ABacus

Release 0.0.5

Data Sapience. Advanced Analytics team

Mar 02, 2024

USER GUIDE

ABacus is a Python library developed for A/B experimentation and testing. It includes versatile instruments for different experimentation tasks like experiment design, sample size determination, results evaluation, visualisation and reporting.

CHAPTER

ONE

IMPORTANT FEATURES

- Experiment design: type I/II errors, effect size, sample size simulations.
- Groups splitting.
- A/A test and evaluation of splitter accuracy.
- Evaluation of experiment results with various statistical tests and approaches.
- Sensitivity increasing techniques like CUPED, CUPAC.

Note: This project is under active development.

1.1 Installation

You can use **pip** to install **ABacus** directly from PyPI:

```
pip install kolmogorov-abacus
```

or right from Github:

```
pip install git+https://github.com/kolmogorov-lab/abacus
```

Note: ABacus requires Python 3.8+.

1.2 Experiment Initialization

Before actual analysis, you have to define your experiment. Here is how you can do it:

```
from abacus.auto_ab.abtest import ABTest
from abacus.auto_ab.params import ABTestParams, DataParams, HypothesisParams

df = pd.read_csv('./data/ab_data.csv')

data_params = DataParams(
    id_col='user_id',
    group_col='groups',
    control_name='control',
    treatment_name='treatment',
```

(continues on next page)

(continued from previous page)

```
    target='check_rub_campaign',
)

hypothesis_params = HypothesisParams(
    alpha=0.01,
    beta=0.2,
    alternative='greater',
    metric_type='continuous',
    metric_name='95th quantile',
    metric=lambda x: np.quantile(x, 0.95)
)

ab_params = ABTestParams(data_params, hypothesis_params)
ab_test = ABTest(df, ab_params)
```

As you can see, you just need to describe data and your hypothesis.

For data, you have to define columns and their purposes. Required attributes are:

- `id_col` is observation id. It can be user_id or any other id for your rows. Note that if your observations are somehow dependent (e.g. several checks per user), they must have the same id_col.
- `group_col` contains group names. If your data have two groups, then there mush be only two unique values in this column.
- `control_name` and `treatment_name` are group names e.g. 'control', 'treatment', 'A', 'B', 'control group', 'send sms', 'do not send sms', etc.
- `target` is obviously target column containing metric of interest.

Hypothesis is described with:

- `alpha` — type I error.
- `beta` — type II error.
- `alternative` — alternative of hypothesis (two-sided, less, or greater.
- `metric_type` — metric type. There are three of them: continuous, binary, and ratio.
- `metric_name` — metric name, either default ('mean' or 'median') or customer (e.g. '95th percentile').
- `metric` — function for metric calculation if `metric_name` is not default.

1.3 Experiment Evaluation

After the initialization of experiment, we are ready to dive into the analysis.

You have the following options for analysis:

- Statistical Inference
- Metric Transformations
- Increasing Sensitivity (Variance Reduction)
- Visualizations
- Reporting

1.3.1 Statistical Inference

ABacus supports three types of metrics: continuous, binary, and ratio. Each of these types requires its own particular methods to conduct statistical analysis of experiment.

ABacus has the following statistical tests for each type of metric:

1. For continuous metrics: `Welch t-test`, `Mann-Whitney U-test`, `bootstrap`.
2. For binary metrics: `chi-squared test`, `Z-test`, `bootstrap`.
3. For ratio metrics: `delta method`, `Taylor method`.

To get the result of a test, just call the appropriate statistical method on your ABTest instance:

```
ab_test = ABTest(...)

ab_test.test_welch()
{'stat': 5.172, 'p-value': 0.312, 'result': 0}

# or

ab_test.test_mannwhitney()
{'stat': 0.12, 'p-value': 0.67, 'result': 0}
```

As a result, you'll get dictionary with

- statistic of the test,
- p-value of this empirical statistic,
- result in binary form: 0 - H0 is not rejected, 1 - H0 is not accepted.

1.3.2 Metric Transformations

Sometimes experiment data cannot be analyzed directly due to different limitations such as presense of outliers or form of distribution. Metric transformation techniques available in ABacus are:

- **Outliers removal**: direct exclusion of outliers according to some algorithm. There are two methods implemented in **ABacus**: remove `top 5%` observations and `isolation forest`.

```
hypothesis_params = HypothesisParams(..., filter_method='isolation_forest')

ab_test = ABTest(...)
ab_test_2 = ab_test.filter_outliers()

print(ab_test.params.data_params.control)
# 200 000

print(ab_test_2.params.data_params.control)
# 198 201
```

- **Functional transformation**: application of any function to your target metric in order to make it more normal or remove outliers. The following example includes functional transformation with `sqrt` function:

```
hypothesis_params = HypothesisParams(..., metric_transform=np.sqrt)

ab_test = ABTest(...)
ab_test_2 = ab_test.metric_transform()
```

- **Bucketing**: aggregation of target metric into buckets in order to obtain smaller number of points for analysis and from initial distribution to distributions of means.

```
hypothesis_params = HypothesisParams(..., n_buckets=1500)

ab_test = ABTest(...)
ab_test_2 = ab_test.bucketing()
```

- **Linearization**: remove dependence of observations (and move from ratio target) using linearization approach.

```
data_params = DataParams(..., is_grouped=False)

ab_test = ABTest(...)
ab_test_2 = ab_test.linearization()
```

1.3.3 Increasing Sensitivity (Variance Reduction)

As you want to make your metrics more sensitive, you will mostly likely want to use some sensitivity increasing techniques. **ABacus** supports the following options for increasing sensitivity of your experiments:

- **CUPED (Controlled experiment Using Pre-Experiment Data)** uses information about covariate independent from experiment.

```
data_params = DataParams(..., covariate='pre_experiment_metric')

ab_test = ABTest(...)
ab_test_2 = ab_test.cuped()
```

- **CUPAC (Control Using Predictions as Covariate)** predicts variable that can be used as a covariate.

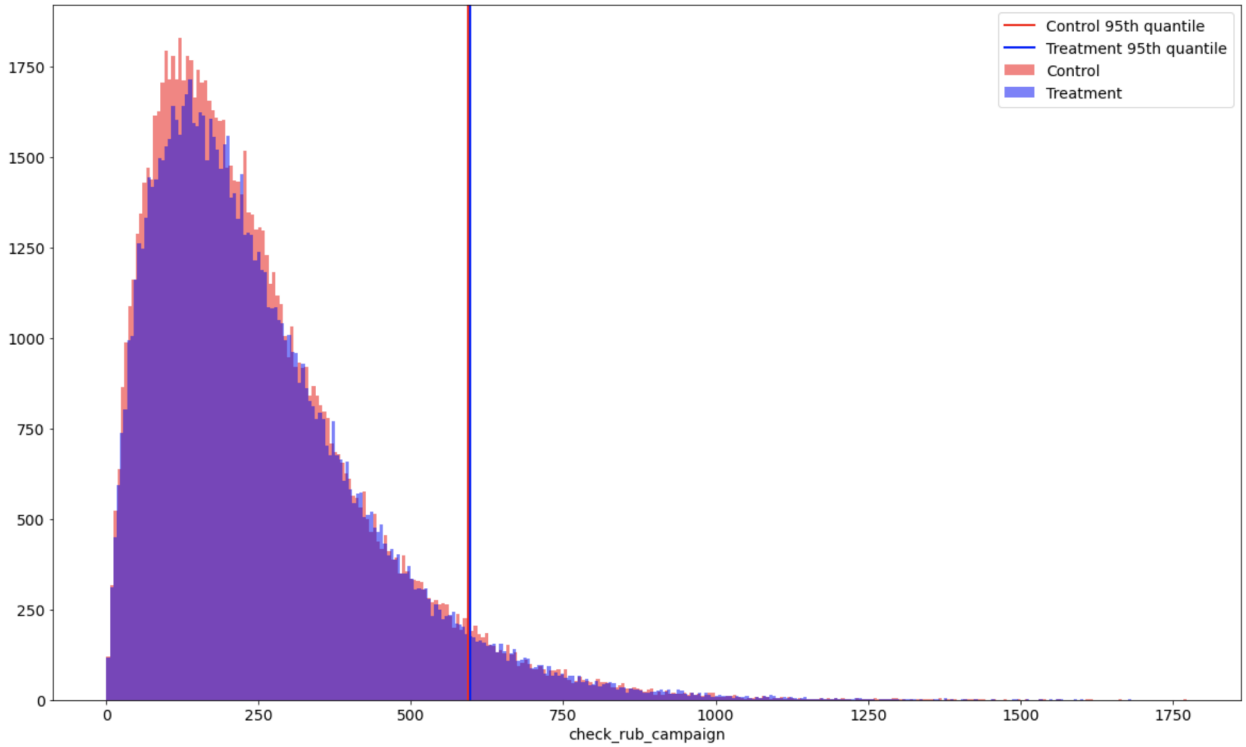
```
data_params = DataParams(..., predictors_prev=['pre_pred_1', 'pre_pred_2'],
                              predictors_now=['now_pred_1', 'now_pred_2'],
                              target_prev='pre_experiment_metric')

ab_test = ABTest(...)
ab_test_2 = ab_test.cupac()
```

1.3.4 Visualizations

A picture is worth a thousand words. No doubt that you want to visually explore your experiment.

You can plot experiments with continuous and binary variables. Continuous plots illustrates not only distributions of desired targe variable, but also a desired metric of a distribution. You can also plot a bootstrap distribution of differences if you want to estimate your experiment with bootstrap approach.

Here is the output of `ab_test.plot()` method:



1.3.5 Reporting

As you may wish to get some sort of report with information of your experiment, you can definitely do it with ABacus.

You just need to call `ab_test.report()` and get information about preprocessing steps and results of statistical tests:

```
Parameters of experiment:
- Metric type: continuous.
- Metric: mean.
- Errors: alpha = 0.05, beta = 0.2.
- Alternative: greater.

Transformations applied: cuped -> bucketing.

Number of bootstrap iterations: 200.
Number of buckets: 50.

Following statistical tests are used:
- Welch's t-test: 1.14, p-value = 0.1289, H0 is not rejected.
- Mann Whitney's U-test: 1092.00, p-value = 0.8627, H0 is not rejected.
- Bootstrap test: H0 is not rejected.

All three stat. tests showed that H0 is not rejected.
Please note that you should carefully use the results of different statistical
procedures and do not consider all of them at once.

Statistics of experiment groups.
Control group:
- Observations: 50
- Mean: 174.9896
- Median: 174.9915
- 25th quantile: 174.9589
- 75th quantile: 175.0248
- Minimum: 174.9198
- Maximum: 175.0591
- St.deviation: 0.0465
- Variance: 0.0022

Treatment group:
- Observations: 50
- Mean: 174.9995
- Median: 174.9931
- 25th quantile: 174.9721
- 75th quantile: 175.0346
- Minimum: 174.8934
- Maximum: 175.1145
- St.deviation: 0.0465
- Variance: 0.0022
```

Report is available for any metric type. On each metric type, you will get a bit different results.

1.3.6 Everything at once

You can freely mix everything you saw above using **chaining**.

```
ab_test = ABTest(...).filter_outliers().metric_transform().cuped().bucketing()
ab_test.test_welch()
```

As you can see, you just need to call methods one by one. `ab_test.report()` will show information about all applied transformations:

```
Parameters of experiment:
- Metric type: continuous.
- Metric: median.
- Errors: alpha = 0.01, beta = 0.2.
- Alternative: greater.

Control group:
- Observations: 1000
- Mean: 0.1971
- Median: 0.2156
- 25th quantile: 0.0800
- 75th quantile: 0.3312
- Minimum: -0.6130
- Maximum: 0.7552
- St.deviation: 0.1878
- Variance: 0.0353

Treatment group:
- Observations: 1000
- Mean: 0.3136
- Median: 0.3288
- 25th quantile: 0.2071
- 75th quantile: 0.4403
- Minimum: -0.9597
- Maximum: 0.7536
- St.deviation: 0.1878
- Variance: 0.0353

Transformations applied: metric transform -> filter outliers -> linearization -> cuped -> bucketing.

Number of bootstrap iterations: 1000.
Number of buckets: 1000.
Metric transformation applied: sqrt.
Outliers filtering method applied: top_5.

Following statistical tests are used:
- Welch's t-test: 13.78, p-value = 0.0000, H0 is rejected.
- Mann Whitney's U-test: 322535.00, p-value = 1.0000, H0 is not rejected.
- Bootstrap test: H0 is rejected.

Two out of three stat. tests showed that H0 is rejected.
```

1.4 Splitter

Splitter is a core instrument that allows you to get 'equal' groups for your experiment. Groups of an experiment are equal in the sense of users' desired characteristic of experiment are equal.

It is a crucial part of any experiment design - to get approximately equal groups. Splitter in **ABacus** not only allows you to split your observations into groups, but also assesses the quality of this split.

```
df = pd.read_csv('./data/ab_data.csv')

split_builder_params = SplitBuilderParams(
    map_group_names_to_sizes={
        'control': 20_000,
        'target': 30_000
    },
    main_strata_col = "city",
    split_metric_col = "check_rub_campaign",
    id_col = "user_id",
    cols = ["check_rub_pre_campaign"],
    cat_cols=["gender"],
```

(continues on next page)

(continued from previous page)

```
    alpha=0.05,
    n_bins = 6,
    min_cluster_size = 500
)

split_builder = SplitBuilder(df, split_builder_params)
split = split_builder.collect()

split.head()
```

After the application of splitter to your data, you will see two additional columns to your data — **strata** and **group_name**:

	user_id	gender	age	city	check_rub_campaign	check_rub_pre_campaign	has_transaction	clicks	session_duration	strata	group_name
0	189963	0.351604	64	Moscow	337.9	322.498048	0	12	91	Moscow4-1	target
1	42186	0.648396	44	Perm	84.8	80.795804	0	18	7	Perm0-1	control
2	3038	0.648396	35	Moscow	428.9	410.870342	0	13	12	Moscow5-1	control
3	125577	0.351604	49	St.Petersburg	401.4	416.752732	0	3	10	St.Petersburg4-1	control
4	10191	0.351604	53	St.Petersburg	122.8	108.475735	0	29	24	St.Petersburg1-1	target

- `strata`: strata of observation created by clustering algorithm (HDBSCAN).
- `group_name`: groups of experiment. Control group has the same group name - `control`, and the treatment is called `treatment`.

1.5 MDE Researcher

MDE Researcher makes experimental design in order to get all the information about your experiment. The main purpose of its usage is calculation of samples size needed to detect particular effect size based on type I and II errors, directionality of hypothesis and other parameters.

There are three components in experimental design:

- data and hypothesis parameters;
- splitter parameters;
- actual experimental design parameters (this section).

This is an example of everything at once for experimental design:

```
from abacus.splitter.split_builder import SplitBuilder
from abacus.splitter.params import SplitBuilderParams
from abacus.mde_researcher.params import MdeParams
from abacus.mde_researcher.mde_research_builder import MdeResearchBuilder
from abacus.mde_researcher.multiple_split_builder import MultipleSplitBuilder

# data, data params and hypothesis
df = pd.read_csv('./data/ab_data.csv')
data_params = DataParams(
    id_col='user_id',
    group_col='groups',
    control_name='control',
    treatment_name='treatment',
```

(continues on next page)

(continued from previous page)

```
    is_grouped=True,
    target='check_rub_campaign'
)
hypothesis_params = HypothesisParams(
    alpha=0.01,
    beta=0.2,
    alternative='greater',
    metric_type='continuous',
    metric_name='mean',
)
ab_params = ABTestParams(data_params, hypothesis_params)

# splitter params
split_builder_params = SplitBuilderParams(
    map_group_names_to_sizes={
        'control': None,
        'target': None
    },
    main_strata_col = "city",
    split_metric_col = "check_rub_campaign",
    id_col = "user_id",
    cols = ["check_rub_pre_campaign"],
    cat_cols=["gender"],
    alpha=0.05,
    n_bins = 6,
    min_cluster_size = 500
)

# design params
experiment_params = MdeParams(
    metrics_names=['check_rub_campaign'],
    injects=[1.010, 1.013, 1.015, 1.018, 1.02, 1.030],
    min_group_size=5_000,
    max_group_size=30_000,
    step=5_000,
    variance_reduction=None,
    use_buckets=False,
    stat_test=ABTest.test_welch,
    iterations_number=10,
    max_beta_score=0.9,
    min_beta_score=0.2,
)

# simulation of experimental design
design = MdeResearchBuilder(df,
                            ab_params,
                            experiment_params,
                            split_builder_params)
beta, alpha = design.collect()
```

As a result, you will see something similar to the following tables:

- for type II error (β)

	split_rate	(5000, 5000)	(10000, 10000)	(15000, 15000)	(20000, 20000)	(25000, 25000)	(30000, 30000)
metric	MDE						
check_rub_campaign	1.0%	>=0.9	>=0.9	>=0.9	>=0.9	>=0.9	>=0.9
	1.3%	>=0.9	>=0.9	>=0.9	>=0.9	0.8	0.4
	1.5%	>=0.9	>=0.9	>=0.9	0.5	<=0.2	<=0.2
	1.8%	>=0.9	>=0.9	0.6	0.3	<=0.2	<=0.2
	2.0%	>=0.9	0.8	0.3	<=0.2	<=0.2	<=0.2
	3.0%	>=0.9	<=0.2	<=0.2	<=0.2	<=0.2	<=0.2

Table should be read as follow: if you think that effect size of the experiment will be 1.8% and you want to constraint type II error by 20%, then the minimum number of observations in each group must be at least 25 000.

- for type I error (α)

split_rate	(5000, 5000)	(10000, 10000)	(15000, 15000)	(20000, 20000)	(25000, 25000)	(30000, 30000)
metric						
check_rub_campaign	0.0	0.0	0.0	0.0	0.0	0.0

1.6 Auto A/B

1.6.1 ABTest

class abacus.auto_ab.**ABTest**(*dataset: DataFrame | None*, *params:* ABTestParams)

Performs different calculations of A/B-test.

- Results evaluation for different metric types (continuous, binary, ratio).
- Bucketing (decrease number of points, normal distribution of metric of interest)

Example:

```
from abacus.auto_ab.abtest import ABTest
from abacus.auto_ab.params import ABTestParams, DataParams, HypothesisParams

data_params = DataParams(...)
hypothesis_params = HypothesisParams(...)
ab_params = ABTestParams(data_params, hypothesis_params)

df = pd.read_csv('data.csv')
ab_test = ABTest(df, ab_params)
ab_test.test_welch()
# {'stat': 5.172, 'p-value': 0.312, 'result': 0}
```

Attributes

dataset

Methods

bucketing()	Performs bucketing in order to accelerate results computation.
cupac()	Performs CUPAC for variance reduction.
cuped()	Performs CUPED for variance reduction.
linearization()	Creates linearized continuous metric based on ratio-metric.
plot([kind, save_path])	Plot experiment.
resplit_df()	Resplit dataframe.
test_boot_confint()	Performs bootstrap confidence interval and zero statistical significance.
test_boot_fp()	Performs bootstrap hypothesis testing by calculation of false positives.
test_boot_ratio()	Performs bootstrap for ratio-metric.
test_boot_welch()	Performs Welch's t-test for independent samples with unequal number of observations and variance.
test_buckets()	Performs buckets hypothesis testing.
test_chisquare()	Performs Chi-Square test.
test_delta_ratio()	Delta method with bias correction for ratios.
test_mannwhitney()	Performs Mann-Whitney U test.
test_taylor_ratio()	Calculate expectation and variance of ratio for each group and then use t-test for hypothesis testing.
test_welch()	Performs Welch's t-test.
test_z_proportions()	Performs z-test for proportions.

filter_outliers
metric_transform
report

ABTest	Performs different calculations of A/B-test.
ABTest.bucketing	Performs bucketing in order to accelerate results computation.
ABTest.cupac	Performs CUPAC for variance reduction.
ABTest.cuped	Performs CUPED for variance reduction.
ABTest.linearization	Creates linearized continuous metric based on ratio-metric.
ABTest.plot	Plot experiment.
ABTest.resplit_df	Resplit dataframe.
ABTest.test_boot_confint	Performs bootstrap confidence interval and zero statistical significance.
ABTest.test_boot_fp	Performs bootstrap hypothesis testing by calculation of false positives.
ABTest.test_boot_ratio	Performs bootstrap for ratio-metric.
ABTest.test_boot_welch	Performs Welch's t-test for independent samples with unequal number of observations and variance.
ABTest.test_buckets	Performs buckets hypothesis testing.
ABTest.test_chisquare	Performs Chi-Square test.
ABTest.test_delta_ratio	Delta method with bias correction for ratios.
ABTest.test_mannwhitney	Performs Mann-Whitney U test.
ABTest.test_taylor_ratio	Calculate expectation and variance of ratio for each group and then use t-test for hypothesis testing.
ABTest.test_welch	Performs Welch's t-test.
ABTest.test_z_proportions	Performs z-test for proportions.

abacus.auto_ab.ABTest.**__bucketize**(*self*, *x: List[float] | ndarray[Any, dtype[ScalarType]] | Series*) → ndarray

Split array `x` into N non-overlapping buckets.

There are two purposes for these actions:

1. Decrease number of data points of experiment.
2. Get normal distribution of a metric of interest.

Procedure:

1. Shuffle elements of an array.
2. Split points into N non-overlapping buckets.
3. On every bucket calculate metric of interest.

Parameters
x (*np.ndarray*) – Array to split.

Returns
Splitted into buckets array.

Return type
np.ndarray

abacus.auto_ab.ABTest.**__check_required_columns**(*self*, *df: DataFrame*, *method: str*) → None

Check presence of columns in dataframe.

Parameters

- **df** (*pandas.DataFrame*) – DataFrame to check.
- **method** (*str*) – Stage of A/B process which you'd like to test.

Raises

- **ValueError** – If *is_valid_col* is False. Experiment cannot be provided
- **if required columns are absent.** –

abacus.auto_ab.ABTest.**__get_group**(*self*, *group_label: str*, *df: DataFrame | None = None*) → ndarray

Gets target metric column based on desired group label.

Parameters

- **group_label** (*str*) – Group label, e.g. 'A', 'B'.
- **df** (*DataFrameType, optional*) – DataFrame to query from.

Returns
Target column for a desired group.

Return type
numpy.ndarray

abacus.auto_ab.ABTest.**__delta_params**(*self*, *x: DataFrame*) → Tuple[float, float]

Calculated expectation and variance for ratio metric using delta approximation.

Source: https://arxiv.org/pdf/1803.06336.pdf.

Parameters
x (*pandas.DataFrame*) – Pandas DataFrame of particular group (A, B, etc).

Returns
Mean and variance of ratio metric.

Return type
Tuple[float, float]

abacus.auto_ab.ABTest.**__manual_ttest**(*self*, *ctrl_mean: float*, *ctrl_var: float*, *ctrl_size: int*, *treat_mean: float*, *treat_var: float*, *treat_size: int*) → Dict[str, int | float]

Performs Welch's t-test based on aggregation of metrics instead of datasets.

For empirical calculation of T-statistic we need: expectation, variance, array size for each group.

Parameters

- **ctrl_mean** (*float*) – Mean of control group.
- **ctrl_var** (*float*) – Variance of control group.
- **ctrl_size** (*int*) – Size of control group.
- **treat_mean** (*float*) – Mean of treatment group.
- **treat_var** (*float*) – Variance of treatment group.
- **treat_size** (*int*) – Size of treatment group.

Returns
Dictionary with following properties: test statistic, p-value, test result. Test result: 1 - significant different, 0 - insignificant difference.

Return type
stat_test_typing

abacus.auto_ab.ABTest.__taylor_params(*self*, *x: DataFrame*) → Tuple[float, float]

Calculated expectation and variance for ratio metric using Taylor expansion approximation.

Source: https://www.stat.cmu.edu/~hseltman/files/ratio.pdf.

Parameters
x (*pandas.DataFrame*) – Pandas DataFrame of particular group (A, B, etc).

Returns
Mean and variance of ratio metric.

Return type
Tuple[float, float]

abacus.auto_ab.ABTest.**bucketing**(*self*) → *ABTest*

Performs bucketing in order to accelerate results computation.

Returns
New instance of ABTest class with modified control and treatment.

Return type
ABTest

abacus.auto_ab.ABTest.**cupac**(*self*) → *ABTest*

Performs CUPAC for variance reduction.

Returns
New instance of ABTest class with modified control and treatment.

Return type
ABTest

abacus.auto_ab.ABTest.**cuped**(*self*) → *ABTest*

Performs CUPED for variance reduction.

Returns
New instance of ABTest class with modified control and treatment.

Return type
ABTest

abacus.auto_ab.ABTest.**linearization**(*self*) → *ABTest*

Creates linearized continuous metric based on ratio-metric. Important: there is an assumption that all data is already grouped by user s.t. numerator for user = sum of numerators for user for different time periods and denominator for user = sum of denominators for user for different time periods

Source: https://research.yandex.com/publications/148.

abacus.auto_ab.ABTest.**plot**(*self*, *kind: str = 'experiment'*, *save_path: str | None = None*) → None

Plot experiment.

Parameters

- **kind** (*str*) – Kind of plot: ‘experiment’, ‘bootstrap’.
- **save_path** (*str, optional*) – Path where to save image.

Raises
ValueError – If *kind* is not in [‘experiment’, ‘bootstrap’].

abacus.auto_ab.ABTest.**resplit_df**(*self*) → *ABTest*

Resplit dataframe.

Returns
Instance of `ABTest` class with modified control and treatment.

Return type
ABTest

abacus.auto_ab.ABTest.**test_boot_confint**(*self*) → Dict[str, int | float]

Performs bootstrap confidence interval and zero statistical significance.

Returns
Dictionary with following properties: `test statistic`, `p-value`, `test result`. Test result: 1 - significant different, 0 - insignificant difference.

Return type
stat_test_typing

abacus.auto_ab.ABTest.**test_boot_fp**(*self*) → Dict[str, int | float]

Performs bootstrap hypothesis testing by calculation of false positives.

Returns
Dictionary with following properties: `test statistic`, `p-value`, `test result`. Test result: 1 - significant different, 0 - insignificant difference.

Return type
stat_test_typing

abacus.auto_ab.ABTest.**test_boot_ratio**(*self*) → Dict[str, int | float]

Performs bootstrap for ratio-metric.

Returns
Dictionary with following properties: `test statistic`, `p-value`, `test result`. Test result: 1 - significant different, 0 - insignificant difference.

Return type
stat_test_typing

abacus.auto_ab.ABTest.**test_boot_welch**(*self*) → Dict[str, int | float]

Performs Welch's t-test for independent samples with unequal number of observations and variance.

Welch's t-test is used as a wider approaches with fewer restrictions on samples size as in Student's t-test.

Statistic of the test:

$$t = \frac{\hat{X}_1 - \hat{X}_2}{\sqrt{\frac{s_1}{\sqrt{N_1}} + \frac{s_2}{\sqrt{N_2}}}}.$$

Returns
Dictionary with following properties: `test statistic`, `p-value`, `test result`. Test result: 1 - significant different, 0 - insignificant difference.

Return type
stat_test_typing

abacus.auto_ab.ABTest.**test_buckets**(*self*) → Dict[str, int | float]

Performs buckets hypothesis testing.

Returns
Dictionary with following properties: `test statistic`, `p-value`, `test result`. Test result: 1 - significant different, 0 - insignificant difference.

Return type
stat_test_typing

abacus.auto_ab.ABTest.**test_chisquare**(*self*) → Dict[str, int | float]

Performs Chi-Square test.

Returns
Dictionary with following properties: `test statistic`, `p-value`, `test result`. Test result: 1 - significant different, 0 - insignificant difference.

Return type
stat_test_typing

abacus.auto_ab.ABTest.**test_delta_ratio**(*self*) → Dict[str, int | float]

Delta method with bias correction for ratios.

Source: https://arxiv.org/pdf/1803.06336.pdf.

Returns
Dictionary with following properties: `test statistic`, `p-value`, `test result`. Test result: 1 - significant different, 0 - insignificant difference.

Return type
stat_test_typing

abacus.auto_ab.ABTest.**test_mannwhitney**(*self*) → Dict[str, int | float]

Performs Mann-Whitney U test.

Test works on continues metrics and their ranks.

Assumptions of Mann-Whitney test:

1. Independence of observations.
2. Same shape of metric distributions.

Statistic of the test:

$$U = \sum_{i=1}^{n} \sum_{j=1}^{m} S(X_i, Y_j).$$

Returns
Dictionary with following properties: `test statistic`, `p-value`, `test result`. Test result: 1 - significant different, 0 - insignificant difference.

Return type
stat_test_typing

abacus.auto_ab.ABTest.**test_taylor_ratio**(*self*) → Dict[str, int | float]

Calculate expectation and variance of ratio for each group and then use t-test for hypothesis testing.

Source: http://www.stat.cmu.edu/~hseltman/files/ratio.pdf.

Returns
Dictionary with following properties: `test statistic`, `p-value`, `test result`. Test result: 1 - significant different, 0 - insignificant difference.

Return type
stat_test_typing

abacus.auto_ab.ABTest.**test_welch**(*self*) → Dict[str, int | float]

Performs Welch's t-test.

Returns
Dictionary with following properties: `test statistic`, `p-value`, `test result`. Test result: 1 - significant different, 0 - insignificant difference.

Return type
stat_test_typing

abacus.auto_ab.ABTest.**test_z_proportions**(*self*) → Dict[str, int | float]

Performs z-test for proportions.

The two-proportions z-test is used to compare two observed proportions.

Statistic of the test:

$$Z = \frac{\hat{p}_1 - \hat{p}_2}{\sqrt{\hat{p}(1-\hat{p})(\frac{1}{n_1} + \frac{1}{n_2})}}.$$

Returns
Dictionary with following properties: `test statistic`, `p-value`, `test result`. Test result: 1 - significant different, 0 - insignificant difference.

Return type
stat_test_typing

1.6.2 VarianceReduction

class abacus.auto_ab.**VarianceReduction**

Implementation of sensitivity increasing approaches.

As it is easier to apply variance reduction techniques directly to experiment, all approaches should be called on `ABTest` class instance.

Example:

```
from abacus.auto_ab.abtest import ABTest
from abacus.auto_ab.params import ABTestParams, DataParams, HypothesisParams

data_params = DataParams(...)
hypothesis_params = HypothesisParams(...)
ab_params = ABTestParams(data_params, hypothesis_params)

df = pd.read_csv('data.csv')
ab_test = ABTest(df, ab_params)
ab_test = ab_test.cuped()
```

Methods

cupac(x, target_prev_col, target_now_col, ...)	Perform CUPED on target variable with covariate calculated as a prediction from a linear regression model.
cuped(df, target_col, groups_col, covariate_col)	Perform CUPED on target variable with predefined covariate.

abacus.auto_ab.VarianceReduction.**_target_encoding**(*x: DataFrame*, *encoding_columns: List[str] | ndarray[Any, dtype[ScalarType]] | Series*, *target_column: str*) → DataFrame

Encodes target column.

abacus.auto_ab.VarianceReduction.**_predict_target**(*x: DataFrame*, *target_prev_col: str*, *factors_prev_cols: List[str] | ndarray[Any, dtype[ScalarType]] | Series*, *factors_now_cols: List[str] | ndarray[Any, dtype[ScalarType]] | Series*) → List[float] | ndarray[Any, dtype[ScalarType]] | Series

Covariate prediction with linear regression model.

Parameters

- **x** (*pandas.DataFrame*) – Pandas DataFrame.
- **target_prev_col** (*str*) – Target on previous period column name.
- **factors_prev_cols** (*List[str]*) – Factor columns for modelling.
- **factors_now_cols** (*List[str]*) – Factor columns for prediction on current period.

Returns
Pandas Series with predicted values

Return type
pandas.Series

abacus.auto_ab.VarianceReduction.**cuped**(*df: DataFrame*, *target_col: str*, *groups_col: str*, *covariate_col: str*) → DataFrame

Perform CUPED on target variable with predefined covariate.

Covariate has to be chosen with regard to the following restrictions:

1. Covariate is independent of an experiment.
2. Covariate is highly correlated with target variable.
3. Covariate is continuous variable.

Original paper: https://exp-platform.com/Documents/2013-02-CUPED-ImprovingSensitivityOfControlledExperiments.pdf.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame for analysis.
- **target_col** (*str*) – Target column name.
- **groups_col** (*str*) – Groups A and B column name.

- **covariate_col** (*str*) – Covariate column name. If None, then most correlated column in considered as covariate.

Returns
Pandas DataFrame with additional target CUPEDed column

Return type
pandas.DataFrame

abacus.auto_ab.VarianceReduction.**cupac**(*x: DataFrame*, *target_prev_col: str*, *target_now_col: str*, *factors_prev_cols: List[str] | ndarray[Any, dtype[ScalarType]] | Series*, *factors_now_cols: List[str] | ndarray[Any, dtype[ScalarType]] | Series*, *groups_col: str*) → DataFrame

Perform CUPED on target variable with covariate calculated as a prediction from a linear regression model.

Original paper: https://doordash.engineering/2020/06/08/improving-experimental-power-through-control-using-predictions-as-c

Parameters

- **x** (*pandas.DataFrame*) – Pandas DataFrame for analysis.
- **target_prev_col** (*str*) – Target on previous period column name.
- **target_now_col** (*str*) – Target on current period column name.
- **factors_prev_cols** (*List[str]*) – Factor columns for modelling.
- **factors_now_cols** (*List[str]*) – Factor columns for prediction on current period.
- **groups_col** (*str*) – Groups column name.

Returns
Pandas DataFrame with additional columns: target_pred and target_now_cuped

Return type
pandas.DataFrame

1.6.3 Graphics

class abacus.auto_ab.**Graphics**

Illustration of an experiment.

- As it is easier to apply plotting directly to experiment, all methods should be called on `ABTest` class instance.
- Experiment's plot is based on metric type.

Example:

```
from abacus.auto_ab.abtest import ABTest
from abacus.auto_ab.params import ABTestParams, DataParams, HypothesisParams

data_params = DataParams(...)
hypothesis_params = HypothesisParams(...)
ab_params = ABTestParams(data_params, hypothesis_params)
```

(continues on next page)

(continued from previous page)

```
df = pd.read_csv('data.csv')
ab_test = ABTest(df, ab_params)
ab_test.plot()
```

Methods

plot_binary_experiment(params, save_path)	Plot experiment with binary outcome.
plot_bootstrap_confint(params, save_path)	Plot bootstrapped metric of an experiment with its confidence interval and zero value.
plot_continuous_experiment(params, save_path)	Plot distributions of continuous metric and actual experiment metric.

abacus.auto_ab.Graphics.**plot_continuous_experiment**(*params:* ABTestParams, *save_path: str | None*) → None

Plot distributions of continuous metric and actual experiment metric.

Parameters

- **params** (ABTestParams) – Parameters of the experiment.
- **save_path** (*str, optional*) – Path where to save image.

abacus.auto_ab.Graphics.**plot_binary_experiment**(*params:* ABTestParams, *save_path: str | None*) → None

Plot experiment with binary outcome.

Parameters

- **params** (ABTestParams) – Parameters of the experiment.
- **save_path** (*str, optional*) – Path where to save image.

abacus.auto_ab.Graphics.**plot_bootstrap_confint**(*params:* ABTestParams, *save_path: str | None*) → None

Plot bootstrapped metric of an experiment with its confidence interval and zero value.

Parameters

- **x** (*np.ndarray*) – Bootstrap metric.
- **params** (ABTestParams) – Parameters of the experiment.

1.6.4 Params

class abacus.auto_ab.**DataParams**(*id_col: str = 'id', group_col: str = 'groups', control_name: str = 'A', treatment_name: str = 'B', is_grouped: bool | None = True, strata_col: str | None = '', target: str | None = '', numerator: str | None = '', denominator: str | None = '', covariate: str | None = '', target_prev: str | None = '', predictors_now: ~typing.List[str] | ~numpy.ndarray[~typing.Any, ~numpy.dtype[~numpy._typing._generic_alias.ScalarType]] | ~pandas.core.series.Series | None = FieldInfo(default=PydanticUndefined, default_factory=<class 'list'>, extra={}), predictors_prev: ~typing.List[str] | ~numpy.ndarray[~typing.Any, ~numpy.dtype[~numpy._typing._generic_alias.ScalarType]] | ~pandas.core.series.Series | None = FieldInfo(default=PydanticUndefined, default_factory=<class 'list'>, extra={}), control: ~typing.List[float] | ~numpy.ndarray[~typing.Any, ~numpy.dtype[~numpy._typing._generic_alias.ScalarType]] | ~pandas.core.series.Series | None = FieldInfo(default=PydanticUndefined, default_factory=<class 'list'>, extra={}), treatment: ~typing.List[float] | ~numpy.ndarray[~typing.Any, ~numpy.dtype[~numpy._typing._generic_alias.ScalarType]] | ~pandas.core.series.Series | None = FieldInfo(default=PydanticUndefined, default_factory=<class 'list'>, extra={}), transforms: ~typing.List[str] | ~numpy.ndarray[~typing.Any, ~numpy.dtype[~numpy._typing._generic_alias.ScalarType]] | ~pandas.core.series.Series | None = FieldInfo(default=PydanticUndefined, default_factory=<class 'list'>, extra={})*)

Data description as column names of dataset generated during experiment.

Parameters

- **id_col** (*str*) – ID of observations.
- **group_col** (*str*) – Group of experiment.
- **control_name** (*str*) – Name of control group in `group_col`.
- **treatment_name** (*str*) – Name of treatment group in `group_col`.
- **is_grouped** (*bool, Optional*) – Flag that shows whether observations are grouped.
- **strata_col** (*str, Optional*) – Name of stratification column. Stratification column must be categorical.
- **target** (*str, Optional*) – Target column name of continuous or binary metric.
- **numerator** (*str, Optional*) – Numerator for ratio metric.
- **denominator** (*str, Optional*) – Denominator for ratio metric.
- **covariate** (*str, Optional*) – Covariate column for CUPED.
- **target_prev** (*str, Optional*) – Target column name for previous period of continuous metric.
- **predictors_now** (*List[str], Optional*) – List of columns to predict covariate.
- **predictors_prev** (*List[str], Optional*) – List of columns to create linear model for covariate prediction.
- **control** (*ArrayNumType, Optional*) – Control group data used for quick access and excluding querying dataset.

- **treatment** (*ArrayNumType, Optional*) – Treatment group data used for quick access and excluding querying dataset.
- **transforms** (*ArrayStrType, Optional*) – List of transformations applied to experiment.

class abacus.auto_ab.**HypothesisParams**(*alpha: float | None = 0.05, beta: float | None = 0.2, alternative: str | None = 'two-sided', metric_type: str | None = 'continuous', metric_name: str | None = 'mean', metric: ~typing.Callable[[~typing.List[float] | ~numpy.ndarray[~typing.Any, ~numpy.dtype[~numpy._typing._generic_alias.ScalarType]] | ~pandas.core.series.Series], float] | None = <function mean>, metric_transform: ~typing.Callable[[~numpy.ndarray], ~numpy.ndarray] | None = <function mean>, metric_transform_info: ~typing.Dict[str, ~typing.Dict[str, ~typing.Any]] | None = FieldInfo(default=PydanticUndefined, default_factory=<class 'dict'>, extra={}), filter_method: str | None = 'top_5', n_boot_samples: int | None = 200, n_buckets: int | None = 100, strata: str | None = '', strata_weights: ~typing.Dict[str, float] | None = FieldInfo(default=PydanticUndefined, default_factory=<class 'dict'>, extra={})*)

Description of hypothesis parameters.

Parameters

- **alpha** (*float*) – type I error.
- **beta** (*float*) – type II error.
- **alternative** (*str*) – directionality of hypothesis: less, greater, two-sided.
- **metric_type** (*str*) – metric type: continuous, binary, ratio.
- **metric_name** (*str*) – metric name: mean, median. If custom metric, then use here appropriate name.
- **metric** (*Callable[[Iterable[float]], np.ndarray], Optional*) – if metric_name is custom, then you must define metric function.
- **metric_transform** (*Callable[[np.ndarray], np.ndarray], Optional*) – applied transformations to experiment.
- **metric_transform_info** (*Dict[str, Dict[str, Any]], Optional*) – information of applied transformations.
- **filter_method** (*str, Optional*) – method for filtering outliers: top_5, isolation_forest.
- **n_boot_samples** (*int, Optional*) – number of bootstrap iterations.
- **n_buckets** (*int, Optional*) – number of buckets.
- **strata** (*str, Optional*) – stratification column.
- **strata_weights** (*Dict[str, float], Optional*) – historical strata weights.

Methods

metric([axis, dtype, out, keepdims, where])	Compute the arithmetic mean along the specified axis.
metric_transform([axis, dtype, out, ...])	Compute the arithmetic mean along the specified axis.

alpha_validator
alternative_validator
beta_validator
metric_type_validator

class abacus.auto_ab.**ABTestParams**(*data_params: 'DataParams' = FieldInfo(default=DataParams(id_col='id', group_col='groups', control_name='A', treatment_name='B', is_grouped=True, strata_col='', target='', numerator='', denominator='', covariate='', target_prev='', predictors_now=[], predictors_prev=[], control=[], treatment=[], transforms=[]), extra={}), hypothesis_params: 'HypothesisParams' = FieldInfo(default=HypothesisParams(alpha=0.05, beta=0.2, alternative='two-sided', metric_type='continuous', metric_name='mean', metric=<function mean at 0x7f06d0e5c4c0>, metric_transform=<function mean at 0x7f06d0e5c4c0>, metric_transform_info={}, filter_method='top_5', n_boot_samples=200, n_buckets=100, strata='', strata_weights={}), extra={}))*

1.7 Splitter

SplitBuilder	Builds stratification split for DataFrame.
SplitBuilderParams	Split experiment parameters class.

1.7.1 Split Builder

class abacus.splitter.**SplitBuilder**(*split_data: DataFrame*, *params:* SplitBuilderParams)

Builds stratification split for DataFrame.

Methods

collect()	Calculated splits for init dataframe

1.7.2 Params

class abacus.splitter.**SplitBuilderParams**(*map_group_names_to_sizes: ~typing.Dict[str, int | None], main_strata_col: str, split_metric_col: str, metric_type: str = 'continuous', id_col: str = 'customer_id', cols: ~typing.List[str] = FieldInfo(default=PydanticUndefined, default_factory=<class 'list'>, extra={}), cat_cols: ~typing.List[str] = FieldInfo(default=PydanticUndefined, default_factory=<class 'list'>, extra={}), n_bins: int = 3, min_cluster_size: int = 100, strata_outliers_frac: float = 0.01, alpha: float = 0.05*)

Split experiment parameters class.

Parameters

- **map_group_names_to_sizes** (*Dict*) – dictionary with group names and sizes. Key with name "control" is obligatory
- **main_strata_col** (*str*) – the name of the column to be used first for splitting
- **split_metric_col** (*str*) – the name of the column to be binning data for splitting
- **id_col** (*str*) – the name of the column with id
- **cols** – columns for stratification data
- **cat_cols** – categorical columns that are using for stratification. These columns will be encoded as category features
- **n_bins** – number of bins to be created based on split_metric_col
- **min_cluster_size** – min count of samples in HDBSCAN cluster
- **strata_outliers_frac** – frequency of outliers in strata
- **alpha** – significance level for A/A test for split

Methods

alpha_validator
metric_type_validator

1.8 Resplitter

ResplitBuilder	Builds stratification split for DataFrame.
ResplitParams	Resplit params class.

1.8.1 Resplit Builder

class abacus.resplitter.**ResplitBuilder**(*df: DataFrame*, *resplit_params:* ResplitParams)

Builds stratification split for DataFrame.

Methods

collect()	Method recalculate fractions of each strata in dataframe.

abacus.resplitter.ResplitBuilder.**collect**(*self*) → DataFrame

Method recalculate fractions of each strata in dataframe.

Returns
DataFrame with recalculated strata fractions.

Return type
pandas.DataFrame

1.8.2 Params

class abacus.resplitter.**ResplitParams**(*group_names: GroupNames*, *strata_col: str = 'strata'*, *group_col: str = 'group_col'*)

Resplit params class.

Parameters

- **group_names** (*GroupNames*) – group names
- **strata_col** (*str*) – name of column with strata
- **group_col** (*str*) – name of column with groups split

1.9 MDE Researcher

AbstractMdeResearchBuilder	Base class for Experiment Builders.
BaseSplitElement	Dataclass with data params for experiment calculations
MdeAlphaExperiment	Dataclass for I type error calculations.
MdeBetaExperiment	Dataclass for II type error calculations.
MdeResearchBuilder	Calculates I and II type errors for different group sizes and injects.
MultipleSplitBuilder	Columns with splits and injects will be added
MdeParams	MDE research experiment parameters class.

1.9.1 Abstract MDE Experiment

class abacus.mde_researcher.**AbstractMdeResearchBuilder**(*guests: DataFrame*, *abtest_params:* ABTestParams, *experiment_params:* MdeParams)

Base class for Experiment Builders.

Attributes

experiment_params
group_sizes

abacus.mde_researcher.AbstractMdeResearchBuilder.**_build_group_sizes**(*self*)

Build list of groups sizes tuples.

Returns
List of groups sizes pairs.

Return type
List[int]

1.9.2 Experiment Structures

class abacus.mde_researcher.**BaseSplitElement**(*group_sizes: tuple*, *split_number: int*, *control_group_size: int = FieldInfo(default=PydanticUndefined, extra={'init': False})*, *target_group_size: int = FieldInfo(default=PydanticUndefined, extra={'init': False})*)

Dataclass with data params for experiment calculations

Parameters

- **group_sizes** (*tuple*) – tuple with group sizes. Should have control group size on the 0 index position and target group size on the 1 index position
- **split_number** (*int*) – params with number of split

class abacus.mde_researcher.**MdeAlphaExperiment**(*group_sizes: tuple*, *split_number: int*, *control_group_size: int = FieldInfo(default=PydanticUndefined, extra={'init': False})*, *target_group_size: int = FieldInfo(default=PydanticUndefined, extra={'init': False})*, *metric_name: str = FieldInfo(default=PydanticUndefined, extra={'init': False})*)

Dataclass for I type error calculations.

class abacus.mde_researcher.**MdeBetaExperiment**(*group_sizes: tuple*, *split_number: int*, *control_group_size: int = FieldInfo(default=PydanticUndefined, extra={'init': False})*, *target_group_size: int = FieldInfo(default=PydanticUndefined, extra={'init': False})*, *metric_name: str = FieldInfo(default=PydanticUndefined, extra={'init': False})*, *inject: float = FieldInfo(default=PydanticUndefined, extra={'init': False})*)

Dataclass for II type error calculations.

1.9.3 MDE Research Builder

class abacus.mde_researcher.**MdeResearchBuilder**(*guests: DataFrame*, *abtest_params:* ABTestParams, *experiment_params:* MdeParams, *stratification_params:* SplitBuilderParams)

Calculates I and II type errors for different group sizes and injects.

Attributes

experiment_params
group_sizes

Methods

calc_alpha(guests[, is_splitted])	Calculates I type error.
collect([fill_with_default])	Calculates I and II types error using prepilot parameters.

abacus.mde_researcher.MdeResearchBuilder.**calc_alpha**(*self*, *guests: DataFrame*, *is_splitted: bool = False*) → DataFrame

Calculates I type error.

Parameters

- **guests** (*pandas.DataFrame*) – Dataframe with guests.
- **is_splitted** (*bool*) – If False guests must contain splits for calculation. Otherwise splits will be compute for guests.

Returns
Pandas DataFrame with I type error.

Return type
pandas.DataFrame

abacus.mde_researcher.MdeResearchBuilder.**collect**(*self*, *fill_with_default: bool = True*) → DataFrame

Calculates I and II types error using prepilot parameters.

Parameters
fill_with_default (*bool*) – Fill calculated vaules with defaults.

Returns
Pandas DataFrames with aggregated results of experiment.

Return type
pandas.DataFrame

1.9.4 Multiple Split Builder

class abacus.mde_researcher.**MultipleSplitBuilder**(*guests: DataFrame*, *metrics_names: List[str]*, *injects: List[float]*, *group_sizes: List[int]*, *stratification_params:* SplitBuilderParams, *iterations_number: int = 10*)

Columns with splits and injects will be added

Methods

calc_injected_metrics(guests_for_injects)	Calculates injected metrics for guests df.
collect()	Calculate multiple split with stratification.

abacus.mde_researcher.MultipleSplitBuilder.**_build_splits_grid**(*self*)

abacus.mde_researcher.MultipleSplitBuilder.**_update_strat_params**(*self*)

Update stratification columns, because of columns names duplicated problem.

abacus.mde_researcher.MultipleSplitBuilder.**_build_split**(*self*, *guests_with_strata: DataFrame*, *control_group_size: int*, *target_group_size: int*, *split_number: int = 1*)

Calculate one split with stratification.

Parameters

- **guests_with_strata** (*pandas.DataFrame*) – Dataframe with stratas.
- **control_group_size** (*int*) – Control group size.
- **target_group_size** (*int*) – Target group size.
- **split_number** (int, default = 1) – Number of split. Uses as suffix for new column.

Returns
pandas DataFrame with split.

Return type
pandas.DataFrame

abacus.mde_researcher.MultipleSplitBuilder.**calc_injected_metrics**(*self*, *guests_for_injects: DataFrame*) → DataFrame

Calculates injected metrics for guests df.

Parameters
guests_for_injects (*pandas.DataFrame*) – Dataframe with metrics columns.

Returns
Dataframe with injected metrics columns.

Return type
pandas.DataFrame

abacus.mde_researcher.MultipleSplitBuilder.**collect**(*self*) → DataFrame

Calculate multiple split with stratification.

Returns
Pandas DataFrame with split columns.

Return type
pandas.DataFrame

1.9.5 Params

class abacus.mde_researcher.**MdeParams**(*metrics_names: ~typing.List[str], injects: ~typing.List[float], min_group_size: int, max_group_size: int, step: int, variance_reduction: ~typing.Callable[[~abacus.auto_ab.abtest.ABTest], ~abacus.auto_ab.abtest.ABTest] | None = None, use_buckets: bool = False, transformations: ~typing.Any = None, stat_test: ~typing.Callable[[~abacus.auto_ab.abtest.ABTest], ~typing.Dict[str, int | float]] = <function ABTest.test_boot_confint>, iterations_number: int = 10, max_beta_score: float = 0.2, min_beta_score: float = 0.05*)

MDE research experiment parameters class.

Parameters

- **metrics_names** – Metrics which will be compare in experiments.
- **injects** – Injects represent MDE values.
- **min_group_size** – Minimal value of groups sizes.
- **max_group_size** – Maximal value of groups sizes.
- **step** – Spacing between min_group_size and max_group_size.
- **variance_reduction** – ABTest methods for variance reduction.
- **use_buckets** – Use bucketize method.
- **transformations** – Pipeline of experiment. Will be calulted in __post_init__.
- **stat_test** – Statistical test type.
- **iterations_number** – Count of splits for each element in group_sizes.
- **max_beta_score** – Maximum level of II type error.
- **min_beta_score** – Minimum level of II type error.

Attributes

transformations
variance_reduction

Methods

`stat_test`()	Performs bootstrap confidence interval and zero statistical significance.

groups_sizes_validator
stat_test_validator
variance_reduction_validator

CHAPTER

TWO

EXAMPLES

For more details, see the examples.

CHAPTER

THREE

COMMUNICATION

Developers and authors:

- Vadim Glukhov
- Egor Shishkovets
- Dmitry Zabavin

INDEX

Symbols

A

B

C

D

G

H

L

M

P

R

S

T

V