_sigma, size=(batch_size, pred_len))
return {'prediction': prediction}

23.3 Test the New Model with Forecaster

Once the model class is defined. User can initialize an object and build a forecaster for fit and predict purpose. Before doing that, the demo provides a simulated dataset here.

23.3.1 Data Simulation

[5]: x, y, coefs = make_regression(120, [3.0, -1.0], bias=1.0, scale=1.0)

[6]: df = pd.DataFrame(
    np.concatenate([y.reshape(-1, 1), x], axis=1), columns=['y', 'x1', 'x2'])
df['week'] = pd.date_range(start='2016-01-04', periods=len(y), freq='7D')

[7]: df.head(5)

<table>
<thead>
<tr>
<th></th>
<th>y</th>
<th>x1</th>
<th>x2</th>
<th>week</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.382337</td>
<td>0.345584</td>
<td>0.000000</td>
<td>2016-01-04</td>
</tr>
<tr>
<td>1</td>
<td>2.812929</td>
<td>0.330437</td>
<td>-0.000000</td>
<td>2016-01-11</td>
</tr>
<tr>
<td>2</td>
<td>3.600130</td>
<td>0.905356</td>
<td>0.446375</td>
<td>2016-01-18</td>
</tr>
<tr>
<td>3</td>
<td>-0.884275</td>
<td>-0.000000</td>
<td>0.581118</td>
<td>2016-01-25</td>
</tr>
<tr>
<td>4</td>
<td>2.704941</td>
<td>0.364572</td>
<td>0.294132</td>
<td>2016-02-01</td>
</tr>
</tbody>
</table>

[8]: test_size = 20
test_df = df[-test_size:]
23.3.2 Create the Forecaster

As mentioned previously, model is the inner object to control the math. To use it for fit and predict purpose, we need a Forecaster. Since the model is written in Pyro, the pick here should be SVIForecaster.

```python
[9]: model = BayesLinearRegression(
    regressor_col=['x1', 'x2'],
)

[10]: blr = SVIForecaster(
    model=model,
    response_col='y',
    date_col='week',
    estimator_type=PyroEstimatorSVI,
    verbose=True,
    num_steps=501,
    seed=2021,
)

[11]: blr
```

Now, an object `blr` is instantiated as a SVIForecaster object and is ready for fit and predict.

```python
[12]: blr.fit(train_df)
```

```
2024-03-13 21:44:00 - orbit - INFO - Using SVI (Pyro) with steps: 501, samples: 100,␣
˓→learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
2024-03-13 21:44:00 - orbit - INFO - step 0 loss = 27333, scale = 0.077497
INFO:orbit:step 0 loss = 27333, scale = 0.077497
2024-03-13 21:44:04 - orbit - INFO - step 100 loss = 12594, scale = 0.0092399
INFO:orbit:step 100 loss = 12594, scale = 0.0092399
2024-03-13 21:44:06 - orbit - INFO - step 200 loss = 12591, scale = 0.0095592
INFO:orbit:step 200 loss = 12591, scale = 0.0095592
2024-03-13 21:44:08 - orbit - INFO - step 300 loss = 12593, scale = 0.0094199
INFO:orbit:step 300 loss = 12593, scale = 0.0094199
2024-03-13 21:44:10 - orbit - INFO - step 400 loss = 12591, scale = 0.0092691
INFO:orbit:step 400 loss = 12591, scale = 0.0092691
2024-03-13 21:44:12 - orbit - INFO - step 500 loss = 12591, scale = 0.0095463
INFO:orbit:step 500 loss = 12591, scale = 0.0095463
```

```
[12]: <orbit.forecaster.svi.SVIForecaster at 0x29d220fd0>
```

23.3.3 Compare Coefficients with Truth

```python
[13]: estimated_weights = blr.get_posterior_samples()['weight']
```

The code below is to compare the median of coefficients posteriors which is labeled as `weight` with the truth.

```python
[14]: print("True Coef: {:.3f}, {:.3f}".format(coefs[0], coefs[1]))
estimated_coef = np.median(estimated_weights, axis=0)
print("Estimated Coef: {:.3f}, {:.3f}".format(estimated_coef[0], estimated_coef[1]))
```
23.3.4 Examine Forecast Accuracy

[15]: `predicted_df = blr.predict(df)`

[16]: `_ = plot_predicted_data(train_df, predicted_df, 'week', 'y', test_actual_df=test_df, prediction_percentiles=[5, 95])`

23.3.5 Additional Notes

In general, most of the diagnostic tools in orbit such as posteriors checking and plotting is applicable in the model created in this style. Also, users can provide `point_method='median'` in the `fit()` under the `SVIForecaster` to extract median of posteriors directly.
24.1 orbit package

24.1.1 Subpackages

orbit.constants package

Submodules

orbit.constants.constants module

class orbit.constants.constants.\texttt{BacktestFitKeys}(value)
    
    Bases: Enum

    column names of the dataframe used in the output from the backtest.BackTester.fit_predict() or any labels of the intermediate variables to generate such outcome dataframe

    \begin{itemize}
    \item \texttt{ACTUAL} = 'actual'
    \item \texttt{DATE} = 'date'
    \item \texttt{METRIC\_NAME} = 'metric\_name'
    \item \texttt{METRIC\_VALUES} = 'metric\_values'
    \item \texttt{PREDICTED} = 'prediction'
    \item \texttt{SPLIT\_KEY} = 'split\_key'
    \item \texttt{TEST\_ACTUAL} = 'test\_actual'
    \item \texttt{TEST\_PREDICTED} = 'test\_prediction'
    \item \texttt{TRAIN\_ACTUAL} = 'train\_actual'
    \item \texttt{TRAIN\_FLAG} = 'training\_data'
    \item \texttt{TRAIN\_METRIC\_FLAG} = 'is\_training\_metric'
    \item \texttt{TRAIN\_PREDICTED} = 'train\_prediction'
    \end{itemize}
class orbit.constants.constants.CompiledStanModelPath
    Bases: object
    the directory path for compiled stan models
    CHILD = 'stan'
    PARENT = 'orbit'

class orbit.constants.constants.EstimatorsKeys(value)
    Bases: Enum
    alias for all available estimator types when they are called under model wrapper functions
    CmdStanMAP = 'cmdstan-map'
    CmdStanMCMC = 'cmdstan-mcmc'
    PyroSVI = 'pyro-svi'
    StanMAP = 'stan-map'
    StanMCMC = 'stan-mcmc'

class orbit.constants.constants.KTRTimePointPriorKeys(value)
    Bases: Enum
    hash table keys for the dictionary of back-test aggregation analysis result
    NAME = 'name'
    PRIOR_END_TP_IDX = 'prior_end_tp_idx'
    PRIOR_MEAN = 'prior_mean'
    PRIOR_REGRESSOR_COL = 'prior_regressor_col'
    PRIOR_SD = 'prior_sd'
    PRIOR_START_TP_IDX = 'prior_start_tp_idx'

class orbit.constants.constants.PlotLabels(value)
    Bases: Enum
    used in multiple prediction plots
    ACTUAL_RESPONSE = 'actual_response'
    PREDICTED_RESPONSE = 'predicted_response'
    TRAINING_ACTUAL_RESPONSE = 'training_actual_response'

class orbit.constants.constants.PredictMethod(value)
    Bases: Enum
    The predict method for all of the stan template. Often used are mean and median.
    FULL_SAMPLING = 'full'
    MAP = 'map'
    MEAN = 'mean'
class orbit.constants.constants.PredictionKeys(value)
  Bases: Enum
  column names for the data frame of predicted result with decomposed components
  PREDICTION = 'prediction'
  REGRESSION = 'regression'
  REGRESSOR = 'regressor'
  SEASONALITY = 'seasonality'
  TREND = 'trend'

class orbit.constants.constants.PredictionMetaKeys(value)
  Bases: Enum
  prediction input meta data dictionary processed under Forecaster.predict()
  DATE_ARRAY = 'date_array'
  END = 'prediction_end'
  END_INDEX = 'end'
  FUTURE_STEPS = 'n_forecast_steps'
  PREDICTION_DF_LEN = 'df_length'
  START = 'prediction_start'
  START_INDEX = 'start'

class orbit.constants.constants.TimeSeriesSplitSchemeKeys(value)
  Bases: Enum
  hash table keys for the dictionary of back-test meta data
  MODEL = 'model'
  SPLIT_TYPE_EXPANDING = 'expanding'
  SPLIT_TYPE_ROLLING = 'rolling'
  TEST_IDX = 'test_idx'
  TRAIN_END_DATE = 'train_end_date'
  TRAIN_IDX = 'train_idx'
  TRAIN_START_DATE = 'train_start_date'

class orbit.constants.constants.TrainingMetaKeys(value)
  Bases: Enum
  training meta data dictionary processed under Forecaster.fit()
  DATE_ARRAY = 'date_array'
DATE_COL = 'date_col'
END = 'training_end'
NUM_OF_OBS = 'num_of_obs'
RESPONSE = 'response'
RESPONSE_COL = 'response_col'
RESPONSE_MEAN = 'response_mean'
RESPONSE_SD = 'response_sd'
START = 'training_start'

**orbit/constants/dlt module**

**orbit/constants/lgt module**

**orbit/constants/palette module**

class orbit.constants.palette.KTRPalette(value)
    Bases: Enum
    str
    KNOTS_REGION = '#05A357'
    KNOTS_SEGMENT = '#276ef1'

class orbit.constants.palette.OrbitColorMap(value)
    Bases: Enum
    matplotlib ColorMap
    BLACK_GRADIENT = <matplotlib.colors.LinearSegmentedColormap object>
    BLUE_GRADIENT = <matplotlib.colors.LinearSegmentedColormap object>
    GREEN_GRADIENT = <matplotlib.colors.LinearSegmentedColormap object>
    PURPLE_GRADIENT = <matplotlib.colors.LinearSegmentedColormap object>
    RAINBOW = <matplotlib.colors.LinearSegmentedColormap object>
    RED_GRADIENT = <matplotlib.colors.LinearSegmentedColormap object>
    YELLOW_GRADIENT = <matplotlib.colors.LinearSegmentedColormap object>

class orbit.constants.palette.OrbitPalette(value)
    Bases: Enum
    str
    BLACK = '#000000'
    BLUE = '#276EF1'
BLUE600 = '#174291'
BROWN = '#99644C'
GREEN = '#05A357'
GREEN600 = '#03582F'
ORANGE = '#ED6E33'
PURPLE = '#7356BF'
RED = '#E11900'
WHITE = '#FFFFFF'
YELLOW = '#FFC043'
YELLOW400 = '#FFC043'

class orbit.constants.palette.PredictionPaletteClassic(value)
    Bases: Enum
    str
    ACTUAL_OBS = '#000000'
    HOLDOUT_VERTICAL_LINE = '#000000'
    PREDICTION_INTERVAL = '#276EF1'
    PREDICTION_LINE = '#276EF1'
    TEST_OBS = '#FFC043'

Module contents

orbit.diagnostics package

Submodules

orbit.diagnostics.plot module

orbit.diagnostics.plot.metric_horizon_barplot(df, model_col='model',
    pred_horizon_col='pred_horizon', metric_col='smape',
    bar_width=0.1, path=None, figsize=None,
    fontsize=None, is_visible=False)
compare the distribution of parameters from different models using a boxplot. 

:func:`params_comparison_boxplot`

.. code-block:: python

    def params_comparison_boxplot(data, var_names, model_names, color_list, title, fig_size, box_width, box_distance, showfliers):
        ... parameters

- **fig_size**: tuple figure size
- **box_width**: float width of the boxes in the boxplot
- **box_distance**: float the distance between each boxes in the boxplot
- **showfliers**: boolean show outliers in the chart if set as True

:returns: a boxplot comparing parameter distributions from different models side by side

function to plot and visualize the prediction results from back testing.

:func:`plot_bt_predictions`

.. code-block:: python

    def plot_bt_predictions(bt_pred_df, metrics=<function smape>, split_key_list=None, ncol=2, figsize=None, include_vline=True, title='', fontsize=20, path=None, is_visible=True):
        ... function

bt_pred_df

[data frame] the output of :func:`orbit.diagnostics.backtest.BackTester.fit_predict()`, which includes the actuals/predictions for all the splits

metrics

[callable] the metric function

split_key_list: list; default None

with given model, which split keys to plot. If None, all the splits will be plotted

ncol

[int] number of columns of the panel; number of rows will be decided accordingly
figsize
[tuple] figure size

include_vline
[bool] if plotting the vertical line to cut the in-sample and out-of-sample predictions for each split

title
[std] title of the plot

fontsize: int; optional
fontsize of the title

path
[std] path to save the figure

is_visible
[bool] if displaying the figure

orbit.diagnostics.plot.plot_bt_predictions2(bt_pred_df, metrics=<function smape>,
split_key_list=None, figsize=None, include_vline=True,
title='', fontsize=20, markersize=50, lw=2, fig_dir=None,
is_visible=True, fix_xlim=True, export_gif=False)

a different style backtest plot compare to plot bt_prediction where it writes separate plot for each split; this is
also used to produce an animation to summarize every split

orbit.diagnostics.plot.plot_predicted_components(predicted_df, date_col,
prediction_percentiles=None, plot_components=None,
title='', figsize=None, path=None, fontsize=None, is_visible=True)

Plot predicted components with the data frame of decomposed prediction where components
has been pre-defined as trend, seasonality and regression.

predicted_df
If user provide pred_percentiles_col, it needs to include them as well.

date_col
[std] the date column name

prediction_percentiles
[list] a list should consist exact two elements which will be used to plot as lower and upper bound of
confidence interval

plot_components
[list] a list of strings to show the label of components to be plotted; by default, it uses values in orbit.constants.constants.PredictedComponents.

title
[std; optional] title of the plot

figsize
[tuple; optional] figsize pass through to matplotlib.pyplot.figure()

path
[std; optional] path to save the figure

fontsize
[int; optional] fontsize of the title

is_visible
[boolean] whether we want to show the plot. If called from unittest, is_visible might = False.
orbit.diagnostics.plot.plot_predicted_data(training_actual_df, predicted_df, date_col, actual_col, pred_col='prediction', prediction_percentiles=None, title='', test_actual_df=None, is_visible=True, figsize=None, path=None, fontsize=None, line_plot=False, markersize=50, lw=2, linestyle='-',)

plot training actual response together with predicted data; if actual response of predicted data is there, plot it too.

Parameters

- **training_actual_df** (*pd.DataFrame*) – training actual response data frame. two columns required: actual_col and date_col
- **predicted_df** (*pd.DataFrame*) – predicted data response data frame. two columns required: actual_col and pred_col. If user provide prediction_percentiles, it needs to include them as well in such prediction_{x} where x is the correspondent percentiles
- **prediction_percentiles** (*list*) – list of two elements indicates the lower and upper percentiles
- **date_col** (*str*) – the date column name
- **actual_col** (*str*) –
- **pred_col** (*str*) –
- **title** (*str*) – title of the plot
- **test_actual_df** (*pd.DataFrame*) – test actual response dataframe. two columns required: actual_col and date_col
- **is_visible** (*boolean*) – whether we want to show the plot. If called from unittest, is_visible might = False.
- **figsize** (*tuple*) – figsize pass through to matplotlib.pyplot.figure()
- **path** (*str*) – path to save the figure
- **fontsize** (*int; optional*) – fontsize of the title
- **line_plot** (*bool; default False*) – if True, make line plot for observations; otherwise, make scatter plot for observations
- **markersize** (*int; optional*) – point marker size
- **lw** (*int; optional*) – out-of-sample prediction line width
- **linestyle** (*str*) – linestyle of prediction plot

Return type
matplotlib axes object

orbit.diagnostics.plot.residual_diagnostic_plot(df, dist='norm', date_col='week', residual_col='residual', fitted_col='prediction', sparams=None)

Parameters

- **df** (*pd.DataFrame*) –
- **dist** (*str*) –
- **date_col** (*str*) – column name of date
• **residual_col** *(str)* – column name of residual
• **fitted_col** *(str)* – column name of fitted value from model
• **sparams** *(float or list)* – extra parameters used in distribution such as t-dist

**Notes**

1. residual by time
2. residual vs fitted
3. residual histogram with vertical line as mean
4. residuals qq plot
5. residual ACF
6. residual PACF

**Module contents**

**orbit.estimators package**

**Submodules**

**orbit.estimators.base_estimator module**

class **orbit.estimators.base_estimator.BaseEstimator** *(seed=8888, verbose=True)*

Bases: object

Base Estimator class for both Stan and Pyro Estimator

Parameters

• **seed** *(int)* – seed number for initial random values
• **verbose** *(bool)* – If True (default), output all diagnostics messages from estimators

abstract **fit** *(model_name, model_param_names, data_input, fitter=None, init_values=None)*

Parameters

• **model_name** *(str)* – name of model - used in mapping the right sampling file
  (stan/pyro/…)
• **model_param_names** *(list)* – list of strings of model parameters names to extract
• **data_input** *(dict)* – key-value pairs of data input as required by definition in samplers
  (stan/pyro/…)
• **fitter** – model object used for fitting; this will be used instead of model_name if supplied
to search for model object
• **init_values** *(float or np.array)* – initial sampler value. If None, ‘random’ is used

Returns

• **posteriors** *(dict)* – key value pairs where key is the model parameter name and value is
  num_sample x posterior values
• **training_metrics** *(dict)* – metrics and meta data related to the training process
orbit.estimators.pyro_estimator module

class orbit.estimators.pyro_estimator.PyroEstimator(num_steps=301, learning_rate=0.1, learning_rate_total_decay=1.0, message=100, **kwargs)

    Bases: BaseEstimator

Abstract PyroEstimator with shared args for all PyroEstimator child classes

    Parameters

    - num_steps (int) – Number of estimator steps in optimization
    - learning_rate (float) – Estimator learning rate
    - learning_rate_total_decay (float) – A config re-parameterized from lrd in ClippedAdam. For example, 0.1 means a 90% reduction of the final step as of original learning rate where linear decay is implied along the steps. In the case of 1.0, no decay is applied. All steps will have the constant learning rate specified by learning_rate.
    - seed (int) – Seed int
    - message (int) – Print to console every message number of steps
    - kwargs – Additional BaseEstimator args

    Notes


    abstract fit(model_name, model_param_names, data_input, fitter=None, init_values=None)

    Parameters

    - model_name (str) – name of model - used in mapping the right sampling file (stan/pyro/...)
    - model_param_names (list) – list of strings of model parameters names to extract
    - data_input (dict) – key-value pairs of data input as required by definition in samplers (stan/pyro/...)
    - fitter – model object used for fitting; this will be used instead of model_name if supplied to search for model object
    - init_values (float or np.array) – initial sampler value. If None, ‘random’ is used

    Returns

    - posteriors (dict) – key value pairs where key is the model parameter name and value is num_sample x posterior values
    - training_metrics (dict) – metrics and meta data related to the training process

class orbit.estimators.pyro_estimator.PyroEstimatorSVI(num_sample=100, num_particles=100, init_scale=0.1, **kwargs)

    Bases: PyroEstimator

Pyro Estimator for VI Sampling

    Parameters

    - num_sample (int) – Number of samples ot draw for inference, default 100
• **num_particles** (*int*) – Number of particles used in :class:`~pyro.infer.Trace_ELBO` for SVI optimization

• **init_scale** (*float*) – Parameter used in :class:`pyro.infer.autoguide`; recommend a larger number of small dataset

• **kwargs** – Additional :class:`PyroEstimator` class args

.. _fit:

```
fit(model_name, model_param_names, data_input, sampling_temperature, fitter=None, init_values=None)
```

Parameters

• **model_name** (*str*) – name of model - used in mapping the right sampling file (stan/pyro/...)

• **model_param_names** (*list*) – list of strings of model parameters names to extract

• **data_input** (*dict*) – key-value pairs of data input as required by definition in samplers (stan/pyro/...)

• **fitter** – model object used for fitting; this will be used instead of model_name if supplied to search for model object

• **init_values** (*float or np.array*) – initial sampler value. If None, ‘random’ is used

Returns

• **posteriors** (*dict*) – key value pairs where key is the model parameter name and value is :math:`num_sample \times` posterior values

• **training_metrics** (*dict*) – metrics and meta data related to the training process

```
orbit.estimators.stan_estimator module
```

class **orbit.estimators.stan_estimator.StanEstimator**

```
orbit.estimators.stan_estimator.StanEstimator(num_warmup=900, num_sample=100, chains=4, cores=8, algorithm=None, suppress_stan_log=True, **kwargs)
```

Bases: :class:`BaseEstimator`

Abstract StanEstimator with shared args for all StanEstimator child classes

Parameters

• **num_warmup** (*int*) – Number of samples to warm up and to be discarded, default 900

• **num_sample** (*int*) – Number of samples to return, default 100

• **chains** (*int*) – Number of chains in stan sampler, default 4

• **cores** (*int*) – Number of cores for parallel processing, default max(cores, multiprocessing.cpu_count())

• **algorithm** (*str*) – If None, default to Stan defaults

• **suppress_stan_log** (*bool*) – If False, turn off cmdstanpy logger. Default as False.

• **kwargs** – Additional :class:`BaseEstimator` class args

.. _fit:

```
abstract fit(model_name, model_param_names, data_input, fitter=None, init_values=None)
```

Parameters

• **model_name** (*str*) – name of model - used in mapping the right sampling file (stan/pyro/...
• **model_param_names** *(list)* – list of strings of model parameters names to extract

• **data_input** *(dict)* – key-value pairs of data input as required by definition in samplers (stan/pyro/...)

• **fitter** – model object used for fitting; this will be used instead of model_name if supplied to search for model object

• **init_values** *(float or np.array)* – initial sampler value. If None, ‘random’ is used

Returns

• **posteriors** *(dict)* – key value pairs where key is the model parameter name and value is num_sample x posterior values

• **training_metrics** *(dict)* – metrics and meta data related to the training process

**class** orbit.estimators.stan_estimator.StanEstimatorMAP(stan_map_args=None, **kwargs)**

Bases: StanEstimator

Stan Estimator for MAP Posteriors

**fit**(model_name, model_param_names, data_input, fitter=None, init_values=None)

Parameters

• **model_name** *(str)* – name of model - used in mapping the right sampling file (stan/pyro/...)

• **model_param_names** *(list)* – list of strings of model parameters names to extract

• **data_input** *(dict)* – key-value pairs of data input as required by definition in samplers (stan/pyro/...)

• **fitter** – model object used for fitting; this will be used instead of model_name if supplied to search for model object

• **init_values** *(float or np.array)* – initial sampler value. If None, ‘random’ is used

Returns

• **posteriors** *(dict)* – key value pairs where key is the model parameter name and value is num_sample x posterior values

• **training_metrics** *(dict)* – metrics and meta data related to the training process

**class** orbit.estimators.stan_estimator.StanEstimatorMCMC(stan_mcmc_args=None, **kwargs)**

Bases: StanEstimator

Stan Estimator for MCMC Sampling

**fit**(model_name, model_param_names, sampling_temperature, data_input, fitter=None, init_values=None)

Parameters

• **model_name** *(str)* – name of model - used in mapping the right sampling file (stan/pyro/...)

• **model_param_names** *(list)* – list of strings of model parameters names to extract

• **data_input** *(dict)* – key-value pairs of data input as required by definition in samplers (stan/pyro/...)

• **fitter** — model object used for fitting; this will be used instead of model_name if supplied to search for model object

• **init_values** (*float or np.array*) — initial sampler value. If None, ‘random’ is used

Returns

• **posteriors** (*dict*) — key value pairs where key is the model parameter name and value is num_sample x posterior values

• **training_metrics** (*dict*) — metrics and meta data related to the training process

Module contents

**orbit.models package**

Submodules

**orbit.models.ets module**

**orbit.models.ets.ETS** (*seasonality=None, seasonality_sm_input=None, level_sm_input=None, estimator='stan-mcmc', suppress_stan_log=True, **kwargs*)

Parameters

• **seasonality** (*int*) — Length of seasonality

• **seasonality_sm_input** (*float*) — float value between [0, 1], applicable only if seasonality > 1. A larger value puts more weight on the current seasonality. If None, the model will estimate this value.

• **level_sm_input** (*float*) — float value between [0.0001, 1]. A larger value puts more weight on the current level. If None, the model will estimate this value.

• **estimator** (*string; {‘stan-mcmc’, ‘stan-map’}*) — default to be ‘stan-mcmc’.

• **response_col** (*str*) — Name of response variable column, default ‘y’

• **date_col** (*str*) — Name of date variable column, default ‘ds’

• **n_bootstrap_draws** (*int*) — Number of samples to bootstrap in order to generate the prediction interval. For full Bayesian and variational inference forecasters, samples are drawn directly from original posteriors. For point-estimated posteriors, it will be used to sample noise parameters. When -1 or None supplied, full Bayesian and variational inference forecasters will assume number of draws equal the size of original samples while point-estimated posteriors will mute the draw and output prediction without interval.

• **prediction_percentiles** (*list*) — List of integers of prediction percentiles that should be returned on prediction. To avoid reporting any confident intervals, pass an empty list

• **suppress_stan_log** (*bool*) — If False, turn off cmdstanpy logger. Default as False.

• ****kwargs** — additional arguments passed into orbit.estimators.stan_estimator
orbit.models.lgt.LGT

Parameters

- **seasonality** (int) – Length of seasonality
- **seasonality_sm_input** (float) – float value between [0, 1], applicable only if seasonality > 1. A larger value puts more weight on the current seasonality. If None, the model will estimate this value.
- **level_sm_input** (float) – float value between [0.0001, 1]. A larger value puts more weight on the current level. If None, the model will estimate this value.
- **regressor_col** (list) – Names of regressor columns, if any
- **regressor_sign** (list) – list with values { ‘+’, ‘-’, ‘=’ } such that ‘+’ indicates regressor coefficient estimates are constrained to [0, inf). ‘-’ indicates regressor coefficient estimates are constrained to (-inf, 0]. ‘=’ indicates regressor coefficient estimates can be any value between (-inf, inf). The length of regressor_sign must be the same length as regressor_col. If None, all elements of list will be set to ‘=’.
- **regressor_beta_prior** (list) – list of prior float values for regressor coefficient betas. The length of regressor_beta_prior must be the same length as regressor_col. If None, use non-informative priors.
- **regressor_sigma_prior** (list) – list of prior float values for regressor coefficient sigmas. The length of regressor_sigma_prior must be the same length as regressor_col. If None, use non-informative priors.
- **regression_penalty** ({ ’fixed_ridge’, ’lasso’, ’auto_ridge’ }) – regression penalty method
- **lasso_scale** (float) – float value between [0, 1], applicable only if regression_penalty == ‘lasso’
- **auto_ridge_scale** (float) – float value between [0, 1], applicable only if regression_penalty == ‘auto_ridge’
- **slope_sm_input** (float) – float value between [0, 1]. A larger value puts more weight on the current slope. If None, the model will estimate this value.
- **estimator** (string; {’stan-mcmc’, ’stan-map’, ’pyro-svi’}) – default to be ‘stan-mcmc’.
- **response_col** (str) – Name of response variable column, default ‘y’
- **date_col** (str) – Name of date variable column, default ‘ds’
- **n_bootstrap_draws** (int) – Number of samples to bootstrap in order to generate the prediction interval. For full Bayesian and variational inference forecasters, samples are drawn directly from original posteriors. For point-estimated posteriors, it will be used to sample noise parameters. When -1 or None supplied, full Bayesian and variational inference forecasters will assume number of draws equal the size of original samples while point-estimated posteriors will mute the draw and output prediction without interval.
• **prediction_percentiles**(list) – List of integers of prediction percentiles that should be returned on prediction. To avoid reporting any confident intervals, pass an empty list

• **suppress_stan_log**(bool) – If False, turn off cmdstanpy logger. Default as False.

• **kwargs** – additional arguments passed into orbit.estimators.stan_estimator or orbit.estimators.pyro_estimator

**orbit.models.dlt module**

**orbit.models.dlt.DLT**

```python
orbit.models.dlt.DLT(
    seasonality=None, seasonality_sm_input=None, level_sm_input=None,
    regressor_col=None, regressor_sign=None, regressor_beta_prior=None,
    regressor_sigma_prior=None, regressor_penalty='fixed_ridge', lasso_scale=0.5,
    auto_ridge_scale=0.5, slope_sm_input=None, period=1, damped_factor=0.8,
    global_trend_option='linear', global_cap=1.0, global_floor=0.0,
    global_trend_sigma_prior=None, forecast_horizon=1, estimator='stan-mcmc',
    suppress_stan_log=True, **kwargs)
```

**Parameters**

• **seasonality**(int) – Length of seasonality

• **seasonality_sm_input**(float) – float value between [0, 1], applicable only if seasonality > 1. A larger value puts more weight on the current seasonality. If None, the model will estimate this value.

• **level_sm_input**(float) – float value between [0.0001, 1]. A larger value puts more weight on the current level. If None, the model will estimate this value.

• **regressor_col**(list) – Names of regressor columns, if any

• **regressor_sign**(list) – list with values {‘+’, ‘-’, ‘=’} such that ‘+’ indicates regressor coefficient estimates are constrained to [0, inf). ‘-’ indicates regressor coefficient estimates are constrained to (-inf, 0]. ‘=’ indicates regressor coefficient estimates can be any value between (-inf, inf). The length of **regressor_sign** must be the same length as **regressor_col**. If None, all elements of list will be set to ‘=’.

• **regressor_beta_prior**(list) – list of prior float values for regressor coefficient betas. The length of **regressor_beta_prior** must be the same length as **regressor_col**. If None, use non-informative priors.

• **regressor_sigma_prior**(list) – list of prior float values for regressor coefficient sigmas. The length of **regressor_sigma_prior** must be the same length as **regressor_col**. If None, use non-informative priors.

• **regression_penalty** ({‘fixed_ridge’, ‘lasso’, ‘auto_ridge’}) – regression penalty method

• **lasso_scale**(float) – float value between [0, 1], applicable only if **regression_penalty** == ‘lasso’

• **auto_ridge_scale**(float) – float value between [0, 1], applicable only if **regression_penalty** == ‘auto_ridge’

• **slope_sm_input**(float) – float value between [0, 1]. A larger value puts more weight on the current slope. If None, the model will estimate this value.

• **period**(int) – Used to set time_delta as 1 / max(period, seasonality). If None and no seasonality, then time_delta == 1
• **damped_factor** *(float)* – Hyperparameter float value between [0, 1]. A smaller value further dampens the previous global trend value. Default, 0.8

• **global_trend_option** *(\{'linear', \'loglinear', \'logistic', \'flat'\})* – Transformation function for the shape of the forecasted global trend.

• **global_cap** *(float)* – Maximum value of global logistic trend. Default is set to 1.0. This value is used only when **global_trend_option** = 'logistic'

• **global_floor** *(float)* – Minimum value of global logistic trend. Default is set to 0.0. This value is used only when **global_trend_option** = 'logistic'

• **global_trend_sigma_prior** *(sigma prior of the global trend; default uses 1 standard deviation of response)* –

• **forecast_horizon** *(int)* – forecast_horizon will be used only when users want to specify optimization forecast horizon > 1

• **estimator** *(string; \{'stan-mcmc', \'stan-map'\})* – default to be 'stan-mcmc'.

• **response_col** *(str)* – Name of response variable column, default 'y'

• **date_col** *(str)* – Name of date variable column, default 'ds'

• **n_bootstrap_draws** *(int)* – Number of samples to bootstrap in order to generate the prediction interval. For full Bayesian and variational inference forecasters, samples are drawn directly from original posteriors. For point-estimated posteriors, it will be used to sample noise parameters. When -1 or None supplied, full Bayesian and variational inference forecasters will assume number of draws equal the size of original samples while point-estimated posteriors will mute the draw and output prediction without interval.

• **prediction_percentiles** *(list)* – List of integers of prediction percentiles that should be returned on prediction. To avoid reporting any confident intervals, pass an empty list

• **suppress_stan_log** *(bool)* – If False, turn off cmdstanpy logger. Default as False.

• ****kwargs** – additional arguments passed into orbit.estimators.stan_estimator

**orbit.models.ktrlite module**

**orbit.models.ktrlite.KTRLite**(level_knot_scale=0.1, level_segments=10, level_knot_distance=None, level_knot_dates=None, seasonality=None, seasonality_fs_order=None, seasonality_segments=2, seasonal_initial_knot_scale=1.0, seasonal_knot_scale=0.1, degree_of_freedom=30, date_freq=None, estimator='stan-map', suppress_stan_log=True, **kwargs)

**Parameters**

• **level_knot_scale** *(float)* – sigma for level; default to be .1

• **level_segments** *(int)* – the number of segments partitioned by the knots of level (trend)

• **level_knot_distance** *(int)* – the distance between every two knots of level (trend)

• **level_knot_dates** *(array like)* – list of pre-specified dates for the level knots

• **seasonality** *(int, or list of int)* – multiple seasonality

• **seasonality_fs_order** *(int, or list of int)* – fourier series order for seasonality

• **seasonality_segments** *(int)* – the number of segments partitioned by the knots of seasonality
orbit, Release 1.1.4.4

- **seasonal_initial_knot_scale** *(float)* – scale parameter for seasonal regressors initial coefficient knots; default to be 1
- **seasonal_knot_scale** *(float)* – scale parameter for seasonal regressors drift of coefficient knots; default to be 0.1.
- **degree_of_freedom** *(int)* – degree of freedom for error t-distribution
- **date_freq** *(str)* – date frequency; if not supplied, pd.infer_freq will be used to imply the date frequency.
- **estimator** *(string; {‘stan-map’})* –
- **response_col** *(str)* – Name of response variable column, default ‘y’
- **date_col** *(str)* – Name of date variable column, default ‘ds’
- **n_bootstrap_draws** *(int)* – Number of samples to bootstrap in order to generate the prediction interval. For full Bayesian and variational inference forecasters, samples are drawn directly from original posteriors. For point-estimated posteriors, it will be used to sample noise parameters. When -1 or None supplied, full Bayesian and variational inference forecasters will assume number of draws equal the size of original samples while point-estimated posteriors will mute the draw and output prediction without interval.
- **prediction_percentiles** *(list)* – List of integers of prediction percentiles that should be returned on prediction. To avoid reporting any confident intervals, pass an empty list
- **suppress_stan_log** *(bool)* – If False, turn off cmdstanpy logger. Default as False.
- **** **kwargs** – additional arguments passed into orbit.estimators.stan_estimator

### Module contents

**orbit.pyro package**

**Submodules**

**orbit.pyro.lgt module**

**class** orbit.pyro.lgt.Model(*data*)

Bases: object

max_plate_nesting = 1

### Module contents

**orbit.utils package**

**Submodules**

**orbit.utils.general module**

orbit.utils.general.expand_grid(*base*)

Given a base key values span, expand them into a dataframe covering all combinations :param base: dictionary with keys equal columns name and value equals key values :type base: dict
Returns
    pd.DataFrame
Return type
    dataframe generate based on user specified base

orbit.utils.general.get_parent_path(current_file_path)

Parameters
    • current_file_path (str) – The given file path, should be an absolute path
    • Returns –
    • ---------- – str : The parent path of give file path

orbit.utils.general.is_empty_dataframe(df)

Returns
    bool
Return type
    True if df is none, or if df is an empty dataframe; False otherwise.

orbit.utils.general.is_even_gap_datetime(array)

Returns True if array is evenly distributed

orbit.utils.general.is_ordered_datetime(array)

Returns True if array is ordered and non-repetitive

orbit.utils.general.regenerate_base_df(df, time_col, key_col, val_cols=[], fill_na=None)

Given a dataframe, key column, time column and value column, re-generate multiple time-series to cover full range date-time with all the keys. This can be a useful utils for working multiple time-series.

Parameters
    • df (pd.DataFrame) –
    • time_col (str) –
    • key_col (str) –
    • val_cols (List[str]; values column considered to be imputed) –
    • fill_na (Optional[float]; values to fill when there are missing values of the row) –

orbit.utils.general.update_dict(original_dict, append_dict)

orbit.utils.pyro module

orbit.utils.pyro.get_pyro_model(model_name)
orbit.utils.stan module

orbit.utils.stan.get_compiled_stan_model(stan_model_name: str = '', stan_file_path: str | None = None, exe_file_path: str | None = None, force_compile: bool = False) → CmdStanModel

Return a compiled Stan model using CmdStan. This includes both prepackaged models as well as user provided models through stan_file_path.

Parameters
- **stan_model_name** – The name of the Stan model to use. Use this for the built in models (dlt, ets, ktrlite, lgt)
- **stan_file_path** – The path to the Stan file to use. If not provided, the default is to search for the file in the ‘orbit’ package. If provided, function will ignore the stan_model_name parameter, and will compile the provide stan_file_path into executable in place (same folder as stan_file_path)
- **exe_file_path** – The path to the Stan-exe file to use. If not provided, the default is to search for the file in the ‘orbit’ package. If provided, function will ignore the stan_model_name parameter, and will compile the provide stan_file_path into executable in place (same folder as stan_file_path)

Returns
- **sm** – A compiled Stan model.

Return type
- **CmdStanModel**

class orbit.utils.stan.suppress_stdout_stderr

Bases: object

A context manager for doing a “deep suppression” of stdout and stderr in Python, i.e. will suppress all print, even if the print originates in a compiled C/Fortran sub-function.

This will not suppress raised exceptions, since exceptions are printed to stderr just before a script exits, and after the context manager has exited (at least, I think that is why it lets exceptions through).

Module contents

24.1.2 Submodules

24.1.3 orbit.exceptions module

exception orbit.exceptions.AbstractMethodException

Bases: Exception

exception orbit.exceptions.BacktestException

Bases: Exception

exception orbit.exceptions.DataInputException

Bases: Exception

exception orbit.exceptions.EstimatorException

Bases: Exception

24.1. orbit package
exception orbit.exceptions.ForecasterException
    Bases: Exception
exception orbit.exceptions.IllegalArgumentException
    Bases: Exception
exception orbit.exceptions.ModelException
    Bases: Exception
exception orbit.exceptions.PlotException
    Bases: Exception
exception orbit.exceptions.PredictionException
    Bases: Exception

24.1.4 orbit.orbit module

Top level Orbit class

class orbit.orbit.Orbit
    Bases: object

24.1.5 Module contents
CHAPTER TWENTYFIVE

CHANGELOG

25.1 1.1.4 (2024-01-21) (release notes)

Core Changes

• replace stan sampling engine PyStan2 by cmdstanpy (https://github.com/uber/orbit/pull/801)
• update installation process such that it pre-compile all stan files during wheel building (https://github.com/uber/orbit/pull/833), (https://github.com/uber/orbit/pull/835)

Documentation

• update read the doc process and underlying doc with respect to new changes (https://github.com/uber/orbit/pull/836), (https://github.com/uber/orbit/pull/838)
• prune old examples and duplicates under the example/ folder (https://github.com/uber/orbit/pull/838)

25.2 1.1.3 (2022-11-30) (release notes)

Core changes

• add python 3.8 unit tests (https://github.com/uber/orbit/pull/752)
• optimize interface to be compatible with arviz (https://github.com/uber/orbit/pull/755)
• requirements update (https://github.com/uber/orbit/pull/763)
• code clean up (https://github.com/uber/orbit/pull/765)
• dlt global trend prior adjustment (https://github.com/uber/orbit/pull/786)

Documentation

• tutorial refresh (https://github.com/uber/orbit/pull/795)

Utilities

• uses tqdm in parameters tuning (https://github.com/uber/orbit/pull/762)
• residuals plot (https://github.com/uber/orbit/pull/758)
• simpler stan compile interface (https://github.com/uber/orbit/pull/769)
25.3 1.1.2 (2022-04-28) (release notes)

Core changes

- Add Conda installation option (#679)
- Suppress the lengthy Stan logging message (#696)
- WBIC for pyro SVI sampling and BIC for MAP optimization (#719, #710)
- Backtest module to include confidence intervals (#724)
- Allow configuration for compiled Stan model path (#713)
- Box plot for regression coefficient comparison (#737)
- Bounded logistic growth for DLT model (#712)
- Enhance regression output reporting (#739)

Documentation

- Add blacking linting to Github action workflow (#708)
- Tutorial enhancement

Utilities

- Add a new method `make_future_df` to prepare data frame for forecasting (#695)

25.4 1.1.2alpha (2022-04-06) (release notes)

Core changes

- Add Conda installation option (#679)
- Suppress the lengthy Stan logging message (#696)
- WBIC for pyro SVI sampling and BIC for MAP optimization (#719, #710)
- Backtest module to include confidence intervals (#724)
- Allow configuration for compiled Stan model path (#713)
- Box plot for regression coefficient comparison (#737)
- Bounded logistic growth for DLT model (#712)
- Enhance regression output reporting (#739)

Documentation

- Add blacking linting to Github action workflow (#708)
- Tutorial enhancement

Utilities

- Add a new method `make_future_df` to prepare data frame for forecasting (#695)
25.5 1.1.1 (2022-03-03) (release notes)

Core changes

• fix the mplstyle file path bug (#714)

25.6 1.1.0 (2022-01-11) (release notes)

Core changes

• Redesign the model class structure with three core components: model template, estimator, and forecaster (#506, #507, #508, #513)
• Introduce the Kernel-based Time-varying Regression (KTR) model (#515)
• Implement the negative coefficient for LGT and KTR (#600, #601, #609)
• Allow to handle missing values in response for LGT and DLT (#645)
• Implement WBIC value for model candidate selection (#654)

Documentation

• A new series of tutorials for KTR (#558, #559)
• Migrate the CI from TravisCI to Github Actions (#556)
• Missing value handle tutorial (#645)
• WBIC tutorial (#663)

Utilities

• New Plotting Palette (#571, #589)
• Redesign the diagnostic plotting (#581, #607)
• Raise a warning when date index is not evenly distributed (#639)

25.7 1.0.17 (2021-08-30) (release notes)

Core changes

• Use global mean instead of median in ktrx model before next major release

25.8 1.0.16 (2021-08-27) (release notes)

Core changes

• Bug fix and code improvement before next major release (#540, #541, #546)
25.9 1.0.15 (2021-08-02) (release notes)

Core changes

- Prediction functionality refactoring (#430)
- KTRLite model enhancement and interface cleanup (#440)
- More flexible scheduling config in Backtester (#447)
- Allow extraction of training related metrics (e.g. ELBO loss) in Pyro SVI (#443)
- Add a flag to keep the posterior samples or not in aggregated model (#465)
- Bug fix and code improvement (#428, #438, #459, #470)

Documentation

- Clean up and standardize example notebooks (#462)
- Tutorial update and enhancement (#431, #474)

Utilities

- Diagnostic plot with Arviz (#433)
- Refine plotting palette (#434, #473)
- Create an orbit-featured plotting style (#434)

25.10 1.0.13 (2021-04-02) (release notes)

Core changes

- Implement a new model KTRLite (#380)
- Refactoring of BaseTemplate (#382, #384)
- Add MAPTemplate, FullBayesianTemplate, and AggregatedPosteriorTemplate (#394)
- Remove dependency of scikit-learn (#379, #381)

Documentation

- Add changelogs, release process, and contribution guidance (#363, #369, #370, #372)
- Setup documentation deployment via TravisCI (#291)
- New tutorial of making your own model (#389)
- Tutorial enhancement (#383, #388)

Utilities

- New EDA plot utilities (#403, #407, #408)
- More options for existing plot utilities (#396)
25.11 1.0.12 (2021-02-19) (release notes)

- Documentation update (#354, #362)
- Providing prediction intervals for point posteriors such as AggregatedPosterior and MAP (#357, #359)
- Abstract classes created to refactor posteriors estimation as templates (#360)
- Automating documentation and tutorials; migrating docs to readthedocs (#291)

25.12 1.0.11 (2021-02-18) (release notes)

Core changes
- a simple ETS class is created (#280, #296)
- DLT is replacing LGT as the model used in the quick start and general demos (#305)
- DLT and LGT are refactored to inherit from ETS (#280)
- DLT now supports regression with strictly positive/negative signs (#296)
- deprecation on regression with LGT (#305)
- dependency update; remove enum34 and update other dependencies versions (#301)
- fixed pickle error (#342)

Documentation
- updated tutorials (#309, #329, #332)
- docstring cleanup with inherited classes (#350)

Utilities
- include the provide hyper-parameters tuning (#288)
- include dataloader with a few standard datasets (#352, #337, #277, #248)
- plotting functions now returns the plot object (#327, #325, #287, #279)

25.13 1.0.10 (2020-11-15) (Initial Release)

- dpl v2 for travis config (#295)

25.14 1.0.9 (2020-11-15)

- debug travis pypi deployment (#293)
- Debug travis package deployment (#294)
25.15 1.0.8 (2020-11-15)

- debug travis pypi deployment (#293)

25.16 1.0.7 (2020-11-14)

- #279
- reorder fourier series calculation to match the df (#286)
- plot utility enhancement (#287)
- Setup TravisCI deployment for PyPI (#292)

25.17 1.0.6 (2020-11-13)

- #251
- #257
- #259
- #263
- #248
- #264
- #265
- #270
- #273
- #277
- #281
- #282
INDICES AND TABLES

- genindex
- modindex
- search
PYTHON MODULE INDEX

O
orbit, 170
orbit.constants, 155
orbit.constants.constants, 151
orbit.constants.palette, 154
orbit.diagnostics, 159
orbit.diagnostics.plot, 155
orbit.estimators, 163
orbit.estimators.base_estimator, 159
orbit.estimators.pyro_estimator, 160
orbit.estimators.stan_estimator, 161
orbit.exceptions, 169
orbit.models, 167
orbit.models.dlt, 165
orbit.models.ets, 163
orbit.models.ktrlite, 166
orbit.models.lgt, 164
orbit.orbit, 170
orbit.pyro, 167
orbit.pyro.lgt, 167
orbit.utils, 169
orbit.utils.general, 167
orbit.utils.pyro, 168
orbit.utils.stan, 169
INDEX

A
AbstractMethodException, 169
ACTUAL (orbit.constants.constants.BacktestFitKeys attribute), 151
ACTUAL_OBS (orbit.constants.palette.PredictionPaletteClassic attribute), 155
ACTUAL_RESPONSE (orbit.constants.constants.PlotLabels attribute), 152

B
BacktestException, 169
BacktestFitKeys (class in orbit.constants.constants), 151
BaseEstimator (class in orbit.estimators.base_estimator), 159
BLACK (orbit.constants.palette.OrbitPalette attribute), 154
BLACK_GRADIENT (orbit.constants.palette.OrbitColorMap attribute), 154
BLUE (orbit.constants.palette.OrbitPalette attribute), 154
BLUE600 (orbit.constants.palette.OrbitPalette attribute), 154
BLUE_GRADIENT (orbit.constants.palette.OrbitColorMap attribute), 154
BROWN (orbit.constants.palette.OrbitPalette attribute), 155

C
CHILD (orbit.constants.constants.CompiledStanModelPath attribute), 152
CmdStanMAP (orbit.constants.constants.EstimatorsKeys attribute), 152
CmdStanMCMC (orbit.constants.constants.EstimatorsKeys attribute), 152
CompiledStanModelPath (class in orbit.constants.constants), 151

D
DataInputException, 169
DATE (orbit.constants.constants.BacktestFitKeys attribute), 151
DATE_ARRAY (orbit.constants.constants.PredictionMetaKeys attribute), 153
DATE_ARRAY (orbit.constants.constants.TrainingMetaKeys attribute), 153
DATE_COL (orbit.constants.constants.TrainingMetaKeys attribute), 153
DLT() (in module orbit.models.dlt), 165

E
END (orbit.constants.constants.PredictionMetaKeys attribute), 153
END (orbit.constants.constants.TrainingMetaKeys attribute), 154
END_INDEX (orbit.constants.constants.PredictionMetaKeys attribute), 153
EstimatorException, 169
EstimatorsKeys (class in orbit.constants.constants), 152
ETS() (in module orbit.models.ets), 163
expand_grid() (in module orbit.utils.general), 167

F
fit() (orbit.estimators.base_estimator.BaseEstimator method), 159
fit() (orbit.estimators.pyro_estimator.PyroEstimator method), 160
fit() (orbit.estimators.pyro_estimator.PyroEstimatorSVI method), 161
fit() (orbit.estimators.stan_estimator.StanEstimator method), 161
fit() (orbit.estimators.stan_estimator.StanEstimatorMAP method), 162
fit() (orbit.estimators.stan_estimator.StanEstimatorMCMC method), 162
ForecasterException, 169
FULL_SAMPLING (orbit.constants.constants.PredictMethod attribute), 152
FUTURE_STEPS (orbit.constants.constants.PredictionMetaKeys attribute), 153

G
get_compiled_stan_model() (in module orbit-
bit.utils.stan), 169
get_parent_path() (in module orbit.utils.general), 168
get_pyro_model() (in module orbit.utils.pyro), 168
GREEN (orbit.constants.palette.OrbitPalette attribute), 155
GREEN600 (orbit.constants.palette.OrbitPalette attribute), 155
GREEN_GRADIENT (orbit.constants.palette.OrbitColorMap attribute), 154
H
HOLDOUT_VERTICAL_LINE (orbit.constants.palette.PredictionPaletteClassic attribute), 155
IllegalArgument, 170
is_empty_dataframe() (in module orbit.utils.general), 168
is_even_gap_datetime() (in module orbit.utils.general), 168
is_ordered_datetime() (in module orbit.utils.general), 168
K
KNOTS_REGION (orbit.constants.palette.KTRPalette attribute), 154
KNOTS_SEGMENT (orbit.constants.palette.KTRPalette attribute), 154
KTRLite() (in module orbit.models.ktrlite), 166
KTRPalette (class in orbit.constants.palette), 154
KTRTimePointPriorKeys (class in orbit.constants.constants), 152
L
LGT() (in module orbit.models.lgt), 164
M
MAP (orbit.constants.constants.PredictMethod attribute), 152
max_plate_nesting (orbit.pyro.lgt.Model attribute), 167
MEAN (orbit.constants.constants.PredictMethod attribute), 152
MEDIAN (orbit.constants.constants.PredictMethod attribute), 152
metric_horizon_barplot() (in module orbit.diagnostics.plot), 155
METRIC_NAME (orbit.constants.constants.BacktestFitKeys attribute), 151
METRIC_VALUES (orbit.constants.constants.BacktestFitKeys attribute), 151
Model (class in orbit.pyro.lgt), 167
MODEL (orbit.constants.constants.TimeSeriesSplitSchemeKeys attribute), 153
ModelException, 170
module
orbit, 170
orbit.constants, 155
orbit.constants.constants, 151
orbit.constants.palette, 154
orbit.diagnostics, 159
orbit.diagnostics.plot, 155
orbit.estimators, 163
orbit.estimators.base_estimator, 159
orbit.estimators.pyro_estimator, 160
orbit.estimators.stan_estimator, 161
orbit.exceptions, 169
orbit.models, 167
orbit.models.dlt, 165
orbit.models.ets, 163
orbit.models.ktrlite, 166
orbit.models.lgt, 164
orbit.orbit, 170
orbit.pyro, 167
orbit.pyro.lgt, 167
orbit.utils, 169
orbit.utils.general, 167
orbit.utils.pyro, 168
orbit.utils.stan, 169
N
NAME (orbit.constants.constants.KTRTimePointPriorKeys attribute), 152
NUM_OF_OBS (orbit.constants.constants.TrainingMetaKeys attribute), 154
ORANGE (orbit.constants.palette.OrbitPalette attribute), 155
O
ORANGE (orbit.constants.palette.OrbitPalette attribute), 155
Index 183

params_comparison_boxplot() (in module orbit.diagnostics.plot), 155
PARENT (orbit.constants.constants.CompiledStanModelPath attribute), 152
plot_bt_predictions() (in module orbit.diagnostics.plot), 156
plot_bt_predictions2() (in module orbit.diagnostics.plot), 157
plot_predicted_components() (in module orbit.diagnostics.plot), 157
plot_predicted_data() (in module orbit.diagnostics.plot), 158
PlotException, 170
PlotLabels (class in orbit.constants.constants), 152
PREDICTED (orbit.constants.constants.BacktestFitKeys attribute), 151
PREDICTED_RESPONSE (orbit.constants.constants.PlotLabels attribute), 152
PREDICTION (orbit.constants.constants.PredictionKeys attribute), 153
PREDICTION_DF_LEN (orbit.constants.constants.PredictionMetaKeys attribute), 153
PREDICTION_INTERVAL (orbit.constants.palette.PredictionPaletteClassic attribute), 155
PREDICTION_LINE (orbit.constants.palette.PredictionPaletteClassic attribute), 155
PredictionException, 170
PredictionKeys (class in orbit.constants.constants), 153
PredictionMetaKeys (class in orbit.constants.constants), 153
PredictionPaletteClassic (class in orbit.constants.palette), 155
PredictMethod (class in orbit.constants.constants), 152
PRIOR_END_TP_IDX (orbit.constants.constants.KTRTimePointPriorKeys attribute), 152
PRIOR_MEAN (orbit.constants.constants.KTRTimePointPriorKeys attribute), 152
PRIOR_REGRESSOR_COL (orbit.constants.constants.KTRTimePointPriorKeys attribute), 152
PRIOR_SD (orbit.constants.constants.KTRTimePointPriorKeys attribute), 152
PRIOR_START_TP_IDX (orbit.constants.constants.KTRTimePointPriorKeys attribute), 152
PURPLE (orbit.constants.palette.OrbitPalette attribute), 155
PURPLE_GRADIENT (orbit.constants.palette.OrbitColorMap attribute), 154
PyroEstimator (class in orbit.estimators.pyro_estimator), 160
PyroEstimatorSVI (class in orbit.estimators.pyro_estimator), 160
PyroSVI (orbit.constants.constants.EstimatorsKeys attribute), 152
RAINBOW (orbit.constants.palette.OrbitColorMap attribute), 154
RED (orbit.constants.palette.OrbitPalette attribute), 155
RED_GRADIENT (orbit.constants.palette.OrbitColorMap attribute), 154
regenerate_base_df() (in module orbit.utils.general), 168
REGRESSION (orbit.constants.constants.PredictionKeys attribute), 153

P

params_comparison_boxplot() (in module orbit.diagnostics.plot), 155
PARENT (orbit.constants.constants.CompiledStanModelPath attribute), 152
plot_bt_predictions() (in module orbit.diagnostics.plot), 156
plot_bt_predictions2() (in module orbit.diagnostics.plot), 157
plot_predicted_components() (in module orbit.diagnostics.plot), 157
plot_predicted_data() (in module orbit.diagnostics.plot), 158
PlotException, 170
PlotLabels (class in orbit.constants.constants), 152
PREDICTED (orbit.constants.constants.BacktestFitKeys attribute), 151
PREDICTED_RESPONSE (orbit.constants.constants.PlotLabels attribute), 152
PREDICTION (orbit.constants.constants.PredictionKeys attribute), 153
PREDICTION_DF_LEN (orbit.constants.constants.PredictionMetaKeys attribute), 153
PREDICTION_INTERVAL (orbit.constants.palette.PredictionPaletteClassic attribute), 155
PREDICTION_LINE (orbit.constants.palette.PredictionPaletteClassic attribute), 155
PredictionException, 170
PredictionKeys (class in orbit.constants.constants), 153
PredictionMetaKeys (class in orbit.constants.constants), 153
PredictionPaletteClassic (class in orbit.constants.palette), 155
PredictMethod (class in orbit.constants.constants), 152
PRIOR_END_TP_IDX (orbit.constants.constants.KTRTimePointPriorKeys attribute), 152
PRIOR_MEAN (orbit.constants.constants.KTRTimePointPriorKeys attribute), 152
PRIOR_REGRESSOR_COL (orbit.constants.constants.KTRTimePointPriorKeys attribute), 152
PRIOR_SD (orbit.constants.constants.KTRTimePointPriorKeys attribute), 152
PRIOR_START_TP_IDX (orbit.constants.constants.KTRTimePointPriorKeys attribute), 152
PURPLE (orbit.constants.palette.OrbitPalette attribute), 155
PURPLE_GRADIENT (orbit.constants.palette.OrbitColorMap attribute), 154
PyroEstimator (class in orbit.estimators.pyro_estimator), 160
PyroEstimatorSVI (class in orbit.estimators.pyro_estimator), 160
PyroSVI (orbit.constants.constants.EstimatorsKeys attribute), 152
RAINBOW (orbit.constants.palette.OrbitColorMap attribute), 154
RED (orbit.constants.palette.OrbitPalette attribute), 155
RED_GRADIENT (orbit.constants.palette.OrbitColorMap attribute), 154
regenerate_base_df() (in module orbit.utils.general), 168
REGRESSION (orbit.constants.constants.PredictionKeys attribute), 153
REGRESSOR (orbit.constants.constants.PredictionKeys attribute), 153
residual_diagnostic_plot() (in module orbit.diagnostics.plot), 158
RESPONSE (orbit.constants.constants.TrainingMetaKeys attribute), 154
RESPONSE_COL (orbit.constants.constants.TrainingMetaKeys attribute), 154
RESPONSE_MEAN (orbit.constants.constants.TrainingMetaKeys attribute), 154
RESPONSE_SD (orbit.constants.constants.TrainingMetaKeys attribute), 154
S
SEASONALITY (orbit.constants.constants.PredictionKeys attribute), 153
SPLIT_KEY (orbit.constants.constants.BacktestFitKeys attribute), 151
SPLIT_TYPE_EXPANDING (orbit.constants.constants.TimeSeriesSplitSchemeKeys attribute), 153
SPLIT_TYPE_ROLLING (orbit.constants.constants.TimeSeriesSplitSchemeKeys attribute), 153
StanEstimator (class in orbit.estimators.stan_estimator), 161
StanEstimatorMAP (class in orbit.estimators.stan_estimator), 162
StanEstimatorMCMC (class in orbit.estimators.stan_estimator), 162
StanMAP (orbit.constants.constants.EstimatorsKeys attribute), 152
StanMCMC (orbit.constants.constants.EstimatorsKeys attribute), 152
START (orbit.constants.constants.PredictionMetaKeys attribute), 153
START (orbit.constants.constants.TrainingMetaKeys attribute), 154
START_INDEX (orbit.constants.constants.PredictionMetaKeys attribute), 153
suppress_stdout_stderr (class in orbit.utils.stan), 169
T
TEST_ACTUAL (orbit.constants.constants.BacktestFitKeys attribute), 151
TEST_IDX (orbit.constants.constants.TimeSeriesSplitSchemeKeys attribute), 153
TEST_OBS (orbit.constants.palettes.PredictionPaletteClassic attribute), 155
TEST_PREDICTED (orbit.constants.constants.BacktestFitKeys attribute), 151
TimeSeriesSplitSchemeKeys (class in orbit.constants.constants), 153
TRAIN_ACTUAL (orbit.constants.constants.BacktestFitKeys attribute), 151
TRAIN_END_DATE (orbit.constants.constants.TimeSeriesSplitSchemeKeys attribute), 153
TRAIN_FLAG (orbit.constants.constants.BacktestFitKeys attribute), 151
TRAIN_IDX (orbit.constants.constants.TimeSeriesSplitSchemeKeys attribute), 153
TRAIN_METRIC_FLAG (orbit.constants.constants.BacktestFitKeys attribute), 151
TRAIN_PREDICTED (orbit.constants.constants.BacktestFitKeys attribute), 151
TRAIN_START_DATE (orbit.constants.constants.TimeSeriesSplitSchemeKeys attribute), 153
TRAINING_ACTUAL_RESPONSE (orbit.constants.constants.PlotLabels attribute), 152
TrainingMetaKeys (class in orbit.constants.constants), 153
TREND (orbit.constants.constants.PredictionKeys attribute), 153
U
update_dict() (in module orbit.utils.general), 168
W
WHITE (orbit.constants.palettes.OrbitPalette attribute), 155
YELLOW (orbit.constants.palettes.OrbitPalette attribute), 155
YELLOW400 (orbit.constants.palettes.OrbitPalette attribute), 155
YELLOW_GRADIENT (orbit.constants.palettes.OrbitColorMap attribute), 154