mbtr

Release 0.1.3

Aug 10, 2022

Contents:

CHAPTER 1

Introduction

MBTR is a python package for multivariate boosted tree regressors trained in parameter space. The package can handle arbitrary multivariate losses, as long as their gradient and Hessian are known. Gradient boosted trees are competition-winning, general-purpose, non-parametric regressors, which exploit sequential model fitting and gradient descent to minimize a specific loss function. The most popular implementations are tailored to univariate regression and classification tasks, precluding the possibility of capturing multivariate target cross-correlations and applying conditional penalties to the predictions. This package allows to arbitrarily regularize the predictions, so that properties like smoothness, consistency and functional relations can be enforced.

1.1 Installation

The package can be installed via pip:

```
pip3 install mbtr
```

or cloned from the offical repository mbtr[1]

1.2 Citation

If you make use of this software for your work, we would appreciate it if you would cite us:

```
@article{nespoli2020multivariate,
  title={Multivariate Boosted Trees and Applications to Forecasting and Control},
  author={Nespoli, Lorenzo and Medici, Vasco},
  journal={arXiv preprint arXiv:2003.03835},
  year={2020}
}
```

[1] https://github.com/supsi-dacd-isaac/mbtr

1.3 Aknowledgments

The authors would like to thank the Swiss Federal Office of Energy (SFOE) and the Swiss Competence Center for Energy Research - Future Swiss Electrical Infrastructure (SCCER-FURIES), for their financial and technical support to this research work.

CHAPTER 2

mbtr package

2.1 Submodules

2.2 mbtr.losses module

class mbtr.losses.**FourierLoss**(*lambda_weights: float = 0.1*, *lambda_leaves: float = 0.1*, ***loss_kwargs*)

Bases: *mbtr.losses.Loss*

Loss for the Fourier regression:

$$\mathcal{L} = \|y - Px\|_2^2$$

where P is the projection matrix:

$$P = \left[\left[\cos\left(k\frac{2\pi t}{n_t}\right)^T, \sin\left(k\frac{2\pi t}{n_t}\right)^T\right]^T\right]_{k\in\mathcal{K}}$$

eval_optimal_loss(*G2*, *H*)

Evaluate the optimal loss (using response obtained minimizing the second order loss approximation).

Parameters

- **G2** – squared sum of gradients in the current leaf
- **H** – sum of Hessians diags in the current leaf

Returns optimal loss, scalar

eval_optimal_response(*G*, *H*)

Evaluate optimal response, given G and H.

Parameters

- **G** – mean gradient for the current leaf.
- **H** – mean Hessian for the current leaf.

Returns optimal response under second order loss approximation.

get_initial_guess(*y*)
Return an initial guess for the prediciton. This can be loss-specific.

Parameters **y** – target matrix of the training set

Returns np.ndarray with initial guess

projection_matrix
Return projection matrix for the Fourier coefficient estimation.

Parameters **n** – number of observations

Returns projection matrix P, (2*n_harmonics, n_t) where n_harmonics is the number of harmonics to fit, n_t the

target dimension

required_pars = ['n_harmonics']

set_dimension(*n_dim*)
Initialize all the properties which depends on the dimension of the target

Parameters **n_dims** – dimension of the target

Returns None

class mbtr.losses.**LatentVariable**(*lambda_weights: float = 0.1, lambda_leaves: float = 0.1, **loss_kwargs*)

Bases: *mbtr.losses.Loss*

Loss for the hierarchical reconciliation problem, in the form:

$$\mathcal{L} = \|y - Sx\|_2^2$$

where S is the hierarchy matrix. The initial guess is the mean of the last columns of y.

compute_H_inv

compute_fast_H_hat

eval_optimal_loss(*yy*, *H*)
Evaluate the optimal loss (using response obtained minimizing the second order loss approximation).

Parameters

- **G2** – squared sum of gradients in the current leaf
- **H** – sum of Hessians diags in the current leaf

Returns optimal loss, scalar

eval_optimal_response(*G*, *H*)
Evaluate optimal response, given G and H.

Parameters

- **G** – mean gradient for the current leaf.
- **H** – mean Hessian for the current leaf.

Returns optimal response under second order loss approximation.

get_grad_and_hessian_diags(*y*, *y_hat*, *iteration*, *leaves_idx*)

Return the loss gradient and loss Hessian's diagonals based on the current model estimation y_hat and target y matrices. Instead of returning the full Hessian (a 3rd order tensor), the method returns only the Hessian diagonals for each observation, stored in a (n_obs, n_t) matrix. These diagonals are then used by the loss to reconstruct the full Hessian with appropriate dimensions and structure. Currently, full Hessians inferred by data are not supported.

Parameters

- **y** – target matrix (n_obs, n_t)
- **y_hat** – current target estimation matrix (n_obs, n_t)
- **iteration** – current number of iteration, generally not needed
- **leaves_idx** – leaves' indexes for each observation in y, (n_obs, 1). This is needed for example by *mbtr.losses.QuadraticQuantileLoss*.

Returns grad, hessian_diags tuple, each of which is a (n_obs, n_t) matrix

get_initial_guess(*y*)

The initial guess is generated from the last columns of the target matrix, as:

$$y_0 = (\mathbb{E}y_b)\, S^T$$

where $\mathbb{E}$ is the expectation (row-mean), S is the hierarchy matrix, and y_b stands for the last columns of y, with dimension (n_obs, n_b), where n_b is the number of bottom series.

Parameters **y** – target matrix of the training set

Returns np.ndarray with initial guess

required_pars = ['S', 'precision']

set_dimension(*n_dims*)

For the latent loss the number of dimensions is equal to the second dimension of the S matrix, and must not be inferred from the target

class mbtr.losses.LinRegLoss(*lambda_weights: float = 0.1*, *lambda_leaves: float = 0.1*, ***loss_kwargs*)

Bases: *mbtr.losses.Loss*

eval_optimal_loss(*G*, *x*)

Evaluate the optimal loss (using response obtained minimizing the second order loss approximation).

Parameters

- **G** – gradient for the current leaf.
- **x** – linear regression features for the current leaf.

Returns optimal loss, scalar

eval_optimal_response(*G*, *x*)

Evaluate optimal response, given G and x. This is done computing a Ridge regression with intercept

$$w = \left(\tilde{x}^T \tilde{x} + \lambda I\right)^{-1} \left(\tilde{x}^T G\right)$$

where $\tilde{x}$ is the x matrix augmented with an unitary column and λ is the Ridge coefficient.

Parameters

- **G** – gradient for the current leaf.

- **x** – linear regression features for the current leaf.

Returns optimal response under second order loss approximation.

set_dimension(*n_dim*)
Initialize all the properties which depends on the dimension of the target

Parameters **n_dims** – dimension of the target

Returns None

class mbtr.losses.Loss(*lambda_weights: float = 0.1*, *lambda_leaves: float = 0.1*, ***loss_kwargs*)
Bases: object

Loss function class. A loss is defined by its gradient and Hessians. Note that if your specific loss funciton requires some additional argument, you can specify it in the required_pars. Upon instantiation, this list will be used to check if loss_kwargs contains all the needed parameters. Each class inheriting from *mbtr.losses.Loss* must provide an H_inv method, computing the inverse of the Hessian.

Parameters

- **lambda_weights** – quadratic penalization parameter for the leaves weights
- **lambda_leaves** – quadratic penalization parameter for the number of leaves
- **loss_kwargs** – additional parameters needed for a specific loss type

H_inv
Computes the inverse of the Hessian, given the Hessian's diagonal of the current leave. The default implements MSE inverse.

Parameters **H** – current leaf Hessian's diagonal (n_t)

Returns inv(H), (n_t, n_t)

eval(*y*, *y_hat*, *trees*)
Evaluates the overall loss, which is composed by the tree's loss plus weights and total leaves penalizations

Parameters

- **y** – observations
- **y_hat** – current :class: *mbtr.MBT* estimations
- **trees** – array of fitted trees up to the current iteartion

Returns tree loss and regularizations loss tuple, scalars

eval_optimal_loss(*G2*, *H*)
Evaluate the optimal loss (using response obtained minimizing the second order loss approximation).

Parameters

- **G2** – squared sum of gradients in the current leaf
- **H** – sum of Hessians diags in the current leaf

Returns optimal loss, scalar

eval_optimal_response(*G*, *H*)
Evaluate optimal response, given G and H.

Parameters

- **G** – mean gradient for the current leaf.

- **H** – mean Hessian for the current leaf.

Returns optimal response under second order loss approximation.

get_grad_and_hessian_diags(*y*, *y_hat*, *iteration*, *leaves_idx*)
Return the loss gradient and loss Hessian's diagonals based on the current model estimation y_hat and target y matrices. Instead of returning the full Hessian (a 3rd order tensor), the method returns only the Hessian diagonals for each observation, stored in a (n_obs, n_t) matrix. These diagonals are then used by the loss to reconstruct the full Hessian with appropriate dimensions and structure. Currently, full Hessians inferred by data are not supported.

Parameters

- **y** – target matrix (n_obs, n_t)
- **y_hat** – current target estimation matrix (n_obs, n_t)
- **iteration** – current number of iteration, generally not needed
- **leaves_idx** – leaves' indexes for each observation in y, (n_obs, 1). This is needed for example by *mbtr.losses.QuadraticQuantileLoss*.

Returns grad, hessian_diags tuple, each of which is a (n_obs, n_t) matrix

get_initial_guess(*y*)
Return an initial guess for the prediciton. This can be loss-specific.

Parameters **y** – target matrix of the training set

Returns np.ndarray with initial guess

required_pars = []

set_dimension(*n_dims*)
Initialize all the properties which depends on the dimension of the target

Parameters **n_dims** – dimension of the target

Returns None

tree_loss(*y*, *y_hat*)
Compute the tree loss (without penalizations)

Parameters

- **y** – observations of the target on the traning set
- **y_hat** – current estimation of the MBT

Returns tree loss

class mbtr.losses.MSE(*lambda_weights: float = 0.1*, *lambda_leaves: float = 0.1*, ***loss_kwargs*)
Bases: *mbtr.losses.Loss*

Mean Squared Error loss, a.k.a. L2, Ordinary Least Squares.

$$\mathcal{L} = \|y - w\|_2^2 + \frac{1}{2} w^T \Lambda w$$

where Λ is the quadratic punishment matrix.

Parameters

- **lambda_weights** – quadratic penalization parameter for the leaves weights
- **lambda_leaves** – quadratic penalization parameter for the number of leaves
- **loss_kwargs** – additional parameters needed for a specific loss type

`eval_optimal_loss`(*G2*, *H*)
Evaluate the optimal loss (using response obtained minimizing the second order loss approximation).

Parameters

- `G2` – squared sum of gradients in the current leaf
- `H` – sum of Hessians diags in the current leaf

Returns optimal loss, scalar

`eval_optimal_response`(*G*, *H*)
Evaluate optimal response, given G and H.

Parameters

- `G` – mean gradient for the current leaf.
- `H` – mean Hessian for the current leaf.

Returns optimal response under second order loss approximation.

`set_dimension`(*n_dim*)
Initialize all the properties which depends on the dimension of the target

Parameters `n_dims` – dimension of the target

Returns None

`class mbtr.losses.QuadraticQuantileLoss`(*lambda_weights: float = 0.1*, *lambda_leaves: float = 0.1*, ***loss_kwargs*)
Bases: *mbtr.losses.Loss*

`H_inv`(*H*)
Computes the inverse of the Hessian, given the Hessian's diagonal of the current leave. The default implements MSE inverse.

Parameters `H` – current leaf Hessian's diagonal (n_t)

Returns inv(H), (n_t, n_t)

`exact_response`(*y*)

`get_grad_and_hessian_diags`(*y*, *y_hat*, *iteration*, *leaves_idx*)
Return the loss gradient and loss Hessian's diagonals based on the current model estimation y_hat and target y matrices. Instead of returning the full Hessian (a 3rd order tensor), the method returns only the Hessian diagonals for each observation, stored in a (n_obs, n_t) matrix. These diagonals are then used by the loss to reconstruct the full Hessian with appropriate dimensions and structure. Currently, full Hessians inferred by data are not supported.

Parameters

- `y` – target matrix (n_obs, n_t)
- `y_hat` – current target estimation matrix (n_obs, n_t)
- `iteration` – current number of iteration, generally not needed
- `leaves_idx` – leaves' indexes for each observation in y, (n_obs, 1). This is needed for example by *mbtr.losses.QuadraticQuantileLoss*.

Returns grad, hessian_diags tuple, each of which is a (n_obs, n_t) matrix

`get_initial_guess`(*y*)
The initial guess are the alpha quantiles of the target matrix y.

Parameters `y` – target matrix of the training set

Returns np.ndarray with initial guess

`required_pars = ['alphas']`

`tree_loss`(*y*, *y_hat*)
Compute the tree loss (without penalizations)

Parameters

- `y` – observations of the target on the traning set
- `y_hat` – current estimation of the MBT

Returns tree loss

`class mbtr.losses.QuantileLoss`(*lambda_weights: float = 0.1*, *lambda_leaves: float = 0.1*, ***loss_kwargs*)
Bases: *mbtr.losses.Loss*

`H_inv`(*H*)
Computes the inverse of the Hessian, given the Hessian's diagonal of the current leave. The default implements MSE inverse.

Parameters `H` – current leaf Hessian's diagonal (n_t)

Returns inv(H), (n_t, n_t)

`exact_response`(*y*)

`get_grad_and_hessian_diags`(*y*, *y_hat*, *iteration*, *leaves_idx*)
Return the loss gradient and loss Hessian's diagonals based on the current model estimation y_hat and target y matrices. Instead of returning the full Hessian (a 3rd order tensor), the method returns only the Hessian diagonals for each observation, stored in a (n_obs, n_t) matrix. These diagonals are then used by the loss to reconstruct the full Hessian with appropriate dimensions and structure. Currently, full Hessians inferred by data are not supported.

Parameters

- `y` – target matrix (n_obs, n_t)
- `y_hat` – current target estimation matrix (n_obs, n_t)
- `iteration` – current number of iteration, generally not needed
- `leaves_idx` – leaves' indexes for each observation in y, (n_obs, 1). This is needed for example by *mbtr.losses.QuadraticQuantileLoss*.

Returns grad, hessian_diags tuple, each of which is a (n_obs, n_t) matrix

`get_initial_guess`(*y*)
The initial guess are the alpha quantiles of the target matrix y.

Parameters `y` – target matrix of the training set

Returns np.ndarray with initial guess

`quantile_loss`(*y*, *q_hat*)
Quantile loss function, a.k.a. pinball loss.

$$\epsilon(y, \hat{q})_\alpha = \hat{q}_\alpha - y$$
$$\mathcal{L}(y, \hat{q})_\alpha = \epsilon(y, \hat{q})_\alpha \left(I_{\epsilon_\alpha \geq 0} - \alpha\right)$$

Parameters

- `y` – observations of the target on the traning set

- **q_hat** – current estimation matrix of the quantiles

Returns tree loss

required_pars = ['alphas']

class mbtr.losses.TimeSmoother(*lambda_weights: float = 0.1, lambda_leaves: float = 0.1, **loss_kwargs*)

Bases: *mbtr.losses.Loss*

Time-smoothing loss function. Penalizes the time-derivative of the predicted signal.

$$\mathcal{L} = \frac{1}{2}\|y - w\|_2^2 + \frac{1}{2} w^T \left(\lambda_s D^T D + \lambda I\right) w$$

where D is the second order difference matrix

$$D = \begin{bmatrix} 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \\ 1 & & & & & \end{bmatrix}$$

and λ_s is the coefficient for the quadratic penalization of time-derivatives. Required parameters: lambda_smooth: coefficient for the quadratic penalization of time-derivatives

static build_filter_mat(*n*)

Build the second order difference matrix

Parameters **n** – target dimension

Returns D, second order difference matrix

compute_fast_H_inv

required_pars = ['lambda_smooth']

set_dimension(*n_dim*)

Initialize all the properties which depends on the dimension of the target

Parameters **n_dims** – dimension of the target

Returns None

update_smoothing_mat(*smoothing_weights*)

2.3 mbtr.mbtr module

class mbtr.mbtr.MBT(*n_boosts: int = 20, early_stopping_rounds: int = 3, learning_rate: float = 0.1, val_ratio: int = 0, n_q: int = 10, min_leaf: int = 100, loss_type: str = 'mse', lambda_weights: float = 0.1, lambda_leaves: float = 0.1, verbose: int = 0, refit=True, **loss_kwargs*)

Bases: `object`

Multivariate Boosted Tree class. Fits a multivariate tree using boosting.

Parameters

- **n_boosts** – maximum number of boosting rounds. Default: 20

- **early_stopping_rounds** – if the total loss is non-decreasing after early_stopping_rounds, stop training. The final model is the one which achieved the lowest loss up to the final iteration. Default: 3.
- **learning_rate** – in [0, 1]. A learning rate < 1 helps reducing overfitting. Default: 0.1.
- **val_ratio** – in [0,1]. If provided, the early stop is triggered by the loss computed on a validation set, randomly extracted from the training set. The length of the validation set is val_ratio * len (training set). Default: 0.
- **n_q** – number of quantiles for the split search. Default: 10.
- **min_leaf** – minimum number of observations in one leaf. This parameter greatly affect generalization abilities. Default: 100.
- **loss_type** – loss type for choosing the best splits. Currently the following losses are implemented:

  mse: mean squared error loss, a.k.a. L2, ordinary least squares

  time_smoother: mse with an additional penalization on the second order differences of the response function. Requires to pass also lambda_smooth parameter.

  latent_variable: generate response function of dimension n_t from an arbitrary linear combination of n_r responses. This requires to pass also S and precision pars.

  linear_regression: mse with linear response function. Using this loss function, when calling fit and predict methods, one must also pass x_lr as additional argument, which is the matrix of features used to train the linear response inside the leaf (which can be different from the features used to grow the tree, x).

  fourier: mse with linear response function, fitted on the first n_harmonics (where the fundamental has wave-lenght equal to the target output). This requires to pass also the n_harmonics parameter.

  quantile: quantile loss function, a.k.a. pinball loss. This requires to pass also the alphas parameter, a list of quantiles to be fitted.

  quadratic_quantile: quadratic quantile loss function tailored for trees. It has a non-discontinuos derivative. This requires to pass also the alphas parameter, a list of quantiles to be fitted.
- **lambda_weights** – coefficient for the quadratic regularization of the response's parameters. Default: 0.1
- **lambda_leaves** – coefficient for the quadratic regularization of the total number of leaves. This is only used when the Tree is used as a weak learner by MBT. Default: 0.1
- **verbose** – in {0,1}. If set to 1, the MBT return fitting information at each iteration.
- **refit** – if True, if the loss function has an "exact_response" method, use it to refit the tree
- **loss_kwargs** – possible additional arguments for the loss function

fit(*x*, *y*, *do_plot=False*, *x_lr=None*)

Fits an MBT using the features specified in the matrix $x \in \mathbb{R}^{n_{obs} \times n_f}$, in order to predict the targets in the matrix $y \in \mathbb{R}^{n_{obs} \times n_t}$, where n_{obs} is the number of observations, n_f the number of features and n_t the dimension of the target.

Parameters

- **x** – feature matrix, np.ndarray.
- **y** – target matrix, np.ndarray.
- **x_lr** – features for fitting the linear response inside the leaves. This is only required if a LinearLoss is being used.

predict(*x*, *n=None*, *x_lr=None*)

Predicts the target based on the feature matrix x (and linear regression features x_lr).

Parameters

- **x** – feature matrix, np.ndarray.
- **n** – predict up to the nth fitted tree. If None, predict all the trees. Default: None
- **x_lr** – linear regression feature matrix, np.ndarray. Only required if LinearLoss has been used.

Returns target's predictions

class mbtr.mbtr.Tree(*n_q: int = 10*, *min_leaf: int = 100*, *loss_type: str = 'mse'*, *lambda_weights: float = 0.1*, *lambda_leaves: float = 0.1*, ***loss_kwargs*)

Bases: object

Tree class. Fits both univarite and multivariate targets. It implements histogram search for the decision of the splitting points.

Parameters

- **n_q** – number of quantiles for the split search
- **min_leaf** – minimum number of observations in one leaf. This parameter greatly affect generalization abilities.
- **loss_type** – loss type for choosing the best splits. Currently the following losses are implemented:

  mse: mean squared error loss, a.k.a. L2, ordinary least squares

  time_smoother: mse with an additional penalization on the second order differences of the response function. Requires to pass also lambda_smooth parameter.

  latent_variable: generate response function of dimension n_t from an arbitrary linear combination of n_r responses. This requires to pass also S and precision pars.

  linear_regression: mse with linear response function. Using this loss function, when calling fit and predict methods, one must also pass x_lr as additional argument, which is the matrix of features used to train the linear response inside the leaf (which can be different from the features used to grow the tree, x).

  fourier: mse with linear response function, fitted on the first n_harmonics (where the fundamental has wave-lenght equal to the target output). This requires to pass also the n_harmonics parameter.

  quantile: quantile loss function, a.k.a. pinball loss. This requires to pass also the alphas parameter, a list of quantiles to be fitted.

  quadratic_quantile: quadratic quantile loss function tailored for trees. It has a non-discontinuos derivative. This requires to pass also the alphas parameter, a list of quantiles to be fitted.

- **lambda_weights** – coefficient for the quadratic regularization of the response's parameters
- **lambda_leaves** – coefficient for the quadratic regularization of the total number of leaves. This is only used

when the Tree is used as a weak learner by MBT. :param loss_kwargs: possible additional arguments for the loss function

compute_loss(*G2_left*, *G2_right*, *H_left*, *H_right*, *j*)

fit(*x*, *y*, *hessian=None*, *learning_rate=1.0*, *x_lr=None*)

Fits a tree using the features specified in the matrix $x \in \mathbb{R}^{n_{obs} \times n_f}$, in order to predict the targets in the matrix $y \in \mathbb{R}^{n_{obs} \times n_t}$, where n_{obs} is the number of observations, n_f the number of features and n_t the dimension of the target.

Parameters

- **x** – feature matrix, np.ndarray.
- **y** – target matrix, np.ndarray.
- **hessian** – diagonals of the hessians $\in \mathbb{R}^{n_{obs} \times n_t}$. If None, each entry is set equal to one (this will result in the default behaviour under MSE loss). Default: None
- **learning_rate** – learning rate used by the MBT instance. Default: 1
- **x_lr** – features for fitting the linear response inside the leaves. This is only required if a LinearLoss is being used.

predict(*x*, *x_lr=None*)

Predicts the target based on the feature matrix x (and linear regression features x_lr).

param: x: feature matrix, np.ndarray. param: x_lr: linear regression feature matrix, np.ndarray.

Returns target's predictions

mbtr.mbtr.bin_sums

mbtr.mbtr.leaf_stats

2.4 mbtr.utils module

class mbtr.utils.**LightGBMMISO**(*n_estimators*, *lgb_pars=None*)

Bases: object

fit(*x*, *y*)

predict(*x*)

mbtr.utils.check_pars(*required_pars*, ***kwargs*)

mbtr.utils.download_dataset()

mbtr.utils.load_dataset()

mbtr.utils.set_figure(*size*, *subplots=(1, 1)*, *context='paper'*, *style='darkgrid'*, *font_scale=1*, *l=0.2*, *w=0.1*, *h=0.1*, *b=0.1*)

2.5 Module contents

CHAPTER 3

Examples

Some didactic examples can be found in the `mbtr/examples` dir. All the examples are based on the Hierarchical Demand Forecasting Benchmark[2], which is downloaded at the beginning of the examples. The dataset is downloaded only once, successive calls to `mbtr.ut.load_dataset()` will only read the locally downloaded file.

```
import numpy as np
import mbtr.utils as ut
from scipy.linalg import hankel
import matplotlib.pyplot as plt
from mbtr.mbtr import MBT
from mbtr.utils import set_figure

# --------------------------- Download and format data ---------------------------------
↪------------------------------
# download power data from "Hierarchical Demand Forecasting Benchmark for the␣
↪Distribution Grid" dataset
# from https://zenodo.org/record/3463137#.XruXbx9fiV7 if needed
power_data = ut.load_dataset()
```

3.1 Multivariate forecasts and linear regression

The `examples/multivariate_forecast.py` shows an example of usage of the *mbtr.losses.MSE* and *mbtr.losses.LinRegLoss* losses.

We start creating the training and test set. We use the P_mean signal as a target, and we want to predict the next day ahead only using past values from the same time series. We start downsampling the signal to one hour, and then we embed it in a 2-days matrix. The first 24 columns refers to the previous day in time, and are the signals which will be used to predict the last 24 columns:

[2] https://zenodo.org/record/3463137#.XsIwGR9fiV7

```
total_power = power_data['P_mean']['all'].values.reshape(-1,1)
# down-sample it to 1 hour, bin averages
total_power = np.mean(total_power[:len(total_power)-len(total_power) % 6].reshape(-1, 6),
↪ axis=1, keepdims=True)

# embed the signal in a 2-days matrix
total_power = hankel(total_power, np.zeros((2 * 24, 1)))

# create feature matrix and target for the training and test sets
x = total_power[:, :24]
y = total_power[:, 24:]
n_tr = int(len(x)*0.8)

x_tr, y_tr, x_te, y_te = [x[:n_tr, :], y[:n_tr, :], x[n_tr:, :], y[n_tr:, :]]
```

we can perform a visual check on the features and target signals:

```
# visual check on the first 50 samples of features and targets
fig, ax = set_figure((5,4))
y_min = np.min(y_tr[:50, :]) * 0.9
y_max = np.max(y_tr[:50, :]) * 1.1

for i in range(50):
    ax.cla()
    ax.plot(np.arange(24), x_tr[i,:], label='features')
    ax.plot(np.arange(24) + 24, y_tr[i, :], label='multivariate targets')
    ax.set_xlabel('step ahead [h]')
    ax.set_ylabel('P [kW]')
    ax.legend(loc='upper right')
    ax.set_ylim(y_min, y_max)
    plt.pause(1e-6)
plt.close('all')
```

Now we are ready to fit an MBT instance. We start fitting a mean squared error loss function:

```
print('#'*20 + '    Fitting MBT with mse loss    ' + '#'*20)
m = MBT(n_boosts=30,  min_leaf=100, lambda_weights=1e-3).fit(x_tr, y_tr, do_plot=True)
y_hat = m.predict(x_te)
```

As a comparison, we also fit also 24 different `LightGBM` instances with the utility class `mbtr.ut.LightGBMMISO` :

```
print('#'*20 + '    Fitting 24 MISO LightGBMs    ' + '#'*20)

m_lgb = ut.LightGBMMISO(30).fit(x_tr, y_tr)
y_hat_lgb = m_lgb.predict(x_te)
```

As a last comparison, we fit a linear response MBT, using the *`mbtr.losses.LinRegLoss`*. This class requires as additional input a set of features for fitting the linear response inside each leaf. In order to reduce the computational time, we only use the mean, maximum, minimum and the first and last values of the original regressors matrix `x` as features for finding the best splits of the trees.

```
x_build = np.hstack([np.mean(x, axis=1,keepdims=True), np.max(x,axis=1, keepdims=True),
```

(continues on next page)

(continued from previous page)

```
                    np.min(x, axis=1, keepdims=True), x[:,[0,23]]])
x_build_tr, x_build_te = [x_build[:n_tr, :],x_build[n_tr:, :]]
m_lin = MBT(loss_type='linear_regression', n_boosts=30,  min_leaf=1500,
            lambda_weights=1e-3).fit(x_build_tr, y_tr,x_lr=x_tr, do_plot=True)
y_hat_lin = m_lin.predict(x_build_te, x_lr=x_te)
```

We can now plot some results:

```
fig, ax = set_figure((5,4))
y_min = np.min(y_tr[:150, :]) * 0.9
y_max = np.max(y_tr[:150, :]) * 1.1


for i in range(150):
    ax.cla()
    ax.plot(np.arange(24), y_te[i,:], label='test')
    ax.plot(y_hat[i, :], '--', label='mbtr')
    ax.plot(y_hat_lgb[i, :], '--', label='lgb')
    ax.plot(y_hat_lin[i, :], '--', label='mbtr-lin')
    ax.set_xlabel('step ahead [h]')
    ax.set_ylabel('P [kW]')
    ax.legend(loc='upper right')
    ax.set_ylim(y_min, y_max)
    plt.pause(1e-6)
```

and compare the models in term of RMSE:

```
mean_rmse = lambda x,y : np.mean(np.mean((x-y)**2, axis=1)**0.5)
rmse_mbt = mean_rmse(y_te, y_hat)
rmse_lgb = mean_rmse(y_te, y_hat_lgb)
rmse_mbt_lin = mean_rmse(y_te, y_hat_lin)

print('#'*20 + '    Mean-horizon RMSEs  ' + '#'*20 )
[print('{}: {:0.2e}'.format(n, s)) for n, s in zip(['mbtr', 'lgb', 'mbtr-lin'], [rmse_
↪mbt, rmse_lgb, rmse_mbt_lin])]
```

3.2 Time smoothing and Fourier regression

The `examples/fourier_and_smoothing.py` shows an example of usage of the *`mbtr.losses.TimeSmoother`* and *`mbtr.losses.FourierLoss`* losses. The first part of the code is identical to the one used in the `examples/multivariate_forecast.py` example; we download the dataset and create the training and test sets. We now use the `time_smoother` loss function, which penalize the second order discrete derivative of the response function:

```
print('#'*20 + '    Fitting MBT with smooth loss   ' + '#'*20)
m_sm = MBT(loss_type='time_smoother', lambda_smooth=1, n_boosts=30,
           min_leaf=300, lambda_weights=1e-3).fit(x_tr, y_tr, do_plot=True)
y_hat_sm = m_sm.predict(x_te)
```

Keeping all the other MBT parameters unchanged, we can fit two Fourier losses with different number of harmonics:

```
print('#'*20 + '    Fitting MBT with Fourier loss and 3 harmonics    ' + '#'*20)

m_fou_3 = MBT(loss_type='fourier', n_harmonics=3, n_boosts=30,
              min_leaf=300, lambda_weights=1e-3).fit(x_tr, y_tr, do_plot=True)
y_hat_fou_3 = m_fou_3.predict(x_te)

print('#'*20 + '    Fitting MBT with Fourier loss and 5 harmonics    ' + '#'*20)

m_fou_5 = MBT(loss_type='fourier', n_harmonics=5, n_boosts=30,  min_leaf=300,
              lambda_weights=1e-3).fit(x_tr, y_tr, do_plot=True)
y_hat_fou_5 = m_fou_5.predict(x_te)
```

We can now plot some results from the different fitted losses:

```
fig, ax = set_figure((5,4))
y_min = np.min(y_te[:150, :]) * 0.9
y_max = np.max(y_te[:150, :]) * 1.1

for i in range(150):
    ax.cla()
    ax.plot(np.arange(24), y_te[i,:], label='test')
    ax.plot(y_hat_sm[i, :], '--', label='time-smoother')
    ax.plot(y_hat_fou_3[i, :], '--', label='fourier-3')
    ax.plot(y_hat_fou_5[i, :], '--', label='fourier-5')

    ax.set_xlabel('step ahead [h]')
    ax.set_ylabel('P [kW]')
    ax.legend(loc='upper right')
    ax.set_ylim(y_min, y_max)
    plt.pause(1e-6)
```

Finally, we can compare the models in term of RMSE:

```
mean_rmse = lambda x, y: np.mean(np.mean((x - y) ** 2, axis=1) ** 0.5)
rmse_sm = mean_rmse(y_te, y_hat_sm)
rmse_fou_3 = mean_rmse(y_te, y_hat_fou_3)
rmse_fou_5 = mean_rmse(y_te, y_hat_fou_5)


print('#' * 20 + '    Mean-horizon RMSEs  ' + '#' * 20)
[print('{}: {:0.2e}'.format(n, s)) for n, s in zip(['smoother', 'fourier-3', 'fourier-5
↪'], [rmse_sm, rmse_fou_3, rmse_fou_5])]
```

3.3 Quantile loss

The `examples/quantiles.py` shows an example of usage of the *`mbtr.losses.QuantileLoss`* loss. In this example we aim at predicting the quantiles of the next step ahead, using the previous 24 hours of the signal as covariates. After downloading the dataset as described in the previous example, we build the training and test sets:

```
# embed the signal in a 2-days matrix
total_power = hankel(total_power, np.zeros((25, 1)))[:-25, :]

# create feature matrix and target for the training and test sets
x = total_power[:, :24]
y = total_power[:, 24:]
n_tr = int(len(x)*0.8)
x_tr, y_tr, x_te, y_te = [x[:n_tr, :], y[:n_tr, :], x[n_tr:, :], y[n_tr:, :]]
```

we plot some training instances of the features and the target to have a visual check:

```
fig, ax = set_figure((5,4))
y_min = np.min(y_tr[:50, :]) * 0.9
y_max = np.max(y_tr[:50, :]) * 1.1

for i in range(50):
    ax.cla()
    ax.plot(np.arange(24), x_tr[i,:], label='features')
    ax.scatter(25, y_tr[i, :], label='multivariate targets', marker='.')
    ax.set_xlabel('step ahead [h]')
    ax.set_ylabel('P [kW]')
    ax.legend(loc='upper right')
    ax.set_ylim(y_min, y_max)
    plt.pause(1e-6)
plt.close('all')
```

Finally, we can train a `MBT` instance with a *`mbtr.losses.QuantileLoss`* loss. Note that this loss requires the `alphas` additional parameter. This is an array of quantiles to be fitted:

```
alphas = np.linspace(0.05, 0.95, 7)
m = MBT(loss_type='quantile', alphas=alphas, n_boosts=40,
        min_leaf=300, lambda_weights=1e-3).fit(x_tr, y_tr, do_plot=True)
```

At last, we can plot some predictions for the required quantiles:

```
y_hat = m.predict(x_te)
fig, ax = set_figure((5,4))
y_min = np.min(y_tr[:50, :]) * 0.9
y_max = np.max(y_tr[:50, :]) * 1.1
n_q = y_hat.shape[1]
n_sa = y_te.shape[1]
n_plot = 300
colors = plt.get_cmap('plasma', int(n_q))
for fl in np.arange(np.floor(n_q / 2), dtype=int):
    q_low = np.squeeze(y_hat[:n_plot, fl])
    q_up = np.squeeze(y_hat[:n_plot, n_q - fl - 1])
    x = np.arange(len(q_low))
    ax.fill_between(x, q_low, q_up, color=colors(fl), alpha=0.1 + 0.6*fl/n_q,␣
↪linewidth=0.0)
plt.plot(y_te[:n_plot], linewidth=2)
plt.xlabel('step [h]')
plt.ylabel('P [kW]')
plt.title('Quantiles on first 300 samples')
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

L

M

P

Q

R

S

T

U