Ambrosia

Release 0.4.1

MTS Big Data A/B Team

Apr 21, 2023

QUICKSTART

Ambrosia is a Python library for A/B tests design, split and effect measurement. It provides rich set of methods for conducting full A/B testing pipeline.

The project is intended for use in research and production environments based on data in pandas and Spark format.

CHAPTER

ONE

KEY FUNCTIONALITY

- Pilots design
- Multi-group split
- Matching of new control group to the existing pilot
- Experiments result evaluation as p-value, point estimate of effect and confidence interval
- Data preprocessing
- Experiments acceleration

1.1 Installation

You can always get the newest *Ambrosia* release using `pip`. Stable version is released on every tag to `main` branch.

```
pip install ambrosia
```

Starting from version `0.4.0`, the ability to process PySpark data is optional and can be enabled using `pip` extras during the installation.

```
pip install ambrosia[spark]
```

Python versions support

Ambrosia requires Python 3.7+

1.2 A/B testing with *Ambrosia* in a Nutshell

Imagine that you want to run your own A/B test, and after the product analysis and gathering ideas into a hypothesis, you usually have to go through several routine calculation steps: from collecting and transforming raw data to measuring the statistical significance of the experiment result and confidence intervals construction.

In order to solve the problem of carrying out a large number of calculations using various techniques, in *Ambrosia*, we have identified the following stages of experiments and provide tools and automation for them:

- Process

Raw data aggregation, outliers removal, metric transformation as well as various methods for experiments acceleration. Storable data processing pipelines that can be reused.

- Design

Experiment parameters such as effect uplift, groups size, and experiment statistical power are designed using metrics historical data by a theoretical or empirical approaches.

- Split

Group split methods support different strategies and multi-group split, which allows to quickly create control and test groups of interest. Currently, only batch data splitting methods are supported.

- Test

Tools for the statistical inference are able to calculate relative and absolute effects, construct corresponding confidence intervals for continious and binary variables. A significant number of statistical tests is supported, such as t-test, non-parametric, bootstrap, and others.

1.3 Usage

The main functionality of *Ambrosia* is contained in several core classes and methods, which are autonomic for each stage of an experiment and have very intuitive interface.

Below is a brief overview example of using a set of three classes to conduct some simple experiment.

Designer

```
from ambrosia.designer import Designer
designer = Designer(dataframe=df, effects=1.2, metrics='portfel_clc') # 20% effect, and␣
↪loaded data frame df
designer.run('size')
```

Splitter

```
from ambrosia.splitter import Splitter
splitter = Splitter(dataframe=df, id_column='id') # loaded data frame df with column␣
↪with id - 'id'
splitter.run(groups_size=500, method='simple')
```

Tester

```
from ambrosia.tester import Tester
tester = Tester(dataframe=df, column_groups='group') # loaded data frame df with groups␣
↪info 'group'
tester.run(metrics='retention', method='theory', criterion='ttest')
```

Explore further

To better understand how to use *Ambrosia* in your project and dive into the functionality of the library, check out the Core Functionality and Usage Examples sections of this documentation.

1.4 Data Preprocessing

The tools from this subsection allow to automatically perform various stages of processing experimental data and save the specified configurations for repeated data transformations.

Data preprocessing tools:

1.4.1 Aggregation

AggregatePreprocessor	Preprocessing class for data aggregation.

class ambrosia.preprocessing.**AggregatePreprocessor**(*categorial_method='mode'*, *real_method='sum'*)

Preprocessing class for data aggregation.

Can group data by multiple columns and aggregate it using methods for real and categorial features.

Parameters

categorial_method
[types.MethodType, default: "mode"] Aggregation method for categorial variables that will become as a default behavior.

real_method
[types.MethodType, default: "sum"] Aggregation method for real variables that will become as a default behavior.

Attributes

categorial_method
[types.MethodType] Default aggregation method for categorial variables.

real_method
[types.MethodType] Default aggregation method for real variables.

groupby_columns
[types.ColumnNamesType] Columns which were used for groupping in the last aggregation. Gets value after fitting the class instance.

agg_params
[Dict] Dictionary with aggregation rules which was used in the last aggregation. Gets value after fitting the class instance.

get_params_dict()

Returns dictionary with parameters of the last run() or transform() call.

fit(*dataframe*, *groupby_columns*, *agg_params=None*, *real_cols=None*, *categorial_cols=None*)

Fit preprocessor with parameters of aggregation.

Aggregation will be performed using passed dictionary with defined aggregation conditions for each columns of interest, or lists of columns with default class aggregation behavior.

Parameters

dataframe
[pd.DataFrame] Table with selected columns.

groupby_columns
[types.ColumnNamesType] Columns for GROUP BY.

agg_params
[Dict, optional] Dictionary with aggregation parameters.

real_cols
[types.ColumnNamesType, optional] Columns with real metrics. Overriden by `agg_params` parameter and could be passed if expected default aggregation behavior.

categorial_cols
[types.ColumnNamesType, optional] Columns with categorial metrics Overriden by `agg_params` parameter and could be passed if expected default aggregation behavior.

Returns

self
[object] Instance object.

`transform`(*dataframe*)

Apply table transformation by its aggregation with prefitted parameters.

Parameters

dataframe
[pd.DataFrame] Table to aggregate.

Returns

agg_table
[pd.DataFrame] Aggregated table.

1.4.2 Outliers removal

RobustPreprocessor	Unit for simple robust transformation for avoiding outliers in data.
IQRPreprocessor	Unit for IQR transformation of the data to exclude outliers.

`class` ambrosia.preprocessing.**RobustPreprocessor**(*verbose=True*)

Unit for simple robust transformation for avoiding outliers in data.

It cuts the alpha percentage of distribution from head, tail or both sides for each given metric. The data distribution structure assumed to present as small alpha part of outliers, followed by the normal part of the data with another alpha part of outliers at the end of the distribution.

Parameters

verbose
[bool, default: `True`] If `True` will show info about the transformation of passed columns.

Examples

```
>>> robust = RobustPreprocessor(verbose=True)
>>> robust.fit(dataframe, ['column1', 'column2'], alpha=0.05)
>>> robust.transform(dataframe, inplace=True)
```

You can pass one or number of columns, if several columns are passed it will drop in total alpha percent of extreme values for each column.

Attributes

params
[Dict] Dictionary with operational parameters of the instance. Updated after calling the `fit` method.

verbose
[bool] Verbose info flag.

available_tails
[List] List of the available tail type names to preprocess

non_serializable_params: List
List of the class parameters that should be converted to lists in order to serialize.

fitted
[bool] Fit flag.

`fit`(*dataframe*, *column_names*, *alpha=0.05*, *tail='both'*)

Fit to calculate robust parameters for the selected columns.

Parameters

dataframe
[pd.DataFrame] Dataframe to calculate quantiles.

column_names
[ColumnNamesType] One or number of columns in the dataframe.

alpha
[Union[float, np.ndarray], default: `0.05`] The percentage of removed data from head and tail.

tail
[str, default: `"both"`] Part of distribution to be removed. Can be `"left"`, `"right"` or `"both"`.

Returns

self
[object] Instance object.

`transform`(*dataframe*, *inplace=False*)

Remove objects from the dataframe which are in the head, tail or both alpha parts of chosen metrics distributions.

Parameters

dataframe
[pd.DataFrame] Dataframe to transform.

inplace
[bool, default: `False`] If `True` transforms the given dataframe, otherwise copy and returns an another one.

Returns

df
[Union[pd.DataFrame, None]] Transformed dataframe or None

`fit_transform`(*dataframe*, *column_names*, *alpha=0.05*, *tail='both'*, *inplace=False*)

Fit preprocessor parameters using given dataframe and transform it.

Parameters

dataframe
[pd.DataFrame] Dataframe to calculate quantiles and for further transformation.

column_names
[ColumnNamesType] One or number of columns in the dataframe.

alpha
[Union[float, np.ndarray], default: `0.05`] The percentage of removed data from head and tail.

tail
[str, default: `"both"`] Part of distribution to be removed. Can be `"left"`, `"right"` or `"both"`.

inplace
[bool, default: `False`] If `True` transforms the given dataframe, otherwise copy and returns an another one.

Returns

df
[Union[pd.DataFrame, None]] Transformed dataframe or None

`store_params`(*store_path*)

Parameters

store_path
[Path] Path where parameters will be stored in a json format.

`load_params`(*load_path*)

Parameters

load_path
[Path] Path to json file with parameters.

`class` ambrosia.preprocessing.`IQRPreprocessor`(*verbose=True*)

Unit for IQR transformation of the data to exclude outliers.

It cuts the points from the distribution which are behind the range of 0.25 quantile - 1,5 * iqr and 0.75 quantile + 1,5 * iqr for each given metric.

Parameters

verbose
[bool, default: `True`] If `True` will show info about the transformation of passed columns.

Examples

```
>>> iqr = IQRPreprocessor(verbose=True)
>>> iqr.fit(dataframe, ['column1', 'column2'])
>>> iqr.transform(dataframe, inplace=True)
```

You can pass one or number of columns, if several columns are passed it will drop extreme values for each column.

Attributes

params
[Dict] Dictionary with operational parameters of the instance. Updated after calling the `fit` method.

verbose
[bool] Verbose info flag.

non_serializable_params: List
List of the class parameters that should be converted to lists in order to serialize.

fitted
[bool] Fit flag.

`fit`(*dataframe*, *column_names*)

Fit to calculate iqr parameters for the selected columns.

Parameters

dataframe
[pd.DataFrame] Dataframe to calculate quantiles.

column_names
[ColumnNamesType] One or number of columns in the dataframe.

Returns

self
[object] Instance object.

`transform`(*dataframe*, *inplace=False*)

Remove objects from the dataframe which are behind maximum and minimum values of boxplots for each metric distribution.

Parameters

dataframe
[pd.DataFrame] Dataframe to transform.

inplace
[bool, default: `False`] If `True` transforms the given dataframe, otherwise copy and returns an another one.

Returns

df
[Union[pd.DataFrame, None]] Transformed dataframe or None

`fit_transform`(*dataframe*, *column_names*, *inplace=False*)

Fit preprocessor parameters using given dataframe and transform it.

Parameters

dataframe
[pd.DataFrame] Dataframe to calculate quantiles and for further transformation.

column_names
[ColumnNamesType] One or number of columns in the dataframe.

inplace
[bool, default: `False`] If `True` transforms the given dataframe, otherwise copy and returns an another one.

Returns

df
[Union[pd.DataFrame, None]] Transformed dataframe or None

`store_params`(*store_path*)

Parameters

store_path
[Path] Path where parameters will be stored in a json format.

`load_params`(*load_path*)

Parameters

load_path
[Path] Path to json file with parameters.

1.4.3 Simple metric transformations

BoxCoxTransformer	Unit for a Box-Cox transformation of the pandas data.
LogTransformer	Unit for a logarithmic transformation of the pandas data.

`class` ambrosia.preprocessing.**BoxCoxTransformer**

Unit for a Box-Cox transformation of the pandas data.

A Box Cox transformation helps to transform non-normal dependent variables into a normal shape. All variables values must be positive.

Optimal transformation lambdas are selected automatically during the transformer fit process.

Examples

```
>>> boxcox = BoxCoxTransformer()
>>> boxcox.fit(dataframe, ['column1', 'column2'])
>>> boxcox.transform(dataframe, inplace=True)
```

Attributes

column_names
[List] Names of column which will be selected for the transformation.

lambda_
[np.ndarray] Array of parameters using during the transformation of the selected columns.

fitted
[bool] Fit flag.

`fit`(*dataframe*, *column_names*)

Fit to calculate transformation parameters for the selected columns.

Parameters

dataframe
[pd.DataFrame] Dataframe to calculate optimal transformation parameters.

column_names
[ColumnNamesType] One or number of columns in the dataframe.

Returns

self
[object] Instance object.

`transform`(*dataframe*, *inplace=False*)

Apply Box-Cox transformation for the data.

Parameters

dataframe
[pd.DataFrame] Dataframe to transform.

inplace
[bool, default: `False`] If `True` transforms the given dataframe, otherwise copy and returns an another one.

Returns

df
[Union[pd.DataFrame, None]] Transformed dataframe or None

`fit_transform`(*dataframe*, *column_names*, *inplace=False*)

Fit transformer parameters using given dataframe and transform it.

Parameters

dataframe
[pd.DataFrame] Dataframe for calculation of optimal parameters and further transformation.

column_names
[ColumnNamesType] One or number of columns in the dataframe.

inplace
[bool, default: `False`] If `True` transforms the given dataframe, otherwise copy and returns an another one.

Returns

df
[Union[pd.DataFrame, None]] Transformed dataframe or None

`store_params`(*store_path*)

Parameters

store_path
[Path] Path where parameters will be stored in a json format.

load_params(*load_path*)

Parameters

load_path
[Path] Path to json file with parameters.

class ambrosia.preprocessing.**LogTransformer**

Unit for a logarithmic transformation of the pandas data.

A logarithmic transformation helps to transform some metrics distributions into a more normal shape and reduce the variance. All metrics values must be positive.

Examples

```
>>> log = LogTransformer()
>>> log.fit(dataframe, ['column1', 'column2'])
>>> log.transform(dataframe, inplace=True)
```

Attributes

column_names
[List] Names of column which will be selected for the transformation.

fitted
[bool] Fit flag.

fit(*dataframe*, *column_names*)

Fit names of the selected columns.

Parameters

dataframe
[pd.DataFrame] Dataframe with metrics.

column_names
[ColumnNamesType] One or number of columns in the dataframe.

Returns

self
[object] Instance object.

transform(*dataframe*, *inplace=False*)

Apply log transformation for the data.

Parameters

dataframe
[pd.DataFrame] Dataframe to transform.

inplace
[bool, default: `False`] If `True` transforms the given dataframe, otherwise copy and returns an another one.

Returns

df
[Union[pd.DataFrame, None]] Transformed dataframe or None

fit_transform(*dataframe*, *column_names*, *inplace=False*)

Fit transformer parameters using given dataframe and transform it.

Only column names are fittable.

Parameters

dataframe
[pd.DataFrame] Dataframe to transform.

column_names
[ColumnNamesType] One or number of columns in the dataframe.

inplace
[bool, default: `False`] If `True` transforms the given dataframe, otherwise copy and returns an another one.

Returns

df
[Union[pd.DataFrame, None]] Transformed dataframe or None

store_params(*store_path*)

Parameters

store_path
[Path] Path where parameters will be stored in a json format.

load_params(*load_path*)

Parameters

load_path
[Path] Path to json file with parameters.

1.4.4 Advanced metric transformations

Cuped	Class for data CUPED transformation.
MultiCuped	Class for data Multi CUPED transformation.
MLVarianceReducer	Machine Learning approach for variance reduction.

class ambrosia.preprocessing.**Cuped**(*verbose=True*)

Class for data CUPED transformation.

https://towardsdatascience.com/how-to-double-a-b-testing-speed-with-cuped-f80460825a90 Y_hat = Y - theta * X theta := cov(X, Y) / Var(Y) It is important, that the mean covariance metric did not change over time!!!

Parameters

verbose
[bool, default: `True`] If `True` will print in sys.stdout the information about the variance reduction.

Examples

Suppose we have the dataframe with users info which contains two columns: a "target" columns and a column with metric "income". Let us can assume, that over time, the average of the "income" values do not change. Then, we can use CUPED transformation based on "income" data to reduce "target" column variation.

```
>>> cuped_transformer = Cuped(dataframe, 'target', verbose=True)
>>> cuped_transformer.fit_transform(
>>>     dataframe=dataframe
>>>     target_column='target'
>>>     covariate_column='income',
>>>     transformed_name='cuped_target'
>>>     inplace=True,
>>> )
```

Now in the dataframe a new column "cuped_target" appeared, we can use it to design our experiment and estimate variance reduction. For further CUPED usage in the future experiment, let us store the parameters:

```
>>> cuped_transformer.store_params('cuped_transform_params.json')
```

Now we conduct an experiment and want to transform our data to reduce its variation:

```
>>> cuped_transformation = Cuped()
>>> cuped_transformation.load_params('cuped_transform_params.json')
>>> cuped_transformation.transform(
>>>     dataframe=exp_results,
>>>     inplace=True,
>>> )
```

Attributes

params
: [Dict] Parameters of instance that will be updated after calling fit() method. Include: - target column name - covariate column name - name of column after the transformation - linear coefficient for CUPED transformation. - bias value for mean equality

verbose
: [bool] Verbose info flag.

fitted
: [bool] Flag if class was fitted.

Methods

get_params_dict()	Returns dictionary with params if fit() method has been previously called.
load_params_dict(params)	Load params from a dictionary.
store_params(store_path)	Store params to json file if fit() method has been previously called.
load_params(load_path)	Load params from a json file.
fit(covariate_column)	Fit model using a specific covariate column.
transform(covariate_column, inplace, name)	Transform target column after a class instance fitting.
fit_transform(covariate_column, inplace, name)	Combination of fit() and transform() methods.

`fit`(*dataframe*, *target_column*, *covariate_column*, *transformed_name=None*)

Fit to calculate CUPED parameters for target column using given covariate column and data.

Parameters

dataframe
[pd.DataFrame] Table with data for the calculation of CUPED parameters.

target_column
[ColumnNameType] Column from the dataframe, for which CUPED transformation will be applied.

covariate_column
[ColumnNameType] Column which will be used as the covariate in CUPED transformation.

transformed_name
[ColumnNamesType, optional] Name for the new transformed target column, if is not defined it will be generated automatically.

`transform`(*dataframe*, *inplace=False*)

Make CUPED transformation for the target column.

Could be performed inplace or not.

Parameters

dataframe
[pd.DataFrame] Table with data for CUPED transformation.

inplace
[bool, default: `False`] If is `True`, then method returns `None` and sets a new column for the original dataframe. Otherwise return copied dataframe with a new column.

`fit_transform`(*dataframe*, *target_column*, *covariate_column*, *transformed_name=None*, *inplace=False*)

Combination of fit() and transform() methods.

Parameters

dataframe
[pd.DataFrame] Table with data for fitting and applying CUPED transformation.

target_column
[ColumnNameType] Column from the dataframe, for which CUPED transformation will be applied.

covariate_column
[ColumnNameType] Column which will be used as the covariate.

transformed_name
[ColumnNamesType, optional] Name for the new transformed target column, if is not defined it will be generated automatically.

inplace
[bool, default: `False`] If is `True`, then method returns `None` and sets a new column for the original dataframe. Otherwise return copied dataframe with a new column.

store_params(*store_path*)

Parameters

store_path
[Path] Path where parameters will be stored in a json format.

load_params(*load_path*)

Parameters

load_path
[Path] Path to json file with parameters.

class ambrosia.preprocessing.**MultiCuped**(*verbose=True*)

Class for data Multi CUPED transformation.

Y_hat = Y - X theta (Matrix multiplication) theta := argmin Var (Y - X theta) It is important, that the mean covariance metric do not change over time!!!

Parameters

verbose
[bool, default: `True`] If `True` will print in sys.stdout the information about the variance reduction.

Examples

We have dataframe with users info with column 'target' and columns 'income' and 'age'. We can assume, that over time, the average of this covariate values does not change. Then, we can use multi cuped transformation to reduce variation.

Suppose we have the dataframe with users info which contains two columns: a "target" columns and columns "income" and "age". Let us can assume, that over time, the average of the "income" and "age" values do not change. Then, we can use Multi CUPED transformation based on "income" and "age" data in order to reduce "target" column variation.

```
>>> cuped_transformer = MultiCuped(verbose=True)
>>> cuped_transformer.fit_transform(
>>>     dataframe=dataframe
>>>     target_column='target'
>>>     ['income', 'age'],
>>>     transformed_name='cuped_target'
>>>     inplace=True,
>>> )
```

Now in the dataframe a new column "cuped_target" appeared, we can use it to design our experiment and estimate variance reduction. For further Multi CUPED usage in the future experiment, let us store the parameters:

```
>>> cuped_transformer.store_params('cuped_transform_params.json')
```

Now we conduct an experiment and want to transform our data to reduce its variation:

```
>>> cuped_transformation = MultiCuped()
>>> cuped_transformation.load_params('cuped_transform_params.json')
>>> cuped_transformation.transform(
>>>     exp_results,
>>>     inplace=True,
>>> )
```

Attributes

params
[Dict] Parameters of instance that will be updated after calling fit() method. Include: - target column name - covariate columns names - name of column after the transformation - linear coefficients for Multi CUPED transformation. - bias value for mean equality

verbose
[bool] Verbose info flag.

fitted
[bool] Flag if class was fitted.

Methods

get_params_dict()	Returns dictionary with params if fit() method has been previously called.
load_params_dict(params)	Load params from a dictionary.
store_params(store_path)	Store params to json file if fit() method has been previously called.
load_params(load_path)	Load params from a json file.
fit(covariate_column)	Fit model using covariate columns.
transform(covariate_column, inplace, name)	Transform target column after a class instance fitting.
fit_transform(covariate_column, inplace, name)	Combination of fit() and transform() methods.

fit(*dataframe*, *target_column*, *covariate_columns*, *transformed_name=None*)

Fit to calculate Multi CUPED parameters for target column using selected covariate columns.

Parameters

dataframe
[pd.DataFrame] Table with data for the calculation of CUPED parameters.

target_column
[ColumnNameType] Column from the dataframe, for which CUPED transformation will be applied.

covariate_columns
[ColumnNamesType] Columns which will be used as the covariates in Multi CUPED transformation.

transformed_name
[ColumnNamesType, optional] Name for the new transformed target column, if is not defined it will be generated automatically.

`transform`(*dataframe*, *inplace=False*)

Make Multi CUPED transformation for the target column.

Could be performed inplace or not.

Parameters

dataframe
[pd.DataFrame] Table with data for Multi CUPED transformation.

inplace
[bool, default: `False`] If is `True`, then method returns `None` and sets a new column for the original dataframe. Otherwise return copied dataframe with a new column.

`fit_transform`(*dataframe*, *target_column*, *covariate_columns*, *transformed_name=None*, *inplace=False*)

Combination of fit() and transform() methods.

Parameters

dataframe
[pd.DataFrame] Table with data for fitting and applying Multi CUPED transformation.

target_column
[ColumnNameType] Column from the dataframe, for which CUPED transformation will be applied.

covariate_column
[ColumnNameType] Column which will be used as the covariate.

transformed_name
[ColumnNamesType, optional] Name for the new transformed target column, if is not defined it will be generated automatically.

inplace
[bool, default: `False`] If is `True`, then method returns `None` and sets a new column for the original dataframe. Otherwise return copied dataframe with a new column.

`store_params`(*store_path*)

Parameters

store_path
[Path] Path where parameters will be stored in a json format.

`load_params`(*load_path*)

Parameters

load_path
[Path] Path to json file with parameters.

`class ambrosia.preprocessing.MLVarianceReducer`(*model='boosting'*, *model_params=None*, *scores=None*, *verbose=True*)

Machine Learning approach for variance reduction.

Building a model M, we can make a transformation: Y_hat = Y - M(X) + MEAN(M(X))

It is important, that that the mean of M(X) do not change over time!!! You can choose models from Gradient boosting or Ridge regression or your own model class, for example `sklearn.ensemble.RandomForest`, and pass models params to constructor function for a model assembly.

Parameters

model
[str or model type, default: `"boosting"`] Model which will be used for the transformations.

model_params
[Dict, optional] Dictionary with parameters which will be used in constructor for a model assembly.

scores
[Dict[str, Callable], optional] Scores which will be used.

verbose
[bool, default: `True`] If `True` will print in sys.stdout the information about the reduction in variance.

Examples

We have data table with column 'target' and columns 'feature_1', 'feature_2', 'feature_3'. Let us assume, that means of all these metrics don't change over the time, it can be age for example. We want to reduce variance using the predictions some of ML model, then we can use this class:

```
>>> transformer = MLVarianceReducer() # By default CatBoost model will be choosen
>>> transformer.fit_transform(dataframe, 'target', [feature columns], inplace=True,␣
↪name='new_target')
>>> transformer.store_params('path_ml_params.json')
```

Now to transform the experimental data we use the following commands:

```
>>> transformer = MLVarianceReducer()
>>> transformer.load_params('path_ml_params.json')
>>> transformer.transform(exp_data, inplace=True)
```

Attributes

model
[model type] Model which will be used for the transformations.

params
[Dict] Parameters of instance that will be updated after calling fit() method. Include: - target column name - covariate columns names - name of column after the transformation - additional train bias equals mean(M(X)).

scores
[Dict[str, Callable]] Scores which will be used.

verbose
[bool] Verbose info flag.

fitted
[bool] Fit status flag.

Methods

get_params_dict()	Returns dict with instance fitted parameters.
load_params_dict()	Load parameters from the dict.
store_params(store_path)	Store fitted params in a json file and pickle model file.
load_params(load_path)	Load params from a json file and pickled model.
fit(fit_params)**	Fit model using a train data.
transform(dataframe, inplace)	Transform target column of a data frame.
fit_transform(dataframe, **fit_params, inplace)	Combination of fit() and transform() methods.

`store_params`(*config_store_path*, *model_store_path*)

Store params of model as a json file, available only for CatBoost model.

You can reach model using instance.model and store it by yourself.

Parameters

store_path
[Path] Path where models parameters will be stored in a json format.

`load_params`(*config_load_path*, *model_load_path*)

Load models params from a json file, works only for CatBoost model.

Parameters

load_path: Path
Path to a json file with model parameters.

`fit`(*dataframe*, *target_column*, *covariate_columns*, *transformed_name=None*)

Fit model for transformations.

Parameters

dataframe
[pd.DataFrame] Table with data for model fitting.

target_column
[ColumnNameType] Column from the dataframe, for which transformation will be applied.

covariate_columns: ColumnNamesType
Columns which will be used for the transformation.

transformed_name
[ColumnNamesType, optional] Name for the new transformed target column, if is not defined it will be generated automatically.

`transform`(*dataframe*, *inplace=False*)

Transform data using the fitted model.

Parameters

dataframe
[pd.DataFrame] Table with data for transformation.

inplace
[bool, default: `False`] If is `True`, then method returns `None` and sets a new column for the original dataframe. Otherwise return copied dataframe with a new column.

fit_transform(*dataframe*, *target_column*, *covariate_columns*, *transformed_name=None*, *inplace=False*)

Combinate consequentially `fit()` and `transform()` methods.

Parameters

dataframe
[pd.DataFrame] Table with data for model fitting and further transformation.

target_column
[ColumnNameType] Column from the dataframe, for which transformation will be applied.

covariate_columns: ColumnNamesType
Columns which will be used for the transformation.

transformed_name
[ColumnNamesType, optional] Name for the new transformed target column, if is not defined it will be generated automatically.

inplace
[bool, default: `False`] If is `True`, then method returns `None` and sets a new column for the original dataframe. Otherwise return copied dataframe with a new column.

1.4.5 Preprocessor

Preprocessor	Preprocessor class, implementation is based on the chain pattern.

class ambrosia.preprocessing.**Preprocessor**(*dataframe*, *verbose=True*)

Preprocessor class, implementation is based on the chain pattern.

Parameters

dataframe
[pd.DataFrame] Table with data used for further transformations.

verbose
[bool, default: `True`] If `True` will print in sys.stdout the information about the variance reduction.

Examples

```
>>> transformer = Preprocessor(dataframe)
>>> transformer.aggregate(aggregate_params)
>>>             .robust(robust_params)
>>>             .cuped(cuped_params)
>>>             .data()
```

Attributes

dataframe
[pd.DataFrame] Table with data for transformations.

transformers
[List of transformations] List of transformation that have been called before.

verbose
[bool] Verbose info flag.

Methods

data(copy=True)	Returns a copy or a link for the stored dataframe.
aggregate(groupby_columns, categorial_method, real_method, agg_params,	real_cols, categorial_cols) Aggreagate data by columns.
robust(column_names, alpha=0.05)	Make a robust preprocessing of data.
iqr(column_names, alpha=0.05)	Make an IQR preprocessing of data.
boxcox(column_names, alpha=0.05)	Make a Box-Cox transformation.
log(column_names, alpha=0.05)	Make a log transformation.
cuped(target, by, name, load_path)	Make CUPED transformation for the stored dataframe.
multicuped(target, by, name, load_path)	Make Multi CUPED transformation for the stored dataframe.
transformations()	Returns a list of transformations.
store_transformations(store_path)	Store transformations in a json file.
load_transformations(load_path)	Load transformations from a json file.
apply_transformations()	Apply transformations for the stored dataframe.
transform_from_config(load_path)	Transform inner data frame using pre-saved config file.

`data`(*copy=True*)

Return the inner data frame.

Use after all transformations to get transformed data.

Parameters

copy
[bool, default: `True`] If true returns copy, otherwise link

Returns

dataframe
[pd.DataFrame] Table with the modified data after the sequential preprocessing.

`aggregate`(*groupby_columns=None*, *categorial_method='mode'*, *real_method='sum'*, *agg_params=None*, *real_cols=None*, *categorial_cols=None*, *load_path=None*)

Make an aggregation of the dataframe.

Parameters

groupby_columns
[List of columns, optional] Columns for GROUP BY.

categorial_method
[types.MethodType, default: `"mode"`] Aggregation method that will be applied for all selected categorial variables.

real_method
[types.MethodType, default: `"sum"`] Aggregation method that will be applied for all selected real variables.

agg_params
[Dict, optional] Dictionary with aggregation parameters.

real_cols
[types.ColumnNamesType, optional] Columns with real metrics. Overriden by `agg_params` parameter and could be passed if expected default aggregation behavior.

categorial_cols
[types.ColumnNamesType, optional] Columns with categorial metrics Overriden by `agg_params` parameter and could be passed if expected default aggregation behavior.

Returns

self
[Preprocessor] Instance object

`robust`(*column_names=None*, *alpha=0.05*, *tail='both'*, *load_path=None*)

Make a robust preprocessing of the selected columns to remove outliers.

Removes objects from the dataframe which are in the head, end or both tail parts of the selected metrics distributions.

Parameters

column_names
[ColumnNamesType] One or number of columns in the dataframe.

alpha
[Union[float, np.ndarray], default: `0.05`] The percentage of removed data from head and tail.

tail
[str, default: `"both"`] Part of distribution to be removed. Can be `"left"`, `"right"` or `"both"`.

load_path
[Path, optional] Path to json file with parameters.

Returns

self
[Preprocessor] Instance object

`iqr`(*column_names=None*, *load_path=None*)

Make an IQR preprocessing of the selected columns to remove outliers.

Removes objects from the dataframe which are behind boxplot maximum and minimum of the selected metrics distributions.

Parameters

column_names
[ColumnNamesType, optional] One or number of columns in the dataframe.

load_path
[Path, optional] Path to json file with parameters.

Returns

self
[Preprocessor] Instance object

`boxcox`(*column_names=None*, *load_path=None*)

Make a Box-Cox transformation on the selected columns.

Optimal transformation parameters are selected automatically.

Parameters

column_names
[ColumnNamesType, optional] One or number of columns in the dataframe.

load_path
[Path, optional] Path to json file with parameters.

Returns

self
[Preprocessor] Instance object

`log`(*column_names=None*, *load_path=None*)

Make a logarithmic transformation on the selected columns.

Parameters

column_names
[ColumnNamesType, optional] One or number of columns in the dataframe.

load_path
[Path, optional] Path to json file with parameters.

Returns

self
[Preprocessor] Instance object

`cuped`(*target=None*, *by=None*, *transformed_name=None*, *load_path=None*)

Make CUPED transformation on the selected column.

Parameters

target
[ColumnNameType] Column from the dataframe, for which CUPED transformation will be applied.

by
[ColumnNameType] Covariance column in the dataframe.

transformed_name
[types.ColumnNameType, optional] Name for the new transformed target column, if is not defined it will be generated automatically.

load_path
[Path, optional] Path to json file with parameters.

Returns

self
[Preprocessor] Instance object

`transformations`()

List of all transformations which were called.

Returns

transformers
[List[object]] List of executed transformations

`store_transformations`(*store_path*)

Store transformations with parameters in the json file.

Parameters

store_path
[Path] Path to a json file where transformations will be stored

`load_transformations`(*load_path*)

Load pre-saved transformations from the json file.

Parameters

load_path
[Path] Path to a json file where transformations are stored

`apply_transformations`()

Apply all transformations to the inner data frame.

Returns

dataframe
[pd.DataFrame] Transformed inner data frame

`transform_from_config`(*load_path*)

Run transformations from the config file on the internal data frame.

Parameters

load_path
[Path] Path to a json file where transformations are stored.

Returns

dataframe
[pd.DataFrame] Transformed inner data frame

Chain preprocessing

Almost all separate data transformations are available as sequential methods of the `Preprocessor` class.

1.4.6 Examples of using data transformation tools

1.5 Experiment Design

Ambrosia offers tools for calculating A/B test parameters such as effect uplift, groups size, and experiment statistical power, based on historical metrics values.

Choice of design approach

The theoretical approach to designing experimental parameters is much faster than the empirical one.

Designer	Unit for experiments and pilots parameters design.
load_from_config	Restore a `Designer` class instance from a yaml config.
design	Function wrapper around the `Designer` class.
design_binary	Design of experiment parameters for binary metrics based on a known conversion value.

---

class ambrosia.designer.**Designer**(*dataframe=None*, *sizes=None*, *effects=None*, *first_type_errors=0.05*, *second_type_errors=0.2*, *metrics=None*, *method='theory'*)

Unit for experiments and pilots parameters design.

Enables to design missing experiment parameters using historical data. The main related to each other designable parameters for a single metric are:

- **Effect (Minimal Detectible Effect):**
  old_mean_metric_value * effect_value = new_mean_metric_value
- **Sample size:**
  Number of research objects in sample (for example number of users and their retention).
- **Errors (I type error, II type error):**

  I error (alpha):
  Probability to detect presence of effect for equally distributed samples.

  II error (beta):
  Probability not to find effect for differently distributed samples.

Parameters

dataframe
[PassedDataType, optional] DataFrame with metrics historical values.

sizes
[SampleSizeType, optional] Values of research objects number in groups samples during the experiment.

effects
[EffectType, optional] Effects values that are expected during the experiment.

first_type_errors
[StatErrorType, default: `0.05`] I type error bounds P (detect difference for equal) < alpha.

second_type_errors
[StatErrorType, default: `0.2`] II type error bounds P (suppose equality for different groups) < beta.

metrics
[MetricNamesType, optional] Column names of metrics in dataframe to be designed.

method
[str, optional] Method used for experiment design. Can be `"theory"`, `"empiric"` or `"binary"`.

Notes

Constructors:

```
>>> designer = Designer()
>>> # You can pass an Iterable or single object for some parameters
>>> designer = Designer(
>>>     dataframe=df,
>>>     sizes=[100, 200],
>>>     metrics='LTV',
>>>     effects=1.05
>>> )
>>> designer = Desginer(sizes=1000, metrics=['retention', 'LTV'])
>>> # You can use path to .csv table for pandas
>>> designer = Designer('./data/table.csv')
```

Setters:

```
>>> designer.set_first_errors([0.05, 0.01])
>>> desginer.set_dataframe(df)
```

Run:

```
>>> # One can pass arguments and they will have higher priority
>>> designer.run('size', effects=1.1)
>>> designer.run('effect', sizes=[500, 1000], metrics='retention')
>>> # You can set method (watch below)
>>> designer.run('effect', sizes=[500, 1000], metrics='retention', method='binary')
```

Load from yaml config:

```
>>> config = '''
        !splitter # <--- this is yaml tag (!important)
            effects:
                - 0.9
                - 1.05
            sizes:
                - 1000
    '''
>>> designer = yaml.load(config)
>>> # Or use the implmented function
>>> designer = load_from_config(config)
```

Use standalone function instead of a class:

```
>>> design('size', dataframe=df, effects=1.05, metrics='retention')
```

Examples

We have retention labels for users of mobile app for previous month. Suppose old_retention = `0.3`, that is 30% of users returned to the app in a month after installation.

Let us fix the following parameters:

I type error (alpha) = `0.05` (5% of equal samples we can suppose to be different).

II type error (beta) = `0.2` (20% of different sampels we can suppose to be equal).

We add onboarding to our app and want to estimate an effect, by A/B testing and wish to increase retention value to 31% percents, so our effect parameter gets value of `1.0(3)`. Now we want to find how much users we need in both groups to detect such effect.

We can use `Designer` class in the following way:

```
>>> designer = Designer(dataframe=df, metric='retention', effect=1.033)
>>> designer.run("size")
```

Note, that default values for errors are:

`first_type_error` = `0.05`

`second_type_error` = `0.2`

Then we get dataframe that contains value of sufficient number of users for our experiment.

Attributes

dataframe
[PassedDataType] DataFrame with metrics historocal values.

sizes
[SampleSizeType] Number of research objects in group samples.

effects
[EffectType] Effects values in the experiment.

first_type_errors
[StatErrorType, default: `0.05`] I type errors.

second_type_errors
[StatErrorType, default: `0.2`] II type errors.

metrics
[MetricNamesType] Column names of metrics in dataframe to be designed.

method
[str] Method used for experiment design.

run(*to_design*, *method=None*, *sizes=None*, *effects=None*, *first_type_errors=None*, *second_type_errors=None*, *dataframe=None*, *metrics=None*, ***kwargs*)

Perform an experiment design for chosen parameter and metrics using historical data.

Parameters

to_design
[str] Parameter that will be designed using historical data. Can take the values of `"size"`, `"effect"` or `"power"`.

method
[str, optional] Method used for experiment design. Can be `"theory"`, `"empiric"` or `"binary"`.

sizes
[SampleSizeType, optional] Values of research objects number in groups samples during the experiment. If is not provided, must exist as proper class attribute.

effects
[EffectType, optional] Effects for experiment If is not provided, must exist as proper class attribute.

first_type_errors
[StatErrorType, optional] I type error bounds P (detect difference for equal) < alpha.

second_type_errors
[StatErrorType, optional] II type error bounds P (suppose equality for different groups) < beta.

dataframe
[PassedDataType, optional] DataFrame with metrics historical values. If is not provided, must exist as proper class attribute.

metrics
[MetricNamesType, optional] Column names of metrics in dataframe to be designed. If not provided, must exist as proper class attribute.

****kwargs**
[Dict] Other keyword arguments.

Returns

result
[DesignerResult] Table or dictionary with the results of parameter design for each metric.

Other Parameters

as_numeric
[bool, default: `False`] The result of calculations can be obtained as a percentage string either as a number, this parameter could used to toggle.

groups_ratio
[float, default: `1.0`] Ratio between two groups.

alternative
[str, default: `"two-sided"`] Alternative hypothesis, can be `"two-sided"`, `"greater"` or `"less"`. `"greater"` - if effect is positive. `"less"` - if effect is negative.

stabilizing_method
[str, default: `"asin"`] Effect trasformation. Can be `"asin"` and `"norm"`. For non-binary metrics: only `"norm"` is accceptable. For binary metrics: `"norm"` and `"asin"`, but `"asin"` is more robust and accurate. Acceptable only for `"theory"` method and actual for binary metrics!

ambrosia.designer.**load_from_config**(*yaml_config*, *loader=<class 'yaml.loader.Loader'>*)

Restore a `Designer` class instance from a yaml config.

For yaml_config you can pass file name with config, it must ends with .yaml, for example: “config.yaml”.

For loader you can choose SafeLoader.

ambrosia.designer.**design**(*to_design*, *dataframe*, *metrics*, *sizes=None*, *effects=None*, *first_type_errors=(0.05,)*, *second_type_errors=(0.2,)*, *method='theory'*, ***kwargs*)

Function wrapper around the `Designer` class.

Make experiment design based on historical data using passed arguments.

Creates an instance of the `Designer` class internally and execute run method with corresponding arguments.

Parameters

to_design
[str] Parameter that will be designed using historical data. Can take the values of `"size"`, `"effect"` or `"power"`.

dataframe
[PassedDataType] DataFrame with metrics historical values.

metrics
[MetricNamesType] Column names of metrics in dataframe to be designed.

sizes
[SampleSizeType, optional] Values of research objects number in groups samples during the experiment. If is not provided, `effects` value must be defined.

effects
[EffectType, optional] Effects for experiment If is not provided, `sizes` value must be defined.

first_type_errors
[StatErrorType, default: `(0.05,)`] I type error bounds P (detect difference for equal) < alpha.

second_type_errors
[StatErrorType, default: `(0.2,)`] II type error bounds P (suppose equality for different groups) < beta.

method
[str, default: `"theory"`] Method used for experiment design. Can be `"theory"`, `"empiric"` or `"binary"`.

****kwargs**
[Dict] Other keyword arguments.

Returns

result
[DesignerResult] Table or dictionary with the results of parameter design for each metric.

Other Parameters

as_numeric
[bool, default: `False`] The result of calculations can be obtained as a percentage string either as a number, this parameter could used to toggle.

groups_ratio
[float, default: `1.0`] Ratio between two groups.

alternative
[str, default: `"two-sided"`] Alternative hypothesis, can be `"two-sided"`, `"greater"` or `"less"`. `"greater"` - if effect is positive. `"less"` - if effect is negative.

stabilizing_method
[str, default: `"asin"`] Effect trasformation. Can be `"asin"` and `"norm"`. For non-binary metrics: only `"norm"` is accceptable. For binary metrics: `"norm"` and `"asin"`, but `"asin"` is more robust and accurate. Acceptable only for `"theory"` method and actual for binary metrics!

`ambrosia.designer.`**`design_binary`**(*to_design*, *prob_a*, *sizes=None*, *effects=None*, *first_type_errors=(0.05,)*, *second_type_errors=(0.2,)*, *method='theory'*, *groups_ratio=1.0*, *alternative='two-sided'*, *stabilizing_method='asin'*, ***kwargs*)

Design of experiment parameters for binary metrics based on a known conversion value.

Parameters

to_design
[str] Parameter to design.

prob_a
[float] Probability of success for the control group.

sizes
[SampleSizeType, optional] List or single value of group sizes. For example: `100`, `[100, 200]`.

effects
[EffectType, optional] List of single value of relative effects. For example: 1.05, [1.05, 1.2].

first_type_errors
[StatErrorType, default: `(0.05, )`] I type error bounds P (detect difference for equal) < alpha.

second_type_errors
[StatErrorType, default: `(0.2,)`] II type error bounds P (suppose equality for different groups) < beta.

method: str, default: ``"theory"``
Supports 2 methods: `"theory"` and `"binary"` `"theory"` ~ by formula using statsmodels solve_power mechanism `"binary"` ~ using different types of intervals

groups_ratio
[float, default: `1.0`] Ratio between two groups.

alternative
[str, default: `"two-sided"`] Alternative hypothesis, can be `"two-sided"`, `"greater"` or `"less"`. `"greater"` - if effect is positive. `"less"` - if effect is negative.

stabilizing_method
[str, default: `"asin"`] Effect trasformation. Can be `"asin"` and `"norm"`. For non-binary metrics: only `"norm"` is accceptable. For binary metrics: `"norm"` and `"asin"`, but `"asin"` is more robust and accurate.

****kwargs**
[Dict] Other keyword arguments.

Returns

result_table
[pd.DataFrame] Table with results of design.

1.5.1 Examples of using experiment design tools

1.6 Groups Splitting

The following classes and functions helps to split batch data into experimental groups using different approaches.

Real-time Splitter availability

The real-time splitting tools are under development. This functionality is intended to be applied to batch data only.

Splitter	Unit for creating experimental groups from batch data.
load_from_config	Restore a `Splitter` class instance from a yaml config.
split	Function wrapper around the `Splitter` class.

---

class ambrosia.splitter.**Splitter**(*dataframe=None*, *id_column=None*, *groups_size=None*, *test_group_ids=None*, *fit_columns=None*, *strat_columns=None*)

Unit for creating experimental groups from batch data.

Split your data into groups of selected size with respect to:

- Stratification columns
- Metric distance of objects in feature space
- Set of passed ids

Parameters

dataframe
[PassedDataType, optional] Dataframe or string name of .csv table which contains data used for groups split.

id_column
[IdColumnNameType, optional] Name of id column which is used in hash split.

groups_size
[int, optional] Size of the splitted groups.

test_group_ids
[PeriodColumnNamesType, optional] Ids of objects which are in B(test) group. Used in tasks of post experiment A(control) group pick up.

fit_columns
[PeriodColumnNamesType, optional] List of columns names which values will be interpreted as coordinates of points in multidimensional space during metric split.

strat_columns
[PeriodColumnNamesType, optional] Columns for stratification. https://en.wikipedia.org/wiki/Stratified_sampling

Notes

Main methods for split:

Simple:

- Randomly chosen groups (via `np.random.choice`).

Hash:

- Using hashing of identifiers and distribution by buckets, selects the desired buckets for groups formation.

Metric:

- For a fixed reference group or a randomly selected one, other groups are selected using the nearest neighbor method (for desired list of columns passed in `fit_columns` parameter).

Constructors:

```
>>> # Empty constructor
>>> splitter = Splitter()
>>> # Some data
>>> splitter = Splitter(dataframe=df,
>>>                     id_column='my_id_column',
>>>                     strat_columns=['gender', 'age'],
>>>                     test_group_ids=ids_for_B_group
>>> )
```

Setters:

```
>>> splitter.set_dataframe(dataframe)
>>> # You can pass string for pd.read_csv
>>> splitter.set_dataframe('name_of_table.csv')
>>> # Other setters
>>> splitter.set_group_size(1000)
>>> splitter.set_strat_columns(['age', 'region'])
```

Run:

```
>>> splitter.run(method='hash', groups_size=10000)
>>> splitter.run(method='metric'
>>>              test_group_ids=b_group,
>>>              id_column='id',
>>>              strat_columns=['age', 'city']
>>>              fit_columns=['metric_history_column', 'other_metric']
>>>              method_meric='fast', # It is used as kwarg
>>>              norm='l2' # It is used as kwarg
>>> )
```

Load from yaml config:

```
>>> config = '''
            !splitter # <--- this is yaml tag (important!)
                groups_size:
                    1000
                id_column:
                    id
                strat_columns:
                    - age
                    - country
        '''
>>> splitter = yaml.load(config)
>>> # Or use the implmented function
>>> splitter = load_from_config(config)
```

Examples

Our development team decided to add onboarding to the mobile app. Already knowing the required group size, we would like to select users for groups A and B respectively. Using the splitter class, this task could be done in the following way:

```
>>> splitter = Splitter(dataframe=dataframe)
>>> splitter.run(group_size=1000, method='hash', salt='onboarding')
```

Suppose now, we know that people of different ages and from several countries use our application, so we would like to take this into account during split. To do this, you might use stratification, which can be easily applied by passing only one additional parameter:

```
>>> splitter = Splitter(data=dataframe, strat_columns=['age', 'country'])
>>> splitter.run(group_size=1000, method='hash', salt='onboarding')
```

If we have fixed users for the testing group, this can be specified as a parameter:

```
>>> splitter = Splitter(data=dataframe, strat_columns=['age', 'country'])
>>> splitter.run(method='hash',
>>>              salt='onboarding',
>>>              test_group_ids=B_group_id
>>> )
```

Attributes

dataframe
[PassedDataType] Pandas or Spark dataframe with split data.

id_column
[IdColumnNameType] Name of id column which is used in hash split.

groups_size
[int] Split size of groups.

test_group_ids
[PeriodColumnNamesType] Ids of objects which are in B(test) group.

fit_columns
[PeriodColumnNamesType] List of columns names used for metric split.

strat_columns
[PeriodColumnNamesType] Stratification columns names.

run(*method*, *dataframe=None*, *id_column=None*, *groups_size=None*, *part_of_table=None*, *groups_number=2*, *test_group_ids=None*, *strat_columns=None*, *salt=None*, *fit_columns=None*, ***kwargs*)

Perform a split into groups with selected or saved parameters.

Parameters

method
[str] Split method, for example "hash".

dataframe
[PassedDataType, optional] Dataframe or string name of .csv table which contains data used for groups split.

id_column
[IdColumnNameType, optional] Name of id column which is used in hash split.

groups_size
[int, optional] Size of the splitted groups.

part_of_table: float, optional
Split factor(for group A) for tasks of dataframe full split. If is not `None`, then overrides `groups_size` parameter during the split.

groups_number
[int, default: `2`] Number of groups to be splitted.

test_group_ids
[PeriodColumnNamesType, optional] Ids of objects which are in B(test) group. Used in tasks of post experiment A(control) group pick up.

strat_columns
[PeriodColumnNamesType, optional] Columns for stratification. https://en.wikipedia.org/wiki/Stratified_sampling

salt
[str, optional] Salt for hashing in hash-split.

fit_columns
[PeriodColumnNamesType, optional] List of columns names which values will be interpreted as coordinates of points in multidimensional space during metric split.

****kwargs**
[Dict] Other keyword arguments.

Returns

groups
[pd.DataFrame] Returns a dataframe with groups and label column. Dataframe will contain all columns of the original dataframe.

Other Parameters

threads
[int, default][`1`] Number of threads used for calculations.

`ambrosia.splitter.`**`load_from_config`**(*yaml_config*, *loader=<class 'yaml.loader.Loader'>*)

Restore a `Splitter` class instance from a yaml config.

For yaml_config parameter you can pass file name with config, which must ends with .yaml, for example: "config.yaml". For loader you can choose SafeLoader.

`ambrosia.splitter.`**`split`**(*method*, *dataframe=None*, *id_column=None*, *groups_size=None*, *part_of_table=None*, *groups_number=2*, *test_group_ids=None*, *strat_columns=None*, *salt=None*, *fit_columns=None*, *threads=1*, ***kwargs*)

Function wrapper around the `Splitter` class.

Used to create splitted groups from the dataframe.

Creates an instance of the `Splitter` class internally and execute run method with corresponding arguments.

Parameters

method
[str] Split method, for example "`hash`".

dataframe
[PassedDataType, optional] Dataframe or string name of .csv table which contains data used for groups split.

id_column
[IdColumnNameType, optional] Name of id column which is used in hash split.

groups_size
[int, optional] Size of the splitted groups.

part_of_table: float, optional
Split factor(for group A) for tasks of dataframe full split. If is not `None`, then overrides `groups_size` parameter during the split.

groups_number
[int, default][2] Number of groups to be splitted.

test_group_ids
[PeriodColumnNamesType, optional] Ids of objects which are in B(test) group. Used in tasks of post experiment A(control) group pick up.

strat_columns
[PeriodColumnNamesType, optional] Columns for stratification. https://en.wikipedia.org/wiki/Stratified_sampling

salt
[str, optional] Salt for hashing in hash-split.

fit_columns
[PeriodColumnNamesType, optional] List of columns names which values will be interpreted as coordinates of points in multidimensional space during metric split.

threads
[int, default][1] Number of threads used for calculations.

****kwargs**
[Dict] Other keyword arguments.

Returns

groups
[pd.DataFrame] Returns a dataframe with groups and label column. Dataframe will contain all columns of the original dataframe.

1.6.1 Examples of using groups splitting tools

1.7 Effect Measurement

Tools for assessing the statistical significance of completed experiments and calculating the experimental uplift value with corresponding confidence intervals.

Multiple testing correction

Currently, if multiple hypothesis(number of variants combinations * number of metrics passed) are tested, these groups are compared in pairs and Bonferroni correction is applied to all p-values and confidence intervals.

Tester	Unit for evaluating the results of experiments.
test	Function wrapper around the `Tester` class.

class ambrosia.tester.**Tester**(*dataframe=None*, *df_mapping=None*, *experiment_results=None*, *column_groups=None*, *group_labels=None*, *id_column=None*, *first_type_errors=0.05*, *metrics=None*)

Unit for evaluating the results of experiments.

The experiment evaluation result contains:

- Pvalue for the selected criterion
- Point effect estimation
- Corresponding confidence interval for the effect
- Boolean result - presence / absence of the effect

Parameters

dataframe
[PassedDataType, optional] Dataframe used with experiment results metrics.

df_mapping
[GroupsInfoType, optional] Dataframe which contains group labels of objects.

experiment_results
[ExperimentResults, optional] Dict with separate experiment results for each group. Dict keys are used as groups labels, values must be either pandas or Spark dataframes.

column_groups
[ColumnNameType, optional] Column which contains groups label of objects.

group_labels
[GroupLabelsType, optional] Labels for experimental groups. If `column_groups` contains at least two values, they will choose for labels.

id_column
[ColumnNameType, optional] Name of column with objects ids in `df_mapping` dataframe.

first_type_errors
[StatErrorType, default: `0.05`] I type errors values. Fix P (detect difference for equal) to be less than threshold. Used to construct confidence intervals.

metrics
[MetricNameType, optional] Metrics (columns of dataframe) which is used to calculate experiment result.

Notes

Basic mathematic methods for evaluating experiments:

- **Theory:**
  - Absolute: Using ttest, mann-whitney, others and custom criteria
  - Relative: Using delta method
- **Empiric:**
  - Absolute / Relative: Building empirical distribution for T(A, B)
- **Binary:**
  - Absolute: Using special binary intervals and finding pvalue = inf_a {x : 0 not in interval(x)}
  - Relative: Not implemented yet :(

Constructors:

```
>>> # Empty constructor
>>> tester = Tester()
>>> # You can pass Iterable or single object for some parameters
>>> tester = Tester(
>>>     dataframe=df,
>>>     columns_groups='groups',
>>>     metrics=['ltv', 'retention']
>>> )
>>> tester = Tester(metrics='retention', first_type_errors=[0.01, 0.05])
>>> # You can set a separate table containing information about
>>> # the partitioning in the experiment
>>> tester = tester = Tester(
>>>     dataframe=df, # main dataframe with metrics
>>>     df_mapping=groups, # table with information about groups
>>>     metrics='metric', # Metric to be tested
>>>     column_groups='group', # Column in df_mapping with labels
>>>     id_column='id' # Column with ids in df and df_mapping (for join)
>>> )
```

Setters:

```
>>> tester.set_metrics(['ltv', 'retention'])
>>> tester.set_dataframe(dataframe=dataframe, column_groups='groups')
>>> # You can set separate data of each group packed in special dict form
>>> tester.set_experiment_results(experiment_results=experiment_results)
```

Run:

```
>>> # You can choose effect_type to estimate: relative / absolute
>>> tester.run('absolute')
>>> # Also you can choose method
>>> tester.run('absolute', method='empriric') # emipiric for bootstrap
>>> # One can pass arguments in run() method and they will have
>>> # higher priority
>>> tester.run(metrics='ltv', data_a_group=df_a)
```

Use a function instead of a class:

```
>>> test('absolute', dataframe=df, column_groups='groups', metrics='ltv')
```

Examples

We've experimented with adding onboarding to our mobile app and would like to know about its results in terms of A/B testing. Suppose we have a loaded pandas dataframe with a column responsible for the groups in the testing and columns with metric values, such as retention. Then you can use the tester class the following way:

```
>>> tester = Tester(
>>>     dataframe=df,
>>>     column_groups='groups',
>>>     metrics='retention'
>>> )
>>> tester.run()
>>> # Output
>>> [{
>>>     'first_type_error' : 0.05,
>>>     'pvalue' : 0.03,
>>>     'effect' : 1.05,
>>>     'confidence_interval' : (1.01, 1.10),
>>>     'metric name': 'retention',
>>>     'group A label': 'A',
>>>     'group B label': 'B'
>>> }]
```

Attributes

dataframe
[PassedDataType] Dataframe used with experiment results metrics.

df_mapping
[GroupsInfoType] Dataframe which contains group labels of objects.

experiment_results
[ExperimentResults, optional] Dict with separate experiment results for each group.

column_groups
[ColumnNameType] Column which contains groups label of objects.

group_labels
[GroupLabelsType] Labels for experimental groups.

id_column
[ColumnNameType] Name of column with objects ids in `df_mapping` dataframe.

first_type_errors
[StatErrorType, default: `0.05`] I type errors values.

metrics
[MetricNameType] Columns of dataframe with experiment results.

run(*effect_type='absolute'*, *method='theory'*, *dataframe=None*, *df_mapping=None*, *experiment_results=None*, *id_column=None*, *column_groups=None*, *group_labels=None*, *metrics=None*, *first_type_errors=None*, *criterion=None*, *correction_method='bonferroni'*, *as_table=True*, ***kwargs*)

The main method for testing and evaluating experimental results.

Parameters

effect_type
[str, default: "absolute"] Effect type to calculate. Could be "absolute" or "relative".

method
[str, default: "theory"] Type of testing approach. Can take the values "theory", "empiric" or "binary".

dataframe
[PassedDataType, optional] Data used to calculate the results of an experiment.

df_mapping
[GroupsInfoType, optional] Dataframe which contains group labels of objects.

experiment_results
[ExperimentResults] Dict with separate experiment results for each group. Dict keys are used as groups labels, values must be either pandas or Spark dataframes.

column_groups
[ColumnNameType] Column which contains groups label of objects.

group_labels
[GroupLabelsType] Labels for experimental groups.

id_column
[ColumnNameType] Name of column with objects ids in df_mapping dataframe.

first_type_errors
[StatErrorType, default: 0.05] I type errors values.

metrics
[MetricNameType] Columns of dataframe with experiment results.

criterion
[ABStatCriterion, optional] Statistical criterion for hypotheses testing. If method is "theory" and no criterion provided, ttest for independent samples will be used.

correction_method
[Union[str, None], default: bonferroni] Method for pvalues and confidence intervals multitest correction. Total number of hypothesis is equal to the number of variants combinations * number of metrics passed.

as_table
[bool, default: True] Return the test results as a pandas dataframe. If False, a list of dicts with results will be returned.

****kwargs**
[Dict] Other keyword arguments.

Returns

result
[types.TesterResult] Experiment results as pandas table or list of dicts for each metric and first type error.

ambrosia.tester.**test**(*effect_type='absolute'*, *method='theory'*, *dataframe=None*, *df_mapping=None*, *experiment_results=None*, *id_column=None*, *column_groups=None*, *group_labels=None*, *metrics=None*, *first_type_errors=None*, *criterion=None*, *correction_method='bonferroni'*, *as_table=True*, ***kwargs*)

Function wrapper around the Tester class.

Apply on the experimental data to get the results of an experiment.

Creates an instance of the `Tester` class internally and execute run method with corresponding arguments.

Parameters

effect_type
[str, default: `"absolute"`] Effect type to calculate. Could be `"absolute"` or `"relative"`.

method
[str, default: `"theory"`] Type of testing approach. Can take the values `"theory"`, `"empiric"` or `"binary"`.

dataframe
[PassedDataType, optional] Data used to calculate the results of an experiment.

df_mapping
[GroupsInfoType, optional] Dataframe which contains group labels of objects.

experiment_results
[ExperimentResults] Dict with separate experiment results for each group. Dict keys are used as groups labels, values must be either pandas or Spark dataframes.

column_groups
[ColumnNameType] Column which contains groups label of objects.

group_labels
[GroupLabelsType] Labels for experimental groups.

id_column
[ColumnNameType] Name of column with objects ids in `df_mapping` dataframe.

first_type_errors
[StatErrorType, default: `0.05`] I type errors values.

metrics
[MetricNameType] Columns of dataframe with experiment results.

criterion
[ABStatCriterion, optional] Statistical criterion for hypotheses testing. If `method` is `"theory"` and no criterion provided, ttest for independent samples will be used.

correction_method
[Union[str, None], default: `bonferroni`] Method for pvalues and confidence intervals multitest correction. Total number of hypothesis is equal to the number of variants combinations * number of metrics passed.

as_table
[bool, default: `True`] Return the test results as a pandas dataframe. If `False`, a list of dicts with results will be returned.

****kwargs**
[Dict] Other keyword arguments.

Returns

result
[types.TesterResult] Experiment results as pandas table or list of dicts for each metric and first type error.

1.7.1 Examples of using testing tools

1.8 Development

To install all requirements run

```
make install
```

You must have `python3` and `poetry` installed.

For autoformatting run

```
make autoformat
```

For linters check run

```
make lint
```

For tests run

```
make test
```

For coverage run

```
make coverage
```

To remove virtual environment run

```
make clean
```

1.8.1 Contributing Guide

Ambrosia is an open source project and there are many ways to contribute, from writing tutorials or blog posts, improving the documentation, submitting bug reports and feature requests or writing code which can be incorporated into *Ambrosia* itself.

Bug reports

If you think you have found a bug in *Ambrosia*, first make sure that you are testing against the latest version of package - your issue may already have been fixed. If not, search our issues list on GitHub in case a similar issue has already been opened.

It is very helpful if you can prepare a reproduction of the bug. In other words, provide a small test case which we can run to confirm your bug. It makes it easier to find the problem and to fix it.

Provide as much information as you can. The easier it is for us to recreate your problem, the faster it is likely to be fixed.

Feature requests

If you find yourself wishing for a feature that doesn't exist in *Ambrosia*, you can open an issue on our issues list on GitHub which describes the feature you would like to see, why you need it, and how it should work.

Contributing code and documentation changes

If you have a bugfix or new feature that you would like to contribute to *Ambrosia*, please find or open an issue about it first. Talk about what you would like to do. It may be that somebody is already working on it, or that there are particular issues that you should know about before implementing the change.

There are many approaches to fixing a problem and it is important to find the best approach before writing too much code.

Branching

Those users with Contributor permissions can directly clone the repository and work on a branch within it.

Those without Contibutor permissions will need to fork the main repository to work on your changes. Simply navigate to our GitHub page and click the "Fork" button at the top. Once you have forked the repository, you can clone your new repository and start making edits.

When using git, it is best to isolate each topic or feature into a "topic branch". Branches are a great way to group commits related to one feature together, or to isolate different efforts when you might be working on multiple topics at the same time.

While it takes some experience to get the right feel about how to break up commits, a topic branch should be limited in scope to a single issue. If you are working on multiple issues, please create multiple branches and submit them for review separately.

Pull Request Guidelines

Create a pull request for preliminary review or merging into the project when you are ready.

If you need to make any adjustments to your pull request, just push the updates to your branch. Your pull request will automatically track the changes on your development branch and update.

You may merge the Pull Request in once you have the sign-off of two other developers, or if you do not have permission to do that, you may request the second reviewer to merge it for you. We expect to have a minimum of one approval from someone else on the core team.

1.8.2 Security Policy

Supported Python versions

3.7 or above

Product development security recommendations

1. Update dependencies to last stable version
2. Build SBOM for the project
3. Perform SAST (Static Application Security Testing) where possible

Product development security requirements

1. No binaries in repository
2. No passwords, keys, access tokens in source code
3. No "Critical" and/or "High" vulnerabilities in contributed source code

Vulnerability reports

Please, use email mailto:ambajramk1@mts.ru for reporting security issues or anything that can cause any consequences for security.

Please avoid any public disclosure (including registering issues) at least until it is fixed. Thank you in advance for understanding.

1.9 Release Notes

1.9.1 Version 0.4.1 (21.04.2023)

Hotfix for pyspark import in spark criteria.

1.9.2 Version 0.4.0 (21.04.2023)

- Documentation and usage examples have been substantially reworked and updated.
- The `Designer` class and design methods functionality is updated.
  - Empirical design now supports the choice of hypothesis alternative and group ratio parameter
  - Look of resulting tables with calculated parameters is unified for all design methods
  - Changed multiprocessing strategy for bootstrap criterion
- The `Tester` class functionality is updated.
  - Spark data support for the `Tester` class is added. Independent t-test is available now
  - Bootstrap criterion can now return deterministic output using a `random_seed` parameter
  - Paired bootstrap criterion is now available
  - MHC now is optional and takes into account the number of passed metrics
  - `first_errors` parameter renamed to `first_type_errors`
- `pyspark` package now is optional and could be installed using `pip` extras.
- Fixed a set of bugs.

1.9.3 Version 0.3.0 (15.02.2023)

- The `Designer` class and design methods functionality is updated.
  - Theoretical design now supports the choice of hypothesis alternative and group ratio parameter
  - These calculations now use Statsmodels solvers
  - Experimental parameters for binary data can now also be theoretically designed using both the asin variance-stabilizing transformation and the normal approximation
- All preprocessor classes, except for the `Preprocessor`, have changed their api and have updated functionality
  - Preprocessing classes now use `fit` and `transform` methods to get transformation parameters and apply transformation on pandas tables
  - Fitted classes now can now be saved and loaded from json files
  - Table column names used when fitting class instances are now strictly fixed in instance attributes
- The `Preprocessor` class is updated.
  - Added new transformation methods
  - The executed transformation pipeline can now be saved and loaded from a json file. This can be used to store and load the entire experimental data processing pipeline
  - The data handling methods of the class have changed some parameters to match the changes in the classes used
- The `IQRPreprocessor` class now is available in `ambrosia.preprocessing`.
  - It can be used to remove outliers based on quartile and interquartile range estimates
- The `RobustPreprocessor` class is updated.
  - It now supports different types of tails for removal: `both`, `right` or `left`
  - For each processed column, a separate alpha portion of the distribution can be passed.
- The `BoxCoxTransformer` class now is available in `ambrosia.preprocessing`
  - It can be used for data distribution normalization.
- The `LogTransformer` class now is available in `ambrosia.preprocessing`
  - It can be used to transform data for variance reduction.
- The `MLVarianceReducer` class is updated.
  - Now it can store and load the selected ML model from a single specified path

1.9.4 Version 0.2.0 (22.11.2022)

Library name changed back to `ambrosia`. Naming conflict in PyPI has been resolved. 0.1.x versions are still available in PyPI under `ambrozia` name.

1.9.5 Version 0.1.2 (16.11.2022)

Hotfix for Ttest stat criterion absolute effect calculation. Url to main image deleted from docs.

1.9.6 Version 0.1.1 (04.10.2022)

Hotfix for library naming. Library temprorary renamed to `ambrozia` in PyPI repository due to hidden naming conflict.

1.9.7 Version 0.1.0 (03.10.2022)

First release of `Ambrosia` package:

- Added `Designer` class for experiment parameters design
- Added `Spliiter` class for A/B groups split
- Added `Tester` class for experiment effect measurement
- Added various classes for experiment data preprocessing
- Added A/B testing tools with wide functionality

1.10 Authors

Developers and evangelists:

- Bayramkulov Aslan
- Khakimov Artem
- Vasin Artem

1.11 Pandas Data Examples

1.12 Spark Data Examples

INDEX