orbit

Release 1.1.4.9

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

Mar 31, 2024

BASIC

CHAPTER

ONE

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:

```
@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]

CHAPTER

TWO

INSTALLATION

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 .
```

CHAPTER

THREE

QUICK START

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__)
```

```
1.1.4.6
```

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]:
```
week                datetime64[ns]
claims                     float64
trend.unemploy             float64
trend.filling              float64
trend.job                  float64
sp500                      float64
vix                        float64
dtype: object
```

[4]:
```
df.head(5)
```

[4]:
```
        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
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  0.110445
2  0.532339
3  0.428645
4  0.487404
```

Train-test split.

[5]:
```
test_size = 52
train_df = df[:-test_size]
test_df = df[-test_size:]
```

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`.

[6]:
```
ets = ETS(
    response_col=response_col,
    date_col=date_col,
    seasonality=52,
    seed=2024,
    estimator="stan-mcmc",
    stan_mcmc_args={'show_progress': False},
)
```

[7]:
```
%%time
ets.fit(df=train_df)
```

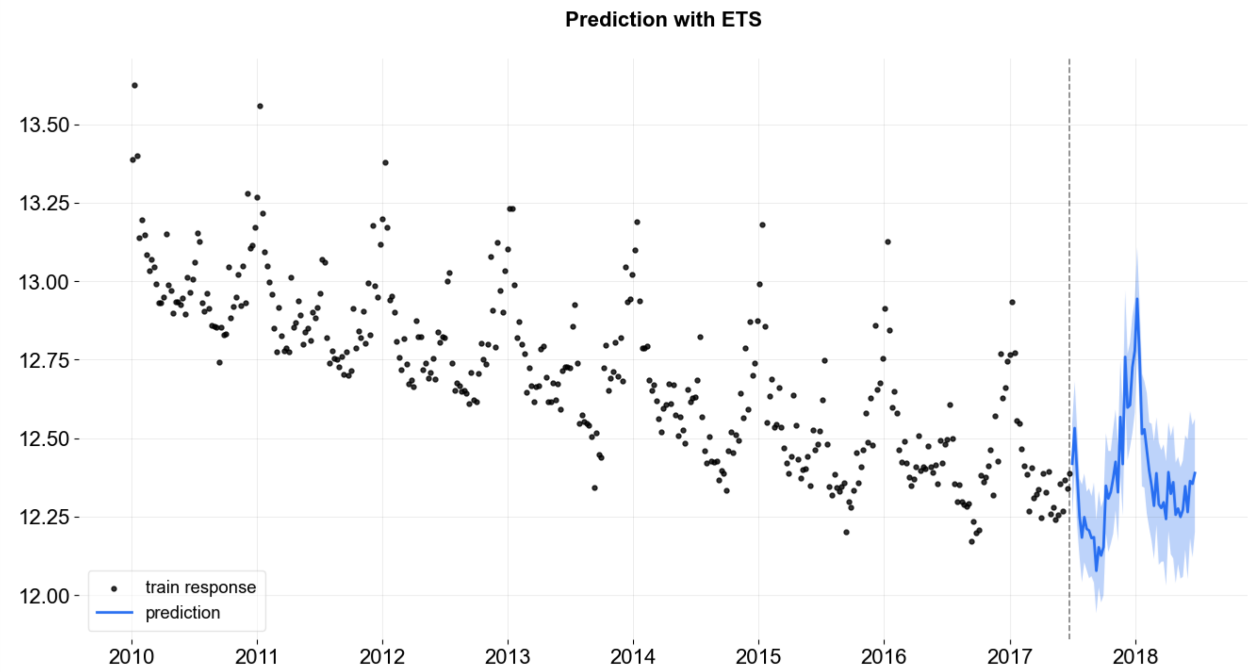
```
2024-03-19 23:42:10 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 1.000, warmups (per chain): 225 and samples(per chain): 25.
```

```
CPU times: user 42.9 ms, sys: 24.3 ms, total: 67.2 ms
Wall time: 613 ms
```

```
[7]: <orbit.forecaster.full_bayes.FullBayesianForecaster at 0x2a5274b50>
```

```
[8]: predicted_df = ets.predict(df=test_df)
```

```
[9]: _ = plot_predicted_data(train_df, predicted_df, date_col, response_col, title=
     ↪'Prediction with ETS')
```



3.4 Extract and Analyze Posterior Samples

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

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

```
[10]: dict_keys(['l', 'lev_sm', 'obs_sigma', 's', 'sea_sm', 'loglk'])
```

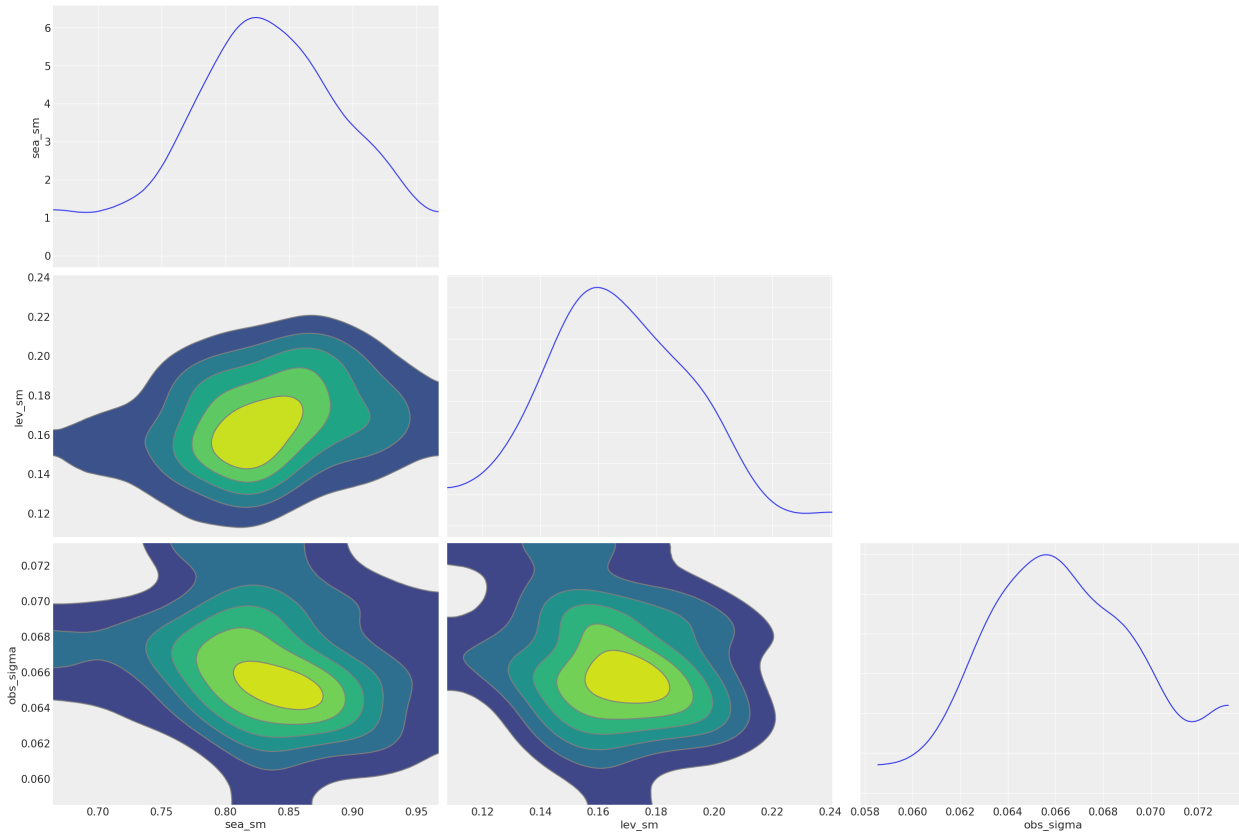
The extracted parameters posteriors are pretty much compatible diagnostic with arviz. To do that, users can set `permute=False` to preserve chain information.

```
[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)

(continued from previous page)

```
    var_names=["sea_sm", "lev_sm", "obs_sigma"],
    kind="kde",
    marginals=True,
    textsize=15,
)
plt.show()
```



For more details in model diagnostics visualization, there is a subsequent section dedicated to it.

CHAPTER

FOUR

METHODS OF ESTIMATIONS AND PREDICTIONS

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

[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__)
```

```
1.1.4.6
```

[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)

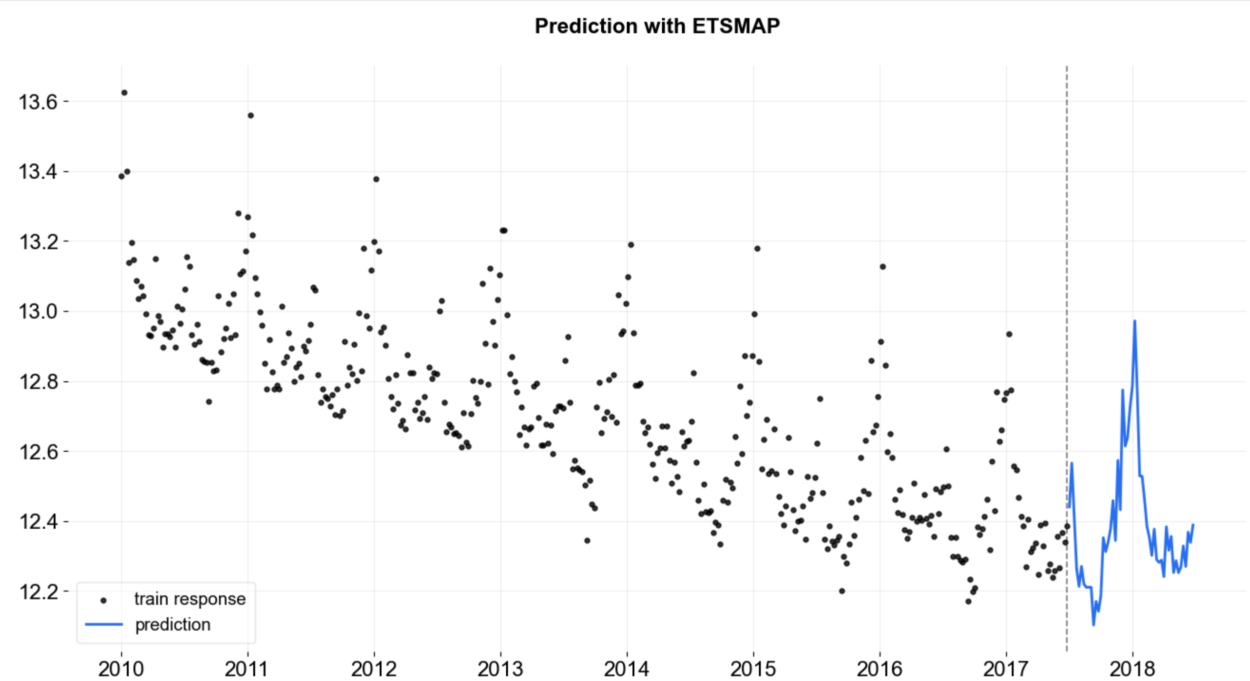
To use MAP method, one can simply specify `estimator='stan-map'` when instantiating a model. The advantage of MAP estimation is a faster computational speed. In MAP, the uncertainty is mainly generated the noise process with bootstrapping. However, the uncertainty would not cover parameters variance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

[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)
```

```
2024-03-19 23:40:31 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
```

```
CPU times: user 9.55 ms, sys: 11.8 ms, total: 21.3 ms
Wall time: 75.3 ms
```

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



To have the uncertainty from MAP, one can speicify `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`.

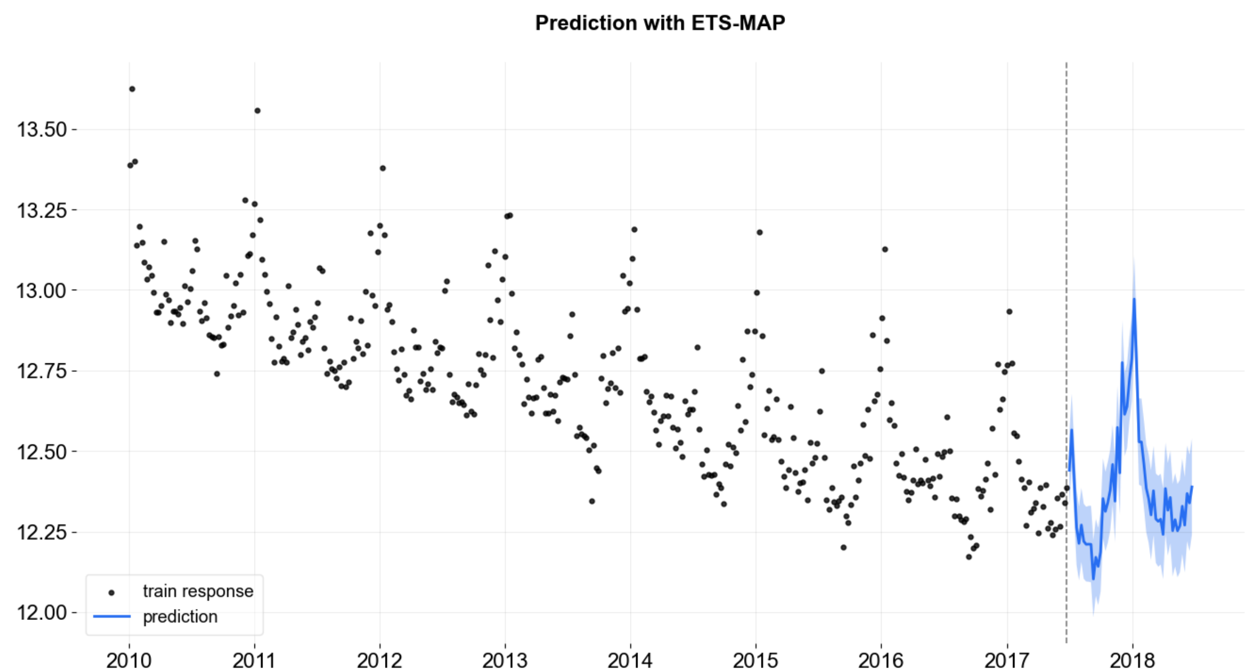
[6]:
```
# default: [10, 90]
prediction_percentiles=[10, 90]
```

(continues on next page)

(continued from previous 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')
```

```
2024-03-19 23:40:32 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
```



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
```

[8]:
```
(1, 391)
```

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.

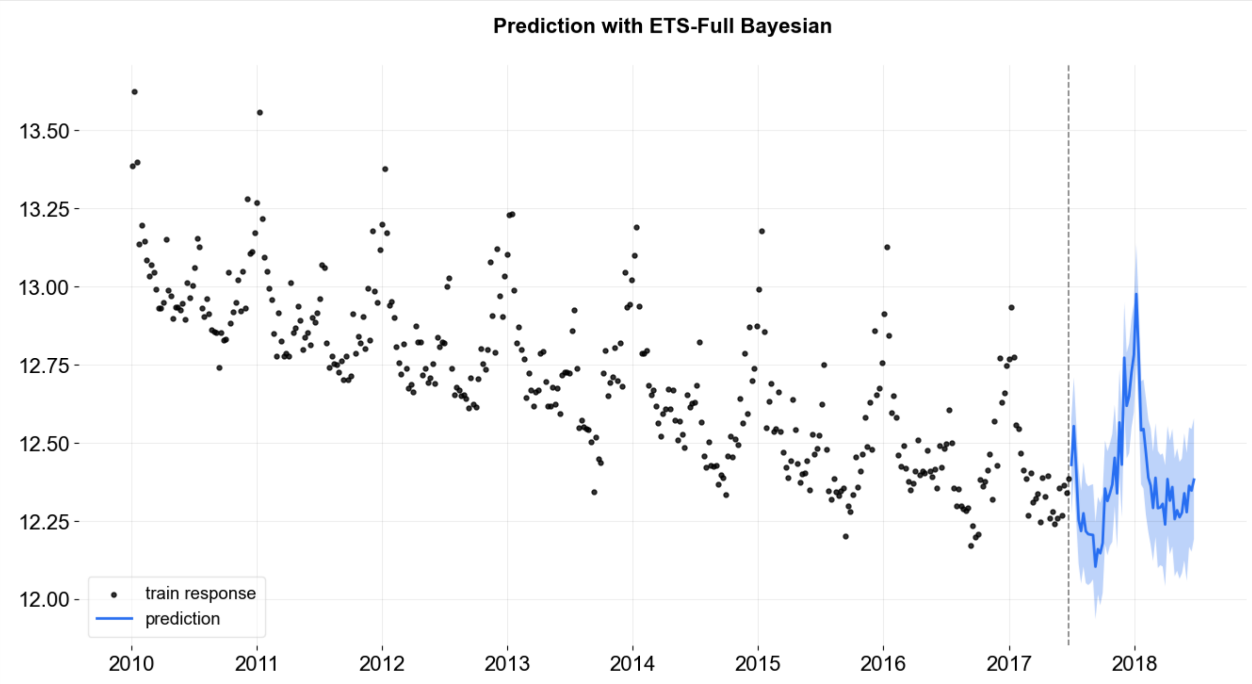
4.3.1 MCMC - Full Bayesian Sampling

[9]:
```
%%time
ets = 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},
)
ets.fit(df=train_df)
predicted_df = ets.predict(df=test_df)
```

```
2024-03-19 23:40:32 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 1.000, warmups (per chain): 100 and samples(per chain): 100.
```

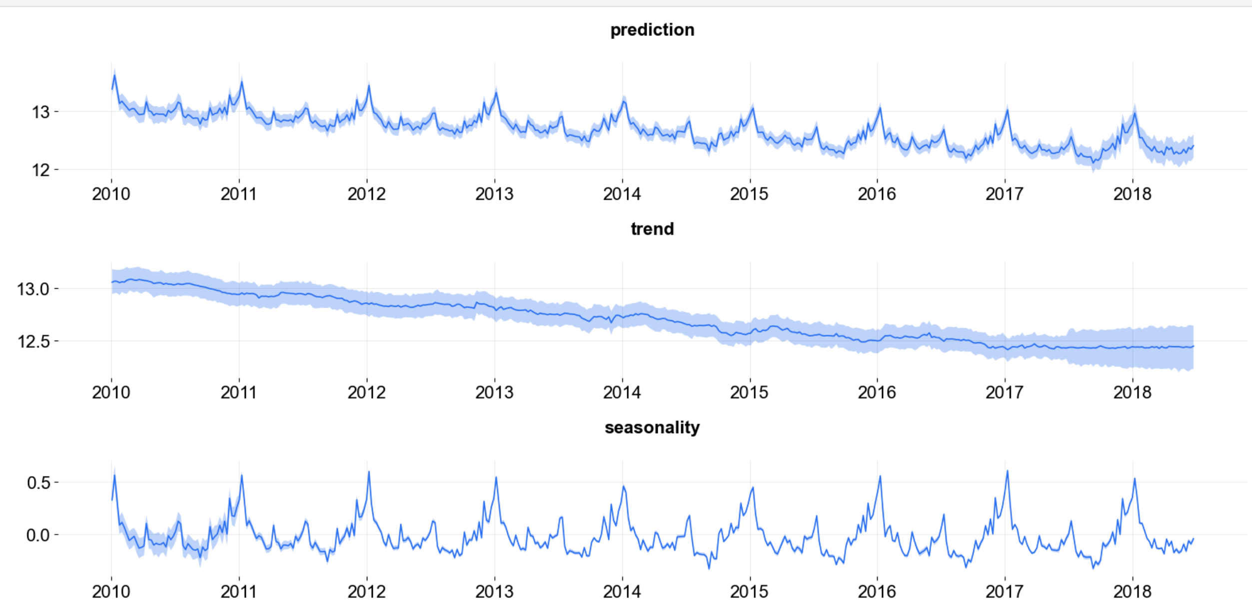
```
CPU times: user 143 ms, sys: 45.5 ms, total: 189 ms
Wall time: 608 ms
```

[10]:
```
_ = 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.

[11]:
```
predicted_df = ets.predict(df=df, decompose=True)
plot_predicted_components(predicted_df, date_col=date_col,
                          plot_components=['prediction', 'trend', 'seasonality'])
```



[11]:
```
array([<Axes: title={'center': 'prediction'}>,
       <Axes: title={'center': 'trend'}>,
       <Axes: title={'center': 'seasonality'}>], dtype=object)
```

Just like the `MAPForecaster`, one can also access the posterior samples by calling the function `.get_posterior_samples()`.

[12]:
```
posterior_samples = ets.get_posterior_samples()
posterior_samples.keys()
```

[12]:
```
dict_keys(['l', 'lev_sm', 'obs_sigma', 's', 'sea_sm', 'loglk'])
```

As mentioned, in **MCMC** (**Full Bayesian**) models, the first dimension reflects the sample size.

[13]:
```
lev = posterior_samples['l']
lev.shape
```

[13]:
```
(400, 391)
```

4.3.2 MCMC - Point Estimation

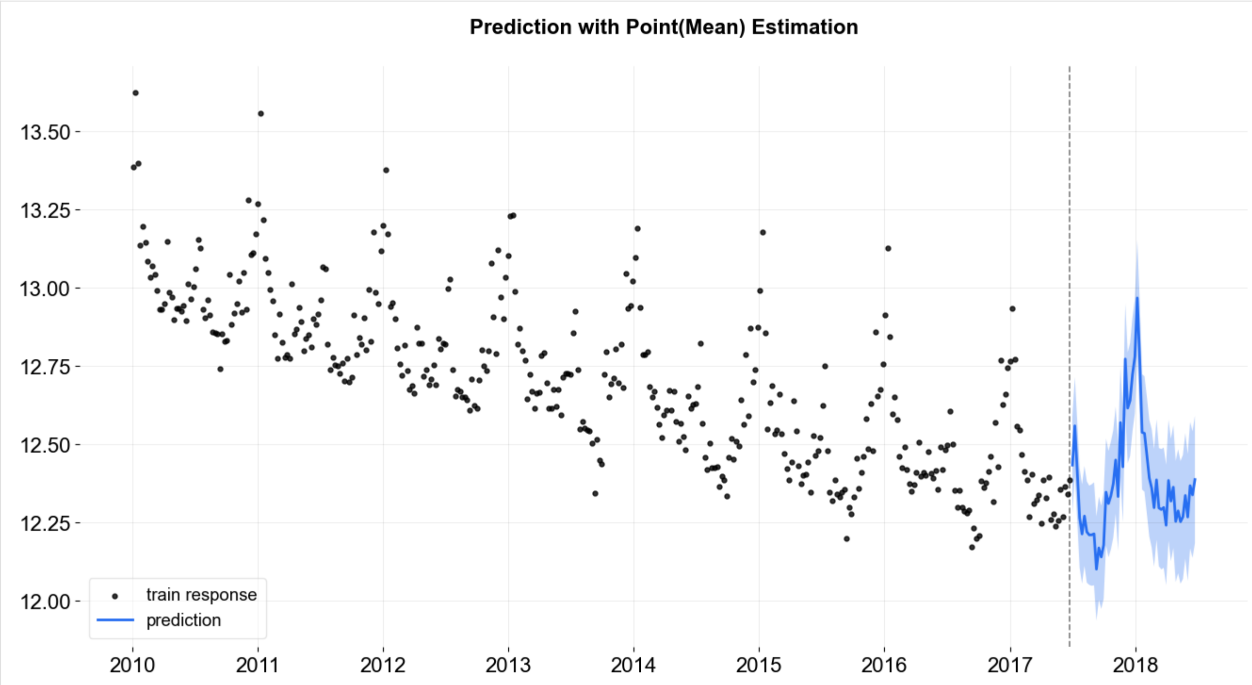
Users can also choose to derive point estimates via MCMC by specifying `point_method` as `mean` or `median` via the call of `.fit`. In that case, posteriors samples are first aggregated by mean or median and store as a point estimate for final prediction. Just like other point estimate, users can specify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

[14]:
```
%%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)
```

```
2024-03-19 23:40:33 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 1.000, warmups (per chain): 225 and samples(per chain): 25.
```

```
CPU times: user 346 ms, sys: 245 ms, total: 591 ms
Wall time: 959 ms
```

[15]:
```
_ = plot_predicted_data(train_df, predicted_df, date_col, response_col,
                        title='Prediction with Point(Mean) Estimation')
```



One can always access the the point estimated posteriors by `.get_point_posteriors()` (including the cases fitting the parameters through MCMC).

[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

[1]:
```
import numpy as np

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

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

```
1.1.4.6
```

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

[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
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  0.110445
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']
```

```
2024-03-19 23:42:23 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:42:23 - 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']
```

```
2024-03-19 23:42:23 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:42:23 - 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)

(continued from previous 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']
```

```
2024-03-19 23:42:24 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 1.000, warmups (per chain): 225 and samples(per chain): 25.
2024-03-19 23:42:36 - 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
```

CHAPTER

SIX

USING PYRO FOR ESTIMATION

Note

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.

[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

from orbit.constants.palette import OrbitPalette

import warnings
warnings.filterwarnings('ignore')
```

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

```
1.1.4.6
```

[3]:
```
df = load_iclaims()
```

[4]:
```
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`.

[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,
)
```

[6]:
```
%%time
lgt_vi.fit(df=train_df)
```

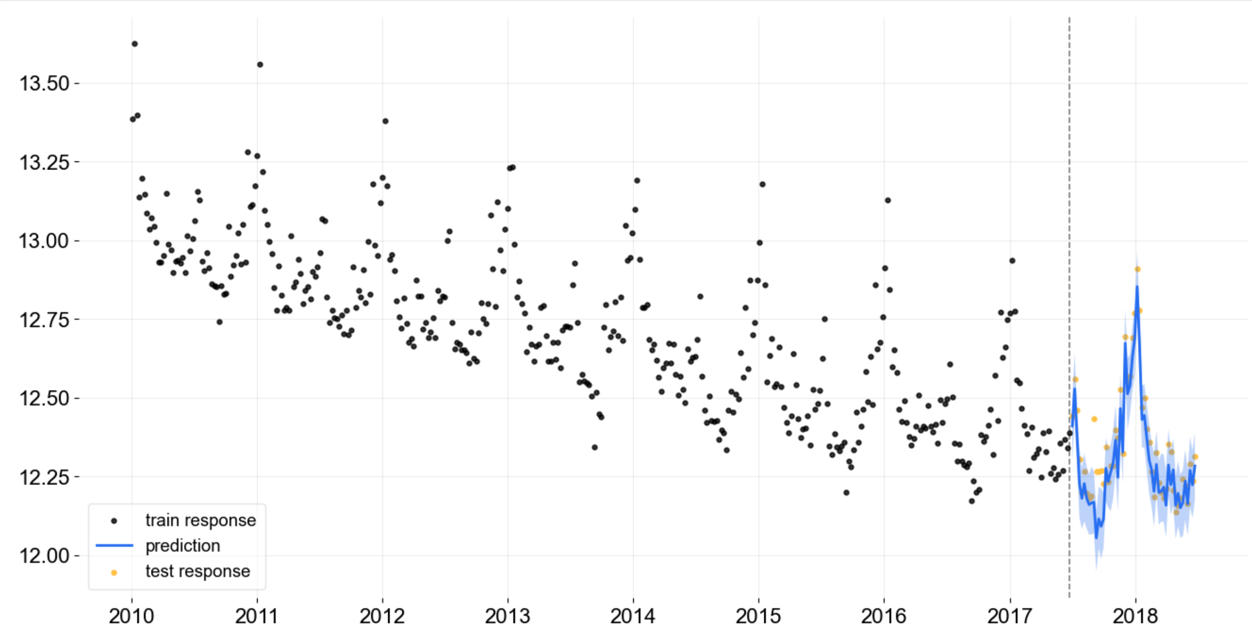
```
2024-03-19 23:40:42 - orbit - INFO - Using SVI (Pyro) with steps: 101, samples: 300,␣
↪learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
2024-03-19 23:40:42 - orbit - INFO - step    0 loss = 658.91, scale = 0.11635
INFO:orbit:step    0 loss = 658.91, scale = 0.11635
2024-03-19 23:40:46 - orbit - INFO - step   50 loss = -432, scale = 0.48623
INFO:orbit:step   50 loss = -432, scale = 0.48623
2024-03-19 23:40:49 - orbit - INFO - step  100 loss = -444.07, scale = 0.34976
INFO:orbit:step  100 loss = -444.07, scale = 0.34976
```

```
CPU times: user 11.8 s, sys: 33.2 s, total: 45 s
Wall time: 7.48 s
```

[6]:
```
<orbit.forecaster.svi.SVIForecaster at 0x289535f10>
```

[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)
```

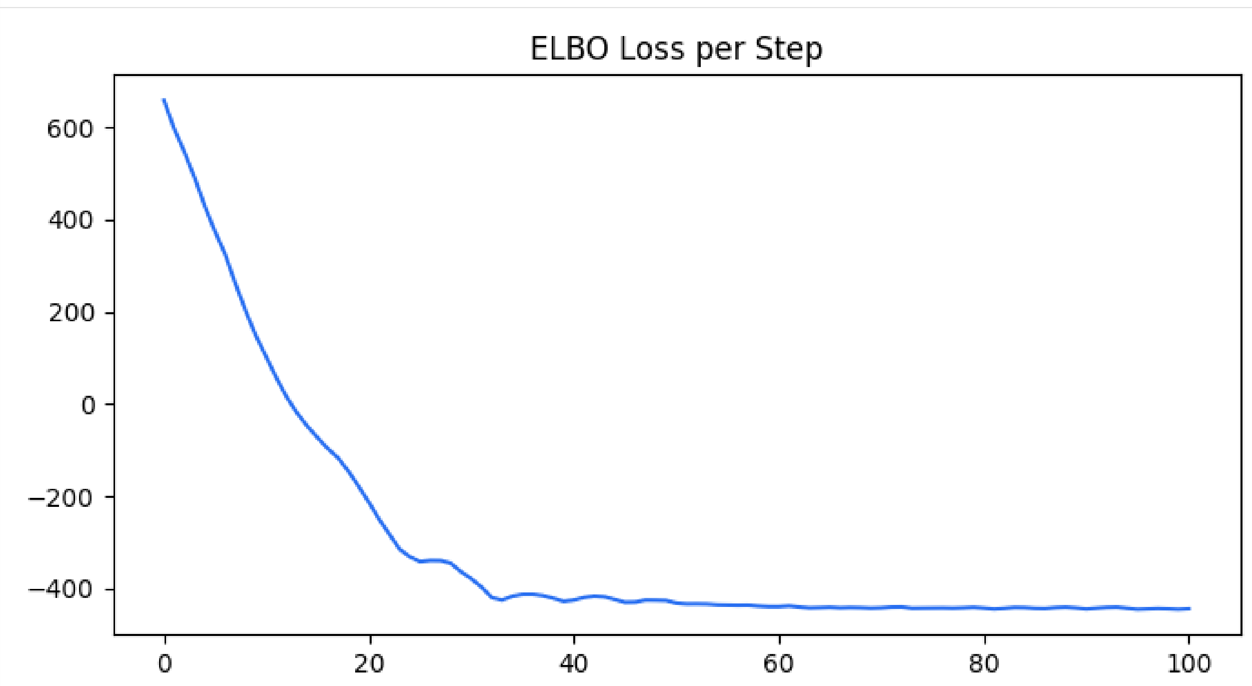


We can also extract the ELBO loss from the training metrics.

[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')
```



CHAPTER

SEVEN

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__)
```

```
1.1.4.6
```

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 Rlgt (Smyl et al., 2019). The model has a structural forecast equations

$$\begin{aligned} y_t &= \mu_t + s_t + r_t + \epsilon_t \\ \mu_t &= g_t + l_{t-1} + \theta b_{t-1} \\ \epsilon_t &\sim \mathtt{Student}(\nu, 0, \sigma) \\ \sigma &\sim \mathtt{HalfCauchy}(0, \gamma_0) \end{aligned}$$

with the update process

$$\begin{aligned} g_t &= D(t) \\ l_t &= \rho_l(y_t - g_t - s_t - r_t) + (1 - \rho_l)(l_{t-1} + \theta b_{t-1}) \\ b_t &= \rho_b(l_t - l_{t-1}) + (1 - \rho_b)\theta b_{t-1} \\ s_{t+m} &= \rho_s(y_t - l_t - r_t) + (1 - \rho_s)s_t \\ r_t &= \Sigma_j \beta_j x_{jt} \end{aligned}$$

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

[3]:
```
# 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:

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

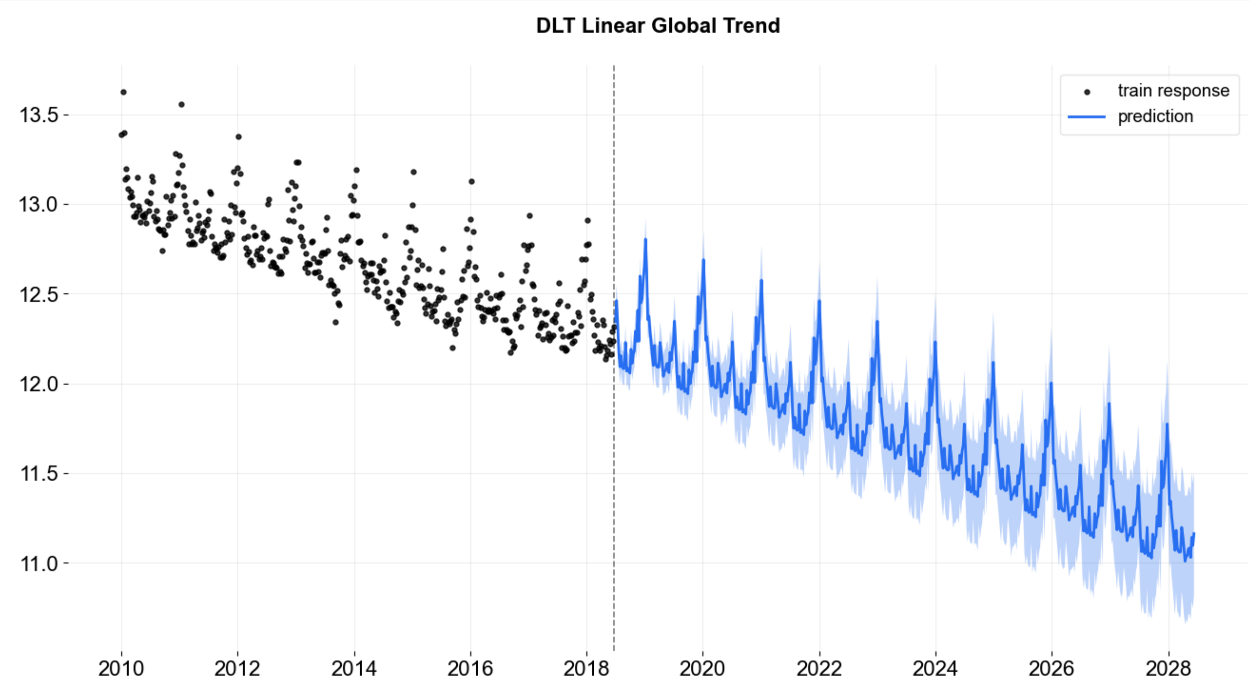
where $\delta_{\text{intercept}}$ and δ_{slope} are fitted parameters and t is rescaled time-step between 0 and T (=number of time steps).

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

[4]:
```
%%time

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')
```

```
2024-03-19 23:38:05 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
```




```
CPU times: user 1.38 s, sys: 538 ms, total: 1.92 s
Wall time: 501 ms
```

[5]:
```
dlt = DLT(
    response_col=response_col,
    date_col=date_col,
    estimator='stan-mcmc',
    seasonality=52,
    seed=8888,
```

(continues on next page)

(continued from previous page)

```
    global_trend_option='linear',
    # for prediction uncertainty
    n_bootstrap_draws=1000,
    stan_mcmc_args={'show_progress': False},
)

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

```
2024-03-19 23:38:05 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 1.000, warmups (per chain): 225 and samples(per chain): 25.
```

[5]:
```
<orbit.forecaster.full_bayes.FullBayesianForecaster at 0x2a9230650>
```

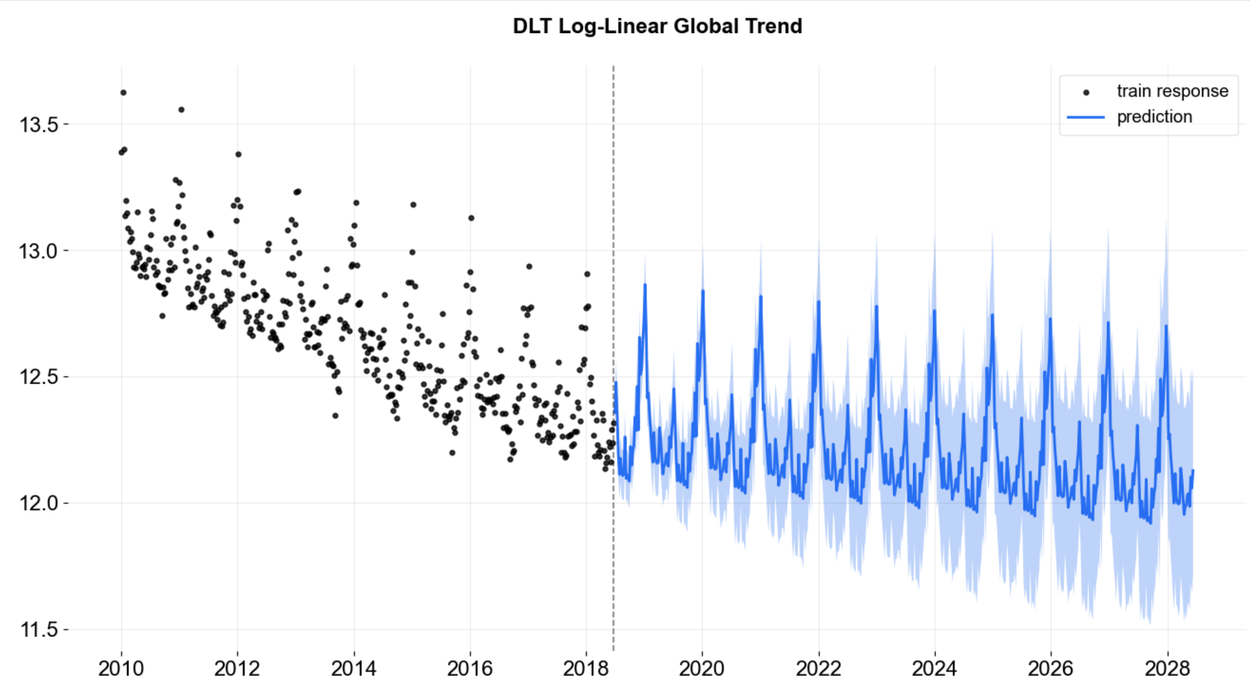
One can use `.get_posterior_samples()` to extract the samples for all sampling parameters.

[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')
```

```
2024-03-19 23:38:08 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
```




```
CPU times: user 1.41 s, sys: 456 ms, total: 1.87 s
Wall time: 540 ms
```

In logistic trend, users need to specify the args `global_floor` and `global_cap`. These args are with default `0` and 1.

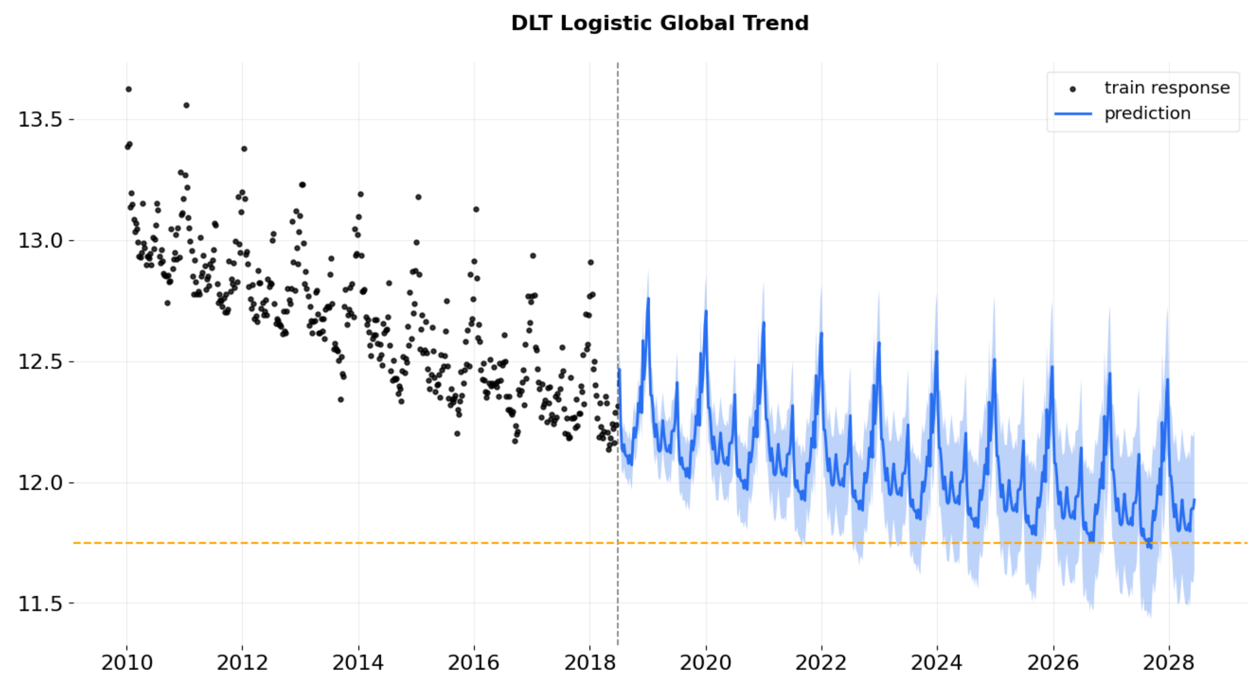
7.2.2 logistic global trend

[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
```

```
2024-03-19 23:38:08 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
```

```
CPU times: user 446 ms, sys: 263 ms, total: 709 ms
Wall time: 419 ms
```

[8]:



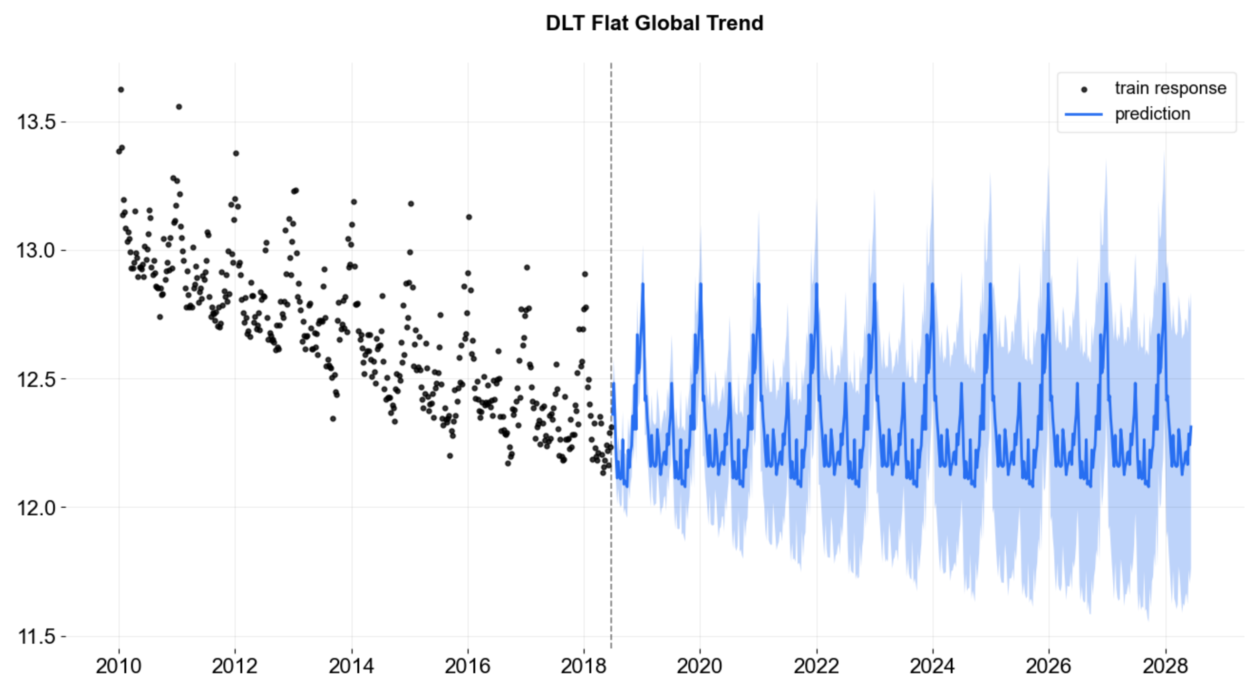
Note: Theoretically, the trend is bounded by the `global_floor` and `global_cap`. However, because of seasonality and regression, the predictions can still be slightly lower than the floor or higher than the cap.

7.2.3 flat trend

[9]:

```
%%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')
```

```
2024-03-19 23:38:09 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
```




```
CPU times: user 1.33 s, sys: 264 ms, total: 1.59 s
Wall time: 681 ms
```

7.3 Regression

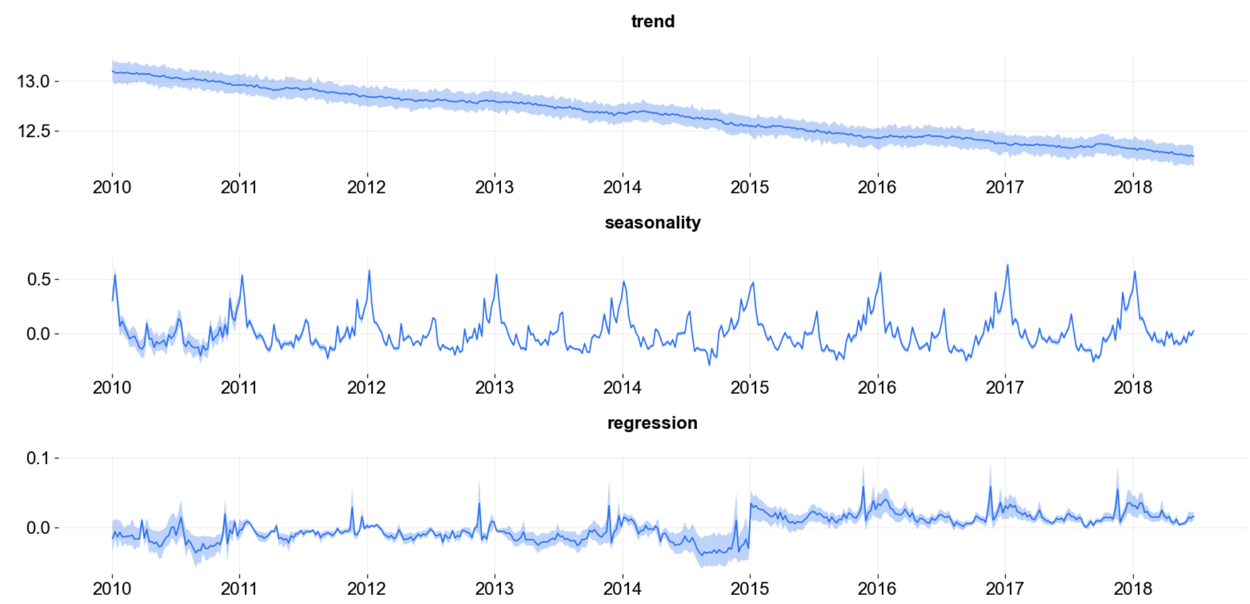
You can also add regressors into the model by specifying `regressor_col`. This serves the purpose of nowcasting or forecasting when exogenous regressors are known such as events and holidays. Without losing generality, the interface is set to be

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

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,

[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);
```

```
2024-03-19 23:38:09 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 1.000, warmups (per chain): 225 and samples(per chain): 25.
```



One can also use `.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.

[11]:
```
dlt.get_regression_coefs()
```

[11]:
```
        regressor regressor_sign  coefficient  coefficient_lower  \
0  trend.unemploy        Regular     0.052448           0.020843
1   trend.filling        Regular     0.084483           0.012782

   coefficient_upper  Pr(coef >= 0)  Pr(coef < 0)
0           0.076498           1.00          0.00
1           0.139546           0.99          0.01
```

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 `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 `examples/` folder for the details.

CHAPTER

EIGHT

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__)
```

```
1.1.4.6
```

8.1 Model Structure

$$
\begin{aligned}
y_t &= \mu_t + s_t + \epsilon_t \\
\mu_t &= l_{t-1} + \xi_1 b_{t-1} + \xi_2 l_{t-1}^{\lambda} \\
\epsilon_t &\sim \mathtt{Student}(\nu, 0, \sigma) \\
\sigma &\sim \mathtt{HalfCauchy}(0, \gamma_0)
\end{aligned}
$$

with the update process,

$$
\begin{aligned}
l_t &= \rho_l(y_t - s_t) + (1 - \rho_l)l_{t-1} \\
b_t &= \rho_b(l_t - l_{t-1}) + (1 - \rho_b)b_{t-1} \\
s_{t+m} &= \rho_s(y_t - l_t) + (1 - \rho_s)s_t
\end{aligned}
$$

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

- local trend takes on a fraction ξ_1 rather than a damped factor
- global trend is with a auto-regrssive term ξ_2 and a power term λ

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

```
[3]: # 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`.

```
[4]: lgt = LGT(
    response_col=response_col,
    date_col=date_col,
    estimator='stan-map',
    seasonality=52,
    seed=8888,
)
```

```
2024-03-19 23:39:41 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
```

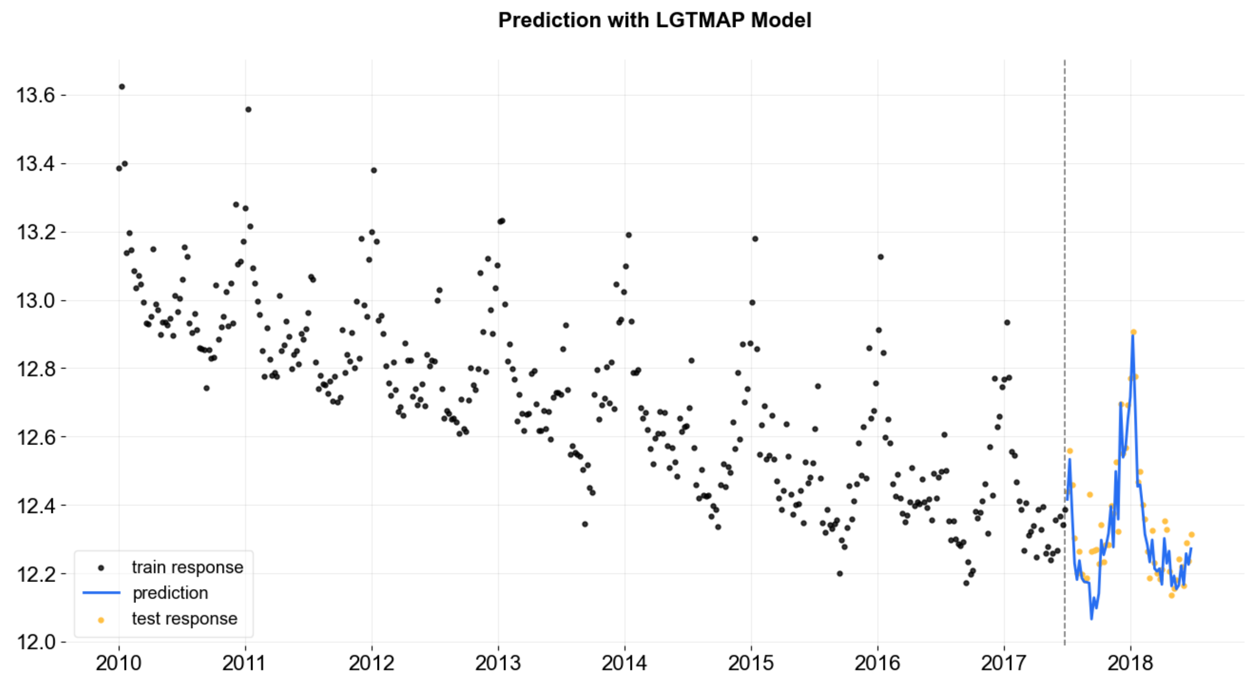
```
[5]: %%time
lgt.fit(df=train_df)
```

```
CPU times: user 9.3 ms, sys: 16.8 ms, total: 26.1 ms
Wall time: 172 ms
```

[5]:
```
<orbit.forecaster.map.MAPForecaster at 0x2919dccd0>
```

[6]:
```
predicted_df = lgt.predict(df=test_df)
```

[7]:
```
_ = 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, title='Prediction with LGTMAP Model')
```



8.2.2 LGT - MCMC

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

- By dedault, 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

[8]:
```
lgt = LGT(
    response_col=response_col,
    date_col=date_col,
    seasonality=52,
    seed=2024,
    stan_mcmc_args={'show_progress': False},
)
```

[9]:
```
%%time
lgt.fit(df=train_df)
```

```
2024-03-19 23:39:42 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 1.000, warmups (per chain): 225 and samples(per chain): 25.
```

```
CPU times: user 87.1 ms, sys: 36.9 ms, total: 124 ms
Wall time: 5.13 s
```

[9]:
```
<orbit.forecaster.full_bayes.FullBayesianForecaster at 0x2a4bc4090>
```

[10]:
```
predicted_df = lgt.predict(df=test_df)
```

[11]:
```
predicted_df.tail(5)
```

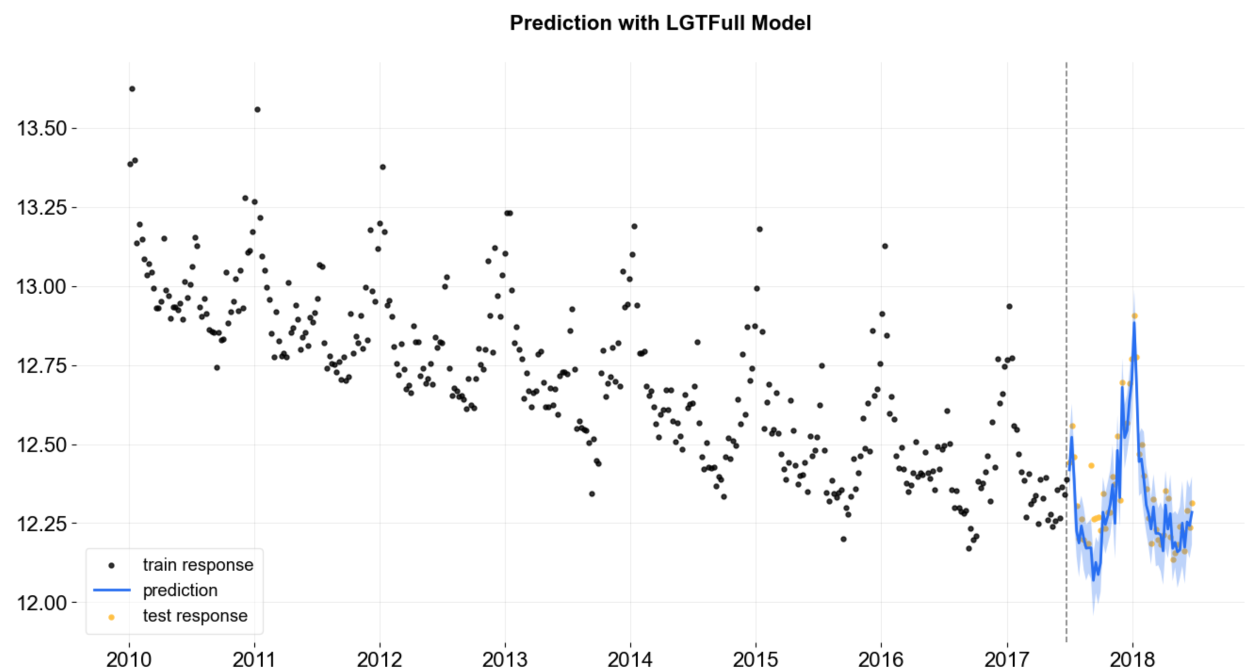
[11]:

	week	prediction_5	prediction	prediction_95
47	2018-05-27	12.114602	12.250131	12.382320
48	2018-06-03	12.058250	12.173431	12.272940
49	2018-06-10	12.164898	12.253941	12.387880
50	2018-06-17	12.138711	12.241891	12.362063
51	2018-06-24	12.182641	12.284261	12.397172

[12]:
```
lgt.get_posterior_samples().keys()
```

[12]:
```
dict_keys(['l', 'b', 'lev_sm', 'slp_sm', 'obs_sigma', 'nu', 'lgt_sum', 'gt_pow', 'lt_coef
↪', 'gt_coef', 's', 'sea_sm', 'loglk'])
```

[13]:
```
_ = 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

[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')
```

```
2024-03-19 23:39:47 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 1.000, warmups (per chain): 225 and samples(per chain): 25.
```

```
CPU times: user 97.2 ms, sys: 41.7 ms, total: 139 ms
Wall time: 4.64 s
```

[15]:
```
<orbit.forecaster.full_bayes.FullBayesianForecaster at 0x2a4d2add0>
```

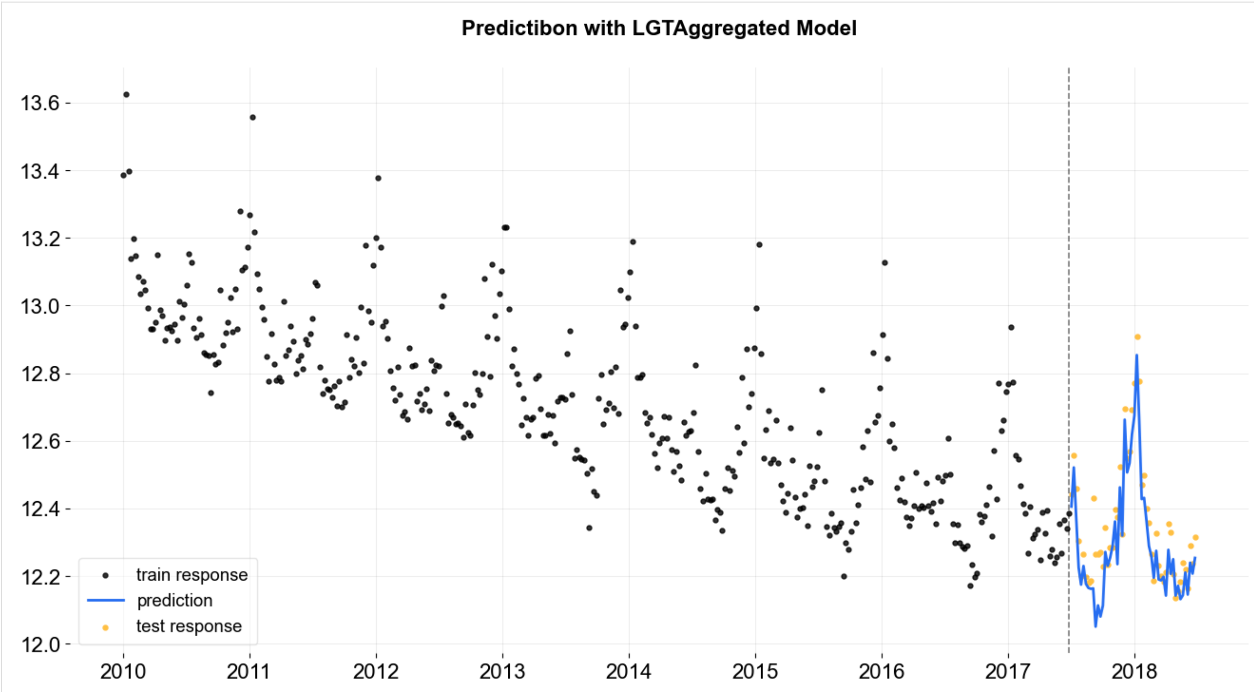
[16]:
```
predicted_df = lgt.predict(df=test_df)
```

[17]:
```
predicted_df.tail(5)
```

[17]:

	week	prediction
47	2018-05-27	12.210257
48	2018-06-03	12.145213
49	2018-06-10	12.239412
50	2018-06-17	12.207138
51	2018-06-24	12.253422

[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='Predictibon with LGTAggregated Model')
```


Predictibon with LGTAggregated Model
13.6
13.4
13.2
13.0
12.8
12.6
12.4
12.2
12.0
2010
2011
2012
2013
2014
2015
2016
2017
2018
train response
prediction
test response

CHAPTER

NINE

REGRESSION PRIORS IN DLT

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.

[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__)
```

```
1.1.4.6
```

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.

[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
```

[3]:
```
week               datetime64[ns]
claims                    float64
trend.unemploy            float64
trend.job                 float64
sp500                     float64
vix                       float64
dtype: object
```

[4]:
```
df.head(5)
```

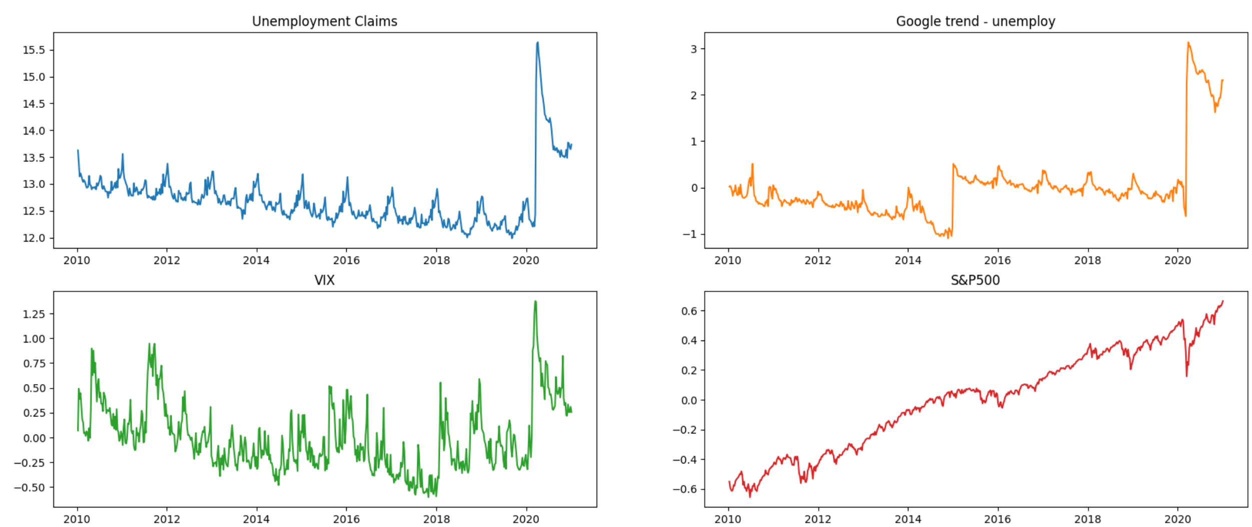
[4]:

	week	claims	trend.unemploy	trend.job	sp500	vix
0	2010-01-10	13.624218	0.016351	0.181862	-0.550891	0.069878
1	2010-01-17	13.398741	0.032611	0.130569	-0.590640	0.491772
2	2010-01-24	13.137549	-0.000179	0.119987	-0.607162	0.388078
3	2010-01-31	13.196760	-0.069172	0.087552	-0.614339	0.446838
4	2010-02-07	13.146984	-0.182500	0.019344	-0.605636	0.308205

We can see form the plot below, there are seasonality, trend, and as well as a huge change point due the impact of COVID-19.

[5]:
```
fig, axs = plt.subplots(2, 2,figsize=(20,8))
axs[0, 0].plot(df['week'], df['claims'])
axs[0, 0].set_title('Unemployment Claims')
axs[0, 1].plot(df['week'], df['trend.unemploy'], 'tab:orange')
axs[0, 1].set_title('Google trend - unemploy')
axs[1, 0].plot(df['week'], df['vix'], 'tab:green')
axs[1, 0].set_title('VIX')
axs[1, 1].plot(df['week'], df['sp500'], 'tab:red')
axs[1, 1].set_title('S&P500')
```

[5]:
```
Text(0.5, 1.0, 'S&P500')
```



[6]:
```
# using relatively updated data
df[['sp500']] = df[['sp500']].diff()
df = df[1:].reset_index(drop=True)
```

(continues on next page)

(continued from previous page)

```
test_size = 12
train_df = df[:-test_size]
test_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)
```

```
2024-03-19 23:42:14 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 1.000, warmups (per chain): 1000 and samples(per chain): 25.
```

```
CPU times: user 126 ms, sys: 39.3 ms, total: 165 ms
Wall time: 11 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 β_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. $\beta_j \in (-\inf, \inf)$
- '+': $\beta_j \sim \mathcal{N}^+(0, \sigma_j^2)$ i.e. $\beta_j \in [0, \inf)$
- '-': $\beta_j \sim \mathcal{N}^-(0, \sigma_j^2)$ i.e. $\beta_j \in (-\inf, 0]$

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,
```

(continues on next page)

(continued from previous page)

```
    regressor_col=['trend.unemploy', 'trend.job', 'sp500', 'vix'],
    seasonality=52,
    seed=8888,
    num_warmup=4000,
    stan_mcmc_args={'show_progress': False}
)
dlt_reg.fit(df=train_df)
predicted_df_reg = dlt_reg.predict(test_df)
```

```
2024-03-19 23:42:26 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 1.000, warmups (per chain): 1000 and samples(per chain): 25.
```

```
CPU times: user 164 ms, sys: 71.3 ms, total: 235 ms
Wall time: 13.3 s
```

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

[9]:
```
dlt_reg.get_regression_coefs()
```

[9]:
```
        regressor regressor_sign  coefficient  coefficient_lower  \
0  trend.unemploy        Regular     0.076715           0.042656
1       trend.job        Regular    -0.038551          -0.081939
2           sp500        Regular    -0.001387          -0.201171
3             vix        Regular     0.011533          -0.013659

   coefficient_upper  Pr(coef >= 0)  Pr(coef < 0)
0           0.105937           1.00          0.00
1           0.019654           0.13          0.87
2           0.195193           0.49          0.51
3           0.035988           0.73          0.27
```

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 (μ, σ) 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 β_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. $\beta_j \in (-\inf, \inf)$
- '+': $\beta_j \sim \mathcal{N}^+(0, \sigma_j^2)$ i.e. $\beta_j \in [0, \inf)$
- '-': $\beta_j \sim \mathcal{N}^-(0, \sigma_j^2)$ i.e. $\beta_j \in (-\inf, 0]$

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 σ 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)
```

```
2024-03-19 23:42:39 - 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()
```

[11]:
```
        regressor regressor_sign  coefficient  coefficient_lower  \
0  trend.unemploy       Positive     0.126584           0.075630
1             vix       Positive     0.019553           0.002202
2           sp500       Negative    -0.032251          -0.087838
3       trend.job        Regular    -0.011294          -0.086100

   coefficient_upper  Pr(coef >= 0)  Pr(coef < 0)
0           0.198016          1.000         0.000
1           0.054368          1.000         0.000
2          -0.002386          0.000         1.000
3           0.058422          0.394         0.606
```

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
```

9.2 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.

CHAPTER

TEN

REGRESSION PENALTIES IN DLT

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.

[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__)
```

```
1.1.4.6
```

10.1 Regression on Simulated Dataset

A simulated dataset is used to demonstrate sparse regression.

[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

$$y_t = l_t + \sum_p^P \beta_p x_{t,p}$$

$$\text{for } t = 1, 2, \cdots, T$$

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.

[4]:
```
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)
```

[5]:
```
# combine trend and the regression
y = trend + regression
```

[6]:
```
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
```

[6]:
```
(50, 12)
```

Here is a peek on the coefficients.

[7]:
```
coefs
```

[7]:
```
array([ 0.38372743, -0.21084054,  0.5404565 , -0.21864409,  0.85529298,
       -0.83838077, -0.54550632,  0.80367924, -0.74643654, -0.26626975])
```

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

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

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

Fixed Ridge Penalty

[8]:
```
%%time
dlt_fridge = DLT(
    response_col=response_col,
    date_col=dt_col,
    regressor_col=x_cols,
    seed=SEED,
    # 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,
)
dlt_fridge.fit(df=df)
```

```
2024-03-19 23:42:19 - 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 81.8 ms, sys: 62.7 ms, total: 145 ms
Wall time: 2.65 s
```

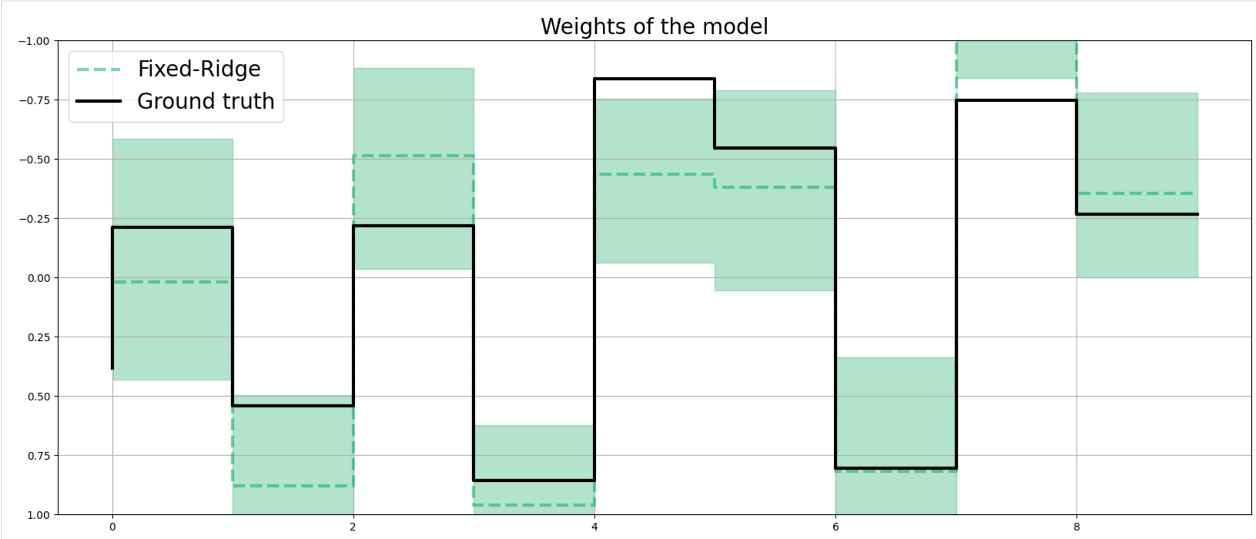
[8]:
```
<orbit.forecaster.full_bayes.FullBayesianForecaster at 0x2a317b0d0>
```

[9]:
```
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=20)
plt.plot(idx, coef_fridge[1], color=OrbitPalette.GREEN.value, linewidth=lw,  drawstyle=
↪'steps', label='Fixed-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.ylim(1, -1)
```

(continues on next page)

(continued from previous page)

```
plt.legend(prop={'size': 20})
plt.grid()
```



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 α. 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)
```

```
2024-03-19 23:42:23 - 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
```

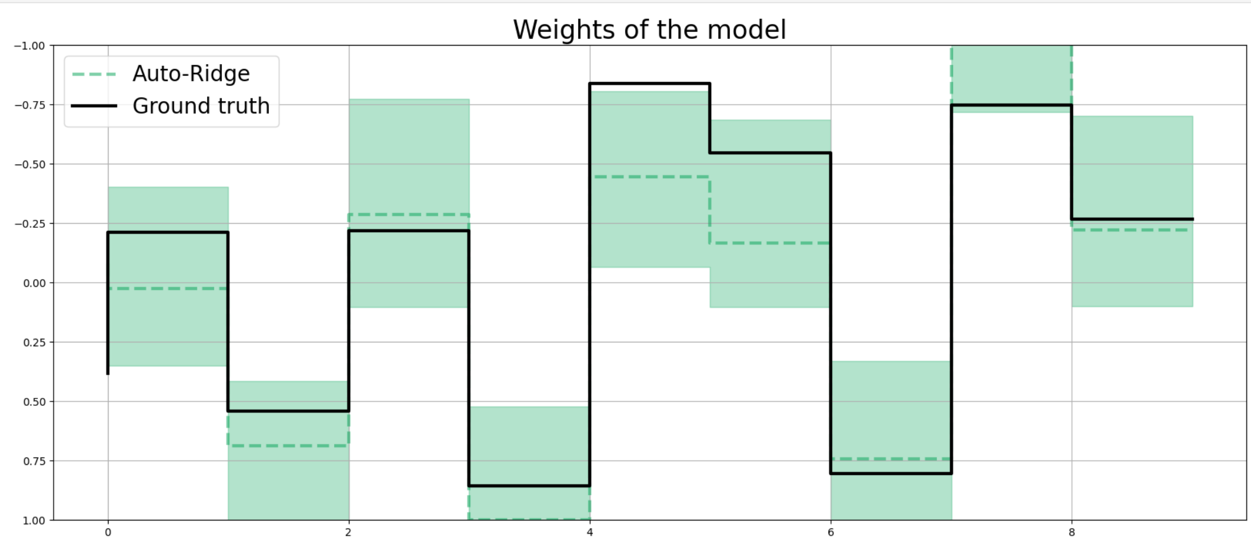
```

CPU times: user 88.3 ms, sys: 52.6 ms, total: 141 ms
Wall time: 4.08 s
```

[10]:

```
<orbit.forecaster.full_bayes.FullBayesianForecaster at 0x2a52ee510>
```

[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='--')
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
```

10.1.2 Sparse Regrssion

In reality, users usually faces 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.

[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
coefs
```

[14]:
```
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.        ,  0.32678039,  0.37436121,  0.38932392,  0.40216056,
        0.        ,  0.        , -0.3076828 , -0.35036047,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
       -0.45838674,  0.3171478 ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.41599814,  0.        , -0.30964341,
       -0.42072894,  0.36255583,  0.        , -0.39326337,  0.44455655,
        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
```

[16]:
```
(50, 52)
```

10.1.3 Fixed Ridge Penalty

[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)
```

```
2024-03-19 23:42:27 - 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
```

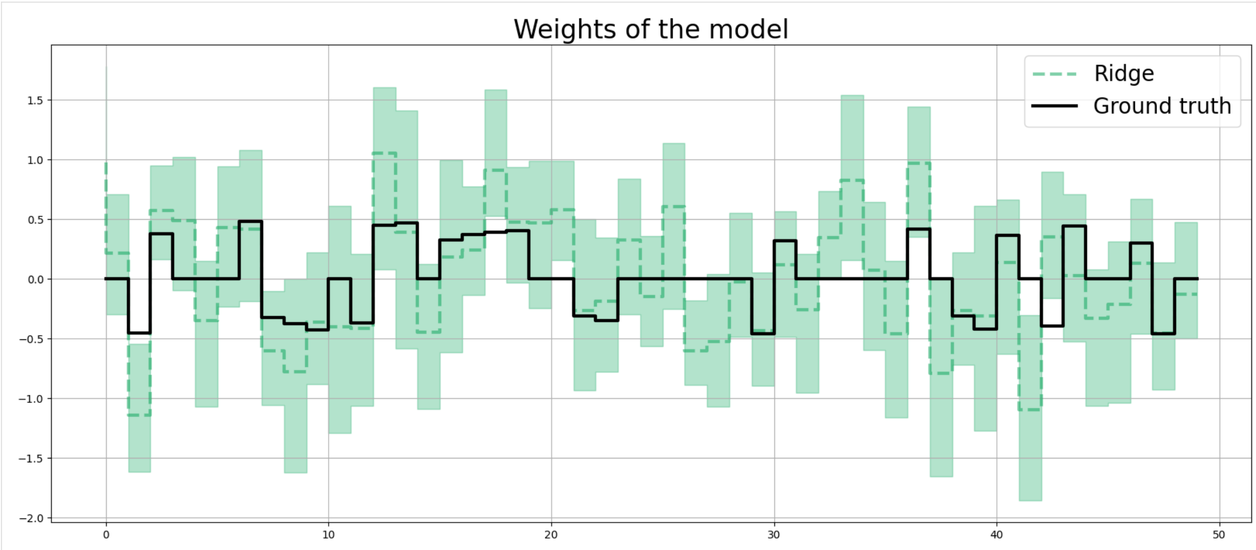
```
chain 2 |          | 00:00 Status
```

```
chain 3 |          | 00:00 Status
```

```
chain 4 |          | 00:00 Status
```

[17]:
```
<orbit.forecaster.full_bayes.FullBayesianForecaster at 0x2a6375650>
```

[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();
```



LASSO Penalty

In high P to N problems, *LASS0* penalty usually shines compared to *Ridge* penalty.

[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)
```

```
2024-03-19 23:42:37 - 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
```

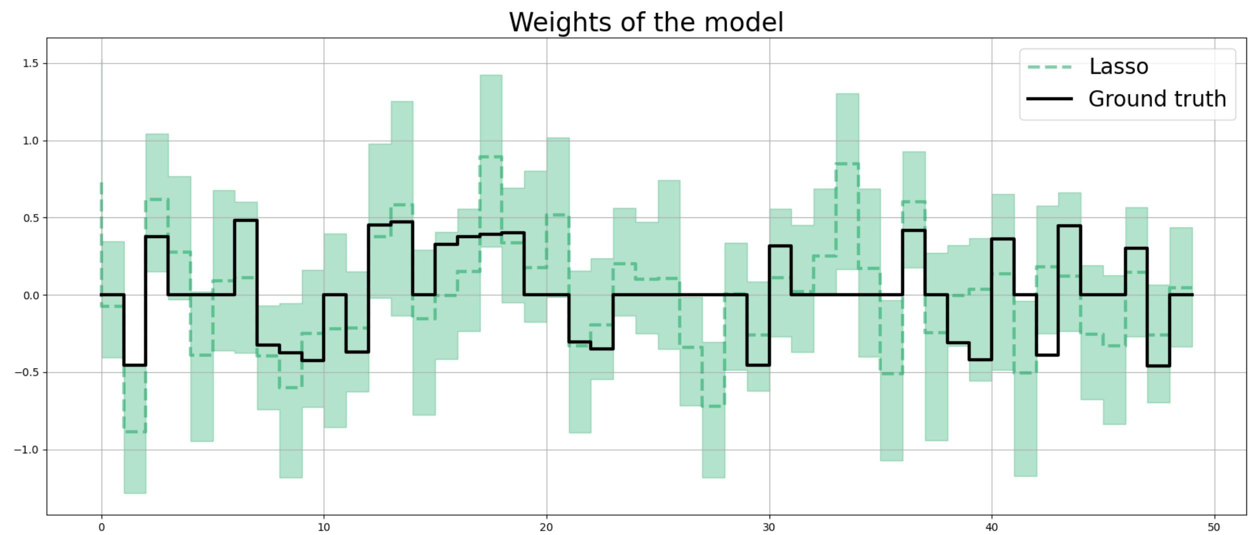
[19]:
```
<orbit.forecaster.full_bayes.FullBayesianForecaster at 0x2a63b9150>
```

[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)

(continued from previous page)

```
plt.plot(coef_lasso[1], color=OrbitPalette.GREEN.value, linewidth=lw,  drawstyle='steps',
↪ label="Lasso", alpha=0.5, linestyle='--')
plt.fill_between(idx, coef_lasso[0], coef_lasso[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();
```



[21]:
```
print('Fixed Ridge MSE:{:.3f}\nLASSO MSE:{:.3f}'.format(
    mse(coef_fridge[1], coefs), mse(coef_lasso[1], coefs)
))
```

```
Fixed Ridge MSE:0.186
LASSO MSE:0.106
```

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.

CHAPTER

ELEVEN

HANDLING MISSING RESPONSE

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.

[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]:
```
'1.1.4.6'
```

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)

(continued from previous page)

```
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  0.110445
2  0.532339
3  0.428645
4  0.487404
```

[4]:
```
test_size=52

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

11.1.1 Define Missing Data

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

[5]:
```
na_idx = np.arange(53, 100, 1)
na_idx
```

[5]:
```
array([53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
       70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86,
       87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99])
```

[6]:
```
train_df_na = train_df.copy()
train_df_na.iloc[na_idx, 1] = np.nan
```

11.2 Exponential Smoothing Examples

11.2.1 ETS

[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)
```

```
2024-03-19 23:38:16 - 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
```

```
chain 3 |          | 00:00 Status
chain 4 |          | 00:00 Status
```

11.2.2 LGT

[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)
```

```
2024-03-19 23:38:17 - 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
chain 3 |          | 00:00 Status
chain 4 |          | 00:00 Status
```

11.2.3 DLT

[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)
```

```
2024-03-19 23:38:21 - 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
chain 3 |          | 00:00 Status
chain 4 |          | 00:00 Status
```

11.2.4 Summary

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

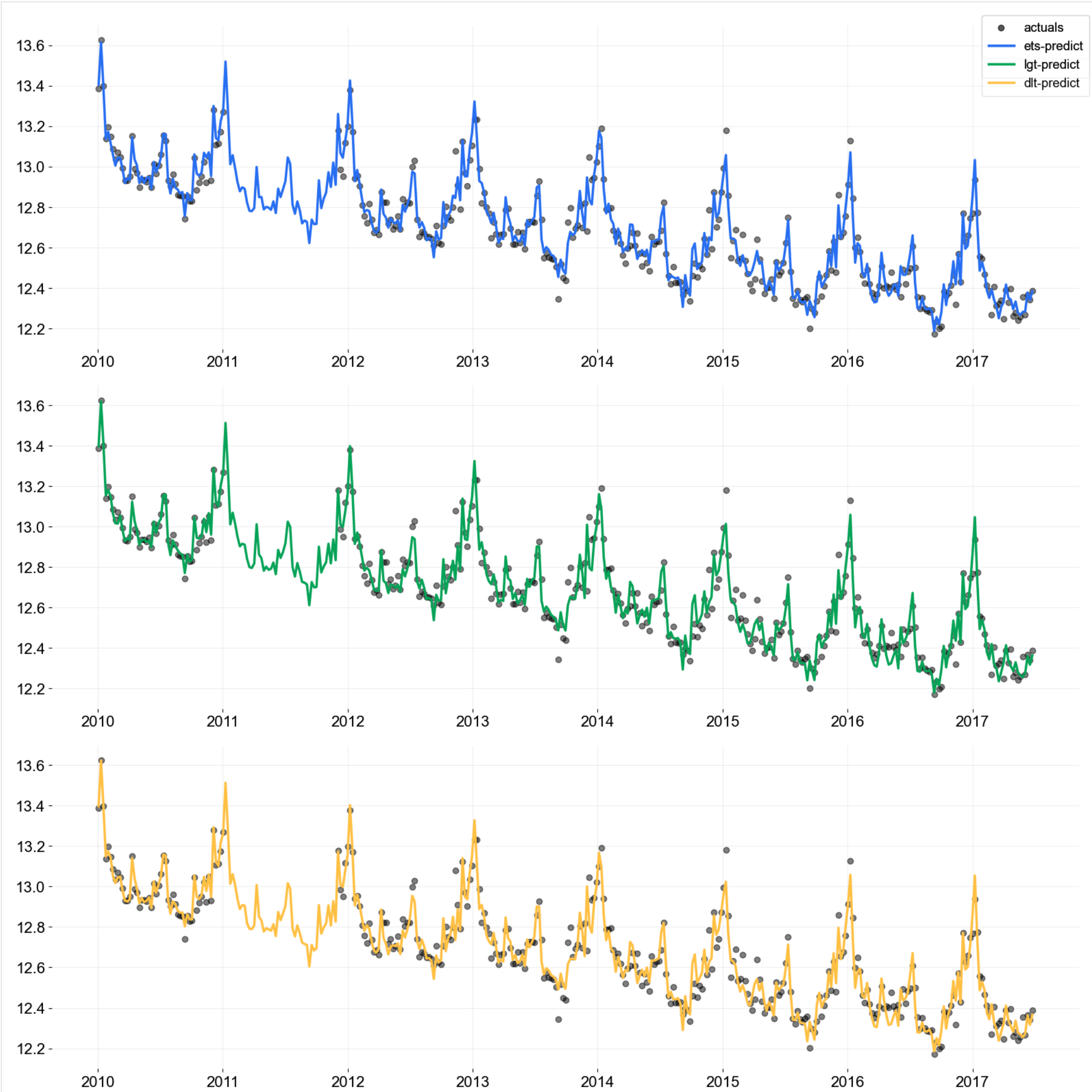
[10]:
```
train_df_na['ets-predict'] = ets_predicted['prediction']
train_df_na['lgt-predict'] = lgt_predicted['prediction']
train_df_na['dlt-predict'] = dlt_predicted['prediction']
```

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

[11]:
```
          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   0.006275 -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.**

[14]:
```
# 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)
```

CHAPTER

TWELVE

KERNEL-BASED TIME-VARYING REGRESSION - PART I

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 Smoothsing. 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]:
```
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]:
```
print(orbit.__version__)
```

```
1.1.4.6
```

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,

$$y_t = l_t + s_t + r_t + \epsilon_t,\ t = 1, \cdots, T,$$

where the ϵ_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

$$y_t = X_t^T \beta_t + \epsilon_t,\ t = 1, \cdots, T,$$

where β_t is a P-dimensional vector of coefficients that vary over time (i.e., β_i is almost certainly different from β_j for $i \neq j$) and X_t P-dimensional covariate vector (i.e., the tth row of X, the design matrix).

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

$$B = Kb^T$$

where - *coefficient matrix* B has size $T \times P$ with rows equal to the β_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 ith row and jth element can be viewed as the normalized weight $k(t_j, t) / \sum_{j=1}^{J} k(t_j, t)$

For the level/trend,

$$l_t = \beta_{t,\text{lev}}$$

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

For the seasonality,

$$B_{\text{seas}} = K_{\text{seas}} b_{\text{seas}}^T$$

$$s_t = X_{t,\text{seas}} \beta_{t,\text{seas}}$$

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.

[3]:
```
# 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()
```

```
(3288, 2)
```

[3]:
```
         date  electricity
0 2000-01-01      9.43760
1 2000-01-02      9.50130
2 2000-01-03      9.63565
3 2000-01-04      9.65392
4 2000-01-05      9.66089
```

[4]:
```
print(f'starts with {df[date_col].min()}\nends with {df[date_col].max()}\nshape: {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.

[5]:
```
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.

[6]:
```
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,
)
```

[7]:
```
ktr.fit(train_df)
```

```
2024-03-19 23:38:25 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:38:25 - orbit - INFO - Using SVI (Pyro) with steps: 301, samples: 100,␣
↪learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
/Users/towinazure/opt/miniconda3/envs/orbit311/lib/python3.11/site-packages/torch/__init_
↪_.py:696: 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.␣
```

(continues on next page)

(continued from previous page)

```
↪(Triggered internally at /Users/runner/work/pytorch/pytorch/pytorch/torch/csrc/tensor/
↪python_tensor.cpp:453.)
  _C._set_default_tensor_type(t)
2024-03-19 23:38:26 - orbit - INFO - step    0 loss = -1946.6, scale = 0.093118
INFO:orbit:step    0 loss = -1946.6, scale = 0.093118
2024-03-19 23:38:29 - orbit - INFO - step  100 loss = -3131.7, scale = 0.01002
INFO:orbit:step  100 loss = -3131.7, scale = 0.01002
2024-03-19 23:38:32 - orbit - INFO - step  200 loss = -3132, scale = 0.010005
INFO:orbit:step  200 loss = -3132, scale = 0.010005
2024-03-19 23:38:35 - orbit - INFO - step  300 loss = -3121.6, scale = 0.0098163
INFO:orbit:step  300 loss = -3121.6, scale = 0.0098163
```

[7]:
```
<orbit.forecaster.svi.SVIForecaster at 0x15e870c90>
```

We can take a look how the level is fitted with the data.

[8]:
```python
predicted_df = ktr.predict(df=df)
predicted_df.head()
```

[8]:

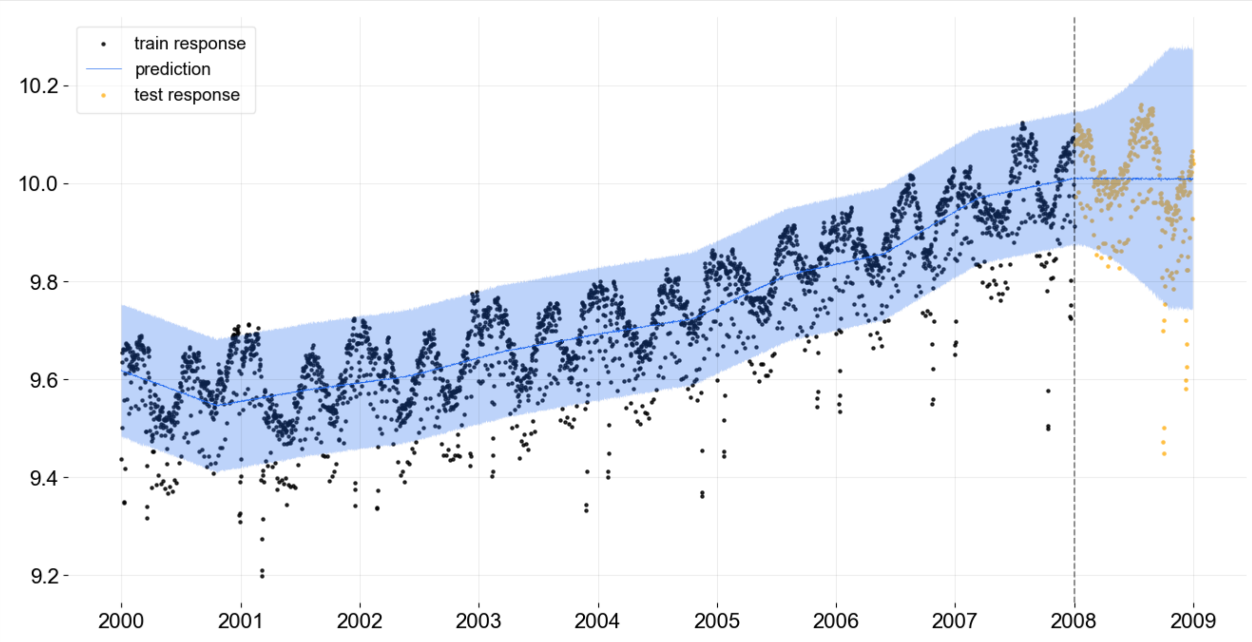
	date	prediction_5	prediction	prediction_95
0	2000-01-01	9.48516	9.61908	9.75428
1	2000-01-02	9.48116	9.61605	9.75010
2	2000-01-03	9.48156	9.61657	9.74966
3	2000-01-04	9.48273	9.61614	9.75022
4	2000-01-05	9.47921	9.61620	9.75183

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

[9]:
```python
ktr.get_posterior_samples().keys()
```

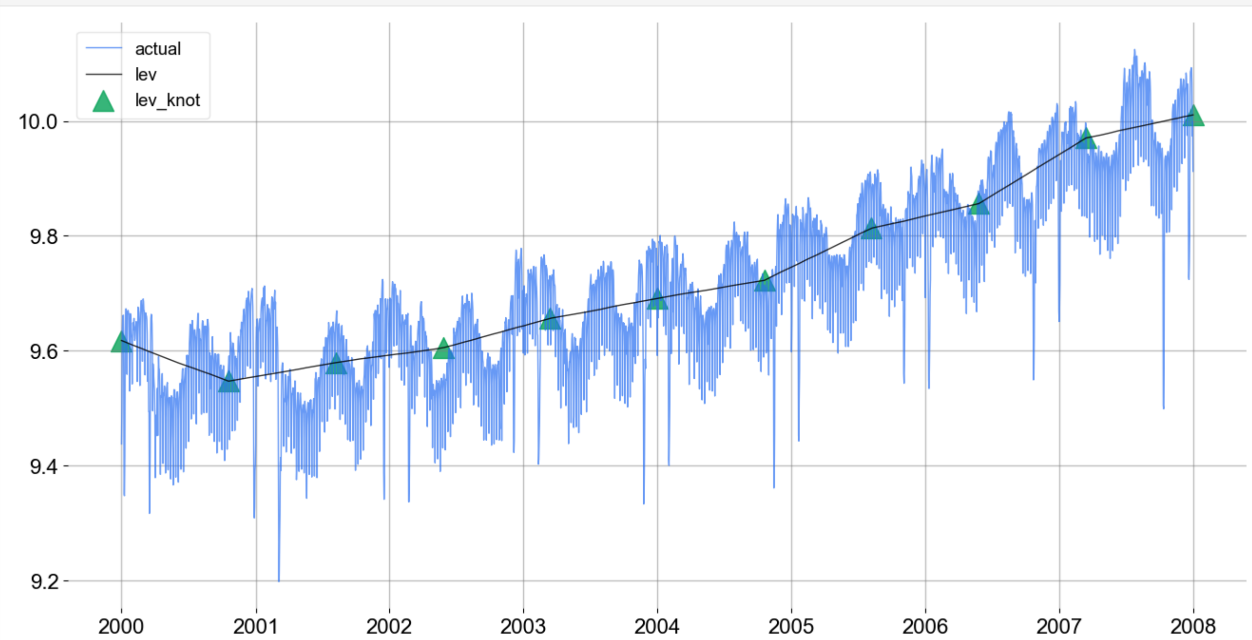
[9]:
```
dict_keys(['lev_knot', 'lev', 'yhat', 'obs_scale'])
```

[10]:
```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.

[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,
)
```

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

```
2024-03-19 23:38:39 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:38:40 - 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.
2024-03-19 23:38:40 - orbit - INFO - step    0 loss = -2190.8, scale = 0.093667
INFO:orbit:step    0 loss = -2190.8, scale = 0.093667
2024-03-19 23:38:42 - orbit - INFO - step  100 loss = -4356.8, scale = 0.0069845
INFO:orbit:step  100 loss = -4356.8, scale = 0.0069845
2024-03-19 23:38:45 - orbit - INFO - step  200 loss = -4301.3, scale = 0.0071019
INFO:orbit:step  200 loss = -4301.3, scale = 0.0071019
2024-03-19 23:38:47 - orbit - INFO - step  300 loss = -4362, scale = 0.0072349
INFO:orbit:step  300 loss = -4362, scale = 0.0072349
```

[13]:
```
<orbit.forecaster.svi.SVIForecaster at 0x2b23a3550>
```

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

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

[15]:
```
        date  prediction_5  prediction  prediction_95  trend_5   trend  \
0 2000-01-01       9.53352     9.61277        9.69350  9.50462 9.58387
1 2000-01-02       9.48100     9.56119        9.64121  9.50211 9.58229
2 2000-01-03       9.54230     9.62288        9.70269  9.50224 9.58282
3 2000-01-04       9.60186     9.68151        9.76075  9.50299 9.58265
4 2000-01-05       9.58472     9.66577        9.74660  9.50168 9.58273

   trend_95  regression_5  regression  regression_95  seasonality_7_5  \
0   9.66460       0.00000     0.00000        0.00000         -0.02784
1   9.66232       0.00000     0.00000        0.00000         -0.07872
2   9.66263       0.00000     0.00000        0.00000         -0.01838
```

(continues on next page)

(continued from previous page)

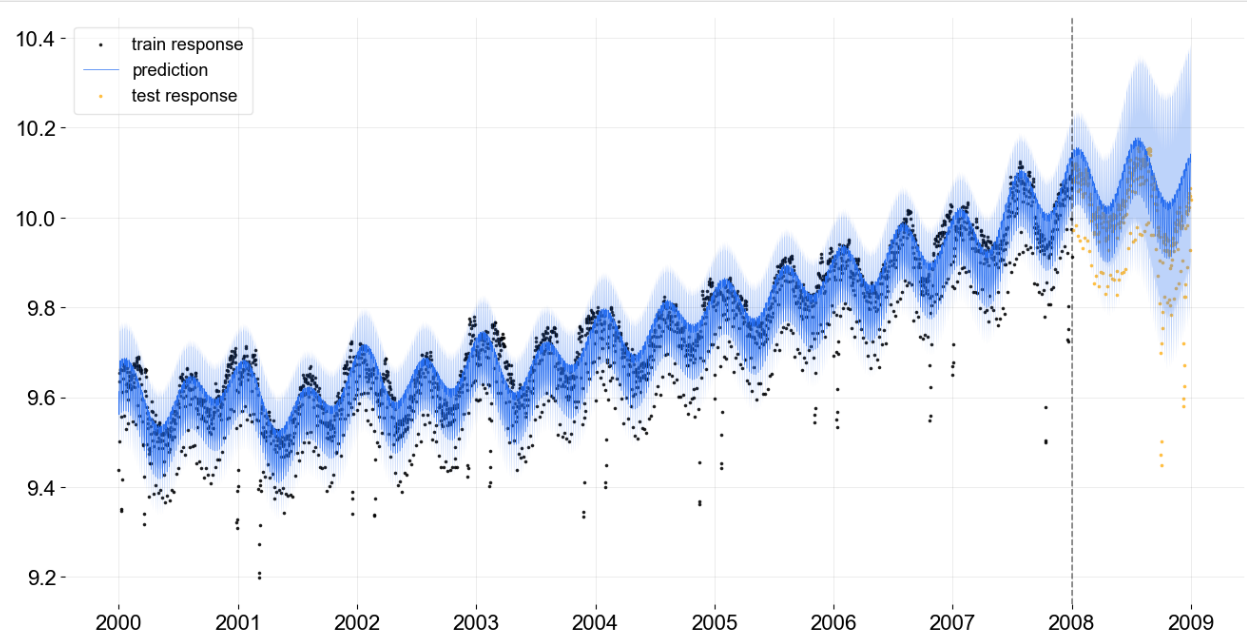
```
3   9.66188      0.00000     0.00000        0.00000          0.03962
4   9.66356      0.00000     0.00000        0.00000          0.02304

   seasonality_7  seasonality_7_95  seasonality_365.25_5  seasonality_365.25  \
0       -0.02784          -0.02784               0.05674             0.05674
1       -0.07872          -0.07872               0.05761             0.05761
2       -0.01838          -0.01838               0.05845             0.05845
3        0.03962           0.03962               0.05924             0.05924
4        0.02304           0.02304               0.05999             0.05999

   seasonality_365.25_95
0                0.05674
1                0.05761
2                0.05845
3                0.05924
4                0.05999
```

Tips: there is an additional arg `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 $<$ seasonality since each order represents adding a pair of sine and cosine regressors.

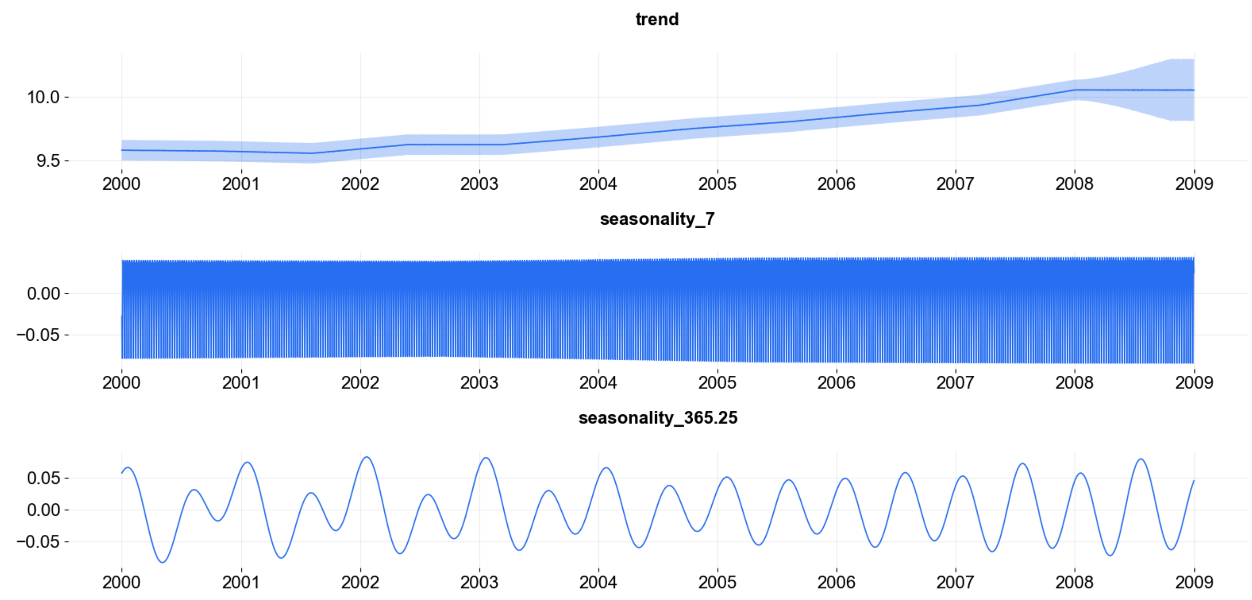
12.5 More Diagnostic and Visualization

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

```
[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)
```



[17]:
```
_ = plot_predicted_components(predicted_df=predicted_df, date_col=date_col, plot_
↪components=['trend', 'seasonality_7', 'seasonality_365.25'])
```



12.6 References

1. Ng, Wang and Dai (2021) Bayesian Time Varying Coefficient Model with Applications to Marketing Mix Modeling, arXiv preprint arXiv:2106.03322
2. Hastie, Trevor and Tibshirani, Robert. (1990), Generalized Additive Models, New York: Chapman and Hall.
3. Wood, S. N. (2006), Generalized Additive Models: an introduction with R, Boca Raton: Chapman & Hall/CRC
4. Harvey, C. A. (1989). Forecasting, Structural Time Series and the Kalman Filter, Cambridge University Press.
5. Durbin, J., Koopman, S. J.. (2001). Time Series Analysis by State Space Methods, Oxford Statistical Science Series

CHAPTER

THIRTEEN

KERNEL-BASED TIME-VARYING REGRESSION - PART II

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__)
```

```
1.1.4.6
```

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

$$y_t = l_t + s_t + r_t + \epsilon_t, \; t = 1, \cdots, T,$$

where

- ϵ_ts comprise a stationary random error process.
- r_t is the regression component which can be further expressed as $\sum_{i=1}^{I} x_{i,t}\beta_{i,t}$ with covariate x and coefficient β 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 β_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 ith row and jth 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_reg as such:

$k_\text{reg}(t, t_j; \rho) = \exp(-\frac{(t-t_j)^2}{2\rho^2}),$

where ρ 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]:
```
def sim_data_seasonal(n, RS):
    """ coefficients curve are sine-cosine like
    """
    np.random.seed(RS)
    # make the time varing 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)

(continued from previous page)

```
        'beta3': 1.01 + np.sin(4*pi*(tau-1/8)),
        'x1': np.random.normal(0, 10, size=n),
        'x2': np.random.normal(0, 10, size=n),
        'x3': np.random.normal(0, 10, size=n),
        'trend': np.cumsum(np.concatenate((np.array([1]), np.random.normal(0, 0.1, n-
↪1)))),
        'error': np.random.normal(0, 1, size=n) #stats.t.rvs(30, size=n),#
    })

    data['y'] = data.x1 * data.beta1 + data.x2 * data.beta2 + data.x3 * data.beta3 +␣
↪data.error
    return data
```

[4]:
```
def sim_data_rw(n, RS, p=3):
    """ coefficients curve are random walk like
    """
    np.random.seed(RS)

    # initializing coefficients at zeros, simulate all coefficient values
    lev = np.cumsum(np.concatenate((np.array([5.0]), np.random.normal(0, 0.01, n-1))))
    beta = np.concatenate(
        [np.random.uniform(0.05, 0.12, size=(1,p)),
         np.random.normal(0.0, 0.01, size=(n-1,p))],
            axis=0)
    beta = np.cumsum(beta, 0)

    # simulate regressors
    covariates = np.random.normal(0, 10, (n, p))

    # observation with noise
    y = lev + (covariates * beta).sum(-1) + 0.3 * np.random.normal(0, 1, n)

    regressor_col = ['x{}'.format(pp) for pp in range(1, p+1)]
    data = pd.DataFrame(covariates, columns=regressor_col)
    beta_col = ['beta{}'.format(pp) for pp in range(1, p+1)]
    beta_data = pd.DataFrame(beta, columns=beta_col)
    data = pd.concat([data, beta_data], axis=1)

    data['y'] = y
    data['date'] = pd.date_range(start='1/1/2018', periods=len(y))

    return data
```

[5]:
```
rw_data = sim_data_rw(n=300, RS=2021, p=3)
rw_data.head(10)
```

[5]:

	x1	x2	x3	beta1	beta2	beta3	y	date
0	14.02970	-2.55469	4.93759	0.07288	0.06251	0.09662	6.11704	2018-01-01
1	6.23970	0.57014	-6.99700	0.06669	0.05440	0.10476	5.35784	2018-01-02
2	9.91810	-6.68728	-3.68957	0.06755	0.04487	0.11624	4.82567	2018-01-03
3	-1.17724	8.88090	-16.02765	0.05849	0.04305	0.12294	3.63605	2018-01-04
4	11.61065	1.95306	0.19901	0.06604	0.03281	0.11897	5.85913	2018-01-05

(continues on next page)

(continued from previous page)

```
5   7.31929  3.36017  -6.09933 0.07825 0.03448 0.10836 5.08805 2018-01-06
6   0.53405  8.80412  -1.83692 0.07467 0.01847 0.10507 4.59303 2018-01-07
7 -16.03947  0.27562 -22.00964 0.06887 0.00865 0.10749 1.26651 2018-01-08
8 -17.72238  2.65195   0.22571 0.07007 0.01008 0.10432 4.10629 2018-01-09
9  -7.39895 -7.63162   3.25535 0.07715 0.01498 0.09356 4.30788 2018-01-10
```

13.2.2 Sine-Cosine Like Simulated Dataset

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

[6]:
```
      tau       date   beta1   beta2   beta3       x1        x2        x3  \
0 0.01250 2018-01-01 0.02500 1.08846 0.02231 14.88609   1.56556 -14.69399
1 0.02500 2018-01-02 0.05000 1.16643 0.05894  6.76011  -0.56861   4.93157
2 0.03750 2018-01-03 0.07500 1.24345 0.11899 -4.18451  -5.38234 -13.90578
3 0.05000 2018-01-04 0.10000 1.31902 0.20098 -8.06521   9.01387  -0.75244
4 0.06250 2018-01-05 0.12500 1.39268 0.30289  5.55876   2.24944  -2.53510
5 0.07500 2018-01-06 0.15000 1.46399 0.42221 -7.05504  12.77788  14.25841
6 0.08750 2018-01-07 0.17500 1.53250 0.55601 11.30858   6.29269   7.82098
7 0.10000 2018-01-08 0.20000 1.59779 0.70098  6.45002   3.61891  16.28098
8 0.11250 2018-01-09 0.22500 1.65945 0.85357  1.06414  36.38726   8.80457
9 0.12500 2018-01-10 0.25000 1.71711 1.01000  4.22155 -12.01221   8.43176

    trend    error         y
0 1.00000 -0.73476   1.01359
1 1.07746 -0.97007  -1.00463
2 1.19201 -0.13891  -8.80009
3 1.22883  0.66550  11.59721
4 1.31341 -1.58259   1.47715
5 1.25911 -0.98049  22.68806
6 1.23484 -0.53751  15.43357
7 1.13237 -1.32858  17.15636
8 1.02834  0.87859  69.01607
9 1.00649 -0.22055 -11.27534
```

13.3 Fitting a Model with Regressors

The metadata for simulated data sets.

[7]:
```
# num of predictors
p = 3
regressor_col = ['x{}'.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 `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.

[8]:
```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',
)
```

Here `predict` has the additional argument `decompose=True`. This returns the compponents (l_t, s_t, and r_t) of the regression along with the prediction.

[9]:
```python
ktr.fit(df=rw_data)
ktr.predict(df=rw_data, decompose=True).head(5)
```

```
2024-03-19 23:38:29 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:38:29 - orbit - INFO - Using SVI (Pyro) with steps: 301, samples: 100,␣
↪learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
/Users/towinazure/opt/miniconda3/envs/orbit311/lib/python3.11/site-packages/torch/__init_
↪_.py:696: 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)
2024-03-19 23:38:30 - orbit - INFO - step    0 loss = 3107.8, scale = 0.091353
INFO:orbit:step    0 loss = 3107.8, scale = 0.091353
2024-03-19 23:38:31 - orbit - INFO - step  100 loss = 307.18, scale = 0.04889
INFO:orbit:step  100 loss = 307.18, scale = 0.04889
2024-03-19 23:38:32 - orbit - INFO - step  200 loss = 299.24, scale = 0.052646
INFO:orbit:step  200 loss = 299.24, scale = 0.052646
2024-03-19 23:38:33 - orbit - INFO - step  300 loss = 314.51, scale = 0.05106
INFO:orbit:step  300 loss = 314.51, scale = 0.05106
```

[9]:
```
        date  prediction_2.5  prediction  prediction_97.5  trend_2.5   trend  \
0 2018-01-01         5.18593     6.31800          7.44407    4.01896 5.17107
1 2018-01-02         3.28170     4.32130          5.31726    4.10800 5.12472
2 2018-01-03         3.39199     4.60176          5.90753    4.06752 5.24010
3 2018-01-04         2.05339     3.21789          4.37279    3.99131 5.11851
4 2018-01-05         4.73718     5.65588          6.60197    4.14160 5.13381

   trend_97.5  regression_2.5  regression  regression_97.5
0     6.39624         0.83381     1.13830          1.45459
1     6.16573        -1.01967    -0.81734         -0.55723
2     6.53175        -0.91607    -0.61901         -0.25905
3     6.26958        -2.30892    -1.89482         -1.38973
4     6.08087         0.31028     0.55018          0.78230
```

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).

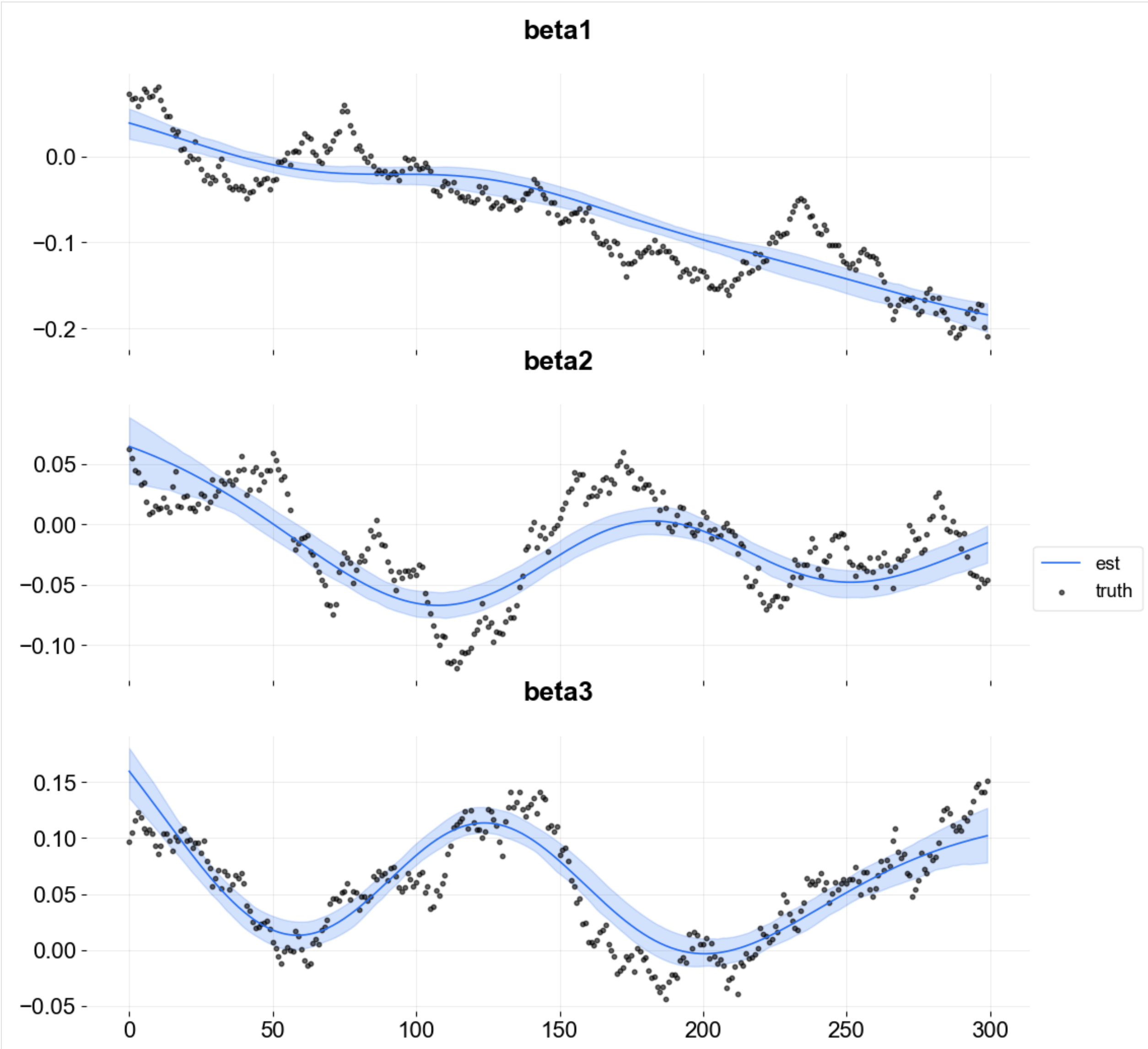
[10]:
```
coef_mid, coef_lower, coef_upper  = ktr.get_regression_coefs(include_ci=True)
```

[11]:
```
coef_mid.head(5)
```

[11]:
```
        date      x1      x2      x3
0 2018-01-01 0.03861 0.06444 0.15967
1 2018-01-02 0.03766 0.06358 0.15637
2 2018-01-03 0.03670 0.06271 0.15305
3 2018-01-04 0.03573 0.06182 0.14970
4 2018-01-05 0.03475 0.06092 0.14633
```

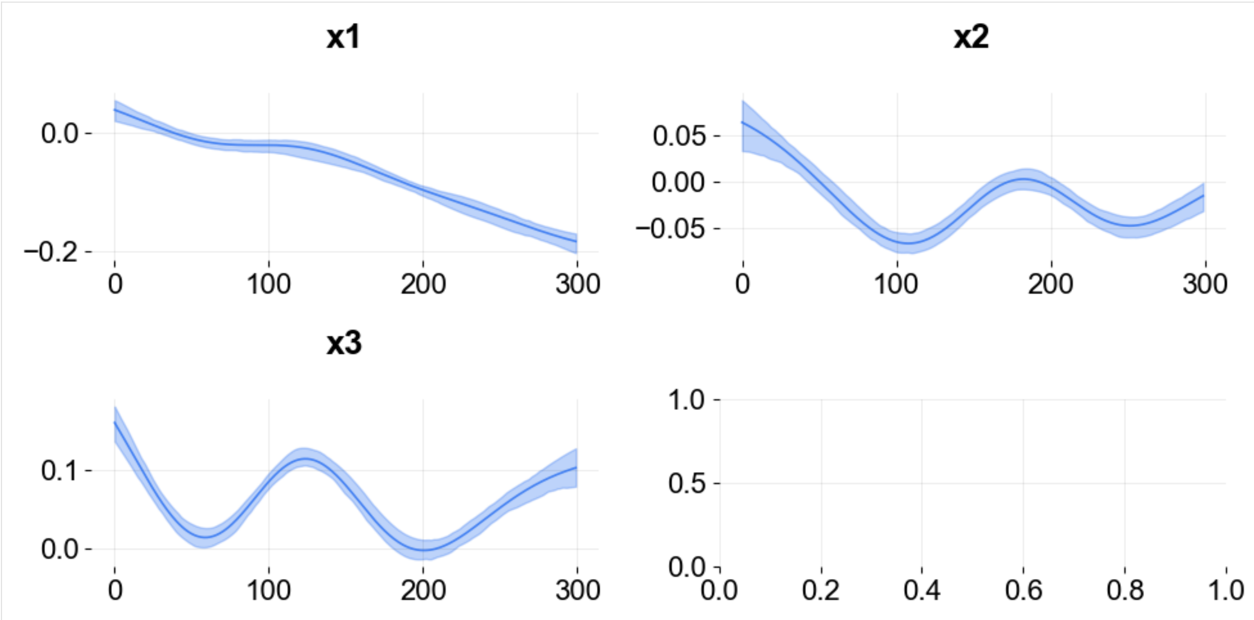
Because this is simulated data it is possible to overlay the estimate with the true coefficients.

[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.

[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** 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.

[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=[10.0] * len(regressor_col),
    regressor_knot_scale=[2.0] * len(regressor_col),
```

(continues on next page)

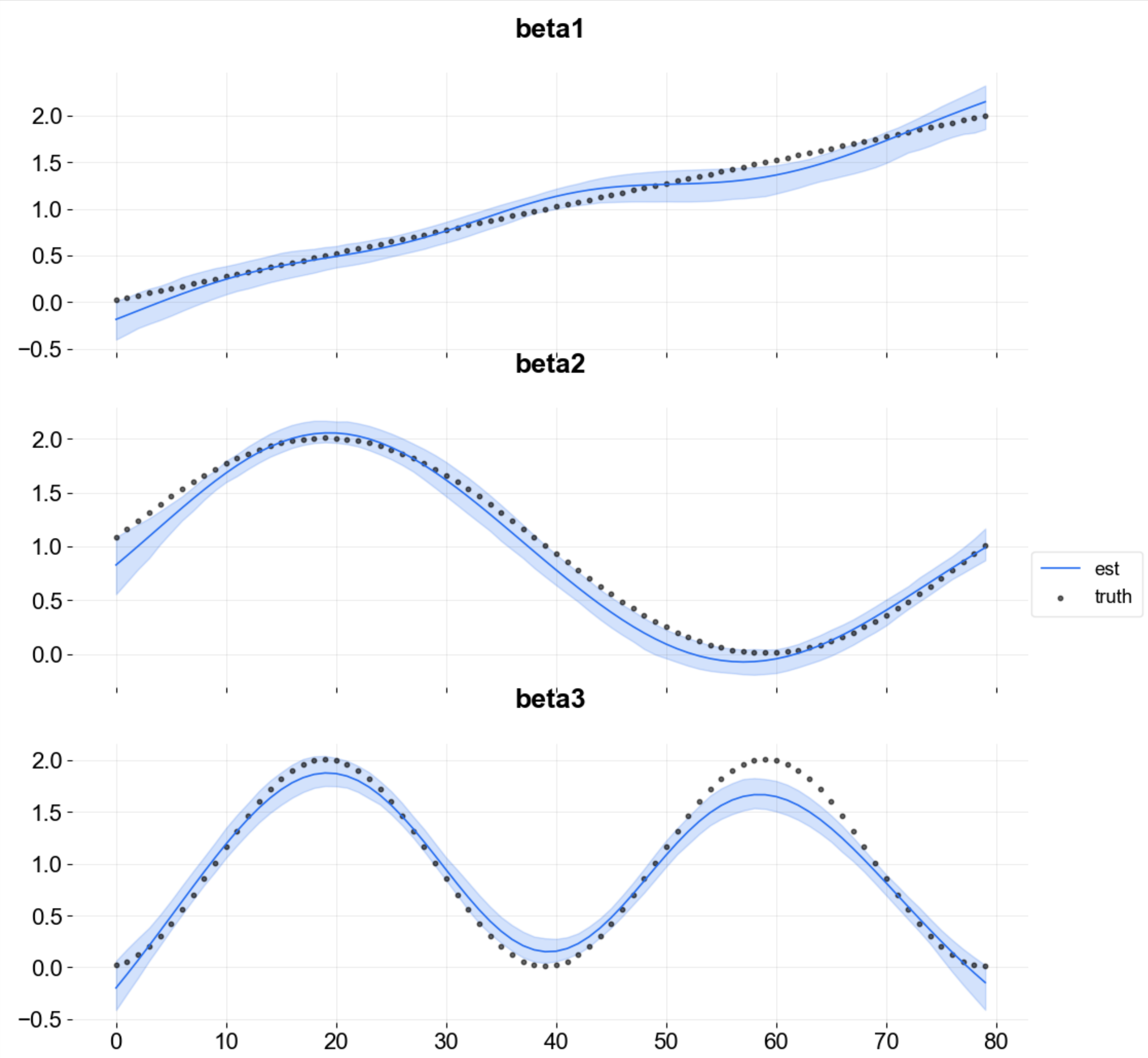
(continued from previous page)

```
    regression_segments=6,

    prediction_percentiles=[2.5, 97.5],
    seed=2021,
    estimator='pyro-svi',
)
ktr.fit(df=sc_data)
```

```
2024-03-19 23:38:33 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:38:33 - 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.
2024-03-19 23:38:33 - orbit - INFO - step    0 loss = 828.02, scale = 0.10882
INFO:orbit:step    0 loss = 828.02, scale = 0.10882
2024-03-19 23:38:34 - orbit - INFO - step  100 loss = 340.58, scale = 0.87797
INFO:orbit:step  100 loss = 340.58, scale = 0.87797
2024-03-19 23:38:35 - orbit - INFO - step  200 loss = 266.67, scale = 0.37411
INFO:orbit:step  200 loss = 266.67, scale = 0.37411
2024-03-19 23:38:36 - orbit - INFO - step  300 loss = 261.21, scale = 0.43775
INFO:orbit:step  300 loss = 261.21, scale = 0.43775
```

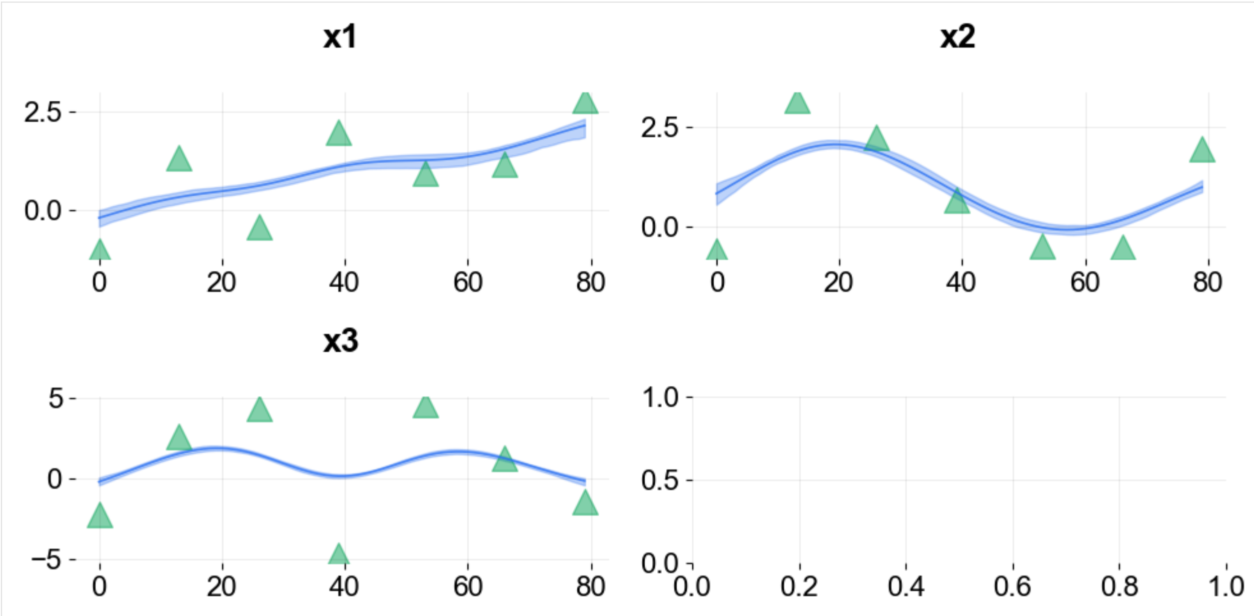
[14]:
```
<orbit.forecaster.svi.SVIForecaster at 0x2b3ab0b10>
```

[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{}'.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, sc_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));
```



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

1. Ng, Wang and Dai (2021). Bayesian Time Varying Coefficient Model with Applications to Marketing Mix Modeling, arXiv preprint arXiv:2106.03322
2. Isabel Casas and Ruben Fernandez-Casal (2021). tvReg: Time-Varying Coefficients Linear Regression for Single and Multi-Equations. https://CRAN.R-project.org/package=tvReg R package version 0.5.4.
3. Steven L Scott and Hal R Varian (2014). Predicting the present with bayesian structural time series. International Journal of Mathematical Modelling and Numerical Optimisation 5, 1-2 (2014), 4–23.

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_segements`
- `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__)
```

```
1.1.4.6
```

14.1 Fitting with iClaims Data

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

[3]:
```
# 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)
```

[3]:
```
        week    claims  trend.unemploy  trend.filling  trend.job    sp500  \
0 2010-01-03 13.38660         0.03493       -0.34414    0.12802 -0.53745
1 2010-01-10 13.62422         0.03493       -0.22053    0.17932 -0.54529
2 2010-01-17 13.39874         0.05119       -0.31817    0.12802 -0.58504
3 2010-01-24 13.13755         0.01840       -0.22053    0.11744 -0.60156
4 2010-01-31 13.19676        -0.05059       -0.26816    0.08501 -0.60874

      vix
0 0.08456
1 0.07235
2 0.49424
3 0.39055
4 0.44931
```

14.1.1 Specifying Levels Segments

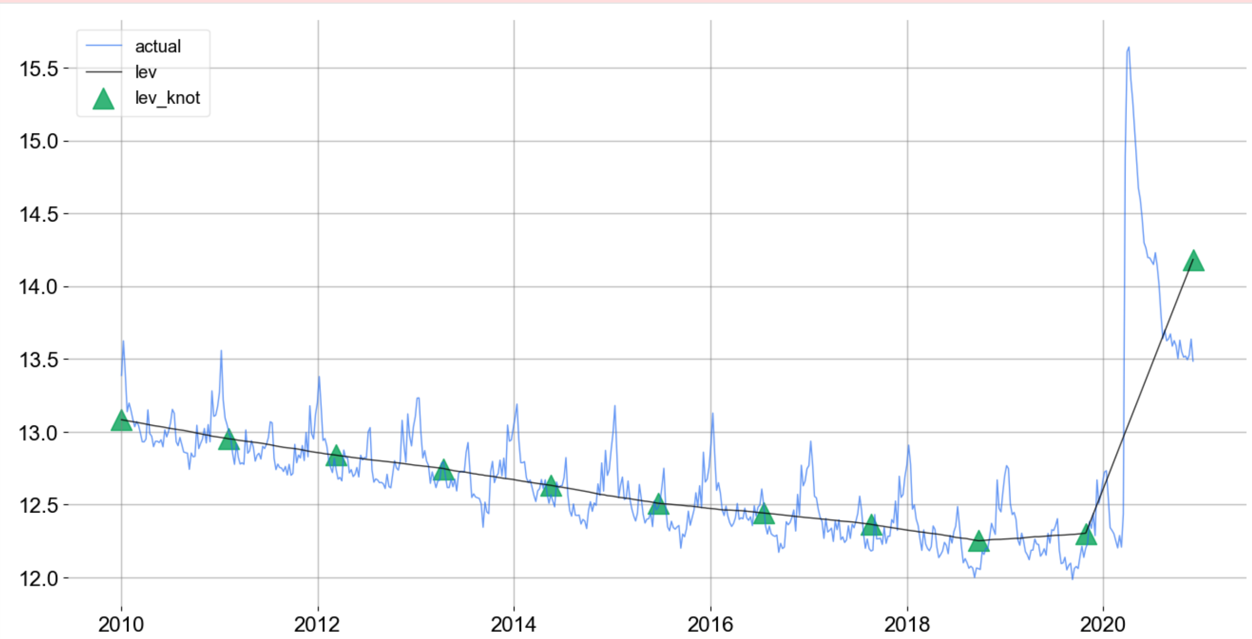
The first way to specify the knot locations and number is the `level_segements` argument. This gives the number of between knot segments; since there is a knot on each end of each the total number of knots would be the number of segments plus one. To illustrate that, try `level_segments=10` (line 5).

[4]:
```
response_col = 'claims'
date_col='week'
```

[5]:
```
ktr = KTR(
    response_col=response_col,
    date_col=date_col,

    level_segments=10,
    prediction_percentiles=[2.5, 97.5],
    seed=2020,
    estimator='pyro-svi'
)
```

[6]:
```
ktr.fit(df=df)
_ = ktr.plot_lev_knots()
```

```
2024-03-19 23:39:34 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:39:34 - orbit - INFO - Using SVI (Pyro) with steps: 301, samples: 100,␣
↪learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
/Users/towinazure/opt/miniconda3/envs/orbit311/lib/python3.11/site-packages/torch/__init_
↪_.py:696: 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)
2024-03-19 23:39:34 - orbit - INFO - step    0 loss = 176.47, scale = 0.083093
INFO:orbit:step    0 loss = 176.47, scale = 0.083093
2024-03-19 23:39:35 - orbit - INFO - step  100 loss = 113.08, scale = 0.046374
INFO:orbit:step  100 loss = 113.08, scale = 0.046374
2024-03-19 23:39:36 - orbit - INFO - step  200 loss = 113.14, scale = 0.046119
INFO:orbit:step  200 loss = 113.14, scale = 0.046119
2024-03-19 23:39:38 - orbit - INFO - step  300 loss = 113.21, scale = 0.046233
INFO:orbit:step  300 loss = 113.21, scale = 0.046233
```



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.

[7]:
```
ktr = KTR(
    response_col=response_col,
    date_col=date_col,
    level_knot_distance=104,
```

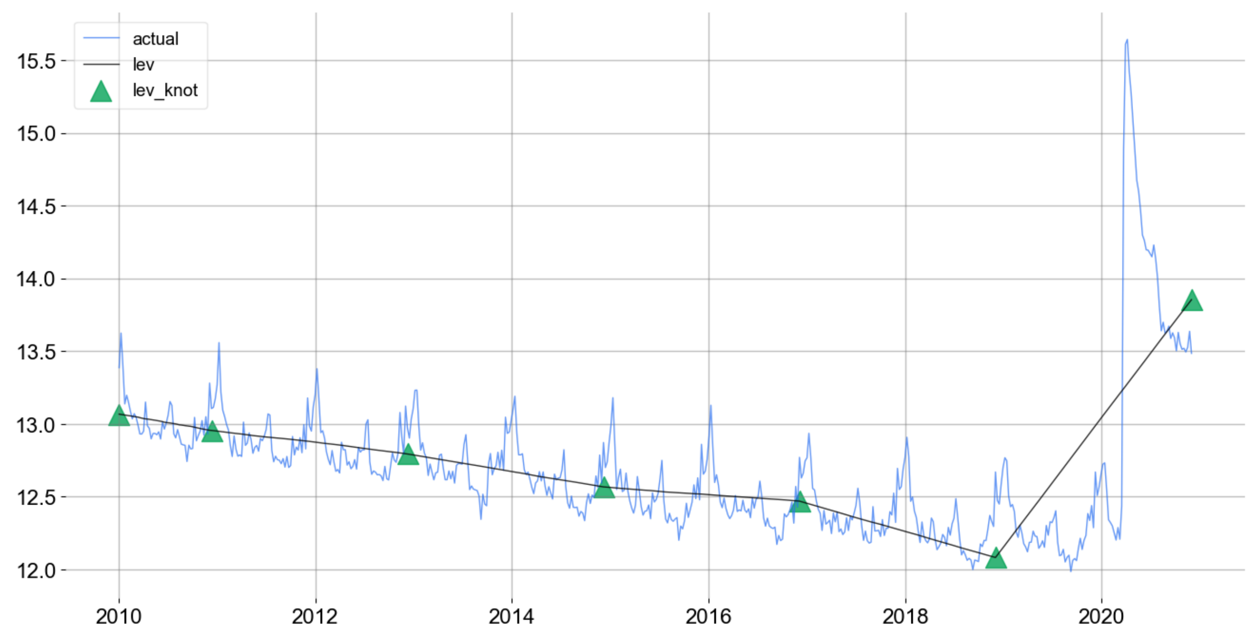
(continues on next page)

(continued from previous 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()
```

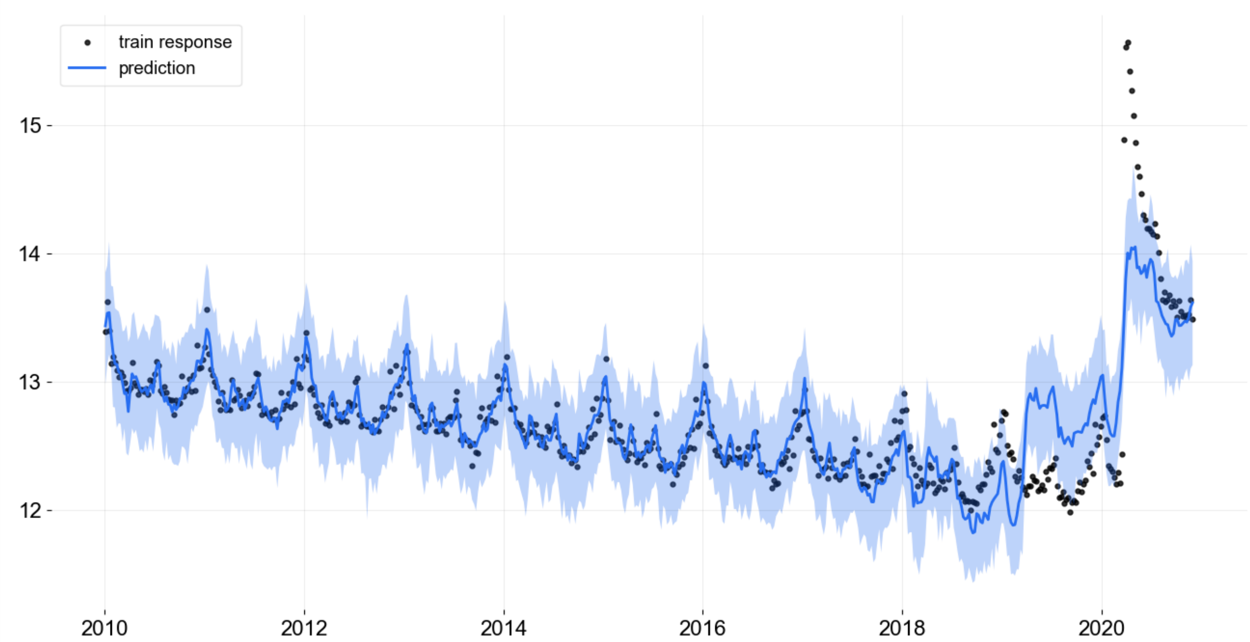
```
2024-03-19 23:39:38 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:39:38 - orbit - INFO - Using SVI (Pyro) with steps: 301, samples: 100,␣
↪learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
INFO:orbit:Using SVI (Pyro) with steps: 301, samples: 100, learning rate: 0.1, learning_
↪rate_total_decay: 1.0 and particles: 100.
2024-03-19 23:39:38 - orbit - INFO - step    0 loss = 145.65, scale = 0.088976
INFO:orbit:step    0 loss = 145.65, scale = 0.088976
2024-03-19 23:39:40 - orbit - INFO - step  100 loss = -5.2369, scale = 0.036939
INFO:orbit:step  100 loss = -5.2369, scale = 0.036939
2024-03-19 23:39:42 - orbit - INFO - step  200 loss = -5.3791, scale = 0.036969
INFO:orbit:step  200 loss = -5.3791, scale = 0.036969
2024-03-19 23:39:46 - orbit - INFO - step  300 loss = -5.5677, scale = 0.037689
INFO:orbit:step  300 loss = -5.5677, scale = 0.037689
```



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.

[9]:
```
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)
```



As the knots are placed evenly the model can not 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.

[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'
)
```

[11]:
```
ktr.fit(df=df)
```

```
2024-03-19 23:39:46 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
```

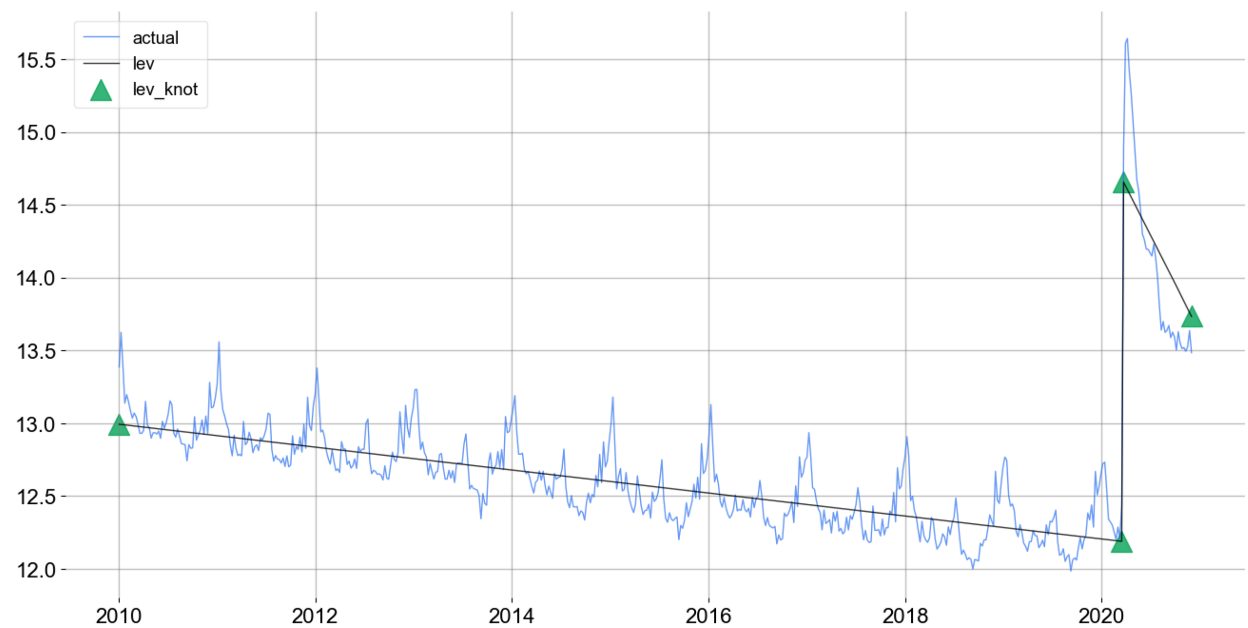
(continues on next page)

(continued from previous page)

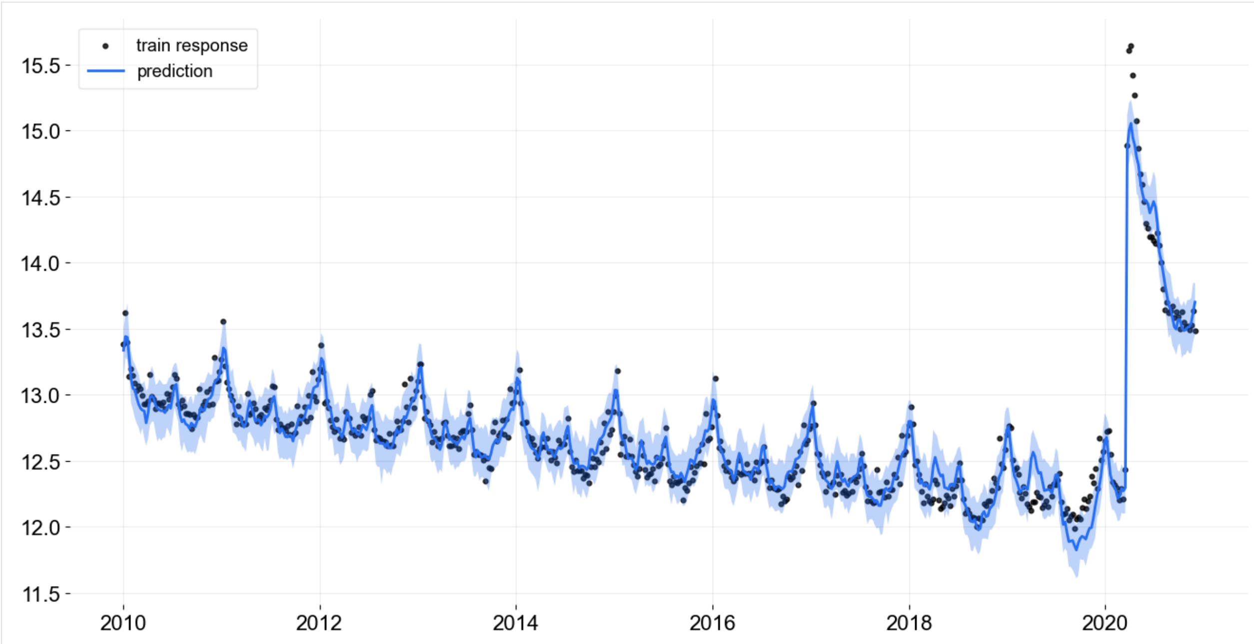
```
2024-03-19 23:39:46 - orbit - INFO - Using SVI (Pyro) with steps: 301, samples: 100,␣
↪learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
INFO:orbit:Using SVI (Pyro) with steps: 301, samples: 100, learning rate: 0.1, learning_
↪rate_total_decay: 1.0 and particles: 100.
2024-03-19 23:39:47 - orbit - INFO - step    0 loss = 99.354, scale = 0.096314
INFO:orbit:step    0 loss = 99.354, scale = 0.096314
2024-03-19 23:39:52 - orbit - INFO - step  100 loss = -440.9, scale = 0.027049
INFO:orbit:step  100 loss = -440.9, scale = 0.027049
2024-03-19 23:39:54 - orbit - INFO - step  200 loss = -446.03, scale = 0.028019
INFO:orbit:step  200 loss = -446.03, scale = 0.028019
2024-03-19 23:39:56 - orbit - INFO - step  300 loss = -445.62, scale = 0.029141
INFO:orbit:step  300 loss = -445.62, scale = 0.029141
```

[11]:
```
<orbit.forecaster.svi.SVIForecaster at 0x2b1b1a810>
```

[12]:
```python
_ = ktr.plot_lev_knots()
```



[13]:
```python
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 fro **KTR** users to explore them.

14.3 References

1. Ng, Wang and Dai (2021). Bayesian Time Varying Coefficient Model with Applications to Marketing Mix Modeling, arXiv preprint arXiv:2106.03322
2. Sean J Taylor and Benjamin Letham. 2018. Forecasting at scale. The American Statistician 72, 1 (2018), 37–45. Package version 0.7.1.

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__)
```

```
1.1.4.6
```

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

[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_col = ['beta{}'.format(pp) for pp in range(1, p+1)]
beta_data = pd.DataFrame(beta[:,1:], columns=beta_col)

data = pd.concat([data, beta_data], axis=1)
```

[4]:
```
data.tail(10)
```

[4]:

	date	y	x1	x2	beta1	beta2
290	2018-10-18	2.15947	-0.62762	0.17840	0.04015	0.02739
291	2018-10-19	2.25871	-0.92975	0.81415	0.04036	0.02723
292	2018-10-20	2.18356	0.82438	-0.92705	0.04057	0.02707
293	2018-10-21	2.26948	1.57181	-0.78098	0.04077	0.02692
294	2018-10-22	2.26375	-1.07504	-0.86523	0.04097	0.02677
295	2018-10-23	2.21349	0.24637	-0.98398	0.04117	0.02663
296	2018-10-24	2.13297	-0.58716	0.59911	0.04136	0.02648
297	2018-10-25	2.00949	-2.01610	0.08618	0.04155	0.02634
298	2018-10-26	2.14302	0.33863	-0.37912	0.04173	0.02620
299	2018-10-27	2.10795	-0.96160	-0.42383	0.04192	0.02606

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

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

(continues on next page)

(continued from previous 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)
```

```
2024-03-19 23:39:37 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:39:37 - orbit - INFO - Using SVI (Pyro) with steps: 301, samples: 100,␣
↪learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
/Users/towinazure/opt/miniconda3/envs/orbit311/lib/python3.11/site-packages/torch/__init_
↪_.py:696: 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)
2024-03-19 23:39:38 - orbit - INFO - step    0 loss = -3.5592, scale = 0.085307
INFO:orbit:step    0 loss = -3.5592, scale = 0.085307
2024-03-19 23:39:39 - orbit - INFO - step  100 loss = -228.48, scale = 0.036575
INFO:orbit:step  100 loss = -228.48, scale = 0.036575
2024-03-19 23:39:40 - orbit - INFO - step  200 loss = -230.1, scale = 0.038104
INFO:orbit:step  200 loss = -230.1, scale = 0.038104
2024-03-19 23:39:41 - orbit - INFO - step  300 loss = -229.33, scale = 0.037629
INFO:orbit:step  300 loss = -229.33, scale = 0.037629
```

[5]:
```
<orbit.forecaster.svi.SVIForecaster at 0x2a836ad90>
```

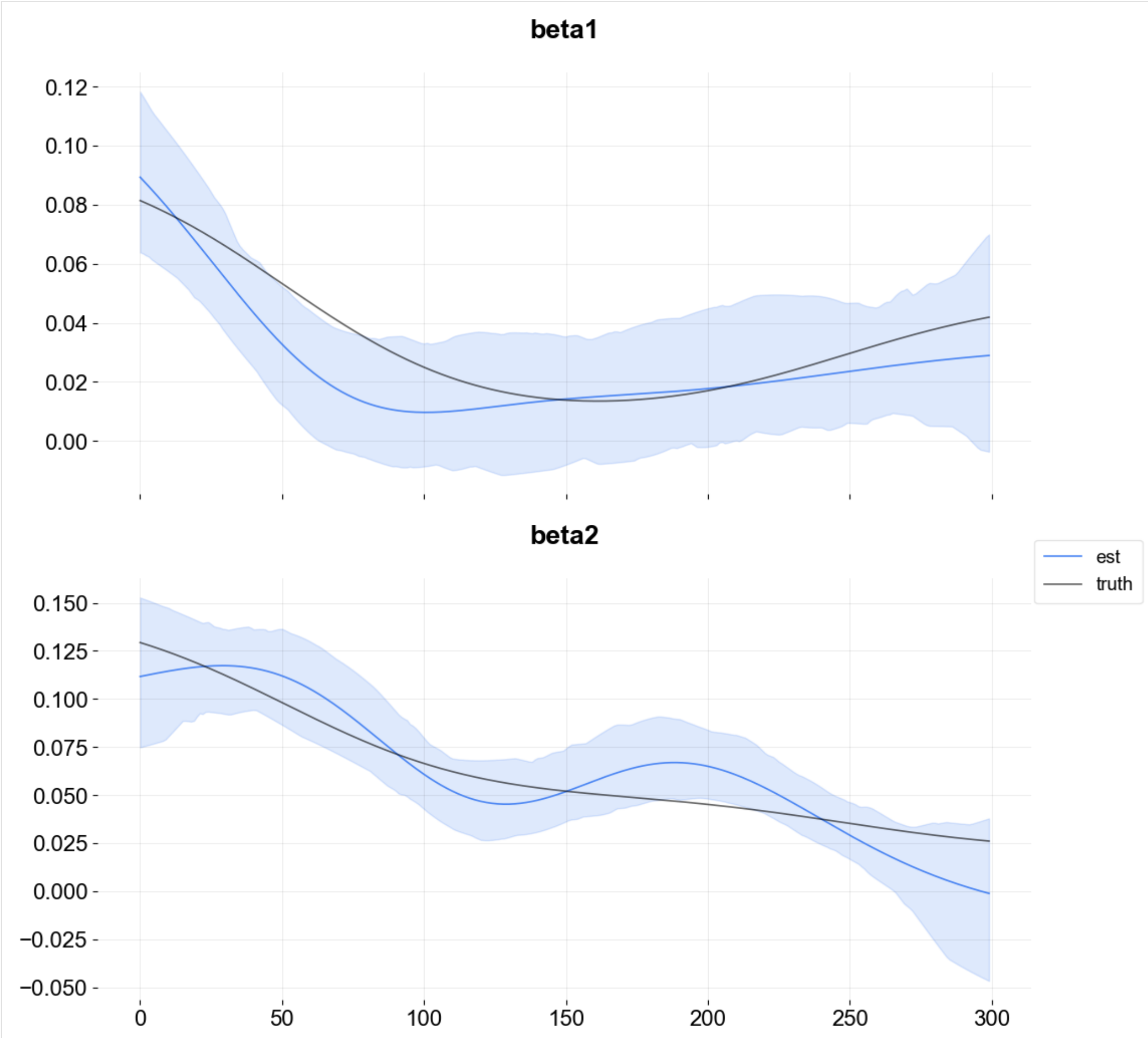
15.2 Visualization of Regression Coefficient Curves

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

[6]:
```
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.

[7]:
```
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));
```



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)

(continued from previous page)

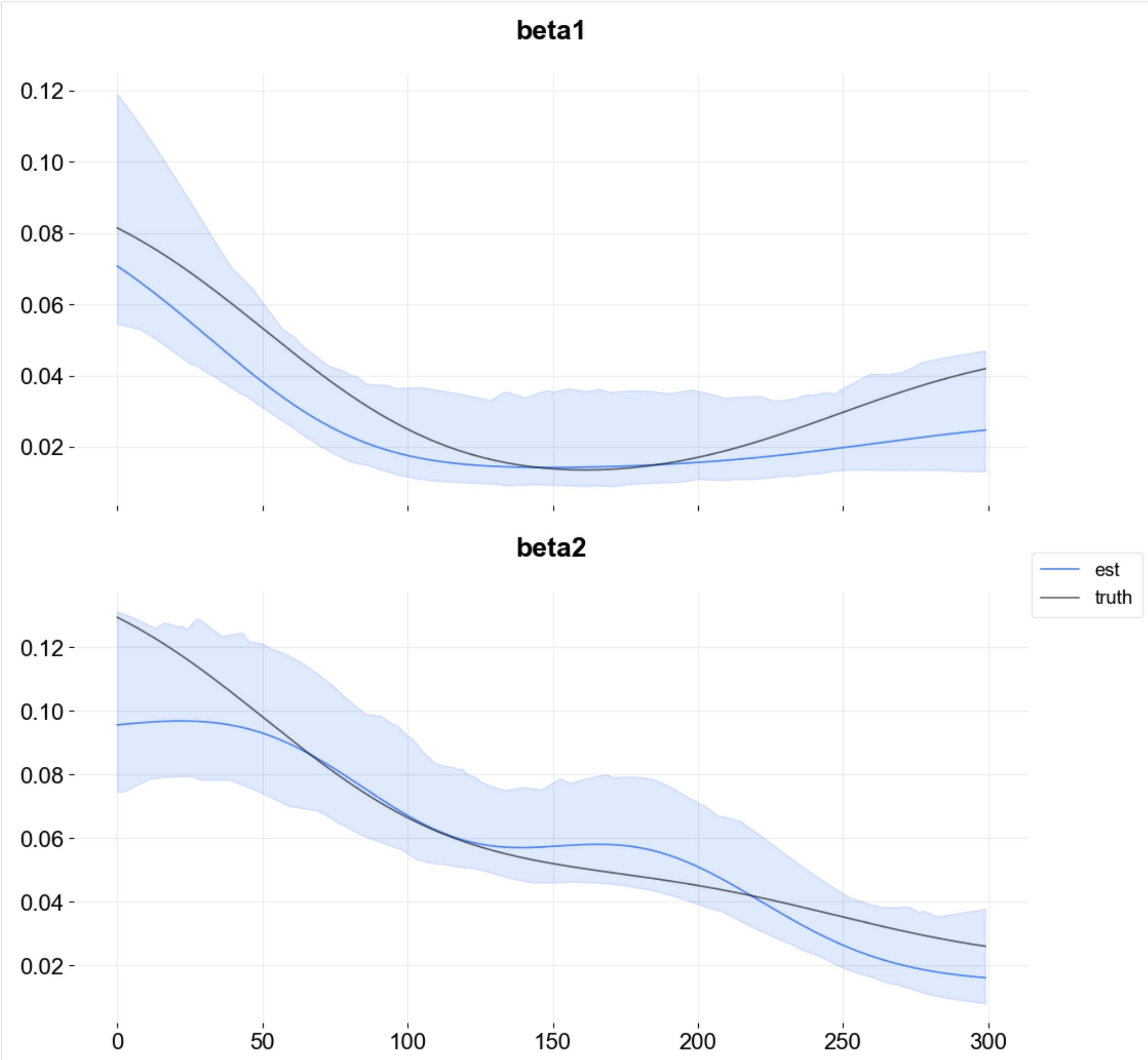
```
    seed=2021,
    estimator='pyro-svi',
)
ktr.fit(df=data)
```

```
2024-03-19 23:39:41 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:39:41 - 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.
2024-03-19 23:39:41 - orbit - INFO - step    0 loss = 9.7371, scale = 0.10482
INFO:orbit:step    0 loss = 9.7371, scale = 0.10482
2024-03-19 23:39:43 - orbit - INFO - step  100 loss = -231.22, scale = 0.41649
INFO:orbit:step  100 loss = -231.22, scale = 0.41649
2024-03-19 23:39:45 - orbit - INFO - step  200 loss = -230.94, scale = 0.42589
INFO:orbit:step  200 loss = -230.94, scale = 0.42589
2024-03-19 23:39:46 - 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 0x2b1d43810>
```

[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))
```

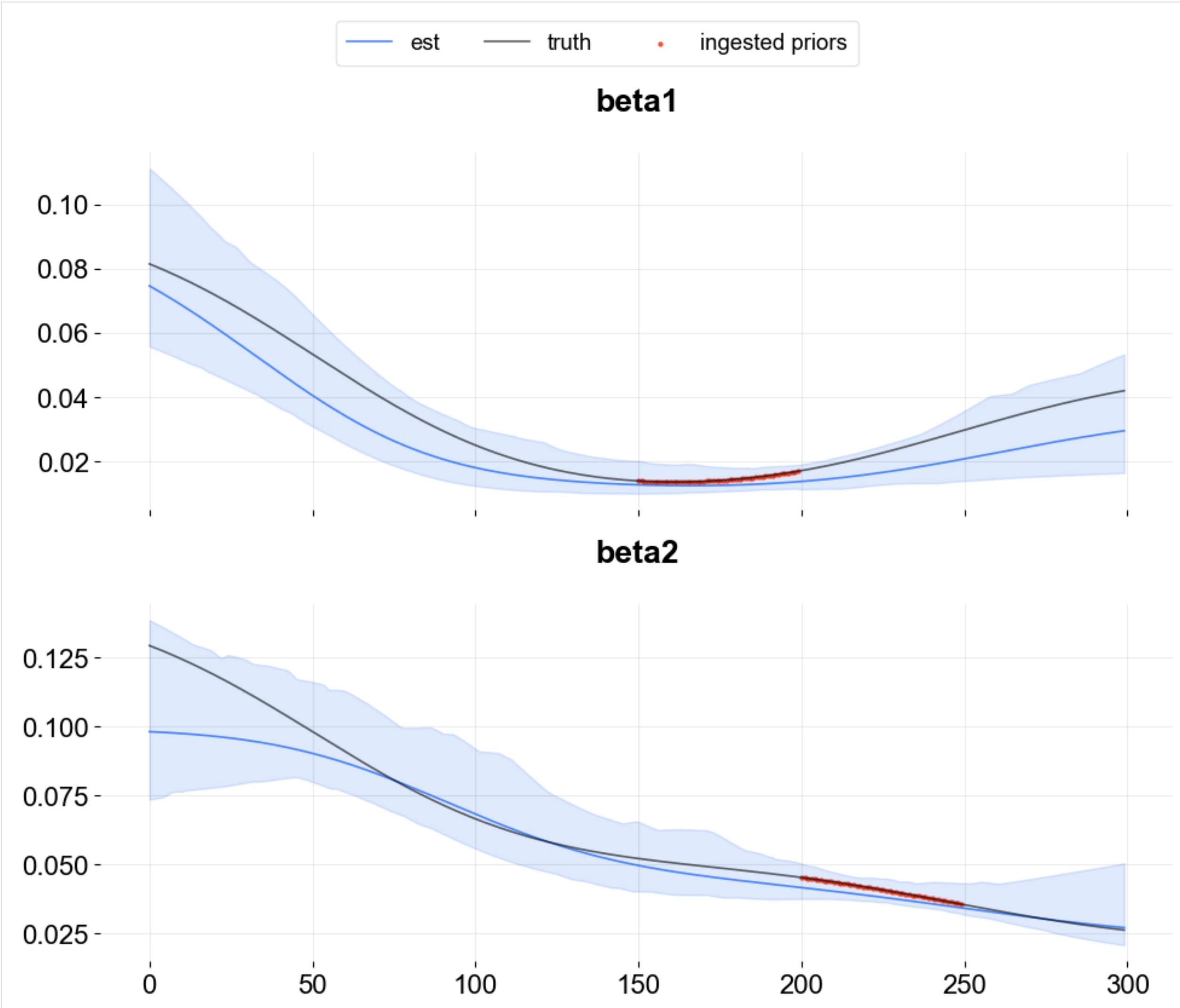
[14]:
```
ktr = KTR(
    response_col='y',
    date_col='date',
    regressor_col=regressor_col,
    regressor_init_knot_scale=[0.1] * p,
    regressor_knot_scale=[0.1] * p,
    regressor_sign=['+'] * p,
    coef_prior_list = coef_list_dict,
    prediction_percentiles=[2.5, 97.5],
    seed=2021,
    estimator='pyro-svi',
)
ktr.fit(df=data)
```

```
2024-03-19 23:39:47 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:39:47 - 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.
2024-03-19 23:39:47 - orbit - INFO - step    0 loss = -5741.9, scale = 0.094521
INFO:orbit:step    0 loss = -5741.9, scale = 0.094521
2024-03-19 23:39:52 - orbit - INFO - step  100 loss = -7140.3, scale = 0.31416
INFO:orbit:step  100 loss = -7140.3, scale = 0.31416
2024-03-19 23:39:55 - orbit - INFO - step  200 loss = -7139.1, scale = 0.31712
INFO:orbit:step  200 loss = -7139.1, scale = 0.31712
2024-03-19 23:39:57 - orbit - INFO - step  300 loss = -7139.5, scale = 0.33039
INFO:orbit:step  300 loss = -7139.5, scale = 0.33039
```

[14]:
```
<orbit.forecaster.svi.SVIForecaster at 0x2b1eb5f10>
```

[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{}'.format(idx + 1)], label='est', 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', alpha=0.6, color =␣
↪OrbitPalette.BLACK.value)
    axes[idx].set_title('beta{}'.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

1. Ng, Wang and Dai (2021). Bayesian Time Varying Coefficient Model with Applications to Marketing Mix Modeling, arXiv preprint arXiv:2106.03322

CHAPTER

SIXTEEN

PREDICTION DECOMPOSITION

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

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]:
```
print(orbit.__version__)
```

```
1.1.4.6
```

[3]:
```
# 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.

[4]:
```
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)
```

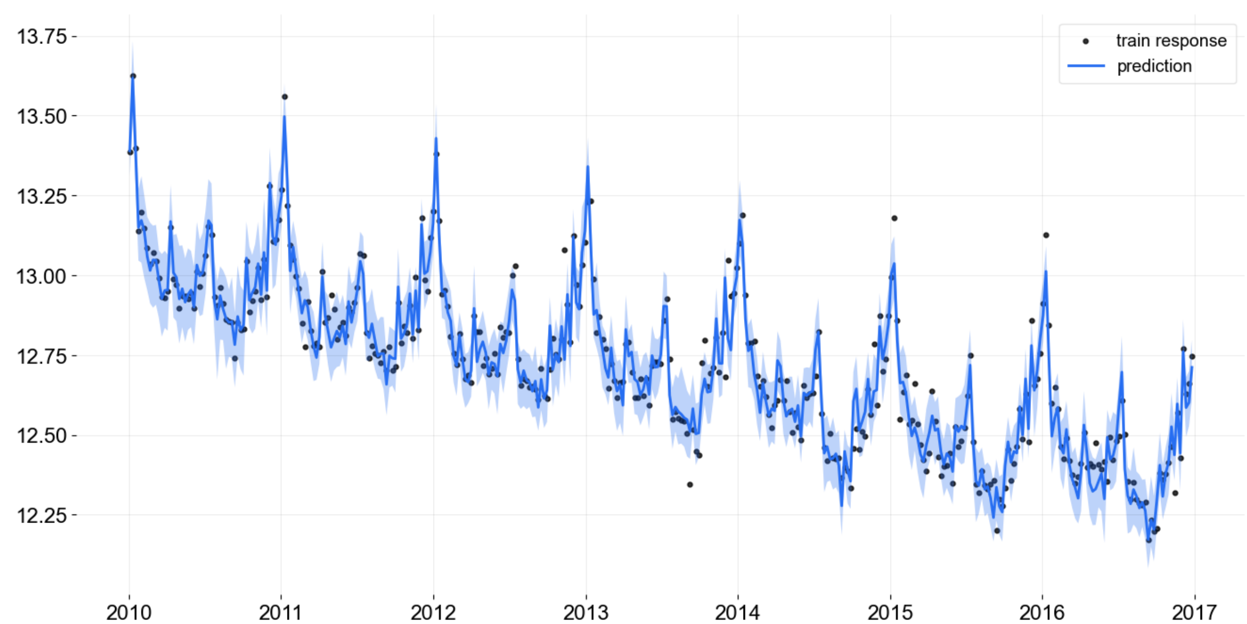
```
2024-03-19 23:38:01 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 1.000, warmups (per chain): 225 and samples(per chain): 25.
```

[4]:
```
<orbit.forecaster.full_bayes.FullBayesianForecaster at 0x2a578a450>
```

16.2 Plot Predictions

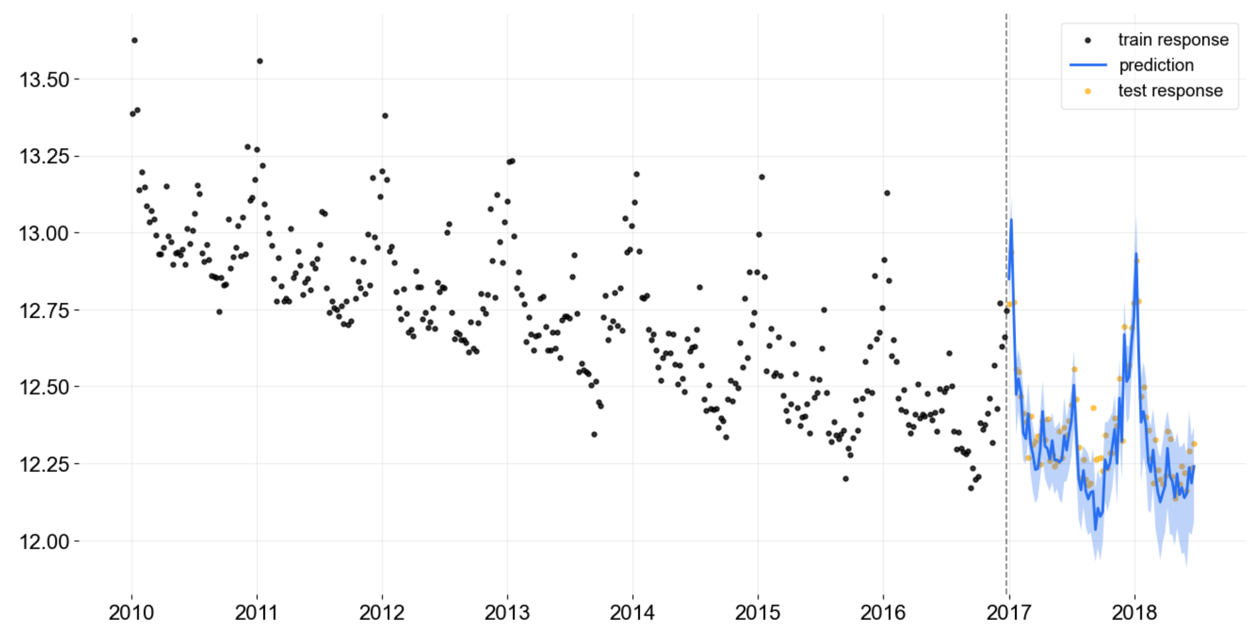
First, we do the prediction on the training data before the year 2017.

[5]:
```
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.

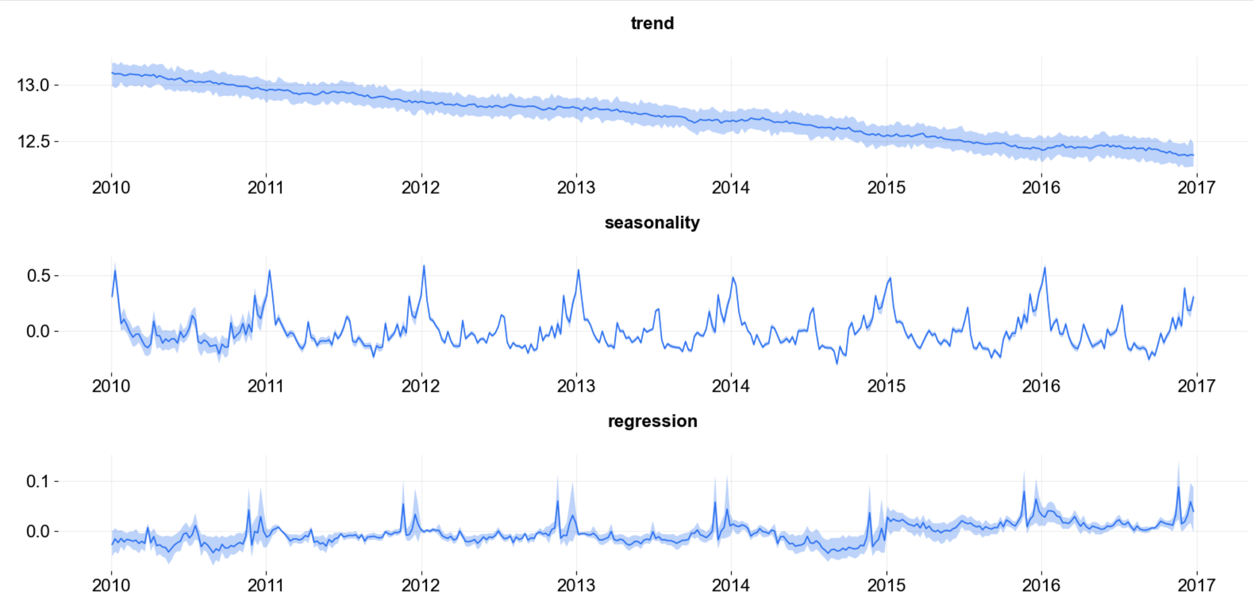
[6]:
```
predicted_df = dlt.predict(df=test_df, decompose=True)

_ = 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)
```



16.3 Plot Predicted Components

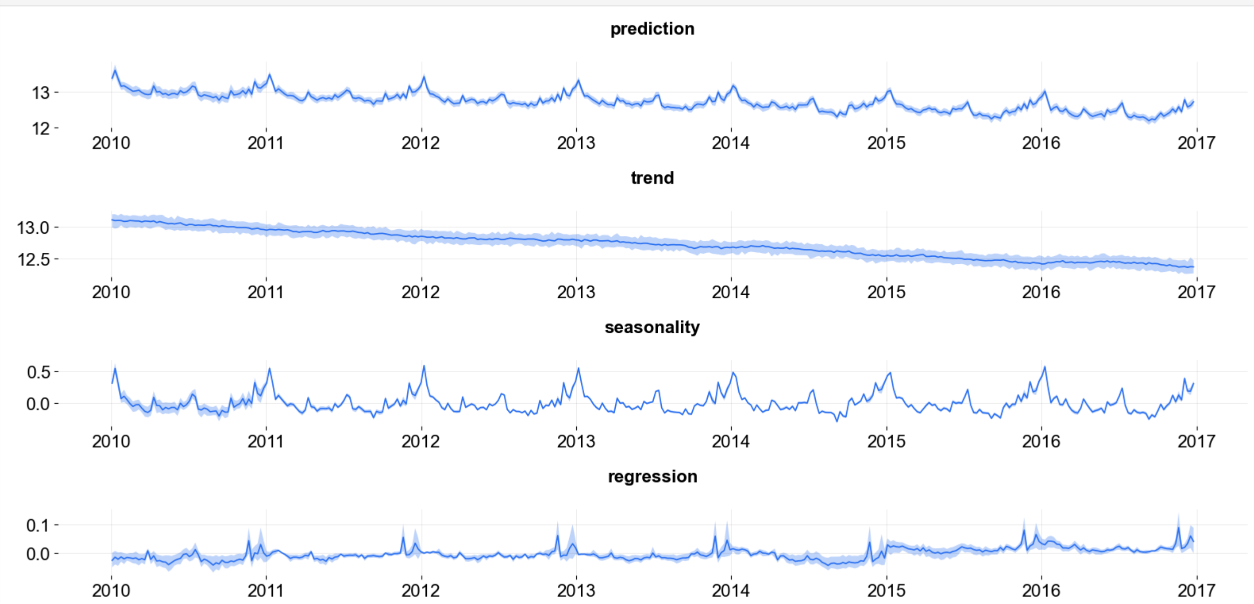
`plot_predicted_components` is the utility to plot each component separately. This is useful when one wants to look into the model prediction results and inspect each component separately.

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

_ = 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'])
```



CHAPTER

SEVENTEEN

MODEL DIAGNOSTICS

In this section, we introduce to 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__)
```

```
1.1.4.6
```

17.1 Load data

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

[3]:
```
week                datetime64[ns]
claims                     float64
trend.unemploy             float64
trend.filling              float64
trend.job                  float64
sp500                      float64
vix                        float64
dtype: object
```

[4]:
```
df.head(5)
```

[4]:
```
        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
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  0.110445
2  0.532339
3  0.428645
4  0.487404
```

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)
```

```
2024-03-19 23:39:45 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 1.000, warmups (per chain): 500 and samples(per chain): 500.
```

[7]:
```
<orbit.forecaster.full_bayes.FullBayesianForecaster at 0x294cf72d0>
```

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`.

[8]:
```
ps = dlt.get_posterior_samples(relabel=True, permute=False)
ps.keys()
```

[8]:
```
dict_keys(['l', 'b', 'lev_sm', 'slp_sm', 'obs_sigma', 'nu', 'lt_sum', 's', 'sea_sm', 'gt_
↪sum', 'gb', 'gl', 'loglk', 'trend.unemploy', 'trend.filling', 'trend.job'])
```

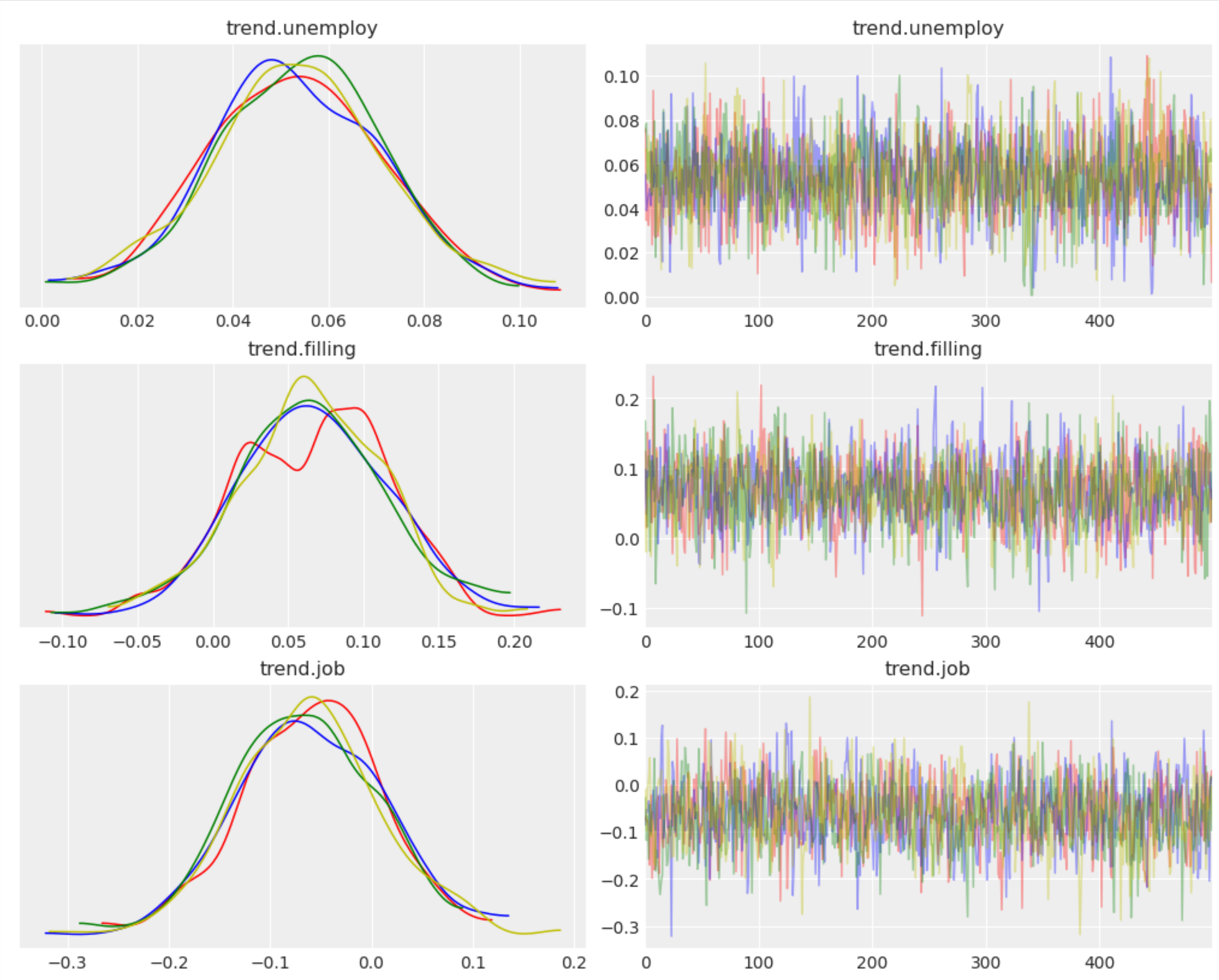
17.3 Diagnostics Visualization

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.

[9]:
```
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),
);
```

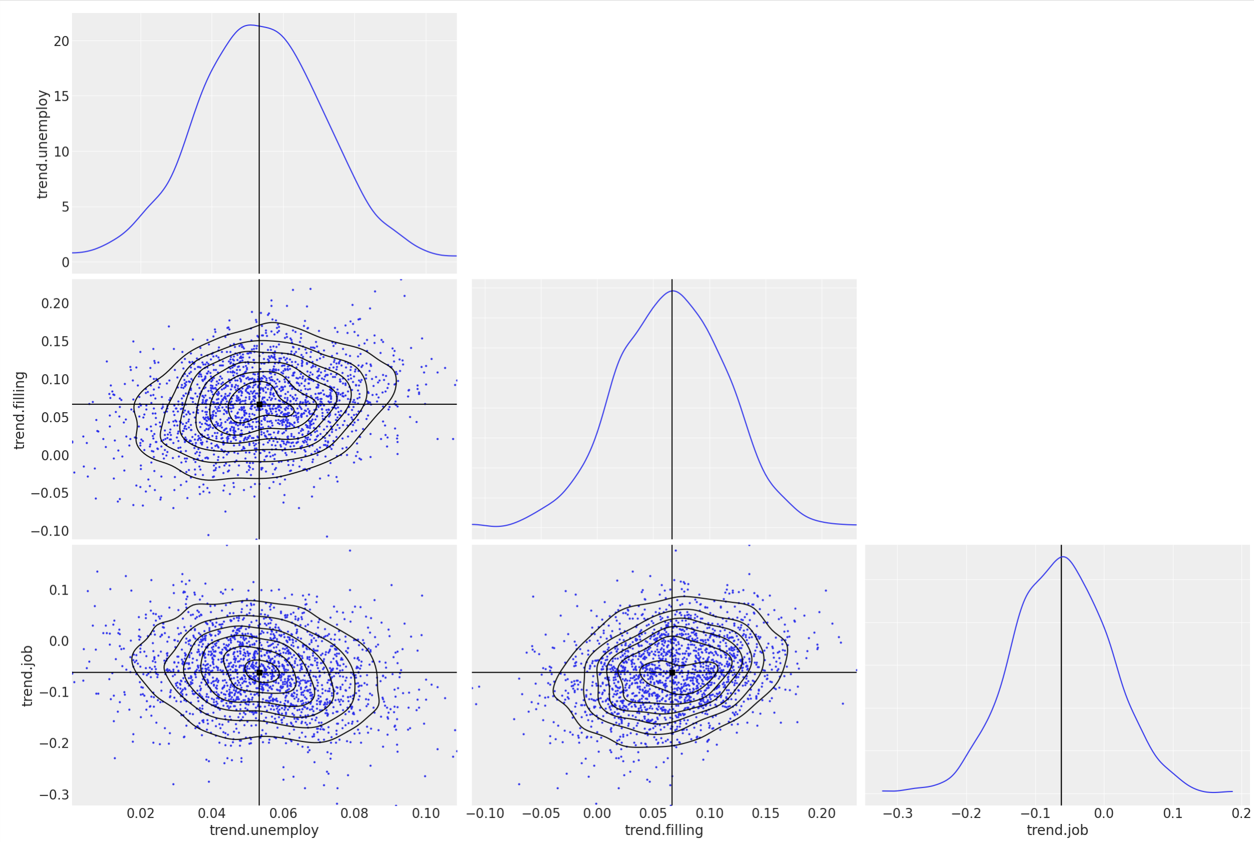


Note that this is only applicable for `FullBayesianForecaster` using sampling method such as MCMC.

17.3.2 Samples Density

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

[10]:
```
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.

[11]:
```
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},
)
dlt_smaller_prior.fit(df=df)
ps_smaller_prior = dlt_smaller_prior.get_posterior_samples(relabel=True, permute=False)
```

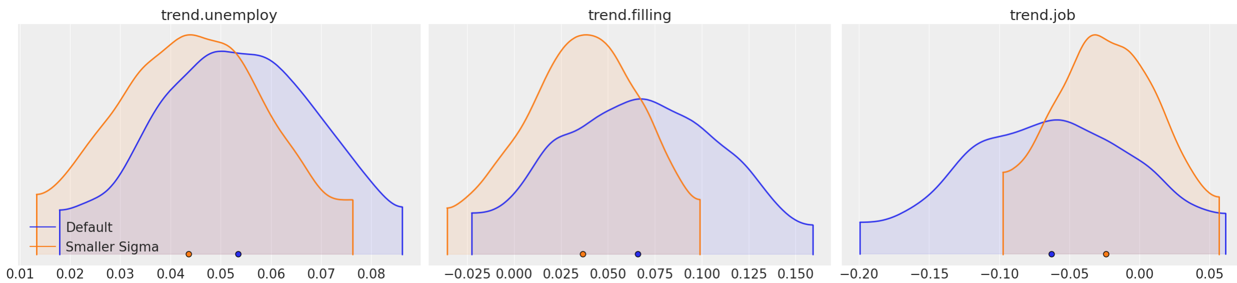
```
2024-03-19 23:39:55 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 1.000, warmups (per chain): 500 and samples(per chain): 500.
```

[12]:
```
az.plot_density(
    [ps, ps_smaller_prior],
    var_names=['trend.unemploy', 'trend.filling', 'trend.job'],
```

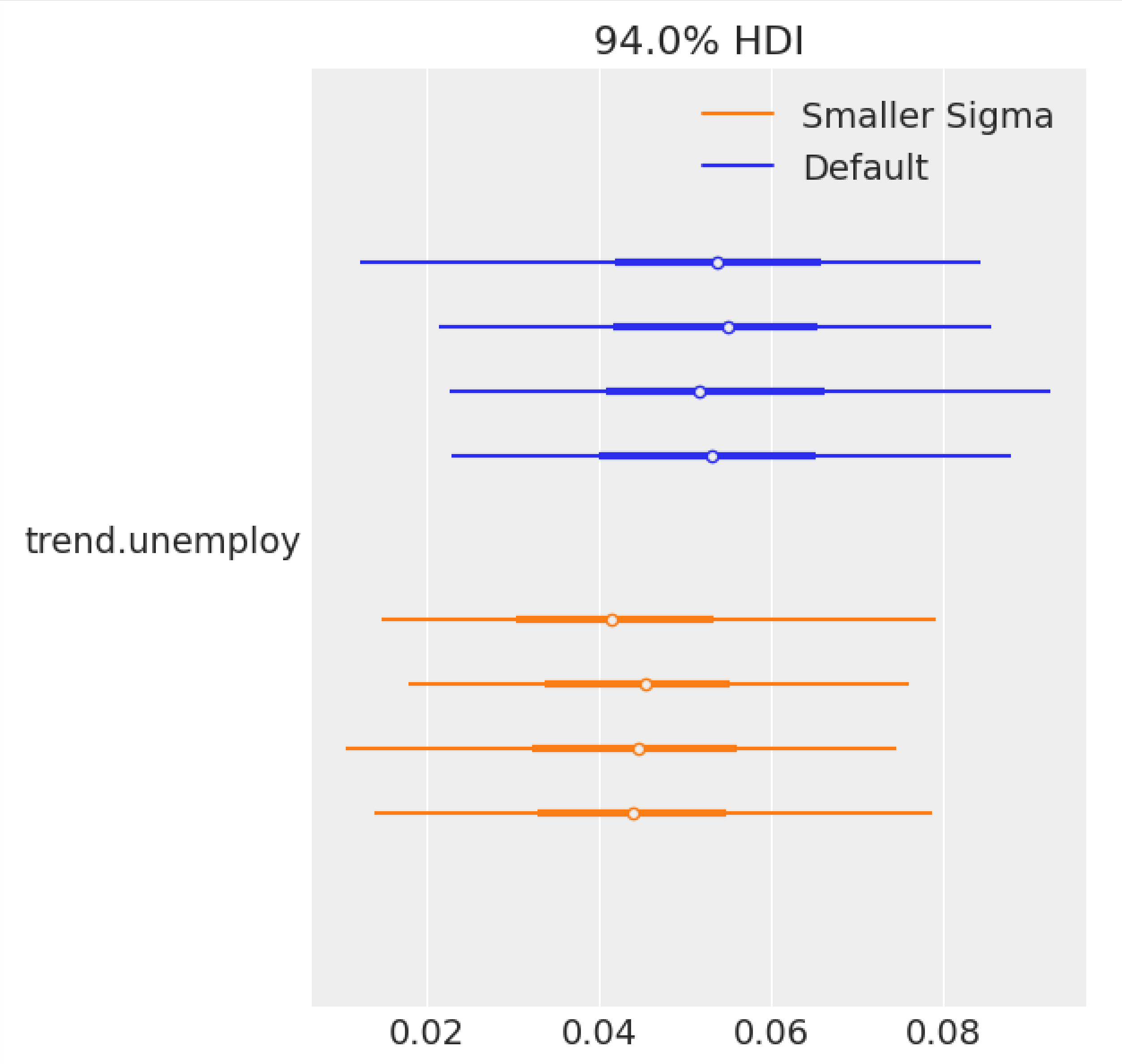
(continues on next page)

(continued from previous page)

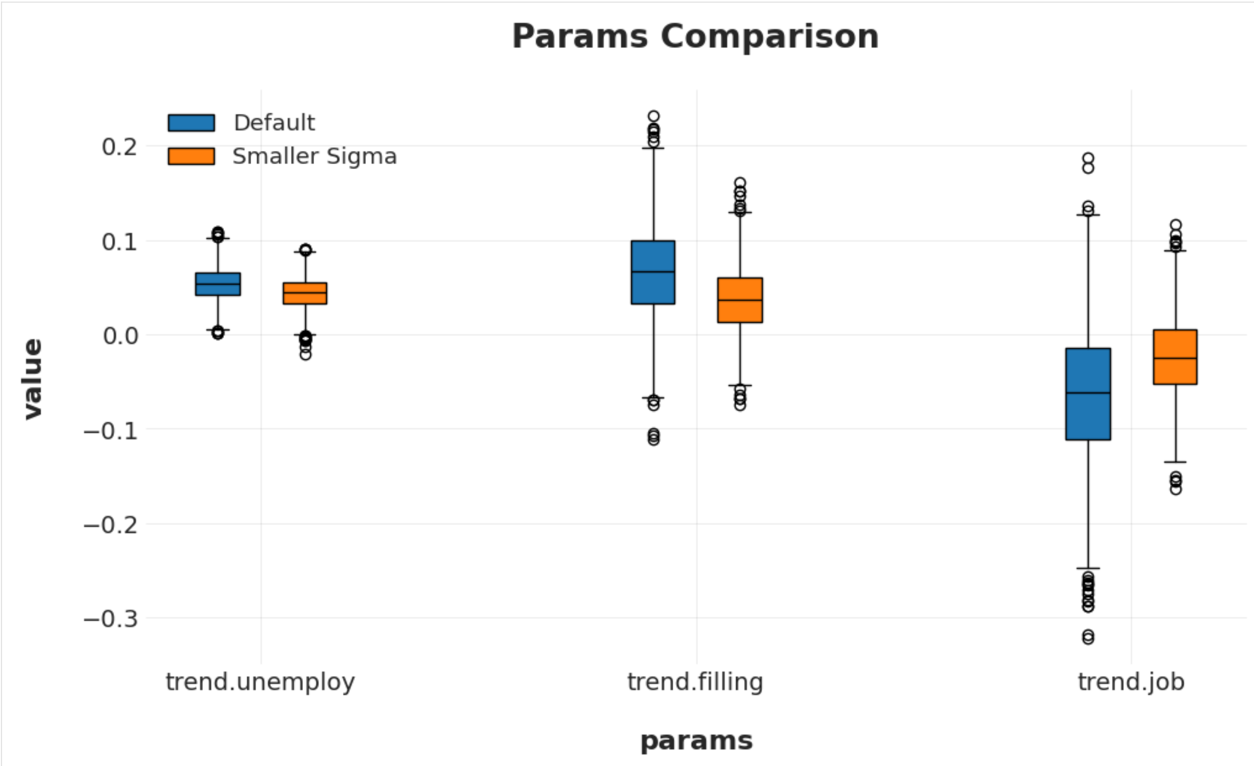
```
    data_labels=["Default", "Smaller Sigma"],
    shade=0.1,
    textsize=18.5,
);
```



[13]:
```
az.plot_forest(
    [ps, ps_smaller_prior],
    var_names=['trend.unemploy'],
    model_names=["Default", "Smaller Sigma"],
);
```



[14]:
```
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.

CHAPTER

EIGHTEEN

BACKTEST

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

[1]:
```
%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')
```

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

```
1.1.4.6
```

[3]:
```
# load log-transformed data
data = load_iclaims()
```

[4]:
```
data.shape
```

[4]:
```
(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. `TimeSeriesSplitter` implemented 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 two main way 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

[5]:
```
# configs
min_train_len = 380 # minimal length of window length
forecast_len = 20 # length forecast window
incremental_len = 20 # step length for moving forward
```

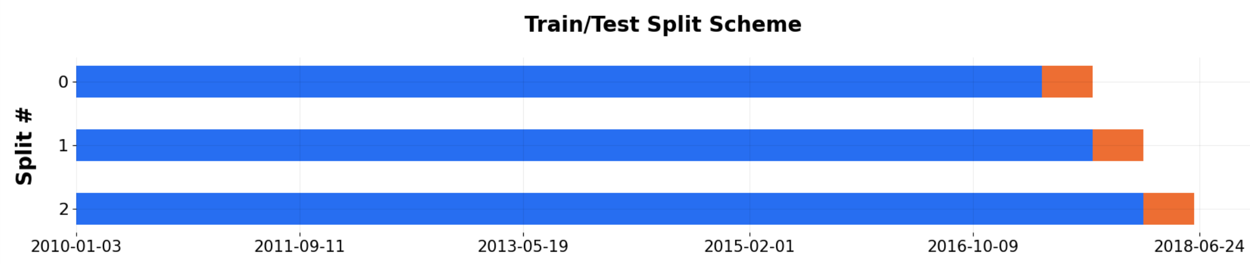
[6]:
```
ex_splitter = TimeSeriesSplitter(df=data,
                                 min_train_len=min_train_len,
                                 incremental_len=incremental_len,
                                 forecast_len=forecast_len,
                                 window_type='expanding',
                                 date_col='week')
```

[7]:
```
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
```

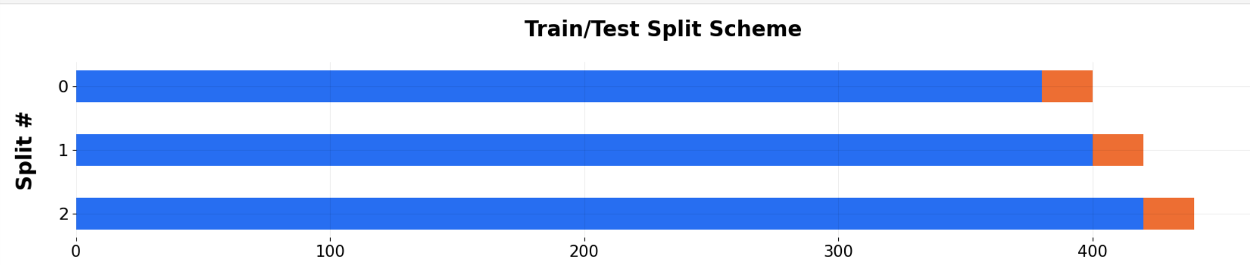
Users can visualize the splits using the internal `plot()` function. One may notice that the last few data points may not be included in the last split, which is expected when `min_train_len` is specified.

[8]:
```
_ = ex_splitter.plot()
```



If users want to visualize the scheme in terms of indcies, one can do the following.

[9]:
```
_ = ex_splitter.plot(show_index=True)
```



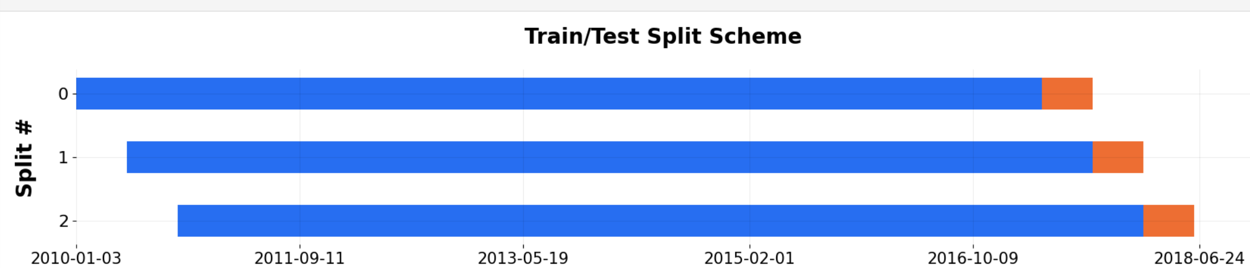
18.1.2 Rolling window

[10]:
```
# 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
```

[11]:
```
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.

[12]:
```
_ = roll_splitter.plot()
```

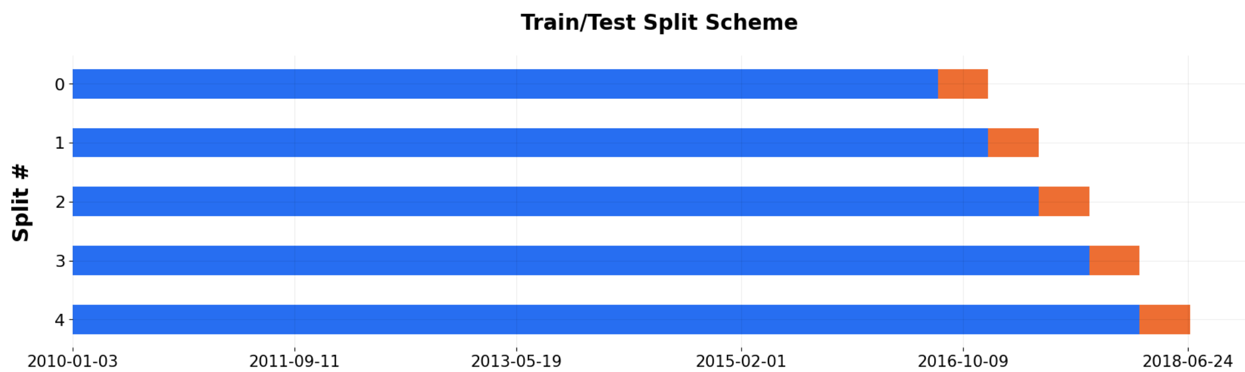


18.1.3 Specifying number of splits

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()
```



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)

(continued from previous 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

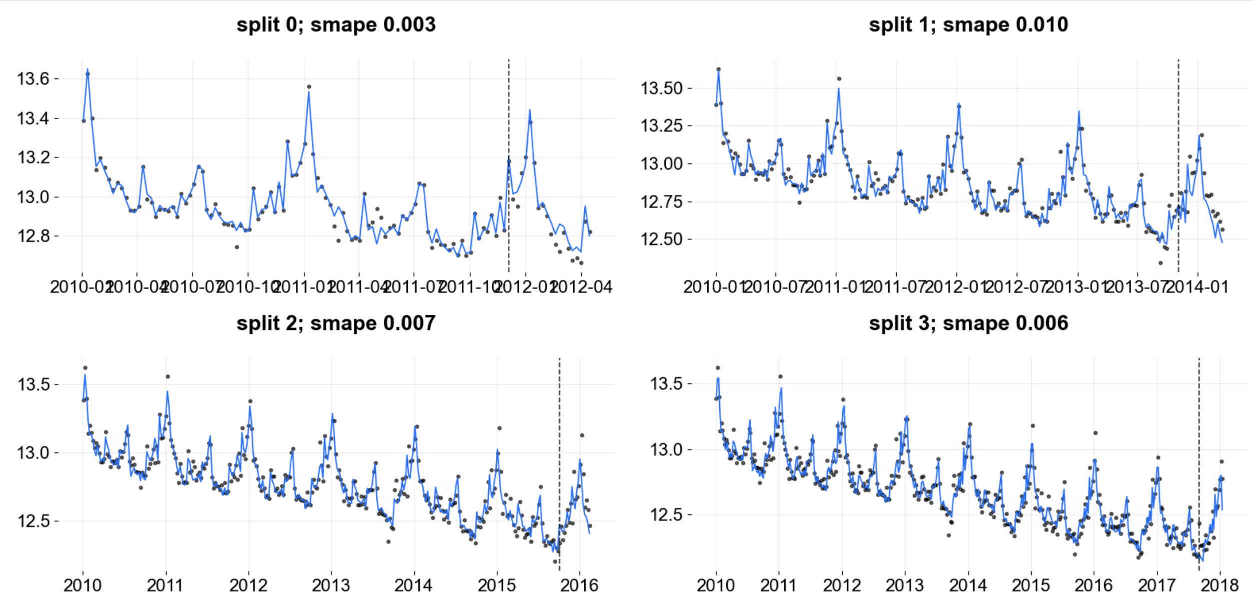
[19]:
```
predicted_df = bt.get_predicted_df()
predicted_df.head()
```

[19]:

	date	actual	prediction	training_data	split_key
0	2010-01-03	13.386595	13.386576	True	0
1	2010-01-10	13.624218	13.649070	True	0
2	2010-01-17	13.398741	13.373163	True	0
3	2010-01-24	13.137549	13.151905	True	0
4	2010-01-31	13.196760	13.187853	True	0

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

[21]:
```
predicted_df.merge(data, left_on='date', right_on='week')
```

[21]:
```
            date     actual  prediction  training_data  split_key       week  \
0     2010-01-03  13.386595   13.386576           True          0 2010-01-03
1     2010-01-10  13.624218   13.649070           True          0 2010-01-10
2     2010-01-17  13.398741   13.373163           True          0 2010-01-17
3     2010-01-24  13.137549   13.151905           True          0 2010-01-24
4     2010-01-31  13.196760   13.187853           True          0 2010-01-31
...          ...        ...         ...            ...        ...        ...
1075 2017-12-17  12.568616   12.566428          False          3 2017-12-17
1076 2017-12-24  12.691451   12.675789          False          3 2017-12-24
1077 2017-12-31  12.769532   12.783320          False          3 2017-12-31
1078 2018-01-07  12.908227   12.800172          False          3 2018-01-07
1079 2018-01-14  12.777193   12.536663          False          3 2018-01-14

         claims  trend.unemploy  trend.filling  trend.job      sp500       vix
0     13.386595        0.219882      -0.318452   0.117500 -0.417633  0.122654
1     13.624218        0.219882      -0.194838   0.168794 -0.425480  0.110445
2     13.398741        0.236143      -0.292477   0.117500 -0.465229  0.532339
3     13.137549        0.203353      -0.194838   0.106918 -0.481751  0.428645
4     13.196760        0.134360      -0.242466   0.074483 -0.488929  0.487404
...         ...             ...            ...        ...       ...       ...
1075  12.568616        0.298663       0.248654  -0.216869  0.434042 -0.482380
1076  12.691451        0.328516       0.233616  -0.358839  0.430410 -0.373389
1077  12.769532        0.503457       0.069313  -0.092571  0.456087 -0.553539
1078  12.908227        0.527849       0.051295   0.029532  0.471673 -0.456456
1079  12.777193        0.465717       0.032946   0.006275  0.480271 -0.352770
```

(continues on next page)

(continued from previous page)

```
[1080 rows x 12 columns]
```

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**.

[22]:
```
bt.score()
```

[22]:
```
  metric_name  metric_values  is_training_metric
0       smape       0.006649               False
1       wmape       0.006645               False
2        mape       0.006631               False
3         mse       0.012889               False
4         mae       0.084415               False
5       rmsse       0.810353               False
```

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 ;)

[23]:
```
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))
```

[24]:
```
bt.score(metrics=[mse_naive, naive_error])
```

[24]:
```
   metric_name  metric_values  is_training_metric
0    mse_naive       0.019628               False
1  naive_error       0.229586               False
```

It doesn't take additional time to refit 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

[25]:
```
bt.score(include_training_metrics=True)
```

[25]:
```
   metric_name  metric_values  is_training_metric
0        smape       0.006649               False
1        wmape       0.006645               False
2         mape       0.006631               False
3          mse       0.012889               False
```

(continues on next page)

(continued from previous page)

```
4          mae       0.084415              False
5        rmsse       0.810353              False
6        smape       0.002738               True
7        wmape       0.002742               True
8         mape       0.002738               True
9          mse       0.003118               True
10         mae       0.035037               True
```

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

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

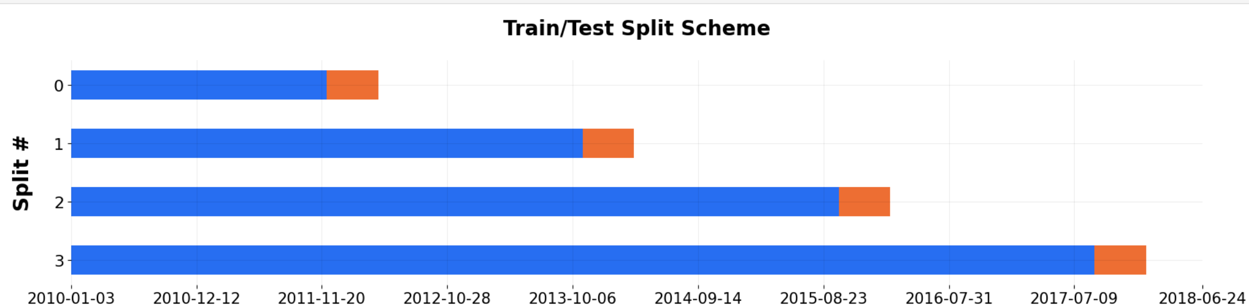
[27]:
```
model_1 = fitted_models[0]
model_1.get_regression_coefs()
```

[27]:
```
        regressor regressor_sign  coefficient
0  trend.unemploy        Regular    -0.048327
1   trend.filling        Regular    -0.120384
2       trend.job        Regular     0.027867
```

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

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



18.6 Hyperparameter Tunning

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
```

[32]:

	level_sm_input	seasonality_sm_input	metrics
0	0.3	0.3	0.004908
1	0.3	0.5	0.004058
2	0.3	0.8	0.003608
3	0.5	0.3	0.007907
4	0.5	0.5	0.006306

[33]:
```
best_params # output best parameters
```

[33]:
```
[{'level_sm_input': 0.3, 'seasonality_sm_input': 0.8}]
```

CHAPTER

NINETEEN

WBIC/BIC

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.

[1]:
```
from datetime import timedelta

import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline

import orbit
from orbit.models import DLT,ETS, KTRLite, LGT
from orbit.utils.simulation import make_trend, make_regression

import warnings
warnings.filterwarnings('ignore')
```

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

```
1.1.4.6
```

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.

[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['date'] = 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 -0.542589 -0.468533
1  5.580432  0.452678  0.223187 -0.000000  0.290559  0.171864  0.105087
2  5.031773  0.182286  0.147066  0.014211  0.273356  0.907347 -0.343835
3  3.264027 -0.368227 -0.081455 -0.241060  0.299423 -1.148723  0.212691
4  5.246511  0.019861 -0.146228 -0.390954 -0.128596  1.269148  0.199272

         x7        x8        x9       x10       date
0 -1.605941 -0.554233  0.657250 -0.445341 2016-01-10
1  1.425865  1.073494  0.000729  0.037389 2016-01-17
2  1.659460 -0.615668 -1.946107 -0.798330 2016-01-24
3 -0.738148  1.958889  0.455206  0.315217 2016-01-31
4  0.701419  0.837458 -0.394186 -1.185948 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.

[5]:
```
%%time
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,
        # fixing the smoothing parameters to learn regression coefficients more␣
↪effectively
        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)
```

```
2024-03-19 23:42:42 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 5.900, warmups (per chain): 1000 and samples(per chain): 1000.
2024-03-19 23:42:54 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 5.900, warmups (per chain): 1000 and samples(per chain): 1000.
```

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

```
2024-03-19 23:43:06 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 5.900, warmups (per chain): 1000 and samples(per chain): 1000.
```

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

```
2024-03-19 23:43:18 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 5.900, warmups (per chain): 1000 and samples(per chain): 1000.
```

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

```
2024-03-19 23:43:29 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 5.900, warmups (per chain): 1000 and samples(per chain): 1000.
```

```
WBIC value with 4 regressors: 1054.616
------------------------------------------------------------------
```

```
2024-03-19 23:43:41 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 5.900, warmups (per chain): 1000 and samples(per chain): 1000.
```

```
WBIC value with 5 regressors: 1059.234
------------------------------------------------------------------
```

```
2024-03-19 23:43:53 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 5.900, warmups (per chain): 1000 and samples(per chain): 1000.
```

```
WBIC value with 6 regressors: 1062.122
------------------------------------------------------------------
```

```
2024-03-19 23:44:05 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 5.900, warmups (per chain): 1000 and samples(per chain): 1000.
```

```
WBIC value with 7 regressors: 1063.653
------------------------------------------------------------------
```

```
2024-03-19 23:44:18 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 5.900, warmups (per chain): 1000 and samples(per chain): 1000.
```

```
WBIC value with 8 regressors: 1071.788
------------------------------------------------------------------
```

```
2024-03-19 23:44:29 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 5.900, warmups (per chain): 1000 and samples(per chain): 1000.
```

```
WBIC value with 9 regressors: 1077.169
------------------------------------------------------------------
WBIC value with 10 regressors: 1082.966
------------------------------------------------------------------
CPU times: user 18.9 s, sys: 962 ms, total: 19.9 s
Wall time: 1min 58s
```

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)

(continued from previous 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)
```

```
2024-03-19 23:44:41 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 5.900, warmups (per chain): 225 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
```

```
2024-03-19 23:44:42 - 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): 1145.178
```

```
2024-03-19 23:44:43 - orbit - INFO - step    0 loss = 324.86, scale = 0.11507
INFO:orbit:step    0 loss = 324.86, scale = 0.11507
2024-03-19 23:44:52 - orbit - INFO - step  100 loss = 116.22, scale = 0.50606
INFO:orbit:step  100 loss = 116.22, scale = 0.50606
2024-03-19 23:45:02 - orbit - INFO - step  200 loss = 116.05, scale = 0.49671
INFO:orbit:step  200 loss = 116.05, scale = 0.49671
2024-03-19 23:45:12 - orbit - INFO - step  300 loss = 116.43, scale = 0.51436
INFO:orbit:step  300 loss = 116.43, scale = 0.51436
2024-03-19 23:45:12 - orbit - INFO - Sampling (CmdStanPy) with chains: 4, cores: 8,␣
↪temperature: 5.900, warmups (per chain): 225 and samples(per chain): 25.
INFO:orbit:Sampling (CmdStanPy) with chains: 4, cores: 8, temperature: 5.900, warmups␣
↪(per chain): 225 and samples(per chain): 25.
```

```
WBIC value for LGT model (pyro SVI): 1121.054
```

```
chain 1 |          | 00:00 Status
```

```
chain 2 |          | 00:00 Status
```

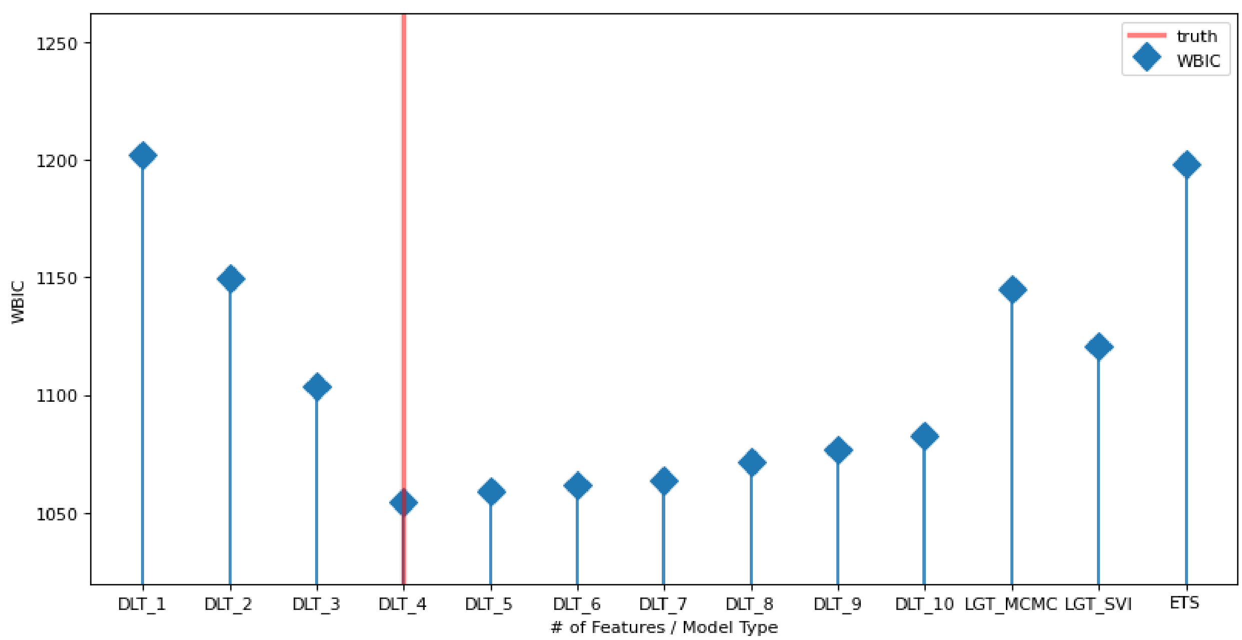
```
chain 3 |          | 00:00 Status
```

```
chain 4 |          | 00:00 Status
```

```
WBIC value for ETS model: 1197.855
CPU times: user 1min 7s, sys: 3min 21s, total: 4min 28s
Wall time: 31.2 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_{}".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)
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("WBIC")
ax.set_xlabel("# of Features / Model Type")
ax.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.

[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)
```

```
2024-03-19 23:45:12 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:45:12 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:45:12 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
```

```
BIC value with 1 regressors: 1247.444
-------------------------------------------------------------------
BIC value with 2 regressors: 1191.892
-------------------------------------------------------------------
BIC value with 3 regressors: 1139.408
-------------------------------------------------------------------
```

```
2024-03-19 23:45:13 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:45:13 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:45:13 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
```

```
BIC value with 4 regressors: 1081.432
-------------------------------------------------------------------
BIC value with 5 regressors: 1080.755
-------------------------------------------------------------------
```

```
2024-03-19 23:45:13 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:45:13 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
```

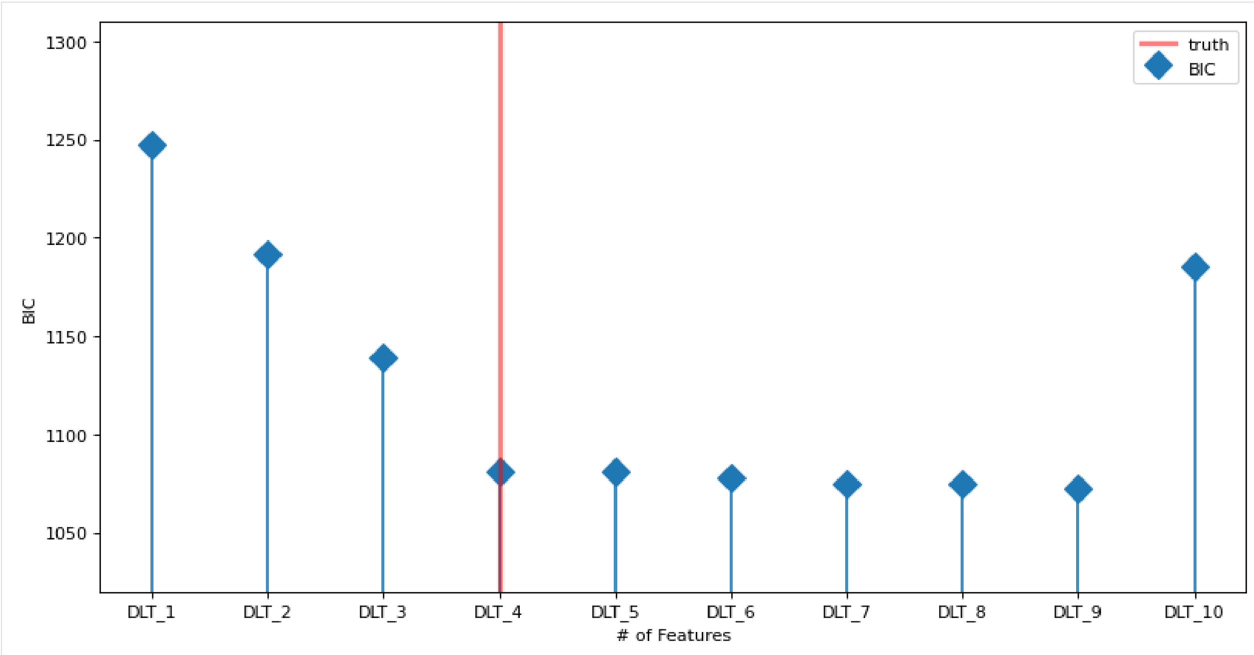
```
BIC value with 6 regressors: 1077.871
------------------------------------------------------------------
BIC value with 7 regressors: 1074.893
------------------------------------------------------------------
```

```
2024-03-19 23:45:13 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
2024-03-19 23:45:13 - orbit - INFO - Optimizing (CmdStanPy) with algorithm: LBFGS.
INFO:orbit:Optimizing (CmdStanPy) with algorithm: LBFGS.
```

```
BIC value with 8 regressors: 1074.881
------------------------------------------------------------------
BIC value with 9 regressors: 1072.668
------------------------------------------------------------------
BIC value with 10 regressors: 1185.180
------------------------------------------------------------------
CPU times: user 110 ms, sys: 210 ms, total: 320 ms
Wall time: 1.11 s
```

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.

```
[9]: 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

1. Watanabe Sumio (2010). "Asymptotic Equivalence of Bayes Cross Validation and Widely Applicable Information Criterion in Singular Learning Theory". Journal of Machine Learning Research. 11: 3571–3594.
2. McElreath Richard (2015). "Statistical Rethinking: A Bayesian course with examples in R and Stan" Secound Ed. 193-221.
3. Vehtari Aki, Gelman Andrew, Gabry Jonah (2016) "Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC"

CHAPTER

TWENTY

EDA UTILITIES

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__)
```

```
1.1.4.6
```

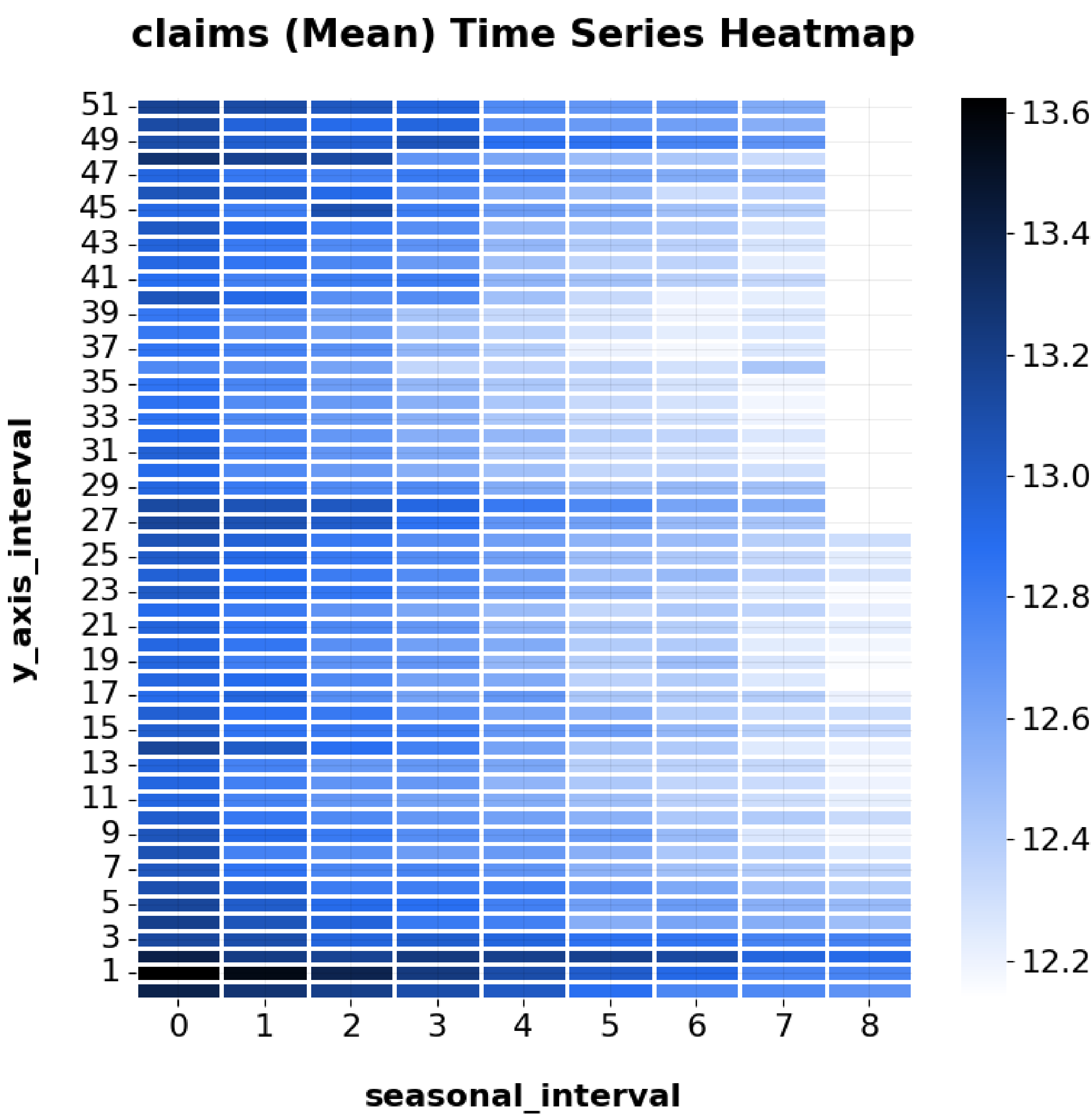
[3]:
```
df = load_iclaims()
df['week'] = pd.to_datetime(df['week'])
```

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

[4]:
```
        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
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  0.110445
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

[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

[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

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

[9]:
```
           week        variable     value
0    2010-01-03  trend.unemploy  0.219882
1    2010-01-10  trend.unemploy  0.219882
2    2010-01-17  trend.unemploy  0.236143
3    2010-01-24  trend.unemploy  0.203353
4    2010-01-31  trend.unemploy  0.134360
...         ...             ...       ...
2210 2018-05-27             vix -0.175192
2211 2018-06-03             vix -0.275119
2212 2018-06-10             vix -0.291676
2213 2018-06-17             vix -0.152422
2214 2018-06-24             vix  0.003284

[2215 rows x 3 columns]
```

[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

%matplotlib inline
```

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

```
1.1.4.6
```

21.1 Trend

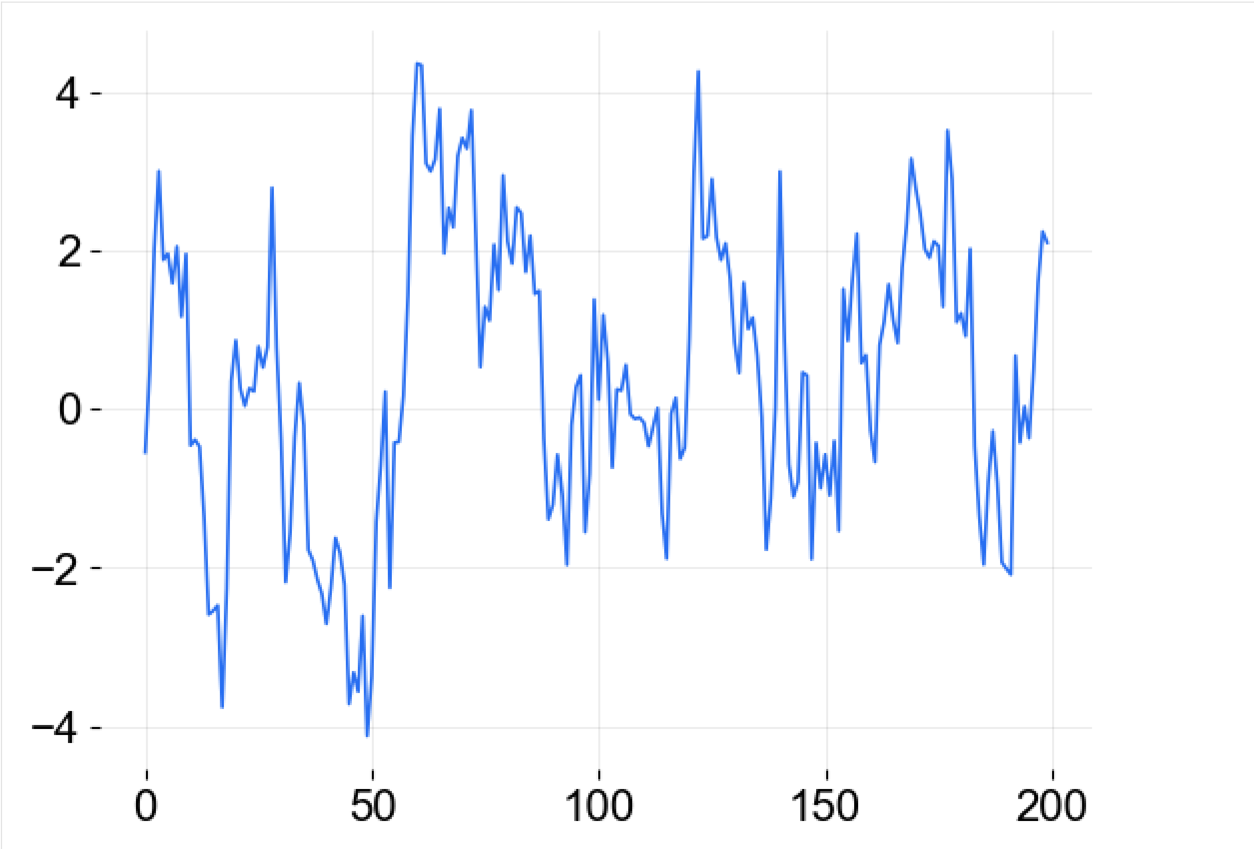
21.1.1 Random Walk

[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

reference for the ARMA process: https://www.statsmodels.org/stable/generated/statsmodels.tsa.arima_process.ArmaProcess.html

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

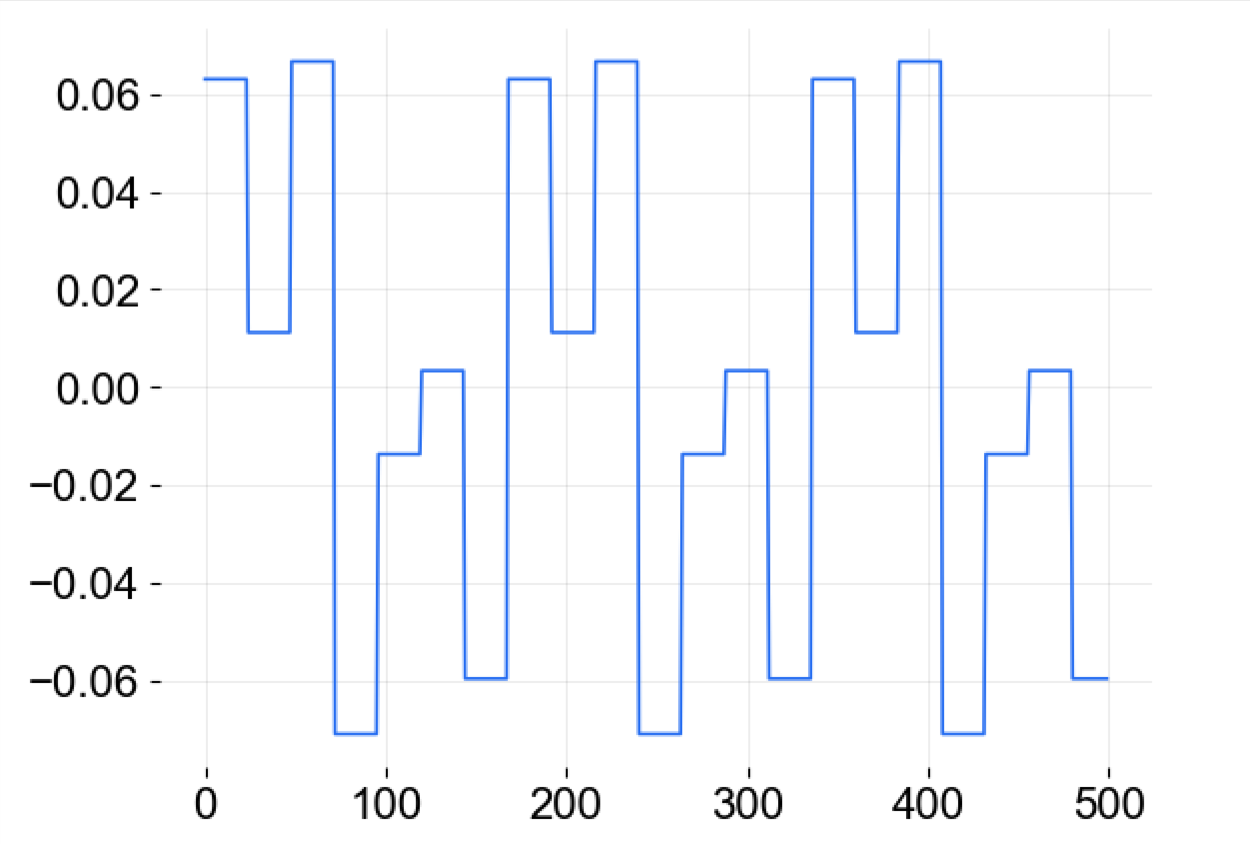


21.2 Seasonality

21.2.1 Discrete

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

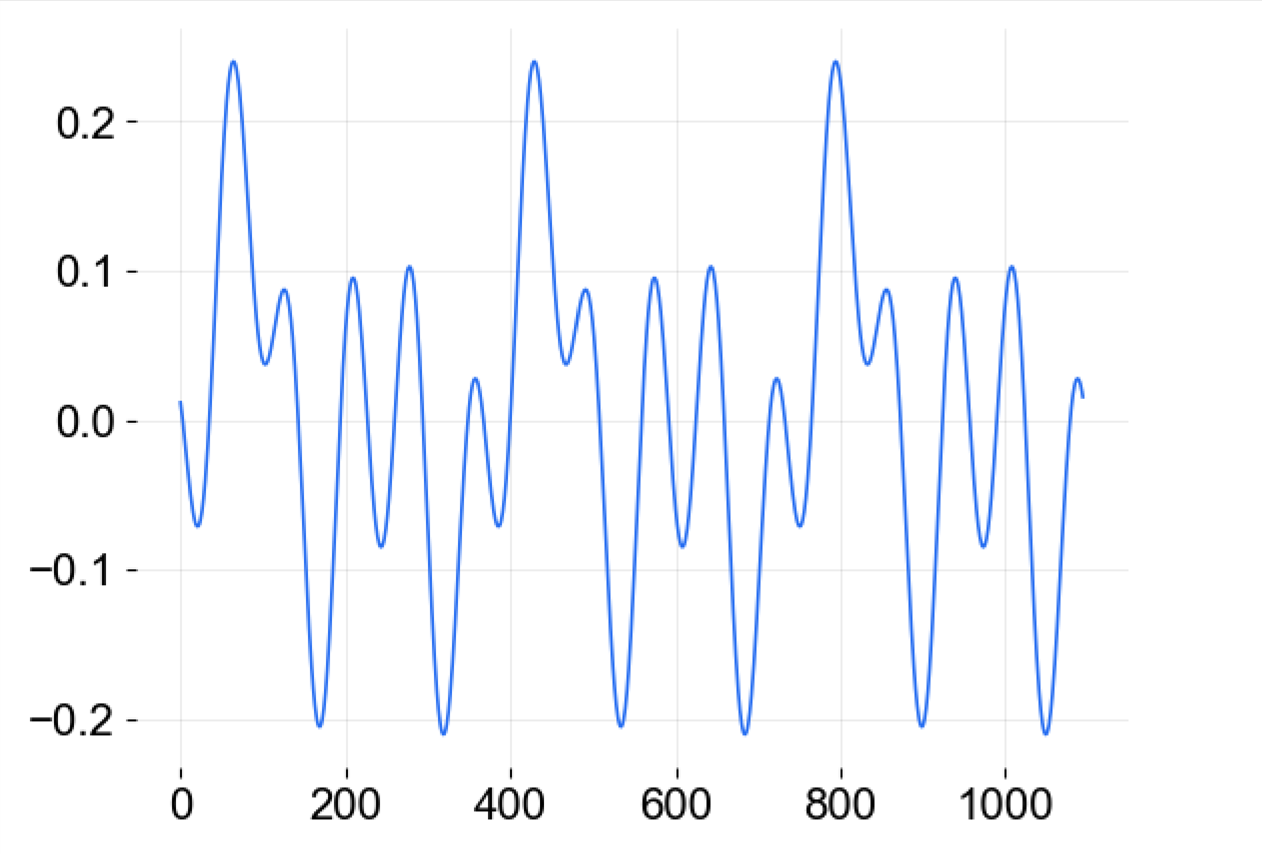
[5]:
```
ds = make_seasonality(500, seasonality=7, duration=24, method='discrete', seed=2020)
_ = plt.plot(ds, color = OrbitPalette.BLUE.value)
```



21.2.2 Fourier

generating a sine-cosine wave seasonality for a annual seasonality(=365) using fourier series

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




```
[7]: fs
```

```
[7]: array([0.01162034, 0.00739299, 0.00282248, ..., 0.02173615, 0.01883928,
            0.01545216])
```

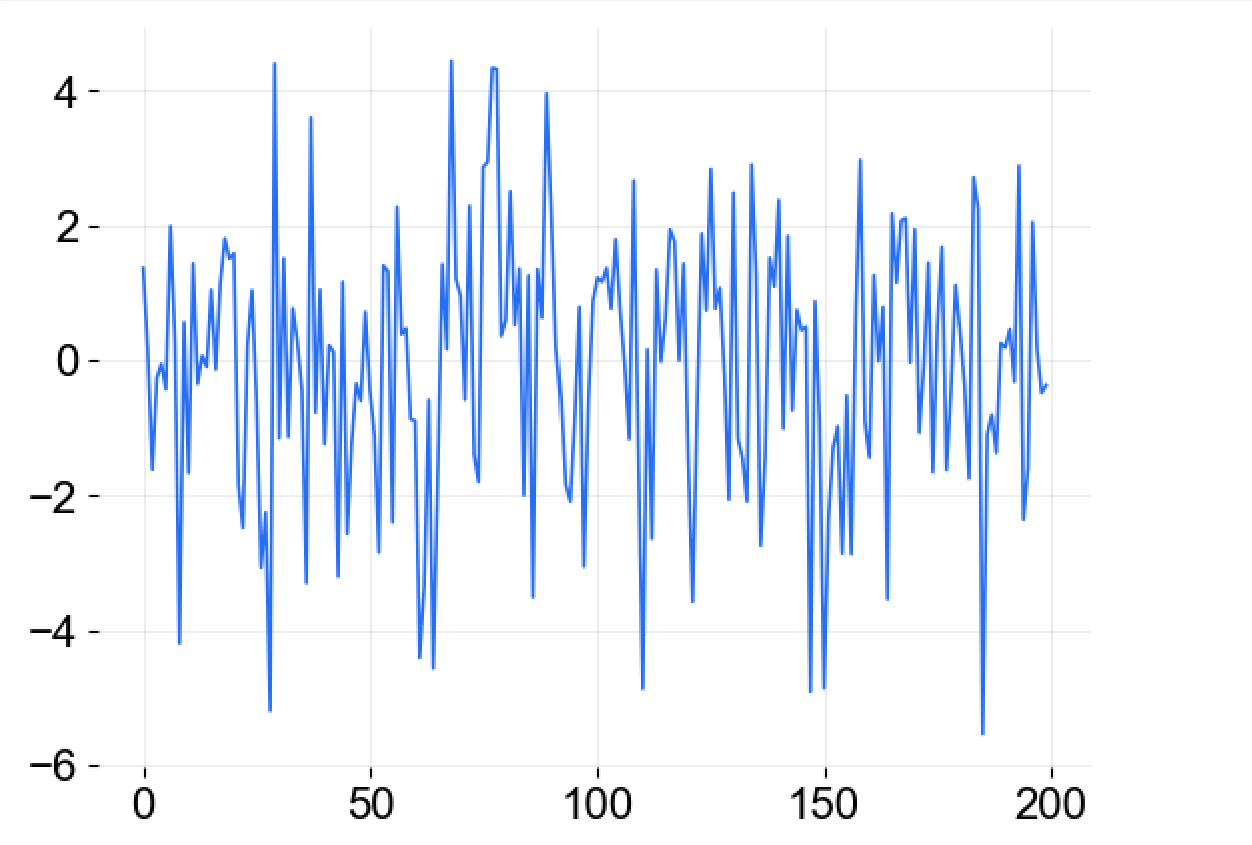
21.3 Regression

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

```
[8]: # define the regression coefficients
coefs = [0.1, -.33, 0.8]
```

```
[9]: x, y, coefs = make_regression(200, coefs, scale=2.0, seed=2020)
```

```
[10]: _ = 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_)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[12], line 1
----> 1 from sklearn.linear_model import LinearRegression
      3 # check if get the coefficients as set up
      4 reg = LinearRegression().fit(x, y)

ModuleNotFoundError: No module named 'sklearn'
```

CHAPTER

TWENTYTWO

OTHER UTILITIES

22.1 Generating Full Span of multiple time-series

[1]:
```
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',
               '2020-05-31', '2020-06-30', '2020-07-31', '2020-08-31',
               '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',
               '2022-01-31', '2022-02-28', '2022-03-31', '2022-04-30',
               '2022-05-31', '2022-06-30', '2022-07-31', '2022-08-31',
               '2022-09-30', '2022-10-31', '2022-11-30', '2022-12-31'],
              dtype='datetime64[ns]', freq='ME')
```

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)
df_base.head(5)
```

```
(360, 3)
```

[3]:
```
  key         dt         x
0  x0 2020-01-31  0.357236
1  x0 2020-02-29 -1.172618
2  x0 2020-03-31 -0.852877
3  x0 2020-04-30  1.080290
4  x0 2020-05-31 -0.641044
```

22.2 Regenerate Multiple Timeseries with Missing rows

Create missing rows.

[4]:
```
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)
```

[4]:
```
  key         dt         x
0  x0 2020-01-31  0.357236
1  x0 2020-02-29 -1.172618
2  x0 2020-03-31 -0.852877
3  x0 2020-04-30  1.080290
4  x0 2020-05-31 -0.641044
```

Use `regenerate_base_df` to regenerate the base dataframe.

[5]:
```
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.

[6]:
```
new_df_base.iloc[drop_idx]
```

[6]:
```
            dt key   x
286 2022-11-30  x7 NaN
274 2021-11-30  x7 NaN
75  2020-04-30  x2 NaN
135 2022-04-30  x3 NaN
43  2020-08-31  x1 NaN
```

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

[7]:
```
new_df_base = regenerate_base_df(df_missing, time_col, key_col, val_cols=['x'], fill_
↪na=0)
```

[8]:
```
new_df_base.iloc[drop_idx]
```

[8]:
```
            dt key    x
286 2022-11-30  x7  0.0
274 2021-11-30  x7  0.0
```

(continues on next page)

(continued from previous page)

```
75  2020-04-30  x2  0.0
135 2022-04-30  x3  0.0
43  2020-08-31  x1  0.0
```

CHAPTER TWENTYTHREE

BUILD YOUR OWN MODEL

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 `fit()` and `predict()` concrete by supplying the `fitter` as a file (**CmdStanPy**) or a callable class (**Pyro**) and the internal `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**). `orbit` is designed to support various APIs such as **CmdStanPy** and **Pyro** (hopefully PyMC3, Numpyro in the future!). The logic separating the call of different APIs with different interface is done by the **Estimator** class which is further inherited in `PyroEstimator` and `StanEstimator`.

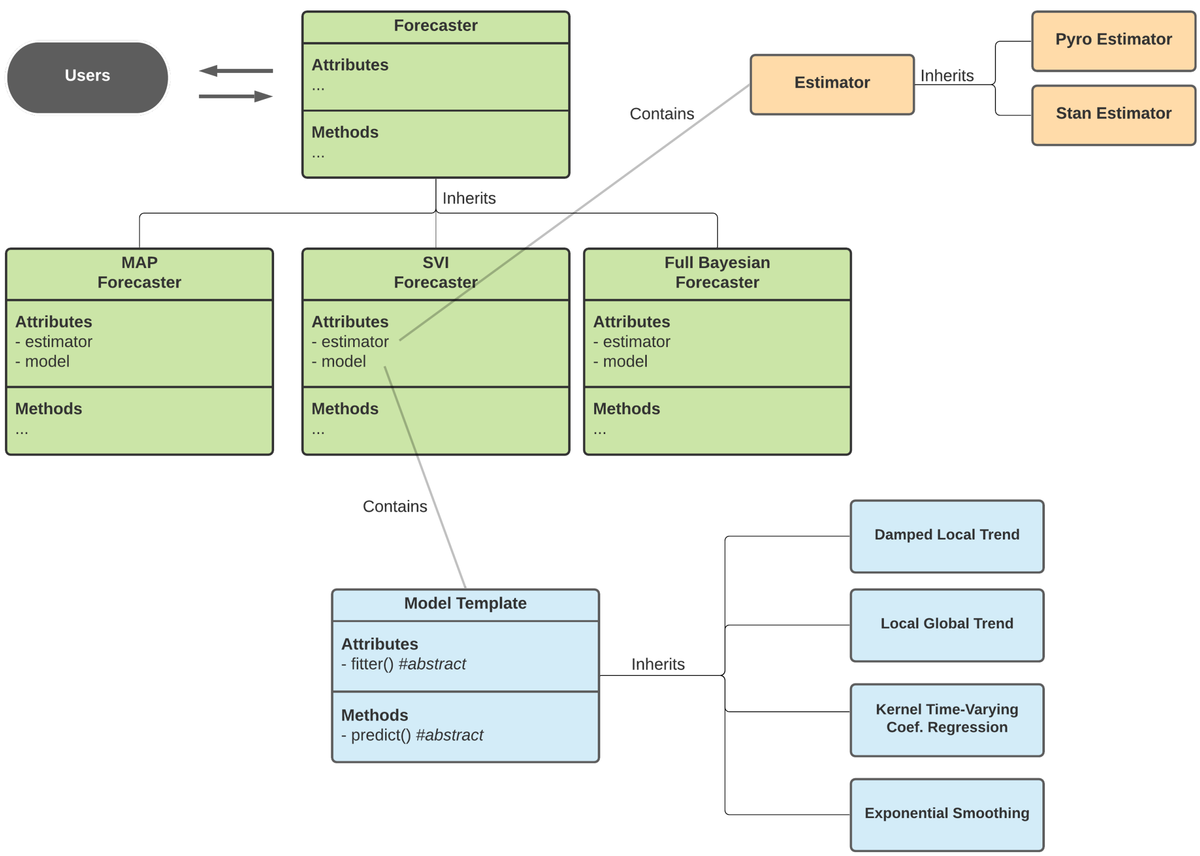


Diagram above shows the interaction across classes under the Orbit package design.

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 α is the intercept, β is the coefficients matrix and ϵ 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__)
```

```
1.1.4.6
```

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)

(continued from previous page)

```
            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**.

[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)

(continued from previous 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)
```

[7]:

	y	x1	x2	week
0	2.382337	0.345584	0.000000	2016-01-04
1	2.812929	0.330437	-0.000000	2016-01-11
2	3.600130	0.905356	0.446375	2016-01-18
3	-0.884275	-0.000000	0.581118	2016-01-25
4	2.704941	0.364572	0.294132	2016-02-01

[8]:
```
test_size = 20
train_df = df[:-test_size]
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`.

[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
```

[11]:
```
<orbit.forecaster.svi.SVIForecaster at 0x2a6164950>
```

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

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

```
2024-03-19 23:37:55 - orbit - INFO - Using SVI (Pyro) with steps: 501, samples: 100,␣
↪learning rate: 0.1, learning_rate_total_decay: 1.0 and particles: 100.
2024-03-19 23:37:56 - orbit - INFO - step    0 loss = 27333, scale = 0.077497
INFO:orbit:step    0 loss = 27333, scale = 0.077497
2024-03-19 23:37:58 - orbit - INFO - step  100 loss = 12594, scale = 0.0092399
INFO:orbit:step  100 loss = 12594, scale = 0.0092399
2024-03-19 23:38:00 - orbit - INFO - step  200 loss = 12591, scale = 0.0095592
INFO:orbit:step  200 loss = 12591, scale = 0.0095592
2024-03-19 23:38:03 - orbit - INFO - step  300 loss = 12593, scale = 0.0094199
INFO:orbit:step  300 loss = 12593, scale = 0.0094199
2024-03-19 23:38:06 - orbit - INFO - step  400 loss = 12591, scale = 0.0092691
INFO:orbit:step  400 loss = 12591, scale = 0.0092691
2024-03-19 23:38:10 - orbit - INFO - step  500 loss = 12591, scale = 0.0095463
INFO:orbit:step  500 loss = 12591, scale = 0.0095463
```

[12]:
```
<orbit.forecaster.svi.SVIForecaster at 0x2a6164950>
```

23.3.3 Compare Coefficients with Truth

[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.

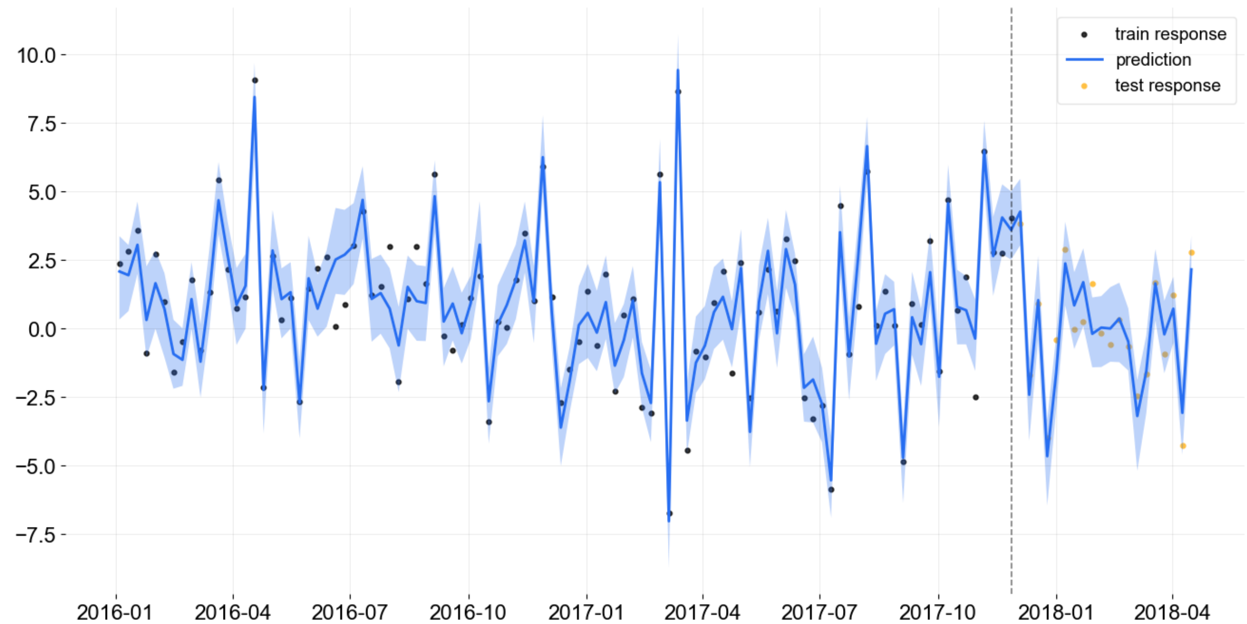
[14]:
```
print("True Coef: {:.3f}, {:.3f}".format(coefs[0], coefs[1]) )
estimated_coef = np.median(estimated_weights, axis=0)
print("Estimated Coef: {:.3f}, {:.3f}".format(estimated_coef[0], estimated_coef[1]))
```

```
True Coef: 3.000, -1.000
Estimated Coef: 2.956, -0.976
```

23.3.4 Examine Forecast Accuracy

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

[16]:
```python
_ = 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.

CHAPTER

TWENTYFOUR

API DOCS

24.1 orbit package

24.1.1 Subpackages

orbit.constants package

Submodules

orbit.constants.constants module

class orbit.constants.constants.**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

ACTUAL = 'actual'

DATE = 'date'

METRIC_NAME = 'metric_name'

METRIC_VALUES = 'metric_values'

PREDICTED = 'prediction'

SPLIT_KEY = 'split_key'

TEST_ACTUAL = 'test_actual'

TEST_PREDICTED = 'test_prediction'

TRAIN_ACTUAL = 'train_actual'

TRAIN_FLAG = 'training_data'

TRAIN_METRIC_FLAG = 'is_training_metric'

TRAIN_PREDICTED = 'train_prediction'

class orbit.constants.constants.**CompiledStanModelPath**

Bases: object

the directory path for compliled 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'

MEDIAN = 'median'

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*) orbit.diagnostics.plot.**params_comparison_boxplot**(*data*, *var_names*, *model_names*, *color_list=[(0.12156862745098039, 0.4666666666666667, 0.7058823529411765), (1.0, 0.4980392156862745, 0.054901960784313725), (0.17254901960784313, 0.6274509803921569, 0.17254901960784313), (0.8392156862745098, 0.15294117647058825, 0.1568627450980392), (0.5803921568627451, 0.403921568627451, 0.7411764705882353), (0.5490196078431373, 0.33725490196078434, 0.29411764705882354), (0.8901960784313725, 0.4666666666666667, 0.7607843137254902), (0.4980392156862745, 0.4980392156862745, 0.4980392156862745), (0.7372549019607844, 0.7411764705882353, 0.13333333333333333), (0.09019607843137255, 0.7450980392156863, 0.8117647058823529)]*, *title='Params Comparison'*, *fig_size=(10, 6)*, *box_width=0.1*, *box_distance=0.2*, *showfliers=False*)

compare the distribution of parameters from different models uisng a boxplot. :param data: a list of dict with keys as the parameters of interest :param var_names: a list of strings, the labels of the parameters to compare :param model_names: a list of strings, the names of models to compare :param color_list: a list of strings, the color to use for differentiating models :param title: string

the title of the chart

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

orbit.diagnostics.plot.**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 to plot and visualize the prediction results from back testing.

bt_pred_df
[data frame] the output of *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
[str] title of the plot

fontsize: int; optional
fontsize of the title

path
[string] 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_xylim=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
[pd.DataFrame] predicted data response data frame. two columns required: actual_col and pred_col. If user provide pred_percentiles_col, it needs to include them as well.

date_col
[str] 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
[str; optional] title of the plot

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

path
[str; 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.

Return type
matplotlib axes object

`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

See http://docs.pyro.ai/en/stable/_modules/pyro/optim/clipped_adam.html for optimizer details

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 *pyro.infer.autoguide*; recommend a larger number of small dataset
- **kwargs** – Additional *PyroEstimator* class args

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 *num_sample* x 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**(*num_warmup=900*, *num_sample=100*, *chains=4*, *cores=8*, *algorithm=None*, *suppress_stan_log=True*, ***kwargs*)

Bases: `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 *BaseEstimator* class args

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

Parameters

stan_mcmc_args (*dict*) – Supplemental stan mcmc args to pass to CmdStandPy.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 module

`orbit.models.lgt.LGT`(*seasonality=None*, *seasonality_sm_input=None*, *level_sm_input=None*, *regressor_col=None*, *regressor_sign=None*, *regressor_beta_prior=None*, *regressor_sigma_prior=None*, *regression_penalty='fixed_ridge'*, *lasso_scale=0.5*, *auto_ridge_scale=0.5*, *slope_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.
- **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**(*seasonality=None*, *seasonality_sm_input=None*, *level_sm_input=None*, *regressor_col=None*, *regressor_sign=None*, *regressor_beta_prior=None*, *regressor_sigma_prior=None*, *regression_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

- **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*)

A simple function to tell whether the passed in df is an empty dataframe or not. :param df: given input dataframe :type df: pd.DataFrame

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`

exception `orbit.exceptions.`**`ForecasterException`**

Bases: `Exception`

exception `orbit.exceptions.`**`IllegalArgument`**

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 procedures and dependencies (https://github.com/uber/orbit/pull/833), (https://github.com/uber/orbit/pull/835), (https://github.com/uber/orbit/pull/821)
- 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 exisiting 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

CHAPTER

TWENTYSIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX

H

I

K

L

M

N

O

P

R

S

T

U

W

Y