Package 'BKTR'

January 20, 2025

Version 0.2.0

Title Bayesian Kernelized Tensor Regression

Description Facilitates scalable spatiotemporally varying coefficient modelling with Bayesian kernelized tensor regression.
The important features of this package are:
(a) Enabling local temporal and spatial modeling of the relationship between the response variable and covariates.
(b) Implementing the model described by Lei et al. (2023) <doi:10.48550/arXiv.2109.00046>.
(c) Using a Bayesian Markov Chain Monte Carlo (MCMC) algorithm to sample from the posterior distribution of the model parameters.
(d) Employing a tensor decomposition to reduce the number of estimated parameters.
(e) Accelerating tensor operations and enabling graphics processing unit (GPU) acceleration with the 'torch' package.

Depends R (>= 4.0.0)

Encoding UTF-8

Imports torch (>= 0.13.0), R6, R6P, ggplot2, ggmap, data.table

License MIT + file LICENSE

RoxygenNote 7.2.3

Collate 'samplers.R' 'tensor_ops.R' 'result_logger.R' 'likelihood_evaluator.R' 'distances.R' 'kernels.R' 'bktr.R' 'examples.R' 'plots.R' 'utils.R'

Suggests knitr, rmarkdown, R.rsp

LazyData true

VignetteBuilder knitr, rmarkdown, R.rsp

BugReports https://github.com/julien-hec/BKTR/issues

NeedsCompilation no

Author Julien Lanthier [aut, cre, cph] (<https://orcid.org/0009-0008-8728-4996>), Mengying Lei [aut] (<https://orcid.org/0000-0001-7343-3323>), Aurélie Labbe [aut] (<https://orcid.org/0000-0002-4207-8143>), Lijun Sun [aut] (<https://orcid.org/0000-0001-9488-0712>)

Maintainer Julien Lanthier <julien.lanthier@hec.ca>

Repository CRAN

Date/Publication 2024-08-18 14:40:02 UTC

Contents

*.Kernel *Operator overloading for kernel multiplication*

Description

Operator overloading for kernel multiplication

Usage

```
## S3 method for class 'Kernel'
k1 * k2
```

Arguments

k1	Kernel: The left kernel to use for composition
k2	Kernel: The right kernel to use for composition

Value

A new KernelMulComposed object.

Examples

```
# Create a new locally periodic kernel
k_loc_per <- KernelSE$new() * KernelPeriodic$new()
# Set the kernel's positions
positions_df <- data.frame(x=c(-4, 0, 3), y=c(-2, 0, 2))
k_loc_per$set_positions(positions_df)
# Generate the kernel's covariance matrix
k_loc_per$kernel_gen()
```

+.Kernel *Operator overloading for kernel addition*

Description

Operator overloading for kernel addition

Usage

```
## S3 method for class 'Kernel'
k1 + k2
```

Arguments

k1	Kernel: The left kernel to use for composition
k2	Kernel: The right kernel to use for composition

Value

A new KernelAddComposed object.

Examples

```
# Create a new additive kernel
k_rq_plus_per <- KernelRQ$new() + KernelPeriodic$new()
# Set the kernel's positions
positions_df <- data.frame(x=c(-4, 0, 3), y=c(-2, 0, 2))
k_rq_plus_per$set_positions(positions_df)
# Generate the kernel's covariance matrix
k_rq_plus_per$kernel_gen()
```

BixiData *BIXI Data Class*

Description

R6 class encapsulating all BIXI dataframes. It is also possible to use a light version of the dataset by using the is_light parameter. In this case, the dataset is reduced to its first 25 stations and first 50 days. The light version is only used for testing and short examples.

Public fields

departure_df The departure dataframe

spatial_features_df The spatial features dataframe

temporal_features_df The temporal features dataframe

spatial_positions_df The spatial positions dataframe

temporal_positions_df The temporal positions dataframe

data_df The data dataframe

is_light Whether the light version of the dataset is used

Methods

Public methods:

- BixiData$new()
- BixiData$clone()

Method new(): Initialize the BIXI data class

Usage:

```
BixiData$new(is_light = FALSE)
```

Arguments:

`is_light` Whether the light version of the dataset is used, defaults to FALSE.

Returns: A new BIXI data instance

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
BixiData$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create a light BIXI data collection instance containing multiple dataframes
# This only uses the first 25 stations and 50 days of the full dataset
bixi_data <- BixiData$new(is_light = TRUE)
# Dataframe containing the position (latitude and longitude) of M stations
bixi_data$spatial_positions_df
# Dataframe containing the time position of N days (0 to N-1)
bixi_data$temporal_positions_df
# Dataframe with spatial and temporal features for each day and station (M x N rows)
bixi_data$data_df
```

`bixi_spatial_features` *Spatial Features of Montreal BIXI Stations in 2019*

Description

These data represent 14 spatial features (columns) for 587 bike sharing stations (rows) located at different geographical coordinates (longitude, latitude) in Montreal. The Montreal based bike sharing company is named BIXI. The first column contains the descriptive label affected to each station and the other columns contain information about the infrastructure, points of interests, walkscore and population surrounding each station for 2019.

Usage

```
data("bixi_spatial_features")
```

Format

A data frame with 587 observations on the following 14 variables.

`location` a character vector

`area_park` a numeric vector

`len_cycle_path` a numeric vector

`len_major_road` a numeric vector

`len_minor_road` a numeric vector

`num_metro_stations` a numeric vector

`num_other_commercial` a numeric vector

`num_restaurants` a numeric vector

`num_university` a numeric vector

`num_pop` a numeric vector

`num_bus_stations` a numeric vector

`num_bus_routes` a numeric vector

`walkscore` a numeric vector

`capacity` a numeric vector

Source

Wang, X., Cheng, Z., Trépanier, M., & Sun, L. (2021). Modeling bike-sharing demand using a regression model with spatially varying coefficients. Journal of Transport Geography, 93, 103059.

References

Reference for the BIXI station informations: BIXI Montréal (2023). "Open Data." Accessed: 2023-07-11, URL https://bixi.com/en/ open-data.

Reference for point of interests and infrastructure informations: DMTI Spatial Inc (2019). "Enhanced Point of Interest (DMTI)." URL https://www.dmtispatial.com.

Reference for Walkscore: Walk Score (2023). "Walk Score Methodology." Accessed: 2023-07-11, URL https://www.walkscore.com/methodology.shtml.

The population information comes from the 2016 Canada census data at a dissemination block level.

---

bixi_spatial_locations

Spatial Locations of Montreal BIXI Stations in 2019

Description

Data points representing the spatial locations of 587 bike sharing stations for the Montreal based bike sharing company named BIXI. The dataframe contains a label to identify each station and its associated longitude and latitude coordinates.

Usage

```
data("bixi_spatial_locations")
```

Format

A data frame with 587 observations on the following 3 variables.

`location` a character vector

`latitude` a numeric vector

`longitude` a numeric vector

Source

Wang, X., Cheng, Z., Trépanier, M., & Sun, L. (2021). Modeling bike-sharing demand using a regression model with spatially varying coefficients. Journal of Transport Geography, 93, 103059.

References

BIXI Montréal (2023). "Open Data." Accessed: 2023-07-11, URL https://bixi.com/en/ opendata.

`bixi_station_departures`

Daily Departure from BIXI Stations in 2019

Description

These data capture the number of daily departure for 587 bike sharing stations (rows) through 197 days (columns). The data is limited to the 2019 season of a Montreal based bike sharing company named BIXI.

Usage

```
data("bixi_station_departures")
```

Format

A data frame with 587 rows and 197 columns.

Source

Wang, X., Cheng, Z., Trépanier, M., & Sun, L. (2021). Modeling bike-sharing demand using a regression model with spatially varying coefficients. Journal of Transport Geography, 93, 103059.

References

BIXI Montréal (2023). "Open Data." Accessed: 2023-07-11, URL https://bixi.com/en/ opendata.

`bixi_temporal_features` *Temporal Features in Montreal applicable to BIXI for 2019*

Description

These data represent the temporal features in Montreal applicable to a Montreal based bike sharing company named BIXI. The data include six features (columns) for 196 days (rows). The time column represent the label associated to each captured time for the 2019 season of BIXI. The other columns contain information about Montral weather and applicable holidays for each day.

Usage

```
data("bixi_temporal_features")
```

Format

A data frame with 196 observations on the following 6 variables.

`time` a IDate

`humidity` a numeric vector

`max_temp_f` a numeric vector

`mean_temp_c` a numeric vector

`total_precip_mm` a numeric vector

`holiday` a numeric vector

Source

Lei, M., Labbe, A., & Sun, L. (2021). Scalable Spatiotemporally Varying Coefficient Modelling with Bayesian Kernelized Tensor Regression. arXiv preprint arXiv:2109.00046.

References

The weather data is sourced from the Environment and Climate Change Canada Historical Climate Data website.

The holiday column is specifying if a date is a holiday or not, according to the Quebec government.

bixi_temporal_locations *Temporal indices for the 2019 BIXI season*

Description

These data represent 196 temporal indices (rows) related to each day of the 2019 season of Montreal based bike sharing company named BIXI. The time column represent the label associated to each day and the time_index column represent the location in time space of each day when compared to each other. Since no days are missing and they are all spaced by exactly one day, the time_index is simply a range from 0 to 195.

Usage

```
data("bixi_temporal_locations")
```

Format

A data frame with 196 observations on the following 2 variables.

`time` a IDate

`time_index` a numeric vector

Source

Lei, M., Labbe, A., & Sun, L. (2021). Scalable Spatiotemporally Varying Coefficient Modelling with Bayesian Kernelized Tensor Regression. arXiv preprint arXiv:2109.00046.

BKTRRegressor *R6 class encapsulating the BKTR regression elements*

Description

A BKTRRegressor holds all the key elements to accomplish the MCMC sampling algorithm (**Algorithm 1** of the paper).

Public fields

`data_df` The dataframe containing all the covariates through time and space (including the response variable)

`y` The response variable tensor

`omega` The tensor indicating which response values are not missing

`covariates` The tensor containing all the covariates

`covariates_dim` The dimensions of the covariates tensor

`logged_params_tensor` The tensor containing all the sampled hyperparameters

`tau` The precision hyperparameter

`spatial_decomp` The spatial covariate decomposition

`temporal_decomp` The temporal covariate decomposition

`covs_decomp` The feature covariate decomposition

`result_logger` The result logger instance used to store the results of the MCMC sampling

`has_completed_sampling` Boolean showing wheter the MCMC sampling has been completed

`spatial_kernel` The spatial kernel used

`temporal_kernel` The temporal kernel used

`spatial_positions_df` The dataframe containing the spatial positions

`temporal_positions_df` The dataframe containing the temporal positions

`spatial_params_sampler` The spatial kernel hyperparameter sampler

`temporal_params_sampler` The temporal kernel hyperparameter sampler

`tau_sampler` The tau hyperparameter sampler

`precision_matrix_sampler` The precision matrix sampler

`spatial_ll_evaluator` The spatial likelihood evaluator

`temporal_ll_evaluator` The temporal likelihood evaluator

`rank_decomp` The rank of the CP decomposition

`burn_in_iter` The number of burn in iterations

`sampling_iter` The number of sampling iterations

`max_iter` The total number of iterations

`a_0` The initial value for the shape in the gamma function generating tau

`b_0` The initial value for the rate in the gamma function generating tau

`formula` The formula used to specify the relation between the response variable and the covariates

`spatial_labels` The spatial labels

`temporal_labels` The temporal labels

`feature_labels` The feature labels

`geo_coords_projector` The geographic coordinates projector

Active bindings

`summary` A summary of the BKTRRegressor instance

`beta_covariates_summary` A dataframe containing the summary of the beta covariates

`y_estimates` A dataframe containing the y estimates

`imputed_y_estimates` A dataframe containing the imputed y estimates

`beta_estimates` A dataframe containing the beta estimates

`hyperparameters_per_iter_df` A dataframe containing the beta estimates per iteration

`decomposition_tensors` List of all used decomposition tensors

Methods

Public methods:

- BKTRRegressor$new()
- BKTRRegressor$mcmc_sampling()
- BKTRRegressor$predict()
- BKTRRegressor$get_iterations_betas()
- BKTRRegressor$get_beta_summary_df()
- BKTRRegressor$clone()

Method `new()`**:** Create a new `BKTRRegressor` object.

Usage:

```
BKTRRegressor$new(
  data_df,
  spatial_positions_df,
  temporal_positions_df,
  rank_decomp = 10,
  burn_in_iter = 500,
  sampling_iter = 500,
  formula = NULL,
  spatial_kernel = KernelMatern$new(smoothness_factor = 3),
  temporal_kernel = KernelSE$new(),
  sigma_r = 0.01,
  a_0 = 1e-06,
  b_0 = 1e-06,
  has_geo_coords = TRUE,
  geo_coords_scale = 10
)
```

Arguments:

`data_df` data.table: A dataframe containing all the covariates through time and space. It is important that the dataframe has a two indexes named 'location' and 'time' respectively. The dataframe should also contain every possible combinations of 'location' and 'time' (i.e. even missing rows should be filled present but filled with NaN). So if the dataframe has 10 locations and 5 time points, it should have 50 rows (10 x 5). If formula is None, the dataframe should contain the response variable 'Y' as the first column. Note that the covariate columns cannot contain NaN values, but the response variable can.

`spatial_positions_df` data.table: Spatial kernel input tensor used to calculate covariates' distance. Vector of length equal to the number of location points.

`temporal_positions_df` data.table: Temporal kernel input tensor used to calculate covariate distance. Vector of length equal to the number of time points.

`rank_decomp` Integer: Rank of the CP decomposition (Paper – R). Defaults to 10.

`burn_in_iter` Integer: Number of iteration before sampling (Paper – K_1). Defaults to 500.

`sampling_iter` Integer: Number of sampling iterations (Paper – K_2). Defaults to 500.

`formula` A Wilkinson R formula to specify the relation between the response variable 'Y' and the covariates. If Null, the first column of the data frame will be used as the response variable and all the other columns will be used as the covariates. Defaults to Null.

`spatial_kernel` Kernel: Spatial kernel Used. Defaults to a KernelMatern(smoothness_factor=3).

`temporal_kernel` Kernel: Temporal kernel used. Defaults to KernelSE().

`sigma_r` Numeric: Variance of the white noise process (τ^{-1}) defaults to 1E-2.

`a_0` Numeric: Initial value for the shape (α) in the gamma function generating tau defaults to 1E-6.

`b_0` Numeric: Initial value for the rate (β) in the gamma function generating tau defaults to 1E-6.

`has_geo_coords` Boolean: Whether the spatial positions df use geographic coordinates (latitude, longitude). Defaults to TRUE.

`geo_coords_scale` Numeric: Scale factor to convert geographic coordinates to euclidean 2D space via Mercator projection using x & y domains of [-scale/2, +scale/2]. Only used if has_geo_coords is TRUE. Defaults to 10.

Returns: A new `BKTRRegressor` object.

Method `mcmc_sampling()`**:** Launch the MCMC sampling process.
For a predefined number of iterations:

1. Sample spatial kernel hyperparameters
2. Sample temporal kernel hyperparameters
3. Sample the precision matrix from a wishart distribution
4. Sample a new spatial covariate decomposition
5. Sample a new feature covariate decomposition
6. Sample a new temporal covariate decomposition
7. Calculate respective errors for the iterations
8. Sample a new tau value
9. Collect all the important data for the iteration

Usage:

```
BKTRRegressor$mcmc_sampling()
```

Returns: NULL Results are stored and can be accessed via summary()

Method `predict()`**:** Use interpolation to predict betas and response values for new data.

Usage:

```
BKTRRegressor$predict(
  new_data_df,
  new_spatial_positions_df = NULL,
  new_temporal_positions_df = NULL,
  jitter = 1e-05
)
```

Arguments:

`new_data_df` data.table: New covariates. Must have the same columns as the covariates used to fit the model. The index should contain the combination of all old spatial coordinates with all new temporal coordinates, the combination of all new spatial coordinates with all old temporal coordinates, and the combination of all new spatial coordinates with all new temporal coordinates.

`new_spatial_positions_df` data.table or NULL: A data frame containing the new spatial positions. Defaults to NULL.

`new_temporal_positions_df` data.table or NULL: A data frame containing the new temporal positions. Defaults to NULL.

`jitter` Numeric or NULL: A small value to add to the diagonal of the precision matrix. Defaults to NULL.

Returns: List: A list of two dataframes. The first represents the beta forecasted for all new spatial locations or temporal points. The second represents the forecasted response for all new spatial locations or temporal points.

Method `get_iterations_betas()`**:** Return all sampled betas through sampling iterations for a given set of spatial, temporal and feature labels. Useful for plotting the distribution of sampled beta values.

Usage:

```
BKTRRegressor$get_iterations_betas(
  spatial_label,
  temporal_label,
  feature_label
)
```

Arguments:

`spatial_label` String: The spatial label for which we want to get the betas

`temporal_label` String: The temporal label for which we want to get the betas

`feature_label` String: The feature label for which we want to get the betas

Returns: A list containing the sampled betas through iteration for the given labels

Method `get_beta_summary_df()`**:** Get a summary of estimated beta values. If no labels are given, then the summary is for all the betas. If labels are given, then the summary is for the given labels.

Usage:

```
BKTRRegressor$get_beta_summary_df(
  spatial_labels = NULL,
  temporal_labels = NULL,
  feature_labels = NULL
)
```

Arguments:

`spatial_labels` vector: The spatial labels used in summary. If NULL, then all spatial labels are used. Defaults to NULL.

`temporal_labels` vector: The temporal labels used in summary. If NULL, then all temporal labels are used. Defaults to NULL.

`feature_labels` vector: The feature labels used in summary. If NULL, then all feature labels are used. Defaults to NULL.

Returns: A new data.table with the beta summary for the given labels.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
BKTRRegressor$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Create a BIXI data collection instance containing multiple dataframes
bixi_data <- BixiData$new(is_light = TRUE) # Use light version for example

# Create a BKTRRegressor instance
bktr_regressor <- BKTRRegressor$new(
  formula = nb_departure ~ 1 + mean_temp_c + area_park,
  data_df <- bixi_data$data_df,
  spatial_positions_df = bixi_data$spatial_positions_df,
  temporal_positions_df = bixi_data$temporal_positions_df,
  burn_in_iter = 5, sampling_iter = 10) # For example only (too few iterations)

# Launch the MCMC sampling
bktr_regressor$mcmc_sampling()

# Get the summary of the bktr regressor
summary(bktr_regressor)

# Get estimated response variables for missing values
bktr_regressor$imputed_y_estimates

# Get the list of sampled betas for given spatial, temporal and feature labels
bktr_regressor$get_iterations_betas(
  spatial_label = bixi_data$spatial_positions_df$location[1],
  temporal_label = bixi_data$temporal_positions_df$time[1],
  feature_label = 'mean_temp_c')

# Get the summary of all betas for the 'mean_temp_c' feature
bktr_regressor$get_beta_summary_df(feature_labels = 'mean_temp_c')


## PREDICTION EXAMPLE ##
# Create a light version of the BIXI data collection instance
bixi_data <- BixiData$new(is_light = TRUE)
# Simplify variable names
data_df <- bixi_data$data_df
spa_pos_df <- bixi_data$spatial_positions_df
temp_pos_df <- bixi_data$temporal_positions_df

# Keep some data aside for prediction
new_spa_pos_df <- spa_pos_df[1:2, ]
new_temp_pos_df <- temp_pos_df[1:5, ]
reg_spa_pos_df <- spa_pos_df[-(1:2), ]
reg_temp_pos_df <- temp_pos_df[-(1:5), ]
reg_data_df_mask <- data_df$location %in% reg_spa_pos_df$location &
  data_df$time %in% reg_temp_pos_df$time
reg_data_df <- data_df[reg_data_df_mask, ]
```

```
new_data_df <- data_df[!reg_data_df_mask, ]

# Launch mcmc sampling on regression data
bktr_regressor <- BKTRRegressor$new(
  formula = nb_departure ~ 1 + mean_temp_c + area_park,
  data_df = reg_data_df,
  spatial_positions_df = reg_spa_pos_df,
  temporal_positions_df = reg_temp_pos_df,
  burn_in_iter = 5, sampling_iter = 10) # For example only (too few iterations)
bktr_regressor$mcmc_sampling()

# Predict response values for new data
bktr_regressor$predict(
  new_data_df = new_data_df,
  new_spatial_positions_df = new_spa_pos_df,
  new_temporal_positions_df = new_temp_pos_df)
```

`CompositionOps` *Kernel Composition Operations*

Description

Kernel Composition Operations Enum. Possibilities of operation between two kernels to generate a new composed kernel. The values are: `MUL` and `ADD`.

Usage

```
CompositionOps
```

Format

An object of class `list` of length 2.

`Kernel` *Base R6 class for Kernels*

Description

Abstract base class for kernels (Should not be instantiated)

Public fields

`kernel_variance` The variance of the kernel

`jitter_value` The jitter value to add to the kernel matrix

`distance_matrix` The distance matrix between points in a tensor format

`name` The kernel's name

`parameters` The parameters of the kernel (list of `KernelParameter`)

`covariance_matrix` The covariance matrix of the kernel in a tensor format

`positions_df` The positions of the points in a dataframe format

`has_dist_matrix` Identify if the kernel has a distance matrix or not

Methods

Public methods:

- `Kernel$new()`
- `Kernel$core_kernel_fn()`
- `Kernel$add_jitter_to_kernel()`
- `Kernel$kernel_gen()`
- `Kernel$set_positions()`
- `Kernel$plot()`
- `Kernel$clone()`

Method `new()`**:** Kernel abstract base constructor

Usage:

```
Kernel$new(kernel_variance, jitter_value)
```

Arguments:

`kernel_variance` Numeric: The variance of the kernel

`jitter_value` Numeric: The jitter value to add to the kernel matrix

Returns: A new `Kernel` object.

Method `core_kernel_fn()`**:** Abstract method to compute the core kernel's covariance matrix

Usage:

```
Kernel$core_kernel_fn()
```

Method `add_jitter_to_kernel()`**:** Method to add jitter to the kernel's covariance matrix

Usage:

```
Kernel$add_jitter_to_kernel()
```

Method `kernel_gen()`**:** Method to compute the kernel's covariance matrix

Usage:

```
Kernel$kernel_gen()
```

Method `set_positions()`**:** Method to set the kernel's positions and compute the distance matrix

Usage:

```
Kernel$set_positions(positions_df)
```

Arguments:

`positions_df` Dataframe: The positions of the points in a dataframe format

Method `plot()`**:** Method to plot the kernel's covariance matrix

Usage:

```
Kernel$plot(show_figure = TRUE)
```

Arguments:

`show_figure` Boolean: If TRUE, the figure is shown, otherwise it is returned

Returns: If `show_figure` is TRUE, the figure is shown, otherwise it is returned

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Kernel$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`KernelAddComposed` *R6 class for Kernels Composed via Addition*

Description

R6 class automatically generated when adding two kernels together.

Super classes

`BKTR::Kernel` -> `BKTR::KernelComposed` -> `KernelAddComposed`

Methods

Public methods:

- `KernelAddComposed$new()`
- `KernelAddComposed$clone()`

Method `new()`**:** Create a new `KernelAddComposed` object.

Usage:

```
KernelAddComposed$new(left_kernel, right_kernel, new_name)
```

Arguments:

`left_kernel` Kernel: The left kernel to use for composition

`right_kernel` Kernel: The right kernel to use for composition

`new_name` String: The name of the composed kernel

Returns: A new `KernelAddComposed` object.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
KernelAddComposed$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create a new additive kernel
k_rq_plus_per <- KernelAddComposed$new(
  left_kernel = KernelRQ$new(),
  right_kernel = KernelPeriodic$new(),
  new_name = 'SE + Periodic Kernel'
)
# Set the kernel's positions
positions_df <- data.frame(x=c(-4, 0, 3), y=c(-2, 0, 2))
k_rq_plus_per$set_positions(positions_df)
# Generate the kernel's covariance matrix
k_rq_plus_per$kernel_gen()
```

`KernelComposed` *R6 class for Composed Kernels*

Description

R6 class for Composed Kernels

Super class

`BKTR::Kernel` -> `KernelComposed`

Public fields

`name` The kernel's name

`parameters` The parameters of the kernel (list of `KernelParameter`)

`left_kernel` The left kernel to use for composition

`right_kernel` The right kernel to use for composition

`composition_operation` The operation to use for composition

`has_dist_matrix` Identify if the kernel has a distance matrix or not

Methods

Public methods:

- KernelComposed$new()
- KernelComposed$core_kernel_fn()
- KernelComposed$set_positions()
- KernelComposed$clone()

Method `new()`**:** Create a new `KernelComposed` object.

Usage:

```
KernelComposed$new(left_kernel, right_kernel, new_name, composition_operation)
```

Arguments:

`left_kernel` Kernel: The left kernel to use for composition

`right_kernel` Kernel: The right kernel to use for composition

`new_name` String: The name of the composed kernel

`composition_operation` CompositionOps: The operation to use for composition

Method `core_kernel_fn()`**:** Method to compute the core kernel's covariance matrix

Usage:

```
KernelComposed$core_kernel_fn()
```

Returns: The core kernel's covariance matrix

Method `set_positions()`**:** Method to set the kernel's positions and compute the distance matrix

Usage:

```
KernelComposed$set_positions(positions_df)
```

Arguments:

`positions_df` Dataframe: The positions of the points in a dataframe format

Returns: NULL, set the kernel's positions and compute the distance matrix

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
KernelComposed$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create a new locally periodic kernel
k_loc_per <- KernelComposed$new(
  left_kernel = KernelSE$new(),
  right_kernel = KernelPeriodic$new(),
  new_name = 'Locally Periodic Kernel',
  composition_operation = CompositionOps$MUL
)
```

```
# Set the kernel's positions
positions_df <- data.frame(x=c(-4, 0, 3), y=c(-2, 0, 2))
k_loc_per$set_positions(positions_df)
# Generate the kernel's covariance matrix
k_loc_per$kernel_gen()
```

KernelMatern *R6 class for Matern Kernels*

Description

R6 class for Matern Kernels

Super class

BKTR::Kernel -> KernelMatern

Public fields

`lengthscale` The lengthscale parameter instance of the kernel

`smoothness_factor` The smoothness factor of the kernel

`has_dist_matrix` Identify if the kernel has a distance matrix or not

Methods

Public methods:

- KernelMatern$new()
- KernelMatern$get_smoothness_kernel_fn()
- KernelMatern$core_kernel_fn()
- KernelMatern$clone()

Method `new()`**:** Create a new KernelMatern object.

Usage:

```
KernelMatern$new(
  smoothness_factor = 5,
  lengthscale = KernelParameter$new(2),
  kernel_variance = 1,
  jitter_value = NULL
)
```

Arguments:

`smoothness_factor` Numeric: The smoothness factor of the kernel (1, 3 or 5)

`lengthscale` KernelParameter: The lengthscale parameter instance of the kernel

`kernel_variance` Numeric: The variance of the kernel

`jitter_value` Numeric: The jitter value to add to the kernel matrix

Method `get_smoothness_kernel_fn()`**:** Method to the get the smoothness kernel function for a given integer smoothness factor

Usage:

```
KernelMatern$get_smoothness_kernel_fn()
```

Returns: The smoothness kernel function

Method `core_kernel_fn()`**:** Method to compute the core kernel's covariance matrix

Usage:

```
KernelMatern$core_kernel_fn()
```

Returns: The core kernel's covariance matrix

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
KernelMatern$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create a new Matern 3/2 kernel
k_matern <- KernelMatern$new(smoothness_factor = 3)
# Set the kernel's positions
positions_df <- data.frame(x=c(-4, 0, 3), y=c(-2, 0, 2))
k_matern$set_positions(positions_df)
# Generate the kernel's covariance matrix
k_matern$kernel_gen()
```

`KernelMulComposed` *R6 class for Kernels Composed via Multiplication*

Description

R6 class automatically generated when multiplying two kernels together.

Super classes

`BKTR::Kernel` -> `BKTR::KernelComposed` -> `KernelMulComposed`

Methods

Public methods:

- KernelMulComposed$new()
- KernelMulComposed$clone()

Method `new()`**:** Create a new `KernelMulComposed` object.

Usage:

```
KernelMulComposed$new(left_kernel, right_kernel, new_name)
```

Arguments:

`left_kernel` Kernel: The left kernel to use for composition

`right_kernel` Kernel: The right kernel to use for composition

`new_name` String: The name of the composed kernel

Returns: A new `KernelMulComposed` object.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
KernelMulComposed$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create a new locally periodic kernel
k_loc_per <- KernelMulComposed$new(
  left_kernel = KernelSE$new(),
  right_kernel = KernelPeriodic$new(),
  new_name = 'Locally Periodic Kernel'
)
# Set the kernel's positions
positions_df <- data.frame(x=c(-4, 0, 3), y=c(-2, 0, 2))
k_loc_per$set_positions(positions_df)
# Generate the kernel's covariance matrix
k_loc_per$kernel_gen()
```

`KernelParameter` *R6 class for kernel's hyperparameter*

Description

KernelParameter contains all information and behaviour related to a kernel parameters.

Public fields

`value` The hyperparameter mean's prior value or its constant value

`is_fixed` Says if the kernel parameter is fixed or not (if fixed, there is no sampling)

`lower_bound` The hyperparameter's minimal value during sampling

`upper_bound` The hyperparameter's maximal value during sampling

`slice_sampling_scale` The sampling range's amplitude

`hparam_precision` Precision of the hyperparameter

`kernel` The kernel associated with the parameter (it is set at kernel instanciation)

`name` The kernel parameter's name

Active bindings

`full_name` The kernel parameter's full name

Methods

Public methods:

- `KernelParameter$new()`
- `KernelParameter$set_kernel()`
- `KernelParameter$clone()`

Method `new()`**:** Create a new `KernelParameter` object.

Usage:

```
KernelParameter$new(
  value,
  is_fixed = FALSE,
  lower_bound = DEFAULT_LBOUND,
  upper_bound = DEFAULT_UBOUND,
  slice_sampling_scale = log(10),
  hparam_precision = 1
)
```

Arguments:

`value` Numeric: The hyperparameter mean's prior value (Paper - ϕ) or its constant value

`is_fixed` Boolean: Says if the kernel parameter is fixed or not (if fixed, there is no sampling)

`lower_bound` Numeric: Hyperparameter's minimal value during sampling (Paper - ϕ_{min})

`upper_bound` Numeric: Hyperparameter's maximal value during sampling (Paper - ϕ_{max})

`slice_sampling_scale` Numeric: The sampling range's amplitude (Paper - ρ)

`hparam_precision` Numeric: The hyperparameter's precision

Returns: A new `KernelParameter` object.

Method `set_kernel()`**:** Set `Kernel` for a given `KernelParameter` object.

Usage:

```
KernelParameter$set_kernel(kernel, param_name)
```

Arguments:

`kernel` Kernel: The kernel to associate with the parameter

`param_name` String: The parameter's name

Returns: NULL, set a new kernel for the parameter

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
KernelParameter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# A kernel parameter can be a constant value
const_param <- KernelParameter$new(7, is_fixed = TRUE)
# It can otherwise be sampled and have its value updated through sampling
samp_param <- KernelParameter$new(1, lower_bound = 0.1,
  upper_bound = 10, slice_sampling_scale = 4)

# A kernel parameter can be associated with any type of kernel
KernelPeriodic$new(period_length = const_param, lengthscale = samp_param)
```

`KernelPeriodic` *R6 class for Periodic Kernels*

Description

R6 class for Periodic Kernels

Super class

`BKTR::Kernel` -> `KernelPeriodic`

Public fields

`lengthscale` The lengthscale parameter instance of the kernel

`period_length` The period length parameter instance of the kernel

`has_dist_matrix` Identify if the kernel has a distance matrix or not

`name` The kernel's name

Methods

Public methods:

- KernelPeriodic$new()
- KernelPeriodic$core_kernel_fn()
- KernelPeriodic$clone()

Method `new()`**:** Create a new `KernelPeriodic` object.

Usage:

```
KernelPeriodic$new(
  lengthscale = KernelParameter$new(2),
  period_length = KernelParameter$new(2),
  kernel_variance = 1,
  jitter_value = NULL
)
```

Arguments:

`lengthscale` KernelParameter: The lengthscale parameter instance of the kernel

`period_length` KernelParameter: The period length parameter instance of the kernel

`kernel_variance` Numeric: The variance of the kernel

`jitter_value` Numeric: The jitter value to add to the kernel matrix

Returns: A new `KernelPeriodic` object.

Method `core_kernel_fn()`**:** Method to compute the core kernel's covariance matrix

Usage:

```
KernelPeriodic$core_kernel_fn()
```

Returns: The core kernel's covariance matrix

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
KernelPeriodic$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create a new Periodic kernel
k_periodic <- KernelPeriodic$new()
# Set the kernel's positions
positions_df <- data.frame(x=c(-4, 0, 3), y=c(-2, 0, 2))
k_periodic$set_positions(positions_df)
# Generate the kernel's covariance matrix
k_periodic$kernel_gen()
```

KernelRQ *R6 class for Rational Quadratic Kernels*

Description

R6 class for Rational Quadratic Kernels

Super class

BKTR::Kernel -> KernelRQ

Public fields

`lengthscale` The lengthscale parameter instance of the kernel

`alpha` The alpha parameter instance of the kernel

`has_dist_matrix` The distance matrix between points in a tensor format

`name` The kernel's name

Methods

Public methods:

- KernelRQ$new()
- KernelRQ$core_kernel_fn()
- KernelRQ$clone()

Method `new()`**:** Create a new KernelRQ object.

Usage:

```
KernelRQ$new(
  lengthscale = KernelParameter$new(2),
  alpha = KernelParameter$new(2),
  kernel_variance = 1,
  jitter_value = NULL
)
```

Arguments:

`lengthscale` KernelParameter: The lengthscale parameter instance of the kernel

`alpha` KernelParameter: The alpha parameter instance of the kernel

`kernel_variance` Numeric: The variance of the kernel

`jitter_value` Numeric: The jitter value to add to the kernel matrix

Returns: A new KernelRQ object.

Method `core_kernel_fn()`**:** Method to compute the core kernel's covariance matrix

Usage:

```
KernelRQ$core_kernel_fn()
```

Returns: The core kernel's covariance matrix

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
KernelRQ$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create a new RQ kernel
k_rq <- KernelRQ$new()
# Set the kernel's positions
positions_df <- data.frame(x=c(-4, 0, 3), y=c(-2, 0, 2))
k_rq$set_positions(positions_df)
# Generate the kernel's covariance matrix
k_rq$kernel_gen()
```

`KernelSE` *R6 class for Square Exponential Kernels*

Description

R6 class for Square Exponential Kernels

Super class

`BKTR::Kernel` -> `KernelSE`

Public fields

`lengthscale` The lengthscale parameter instance of the kernel

`has_dist_matrix` Identify if the kernel has a distance matrix or not

`name` The kernel's name

Methods

Public methods:

- `KernelSE$new()`
- `KernelSE$core_kernel_fn()`
- `KernelSE$clone()`

Method `new()`**:** Create a new `KernelSE` object.

Usage:

```
KernelSE$new(
  lengthscale = KernelParameter$new(2),
  kernel_variance = 1,
  jitter_value = NULL
)
```

Arguments:

`lengthscale` KernelParameter: The lengthscale parameter instance of the kernel

`kernel_variance` Numeric: The variance of the kernel

`jitter_value` Numeric: The jitter value to add to the kernel matrix

Returns: A new `KernelSE` object.

Method `core_kernel_fn()`**:** Method to compute the core kernel's covariance matrix

Usage:

```
KernelSE$core_kernel_fn()
```

Returns: The core kernel's covariance matrix

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
KernelSE$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create a new SE kernel
k_se <- KernelSE$new()
# Set the kernel's positions
positions_df <- data.frame(x=c(-4, 0, 3), y=c(-2, 0, 2))
k_se$set_positions(positions_df)
# Generate the kernel's covariance matrix
k_se$kernel_gen()
```

`KernelWhiteNoise` *R6 class for White Noise Kernels*

Description

R6 class for White Noise Kernels

Super class

`BKTR::Kernel` -> `KernelWhiteNoise`

Public fields

`has_dist_matrix` Identify if the kernel has a distance matrix or not

`name` The kernel's name

Methods

Public methods:

- `KernelWhiteNoise$new()`
- `KernelWhiteNoise$core_kernel_fn()`
- `KernelWhiteNoise$clone()`

Method `new()`:

Usage:

```
KernelWhiteNoise$new(kernel_variance = 1, jitter_value = NULL)
```

Arguments:

`kernel_variance` Numeric: The variance of the kernel

`jitter_value` Numeric: The jitter value to add to the kernel matrix

Returns: A new `KernelWhiteNoise` object.

Method `core_kernel_fn()`: Method to compute the core kernel's covariance matrix

Usage:

```
KernelWhiteNoise$core_kernel_fn()
```

Returns: The core kernel's covariance matrix

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
KernelWhiteNoise$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create a new white noise kernel
k_white_noise <- KernelWhiteNoise$new()
# Set the kernel's positions
positions_df <- data.frame(x=c(-4, 0, 3), y=c(-2, 0, 2))
k_white_noise$set_positions(positions_df)
# Generate the kernel's covariance matrix
k_white_noise$kernel_gen()
```

plot_beta_dists *Plot Beta Coefficients Distribution*

Description

Plot the distribution of beta values for a given list of labels.

Usage

```
plot_beta_dists(
  bktr_reg,
  labels_list,
  show_figure = TRUE,
  fig_width = 9,
  fig_height = 6,
  fig_resolution = 200
)
```

Arguments

bktr_reg	BKTRRegressor: BKTRRegressor object.
labels_list	List: List of labels tuple (spatial, temporal, feature) for which to plot the beta distribution through iterations
show_figure	Boolean: Whether to show the figure. Defaults to True.
fig_width	Integer: Figure width. Defaults to 9.
fig_height	Integer: Figure height. Defaults to 6.
fig_resolution	Numeric: Figure resolution PPI. Defaults to 200.

Value

ggplot or NULL: ggplot object or NULL if show_figure is set to FALSE.

Examples

```
# Launch MCMC sampling on a light version of the BIXI dataset
bixi_data <- BixiData$new(is_light = TRUE)
bktr_regressor <- BKTRRegressor$new(
  data_df <- bixi_data$data_df,
  spatial_positions_df = bixi_data$spatial_positions_df,
  temporal_positions_df = bixi_data$temporal_positions_df,
  burn_in_iter = 5, sampling_iter = 10) # For example only (too few iterations)
bktr_regressor$mcmc_sampling()

# Plot temporal beta coefficients for the first station and the first feature
spa_lab <- bixi_data$spatial_positions_df$location[3]
plot_beta_dists(
  bktr_regressor,
```

```
  labels_list = list(
     c(spa_lab, '2019-04-15', 'area_park'),
     c(spa_lab, '2019-04-16', 'area_park'),
     c(spa_lab, '2019-04-16', 'mean_temp_c')
  ),
)
```

`plot_covariates_beta_dists` *Plot Beta Coefficients Distribution Regrouped by Covariates*

Description

Plot the distribution of beta estimates regrouped by covariates.

Usage

```
plot_covariates_beta_dists(
  bktr_reg,
  feature_labels = NULL,
  show_figure = TRUE,
  fig_width = 9,
  fig_height = 6,
  fig_resolution = 200
)
```

Arguments

`bktr_reg`	BKTRRegressor: BKTRRegressor object.
`feature_labels`	Array or NULL: Array of feature labels for which to plot the beta estimates distribution. If NULL plot for all features.
`show_figure`	Boolean: Whether to show the figure. Defaults to True.
`fig_width`	Integer: Figure width. Defaults to 9.
`fig_height`	Integer: Figure height. Defaults to 6.
`fig_resolution`	Numeric: Figure resolution PPI. Defaults to 200.

Value

ggplot or NULL: ggplot object or NULL if show_figure is set to FALSE.

Examples

```
# Launch MCMC sampling on a light version of the BIXI dataset
bixi_data <- BixiData$new(is_light = TRUE)
bktr_regressor <- BKTRRegressor$new(
  formula = 'nb_departure ~ 1 + area_park + mean_temp_c + total_precip_mm',
  data_df <- bixi_data$data_df,
  spatial_positions_df = bixi_data$spatial_positions_df,
  temporal_positions_df = bixi_data$temporal_positions_df,
  burn_in_iter = 5, sampling_iter = 10) # For example only (too few iterations)
bktr_regressor$mcmc_sampling()

# Plot beta estimates distribution for all features
plot_covariates_beta_dists(bktr_regressor)
# Or plot for a subset of features
plot_covariates_beta_dists(bktr_regressor, c('area_park', 'mean_temp_c'))
```

---

plot_hyperparams_dists

Plot Hyperparameters Distributions

Description

Plot the distribution of hyperparameters through iterations

Usage

```
plot_hyperparams_dists(
  bktr_reg,
  hyperparameters = NULL,
  show_figure = TRUE,
  fig_width = 9,
  fig_height = 6,
  fig_resolution = 200
)
```

Arguments

bktr_reg	BKTRRegressor: BKTRRegressor object.
hyperparameters	Array or NULL: Array of hyperparameters to plot. If NULL, plot all hyperparameters. Defaults to NULL.
show_figure	Boolean: Whether to show the figure. Defaults to True.
fig_width	Integer: Figure width. Defaults to 9.
fig_height	Integer: Figure height. Defaults to 6.
fig_resolution	Numeric: Figure resolution PPI. Defaults to 200.

Value

ggplot or NULL: ggplot object or NULL if show_figure is set to FALSE.

Examples

```
# Launch MCMC sampling on a light version of the BIXI dataset
bixi_data <- BixiData$new(is_light = TRUE)
k_matern <- KernelMatern$new()
k_periodic <- KernelPeriodic$new()
bktr_regressor <- BKTRRegressor$new(
  data_df <- bixi_data$data_df,
  spatial_kernel = k_matern,
  temporal_kernel = k_periodic,
  spatial_positions_df = bixi_data$spatial_positions_df,
  temporal_positions_df = bixi_data$temporal_positions_df,
  burn_in_iter = 5, sampling_iter = 10) # For example only (too few iterations)
bktr_regressor$mcmc_sampling()

# Plot the distribution of all hyperparameters
plot_hyperparams_dists(bktr_regressor)

# Plot the distribution of the spatial kernel hyperparameters
spa_par_name <- paste0('Spatial - ', k_matern$parameters[[1]]$full_name)
plot_hyperparams_dists(bktr_regressor, spa_par_name)

# Plot the distribution of the temporal kernel hyperparameters
temp_par_names <- sapply(k_periodic$parameters, function(x) x$full_name)
temp_par_names <- paste0('Temporal - ', temp_par_names)
plot_hyperparams_dists(bktr_regressor, temp_par_names)
```

---

plot_hyperparams_traceplot

Plot Hyperparameters Traceplot

Description

Plot the evolution of hyperparameters through iterations. (Traceplot)

Usage

```
plot_hyperparams_traceplot(
  bktr_reg,
  hyperparameters = NULL,
  show_figure = TRUE,
  fig_width = 9,
  fig_height = 5.5,
  fig_resolution = 200
)
```

Arguments

bktr_reg	BKTRRegressor: BKTRRegressor object.
hyperparameters	Array or NULL: Array of hyperparameters to plot. If NULL, plot all hyperparameters. Defaults to NULL.
show_figure	Boolean: Whether to show the figure. Defaults to True.
fig_width	Integer: Figure width. Defaults to 9.
fig_height	Integer: Figure height. Defaults to 5.5.
fig_resolution	Numeric: Figure resolution PPI. Defaults to 200.

Value

ggplot or NULL: ggplot object or NULL if show_figure is set to FALSE.

Examples

```
# Launch MCMC sampling on a light version of the BIXI dataset
bixi_data <- BixiData$new(is_light = TRUE)
k_matern <- KernelMatern$new()
k_periodic <- KernelPeriodic$new()
bktr_regressor <- BKTRRegressor$new(
  data_df <- bixi_data$data_df,
  spatial_kernel = k_matern,
  temporal_kernel = k_periodic,
  spatial_positions_df = bixi_data$spatial_positions_df,
  temporal_positions_df = bixi_data$temporal_positions_df,
  burn_in_iter = 5, sampling_iter = 10) # For example only (too few iterations)
bktr_regressor$mcmc_sampling()

# Plot the traceplot of all hyperparameters
plot_hyperparams_traceplot(bktr_regressor)

# Plot the traceplot of the spatial kernel hyperparameters
spa_par_name <- paste0('Spatial - ', k_matern$parameters[[1]]$full_name)
plot_hyperparams_traceplot(bktr_regressor, spa_par_name)

# Plot the traceplot of the temporal kernel hyperparameters
temp_par_names <- sapply(k_periodic$parameters, function(x) x$full_name)
temp_par_names <- paste0('Temporal - ', temp_par_names)
plot_hyperparams_traceplot(bktr_regressor, temp_par_names)
```

---

plot_spatial_betas *Plot Spatial Beta Coefficients*

Description

Create a plot of beta values through space for a given temporal point and a set of feature labels. We use ggmap under the hood, so you need to provide a Google or Stadia API token to plot on a map. See: https://cran.r-project.org/web/packages/ggmap/readme/README.html for more details on how to get an API token.

Usage

```
plot_spatial_betas(
  bktr_reg,
  plot_feature_labels,
  temporal_point_label,
  nb_cols = 1,
  use_dark_mode = TRUE,
  show_figure = TRUE,
  zoom = 11,
  google_token = NULL,
  stadia_token = NULL,
  fig_width = 8.5,
  fig_height = 5.5,
  fig_resolution = 200
)
```

Arguments

bktr_reg	BKTRRegressor: BKTRRegressor object.
plot_feature_labels	Array: Array of feature labels to plot.
temporal_point_label	String: Temporal point label to plot.
nb_cols	Integer: The number of columns to use in the facet grid.
use_dark_mode	Boolean: Whether to use a dark mode for the geographic map or not. Defaults to TRUE.
show_figure	Boolean: Whether to show the figure. Defaults to True.
zoom	Integer: Zoom level for the geographic map. Defaults to 11.
google_token	String or NULL: Google API token to use for the geographic map. Defaults to NULL.
stadia_token	String or NULL: Stadia API token to use for the geographic map. Defaults to NULL.
fig_width	Numeric: Figure width when figure is shown. Defaults to 8.5.
fig_height	Numeric: Figure height when figure is shown. Defaults to 5.5.
fig_resolution	Numeric: Figure resolution PPI. Defaults to 200.

Value

ggplot or NULL: ggplot object or NULL if show_figure is set to FALSE.

Examples

```
# Launch MCMC sampling on a light version of the BIXI dataset
bixi_data <- BixiData$new(is_light = TRUE)
bktr_regressor <- BKTRRegressor$new(
  data_df <- bixi_data$data_df,
  spatial_positions_df = bixi_data$spatial_positions_df,
  temporal_positions_df = bixi_data$temporal_positions_df,
  burn_in_iter = 5, sampling_iter = 10) # For example only (too few iterations)
bktr_regressor$mcmc_sampling()

# Plot spatial beta coefficients for the first time point and the two features
# Use your own Google API token instead of 'GOOGLE_API_TOKEN'
plot_spatial_betas(
  bktr_regressor,
  plot_feature_labels = c('mean_temp_c', 'area_park'),
  temporal_point_label = bixi_data$temporal_positions_df$time[1],
  google_token = 'GOOGLE_API_TOKEN')

# We can also use light mode and plot the maps side by side
# Use your own Stadia API token instead of 'STADIA_API_TOKEN'
plot_spatial_betas(
  bktr_regressor,
  plot_feature_labels = c('mean_temp_c', 'area_park', 'total_precip_mm'),
  temporal_point_label = bixi_data$temporal_positions_df$time[10],
  use_dark_mode = FALSE, nb_cols = 3, stadia_token = 'STADIA_API_TOKEN')
```

---

plot_temporal_betas *Plot Temporal Beta Coefficients*

Description

Create a plot of the beta values through time for a given spatial point and a set of feature labels.

Usage

```
plot_temporal_betas(
  bktr_reg,
  plot_feature_labels,
  spatial_point_label,
  date_format = "%Y-%m-%d",
  show_figure = TRUE,
  fig_width = 8.5,
  fig_height = 5.5,
  fig_resolution = 200
)
```

Arguments

bktr_reg	BKTRRegressor: BKTRRegressor object.
plot_feature_labels	Array: Array of feature labels to plot.
spatial_point_label	String: Spatial point label to plot.
date_format	String: Format of the date to use in bktr dataframes for the time. Defaults to '%Y-%m-%d'.
show_figure	Boolean: Whether to show the figure. Defaults to True.
fig_width	Numeric: Figure width when figure is shown. Defaults to 8.5.
fig_height	Numeric: Figure height when figure is shown. Defaults to 5.5.
fig_resolution	Numeric: Figure resolution PPI. Defaults to 200.

Value

ggplot or NULL: ggplot object or NULL if show_figure is set to FALSE.

Examples

```
# Launch MCMC sampling on a light version of the BIXI dataset
bixi_data <- BixiData$new(is_light = TRUE)
bktr_regressor <- BKTRRegressor$new(
  data_df <- bixi_data$data_df,
  spatial_positions_df = bixi_data$spatial_positions_df,
  temporal_positions_df = bixi_data$temporal_positions_df,
  burn_in_iter = 5, sampling_iter = 10) # For example only (too few iterations)
bktr_regressor$mcmc_sampling()

# Plot temporal beta coefficients for the first station and the two features
plot_temporal_betas(
  bktr_regressor,
  plot_feature_labels = c('mean_temp_c', 'area_park'),
  spatial_point_label = bixi_data$spatial_positions_df$location[1])
```

---

plot_y_estimates *Plot Y Estimates*

Description

Plot y estimates vs observed y values.

Usage

```
plot_y_estimates(
  bktr_reg,
  show_figure = TRUE,
  fig_width = 5,
  fig_height = 5,
  fig_resolution = 200,
  fig_title = "y estimates vs observed y values"
)
```

Arguments

bktr_reg	BKTRRegressor: BKTRRegressor object.
show_figure	Boolean: Whether to show the figure. Defaults to True.
fig_width	Numeric: Figure width when figure is shown. Defaults to 5.
fig_height	Numeric: Figure height when figure is shown. Defaults to 5.
fig_resolution	Numeric: Figure resolution PPI when figure is shown. Defaults to 200.
fig_title	String or NULL: Figure title if provided. Defaults to 'y estimates vs observed y values'

Value

ggplot or NULL: ggplot object or NULL if show_figure is set to FALSE.

Examples

```
# Launch MCMC sampling on a light version of the BIXI dataset
bixi_data <- BixiData$new(is_light = TRUE)
bktr_regressor <- BKTRRegressor$new(
  data_df <- bixi_data$data_df,
  spatial_positions_df = bixi_data$spatial_positions_df,
  temporal_positions_df = bixi_data$temporal_positions_df,
  burn_in_iter = 5, sampling_iter = 10) # For example only (too few iterations)
bktr_regressor$mcmc_sampling()

# Plot Y estimates vs observed y values
plot_y_estimates(bktr_regressor)
```

---

print.BKTRRegressor *Print the summary of a BKTRRegressor instance*

Description

Print the summary of a BKTRRegressor instance

Usage

```
## S3 method for class 'BKTRRegressor'
print(x, ...)
```

Arguments

x	A BKTRRegressor instance
...	Additional arguments to comply with generic function

`reshape_covariate_dfs` *Function used to transform covariates coming from two dataframes one for spatial and one for temporal into a single dataframe with the right shape for the BKTR Regressor. This is useful when the temporal covariates do not vary trough space and the spatial covariates do not vary trough time (Like in the BIXI example). The function also adds a column for the target variable at the beginning of the dataframe.*

Description

Function used to transform covariates coming from two dataframes one for spatial and one for temporal into a single dataframe with the right shape for the BKTR Regressor. This is useful when the temporal covariates do not vary trough space and the spatial covariates do not vary trough time (Like in the BIXI example). The function also adds a column for the target variable at the beginning of the dataframe.

Usage

```
reshape_covariate_dfs(spatial_df, temporal_df, y_df, y_column_name = "y")
```

Arguments

spatial_df	data.table: Spatial covariates dataframe with an index named location and a shape of (n_locations, n_spatial_covariates)
temporal_df	data.table: Temporal covariates dataframe with an index named time and a shape of (n_times, n_temporal_covariates)
y_df	data.table: The dataframe containing the target variable. It should have a shape of (n_locations, n_times). The columns and index names of this dataframe should be correspond to the one of the spatial_df and temporal_df.
y_column_name	string: The name of the target variable column in y_df. Default to 'y'.

Value

data.table: The reshaped covariates dataframe with a shape of (n_locations * n_times, 1 + n_spatial_covariates + n_temporal_covariates). The first two columns are the indexes (location, time), the following column is the target variable and the other columns are the covariates.

Examples

```
# Let's reshape the BIXI dataframes without normalization
new_data_df <- reshape_covariate_dfs(
  spatial_df = BKTR::bixi_spatial_features,
  temporal_df = BKTR::bixi_temporal_features,
  y_df = BKTR::bixi_station_departures,
  y_column_name = 'whole_nb_departure')
# The resulting dataframe has the right shape to be a BKTRRegressor data_df
head(new_data_df)
```

---

simulate_spatiotemporal_data

Simulate Spatiotemporal Data Using Kernel Covariances.

Description

Simulate Spatiotemporal Data Using Kernel Covariances.

Usage

```
simulate_spatiotemporal_data(
  nb_locations,
  nb_time_points,
  nb_spatial_dimensions,
  spatial_scale,
  time_scale,
  spatial_covariates_means,
  temporal_covariates_means,
  spatial_kernel,
  temporal_kernel,
  noise_variance_scale
)
```

Arguments

nb_locations	Integer: Number of spatial locations
nb_time_points	Integer: Number of time points
nb_spatial_dimensions	Integer: Number of spatial dimensions
spatial_scale	Numeric: Spatial scale
time_scale	Numeric: Time scale
spatial_covariates_means	Vector: Spatial covariates means
temporal_covariates_means	Vector: Temporal covariates means

`spatial_kernel` Kernel: Spatial kernel

`temporal_kernel`
Kernel: Temporal kernel

`noise_variance_scale`
Numeric: Noise variance scale

Value

A list containing 4 dataframes: - 'data_df' contains the response variable and the covariates - 'spatial_positions_df' contains the spatial locations and their coordinates - 'temporal_positions_df' contains the time points and their coordinates - 'beta_df' contains the true beta coefficients

Examples

```
# Simulate data with 20 locations, 30 time points, in 2D spatial dimensions
# with 3 spatial covariates and 2 temporal covariates
simu_data <- simulate_spatiotemporal_data(
   nb_locations=20,
   nb_time_points=30,
   nb_spatial_dimensions=2,
   spatial_scale=10,
   time_scale=10,
   spatial_covariates_means=c(0, 2, 4),
   temporal_covariates_means=c(1, 3),
   spatial_kernel=KernelMatern$new(),
   temporal_kernel=KernelSE$new(),
   noise_variance_scale=1)

# The dataframes are similar to bixi_data, we have:
# - data_df
head(simu_data$data_df)
# - spatial_positions_df
head(simu_data$spatial_positions_df)
# - temporal_positions_df
head(simu_data$temporal_positions_df)

# We also obtain the true beta coefficients used to simulate the data
head(simu_data$beta_df)
```

---

summary.BKTRRegressor *Summarize a BKTRRegressor instance*

Description

Summarize a BKTRRegressor instance

Usage

```
## S3 method for class 'BKTRRegressor'
summary(object, ...)
```

Arguments

object	A BKTRRegressor instance
...	Additional arguments to comply with generic function

TensorOperator *R6 singleton that contains the configuration for the tensor backend*

Description

Tensor backend configuration and methods for all the tensor operations in BKTR

Super class

R6P::Singleton -> TensorOperator

Public fields

fp_type The floating point type to use for the tensor operations

fp_device The device to use for the tensor operations

Methods

Public methods:

- TensorOperator$new()
- TensorOperator$set_params()
- TensorOperator$get_default_jitter()
- TensorOperator$tensor()
- TensorOperator$is_tensor()
- TensorOperator$eye()
- TensorOperator$ones()
- TensorOperator$zeros()
- TensorOperator$rand()
- TensorOperator$randn()
- TensorOperator$randn_like()
- TensorOperator$arange()
- TensorOperator$rand_choice()
- TensorOperator$kronecker_prod()
- TensorOperator$khatri_rao_prod()
- TensorOperator$clone()

Method `new()`**:** Initialize the tensor operator with the given floating point type and device

Usage:

```
TensorOperator$new(fp_type = "float64", fp_device = "cpu")
```

Arguments:

`fp_type` The floating point type to use for the tensor operations (either "float64" or "float32")

`fp_device` The device to use for the tensor operations (either "cpu" or "cuda")

Returns: A new tensor operator instance

Method `set_params()`**:** Set the tensor operator parameters

Usage:

```
TensorOperator$set_params(fp_type = NULL, fp_device = NULL, seed = NULL)
```

Arguments:

`fp_type` The floating point type to use for the tensor operations (either "float64" or "float32")

`fp_device` The device to use for the tensor operations (either "cpu" or "cuda")

`seed` The seed to use for the random number generator

Method `get_default_jitter()`**:** Get the default jitter value for the floating point type used by the tensor operator

Usage:

```
TensorOperator$get_default_jitter()
```

Returns: The default jitter value for the floating point type used by the tensor operator

Method `tensor()`**:** Create a tensor from a vector or matrix of data with the tensor operator dtype and device

Usage:

```
TensorOperator$tensor(tensor_data)
```

Arguments:

`tensor_data` The vector or matrix of data to create the tensor from

Returns: A new tensor with the tensor operator dtype and device

Method `is_tensor()`**:** Check if a provided object is a tensor

Usage:

```
TensorOperator$is_tensor(tensor)
```

Arguments:

`tensor` The object to check

Returns: A boolean indicating if the object is a tensor

Method `eye()`**:** Create a tensor with a diagonal of ones and zeros with the tensor operator dtype and device for a given dimension

Usage:

```
TensorOperator$eye(eye_dim)
```

Arguments:

`eye_dim` The dimension of the tensor to create

Returns: A new tensor with a diagonal of ones and zeros with the tensor operator dtype and device

Method `ones()`**:** Create a tensor of ones with the tensor operator dtype and device for a given dimension

Usage:

```
TensorOperator$ones(tsr_dim)
```

Arguments:

`tsr_dim` The dimension of the tensor to create

Returns: A new tensor of ones with the tensor operator dtype and device

Method `zeros()`**:** Create a tensor of zeros with the tensor operator dtype and device for a given dimension

Usage:

```
TensorOperator$zeros(tsr_dim)
```

Arguments:

`tsr_dim` The dimension of the tensor to create

Returns: A new tensor of zeros with the tensor operator dtype and device

Method `rand()`**:** Create a tensor of random uniform values with the tensor operator dtype and device for a given dimension

Usage:

```
TensorOperator$rand(tsr_dim)
```

Arguments:

`tsr_dim` The dimension of the tensor to create

Returns: A new tensor of random values with the tensor operator dtype and device

Method `randn()`**:** Create a tensor of random normal values with the tensor operator dtype and device for a given dimension

Usage:

```
TensorOperator$randn(tsr_dim)
```

Arguments:

`tsr_dim` The dimension of the tensor to create

Returns: A new tensor of random normal values with the tensor operator dtype and device

Method `randn_like()`**:** Create a tensor of random uniform values with the same shape as a given tensor with the tensor operator dtype and device

Usage:

```
TensorOperator$randn_like(input_tensor)
```

Arguments:

`input_tensor` The tensor to use as a shape reference

Returns: A new tensor of random uniform values with the same shape as a given tensor

Method `arange()`**:** Create a tensor of a range of values with the tensor operator dtype and device for a given start and end

Usage:

```
TensorOperator$arange(start, end)
```

Arguments:

`start` The start of the range

`end` The end of the range

Returns: A new tensor of a range of values with the tensor operator dtype and device

Method `rand_choice()`**:** Choose random values from a tensor for a given number of samples

Usage:

```
TensorOperator$rand_choice(
  choices_tsr,
  nb_sample,
  use_replace = FALSE,
  weights_tsr = NULL
)
```

Arguments:

`choices_tsr` The tensor to choose values from

`nb_sample` The number of samples to choose

`use_replace` A boolean indicating if the sampling should be done with replacement. Defaults to FALSE

`weights_tsr` The weights to use for the sampling. If NULL, the sampling is uniform. Defaults to NULL

Returns: A new tensor of randomly chosen values from a tensor

Method `kronecker_prod()`**:** Efficiently compute the kronecker product of two matrices in tensor format

Usage:

```
TensorOperator$kronecker_prod(a, b)
```

Arguments:

`a` The first tensor

`b` The second tensor

Returns: The kronecker product of the two matrices

Method `khatri_rao_prod()`**:** Efficiently compute the khatri rao product of two matrices in tensor format having the same number of columns

Usage:

```
TensorOperator$khatri_rao_prod(a, b)
```

Arguments:

`a` The first tensor

b The second tensor

Returns: The khatri rao product of the two matrices

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TensorOperator$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Set the seed, setup the tensor floating point type and device
TSR$set_params(fp_type='float64', fp_device='cpu', seed=42)
# Create a tensor from a vector
TSR$tensor(c(1, 2, 3))
# Create a tensor from a matrix
TSR$tensor(matrix(c(1, 2, 3, 4), nrow=2))
# Create a 3x3 tensor with a diagonal of ones and zeros elsewhere
TSR$eye(3)
# Create a tensor of ones (with 6 elements, 2 rows and 3 columns)
TSR$ones(c(2, 3))
# Create a tensor of zeros (with 12 elements, 3 rows and 4 columns)
TSR$zeros(c(3, 4))
# Create a tensor of random uniform values (with 6 elements)
TSR$rand(c(2, 3))
# Create a tensor of random normal values (with 6 elements)
TSR$randn(c(2, 3))
# Create a tensor of random normal values with the same shape as a given tensor
tsr_a <- TSR$randn(c(2, 3))
TSR$randn_like(tsr_a)
# Create a tensor of a range of values (1, 2, 3, 4)
TSR$arange(1, 4)
# Choose two random values from a given tensor without replacement
tsr_b <- TSR$rand(6)
TSR$rand_choice(tsr_b, 2)
# Use the tensor operator to compute the kronecker product of two 2x2 matrices
tsr_c <- TSR$tensor(matrix(c(1, 2, 3, 4), nrow=2))
tsr_d <- TSR$tensor(matrix(c(5, 6, 7, 8), nrow=2))
TSR$kronecker_prod(tsr_c, tsr_d) # Returns a 4x4 tensor
# Use the tensor operator to compute the khatri rao product of two 2x2 matrices
TSR$khatri_rao_prod(tsr_c, tsr_d) # Returns a 4x2 tensor
# Check if a given object is a tensor
TSR$is_tensor(tsr_d) # Returns TRUE
TSR$is_tensor(TSR$eye(2)) # Returns TRUE
TSR$is_tensor(1) # Returns FALSE
```

TSR *Tensor Operator Singleton*

Description

Singleton instance of the `TensorOperator` class that contains all informations related the tensor API; tensor methods, used data type and used device.

Usage

```
TSR
```

Format

An object of class `TensorOperator` (inherits from `Singleton`, `R6`) of length 19.

Index