

---

# **modelgym Documentation**

***Release 0.1.3***

**anaderi**

**Apr 13, 2018**



---

## Contents

---

|          |                                        |           |
|----------|----------------------------------------|-----------|
| <b>1</b> | <b>What is this about?</b>             | <b>3</b>  |
| <b>2</b> | <b>Installation</b>                    | <b>5</b>  |
| 2.1      | Installation . . . . .                 | 5         |
| 2.1.1    | Installation without Docker . . . . .  | 5         |
| 2.1.2    | Model Gym With Docker . . . . .        | 6         |
| <b>3</b> | <b>Examples</b>                        | <b>9</b>  |
| 3.1      | Basic Tutorial . . . . .               | 9         |
| 3.1.1    | Define models we want to use . . . . . | 9         |
| 3.1.2    | Get dataset . . . . .                  | 10        |
| 3.1.3    | Create a TPE trainer . . . . .         | 10        |
| 3.1.4    | Optimize hyperparams . . . . .         | 10        |
| 3.1.5    | Report best results . . . . .          | 10        |
| 3.2      | Guru example . . . . .                 | 15        |
| 3.2.1    | Main features . . . . .                | 16        |
| 3.2.2    | dtype with fields . . . . .            | 19        |
| <b>4</b> | <b>Documentaion</b>                    | <b>21</b> |
| 4.1      | Guru . . . . .                         | 21        |
| 4.2      | Models . . . . .                       | 23        |
| 4.2.1    | Model interface . . . . .              | 23        |
| 4.2.2    | XGBoost . . . . .                      | 24        |
| 4.2.3    | LightGBM . . . . .                     | 26        |
| 4.2.4    | RandomForestClassifier . . . . .       | 27        |
| 4.2.5    | Catboost . . . . .                     | 28        |
| 4.2.6    | Ensemble Model . . . . .               | 30        |
| 4.3      | Metrics . . . . .                      | 32        |
| 4.3.1    | Base Class . . . . .                   | 32        |
| 4.3.2    | Sklearn Metrics . . . . .              | 32        |
| 4.4      | Trainers . . . . .                     | 33        |
| 4.4.1    | Hyperopt trainers . . . . .            | 33        |
| 4.4.2    | Skopt trainers . . . . .               | 34        |
| 4.5      | Trackers . . . . .                     | 35        |
| 4.6      | Compare models . . . . .               | 36        |
|          | <b>Python Module Index</b>             | <b>37</b> |



Gym for predictive models



# CHAPTER 1

---

## What is this about?

---

Modelgym is a place (a library?) to get your predictive models as meaningful in a smooth and effortless manner. Modelgym provides the unified interface for

- different kind of Models (XGBoost, CatBoost etc)



## 2.1 Installation

### 2.1.1 Installation without Docker

**Note:** This installation guide was written for python3

#### Starting Virtual Environment

Create directory where you want to clone this rep and switch to it. Install virtualenv and start it:

```
pip3 install virtualenv
python3 -m venv venv
source venv/bin/activate
```

To deactivate simply type `deactivate`

#### Installing Dependencies

Install required python3 packages by running following commands.

1. modelgym:

```
pip3 install git+https://github.com/yandexdataschool/modelgym.git
```

2. jupyter:

```
pip3 install jupyter
```

3. LightGBM. Modelgym works with LightGBM version 2.0.4:

```
pip3 install lightgbm==2.0.4
```

#### 4. XGBoost. Modelgym works with XGBoost version 0.6:

```
git clone --recursive https://github.com/dmlc/xgboost
cd xgboost
git checkout 14fba01b5ac42506741e702d3fde68344a82f9f0
make -j
cd python-package; python3 setup.py install
cd ../../
rm -rf xgboost
```

### Verification If Model Gym Works Correctly

Clone repository:

```
git clone https://github.com/yandexdataschool/modelgym.git
```

Move to example and start jupyter-notebook:

```
cd modelgym/example
jupyter-notebook
```

Open `model_search.ipynb` and run all cells. If there are no errors, everything is alright!

## 2.1.2 Model Gym With Docker

### Getting Started

To run model gym inside Docker container you need to have installed [Docker](#). Also for Mac or Windows you can install instead [Kitematic](#).

Download this repo. All of the needed files are in the `modelgym` directory:

```
$ git clone https://github.com/yandexdataschool/modelgym.git
$ cd ./modelgym
```

### Running Model Gym In A Container Using DockerHub Image

To run docker container with official image `modelgym/jupyter:latest` from DockerHub repo for using model gym via jupyter you simply run the command:

```
$ docker run -ti --rm -v "$(pwd)":/src -p 7777:8888 \
modelgym/jupyter:latest bash --login -ci 'jupyter notebook'
```

If you are using Windows you need to run this instead:

```
$ docker run -ti --rm -v %cd%:/src -p 7777:8888 \
modelgym/jupyter:latest bash --login -ci "jupyter notebook"
```

At first time it downloads container.

### Verification If Model Gym Works Correctly

Firstly you should check inside container that `/src` is not empty.

To connect to jupyter host in browser check your Docker public ip:

```
$ docker-machine ip default
```

Usually the default ip is `192.168.99.100`.

When you start a notebook server with token authentication enabled (default), a token is generated to use for authentication. This token is logged to the terminal, so that you can copy it.

Go to `http://<your published ip>:7777/` and paste auth token.

Open `/example/model_search.ipynb` and try to run all cells. If there are no errors, everything is allright.



### 3.1 Basic Tutorial

Welcome to Modelgym Basic Tutorial.

As an example, we will show you how to use Modelgym for binary classification problem.

In this tutorial we will go through the following steps:

Choosing the models.

Searching for the best hyperparameters on default spaces using TPE algorithm locally.

Visualizing the results.

#### 3.1.1 Define models we want to use

In this tutorial, we will use

1. LightGBMClassifier
2. XGBoostClassifier
3. RandomForestClassifier
4. CatBoostClassifier

```
from modelgym.models import LGBMClassifier, XGBClassifier, RFClassifier, CtBClassifier
```

```
/Users/f-minkin/.pyenv/versions/3.6.2/lib/python3.6/site-packages/sklearn/cross_
→validation.py:41: DeprecationWarning: This module was deprecated in version 0.18 in
→favor of the model_selection module into which all the refactored classes and
→functions are moved. Also note that the interface of the new CV iterators are
→different from that of this module. This module will be removed in 0.20.
    "This module will be removed in 0.20.", DeprecationWarning)
```

```
models = [LGBMClassifier, XGBClassifier, RFClassifier, CtBClassifier]
```

### 3.1.2 Get dataset

For tutorial purposes we will use toy dataset

```
from sklearn.datasets import make_classification
from modelgym.utils import XYCDataset
```

```
X, y = make_classification(n_samples=500, n_features=20, n_informative=10, n_
    ↳classes=2)
dataset = XYCDataset(X, y)
```

### 3.1.3 Create a TPE trainer

```
from modelgym.trainers import TpeTrainer
trainer = TpeTrainer(models)
```

### 3.1.4 Optimize hyperparams

We chose accuracy as a main metric that we rely on when optimizing hyperparams.

Also keep track for RocAuc and F1 measure besides accuracy for our best models.

Please, keep in mind, that now we're optimizing hyperparameters from the default space of hyperparameters. That means, they are not optimal, for optimal ones and complete understanding follow advanced tutorial.

```
from modelgym.metrics import Accuracy, RocAuc, F1
```

Of course, it will take some time.

```
%%time
trainer.crossval_optimize_params(Accuracy(), dataset, metrics=[Accuracy(), RocAuc(),
    ↳F1()])
```

```
/Users/f-minkin/.pyenv/versions/3.6.2/lib/python3.6/site-packages/sklearn/metrics/
    ↳classification.py:1135: UndefinedMetricWarning: F-score is ill-defined and being_
    ↳set to 0.0 due to no predicted samples.
    'precision', 'predicted', average, warn_for)
```

```
CPU times: user 2h 2min 45s, sys: 47min 59s, total: 2h 50min 45s
Wall time: 28min 17s
```

### 3.1.5 Report best results

```
from modelgym.report import Report
```

```
reporter = Report(trainer.get_best_results(), dataset, [Accuracy(), RocAuc(), F1()])
```

## Report in text form

```
reporter.print_all_metric_results()
```

```

~~~~~ accuracy ~~~~~
↪ ~~~~~

          tuned
LGBMClassifier  0.776002 (0.00%)
XGBClassifier   0.838059 (8.00%)
RFCClassifier   0.800075 (3.10%)
CtBClassifier   0.861963 (11.08%)

~~~~~ roc_auc ~~~~~
↪ ~~~~~

          tuned
LGBMClassifier  0.815768 (0.00%)
XGBClassifier   0.904991 (10.94%)
RFCClassifier   0.875230 (7.29%)
CtBClassifier   0.926832 (13.61%)

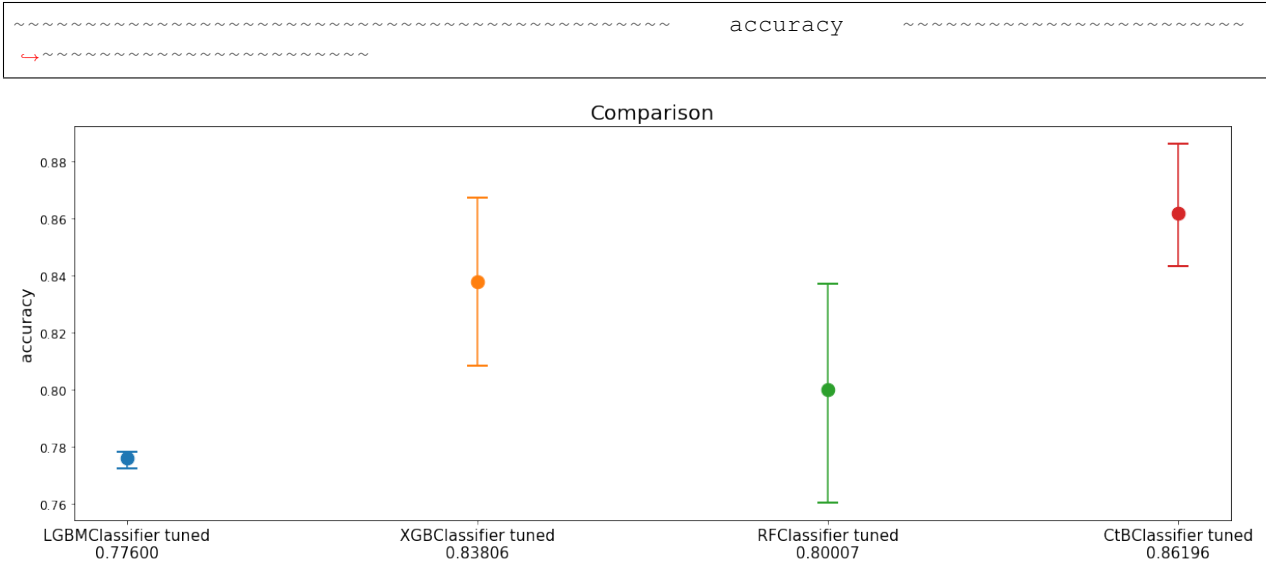
~~~~~ f1_score ~~~~~
↪ ~~~~~

          tuned
LGBMClassifier  0.777157 (0.00%)
XGBClassifier   0.835813 (7.55%)
RFCClassifier   0.792136 (1.93%)
CtBClassifier   0.859078 (10.54%)

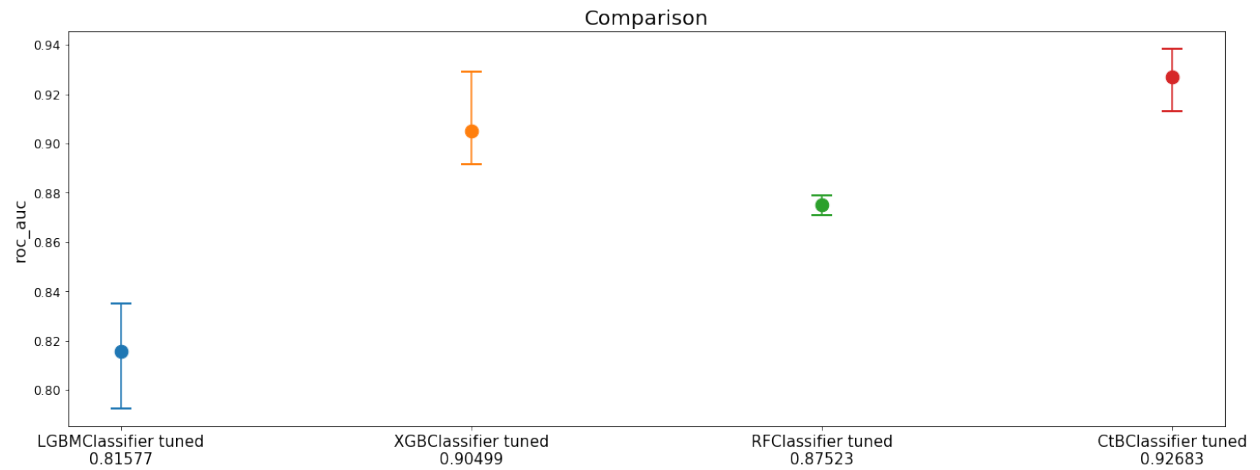
```

## Report plots

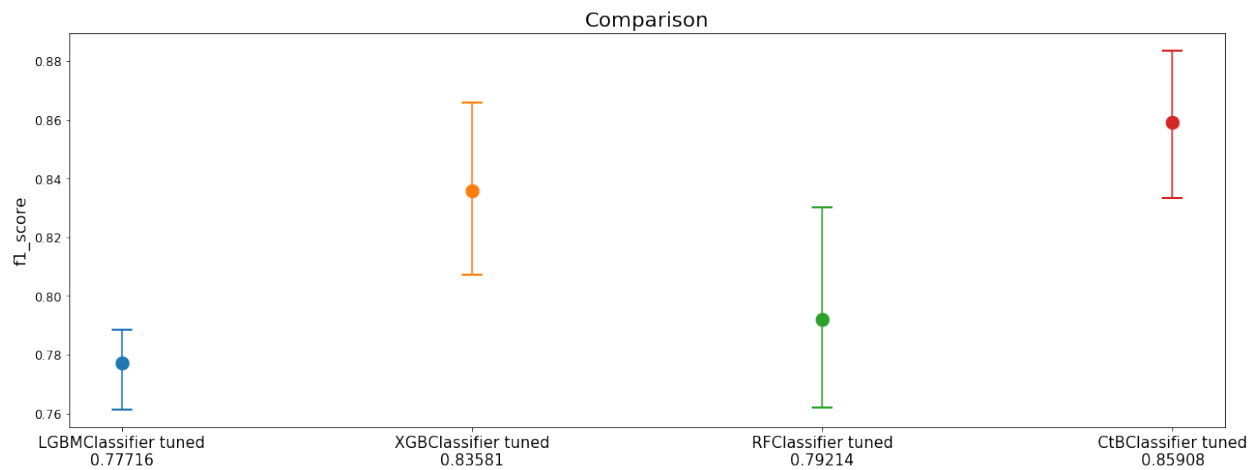
```
reporter.plot_all_metrics()
```



```
~~~~~
roc_auc
~~~~~
↪
```



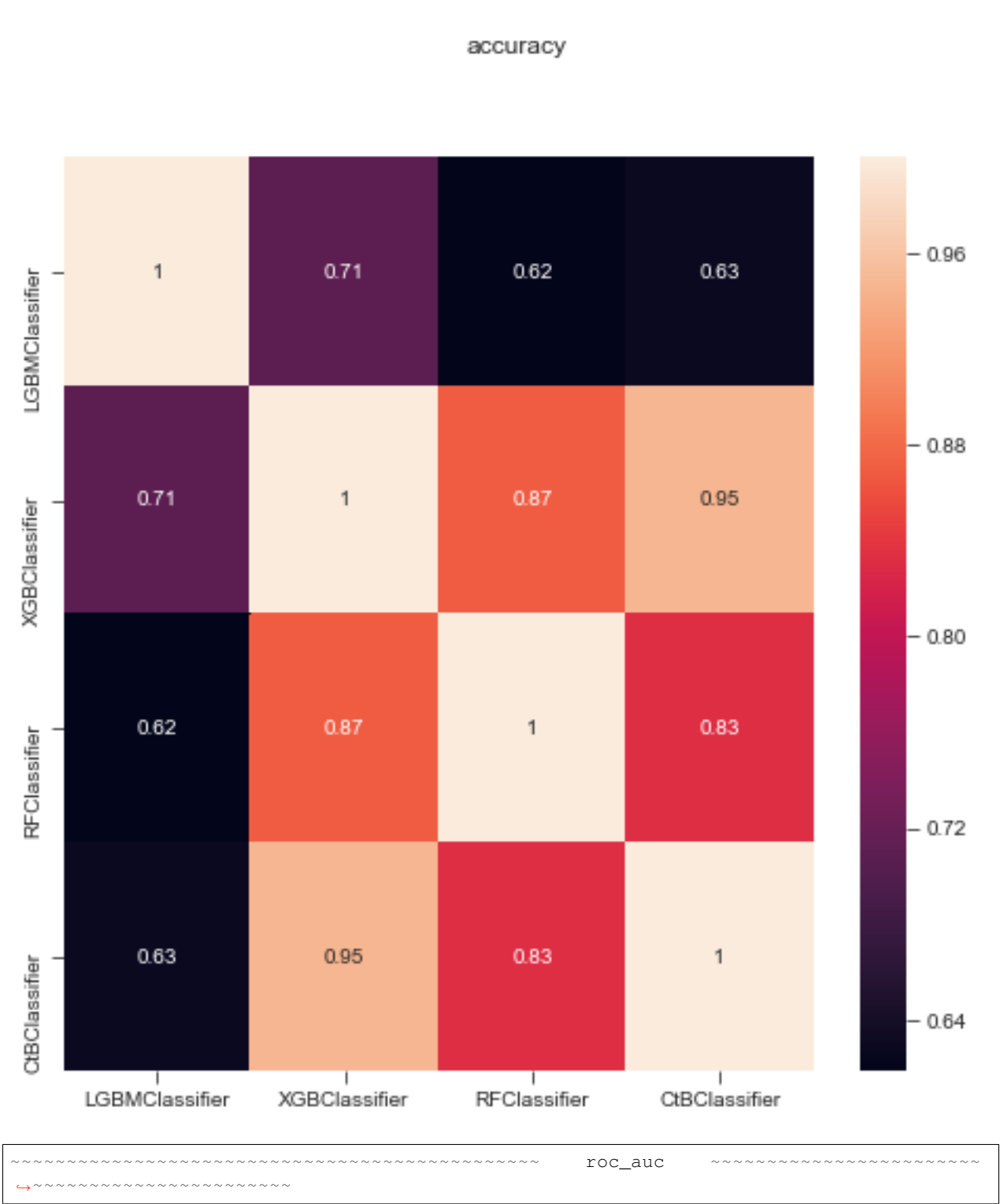
```
~~~~~
f1_score
~~~~~
↪
```

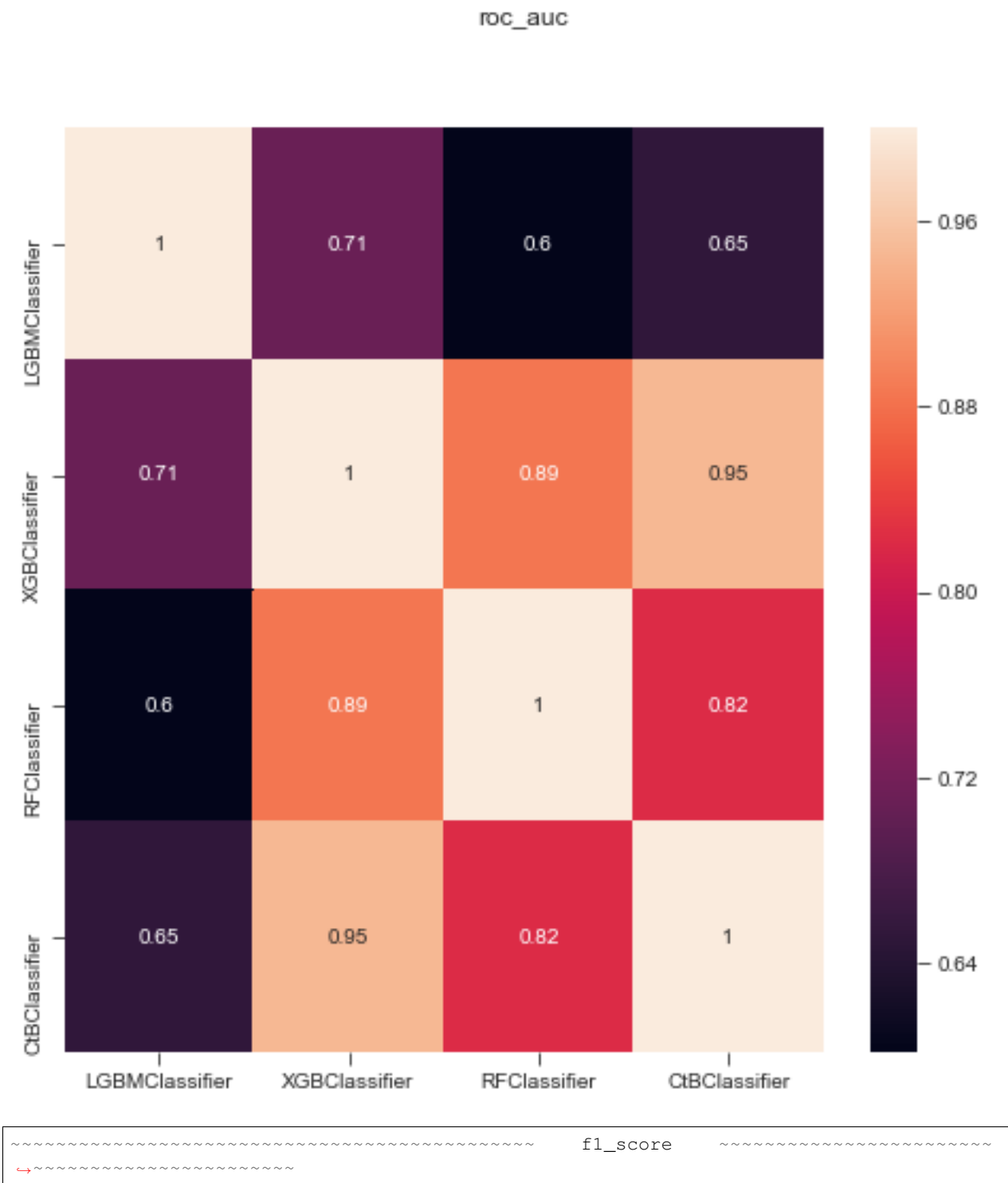


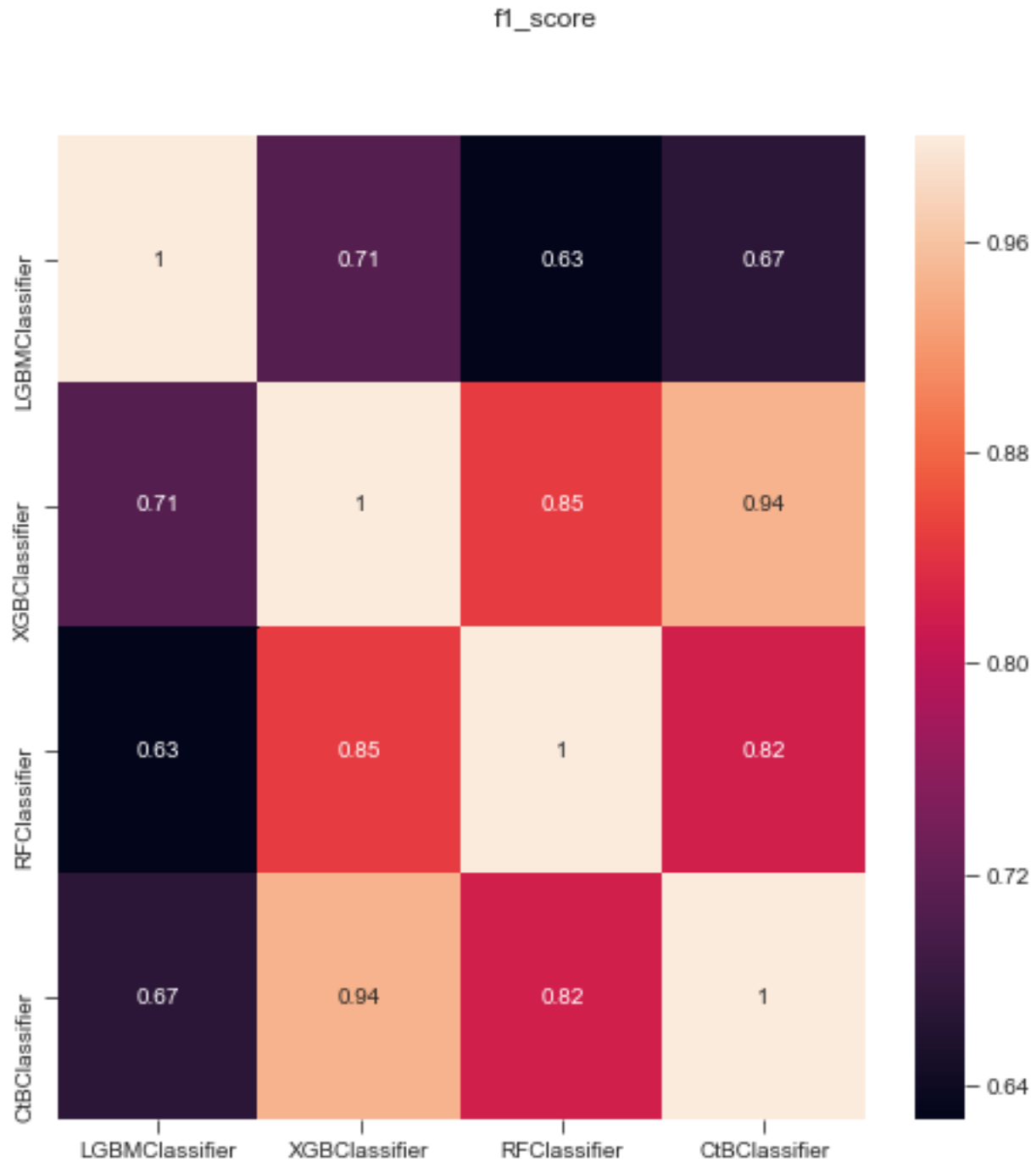
## Report heatmaps for each metric

```
reporter.plot_heatmaps()
```

```
~~~~~
accuracy
~~~~~
↪
```







That's it!

If you like it, please follow the advanced tutorial and learn all features modelgym can provide.

## 3.2 Guru example

```
from modelgym import Guru
import numpy as np
```

## Initialize Guru

```
guru = Guru()
```

## Make toy dataset

```
n = 100
np.random.seed(0)
X = np.zeros((n, 6), dtype=object)

# make not numeric feature
X[:, 0] = 'not a number'

# make categorical feature
X[:, 1] = np.random.binomial(3, 0.6, size=n)

# make sparse feature
X[:, 2] = np.random.binomial(1, 0.05, size=n) * np.random.normal(size=n)

# make correlated features
X[:, 3] = np.random.normal(size=n)
X[:, 4] = X[:, 3] * 50 - 100

# make independent feature
X[:, 5] = np.random.normal(size=n)

# make disbalanced classes
y = np.random.binomial(3, 0.9, size=n)
```

## 3.2.1 Main features

### Looking for categorical features

```
guru.check_categorical(X)
```

Some features are supposed to be categorical. Make sure that **all** categorical features **are** **in** `cat_cols`.  
 Following features are **not** numeric: [0]  
 Following features are **not** variable: [1]

```
defaultdict(list, {'not numeric': [0], 'not variable': [1]})
```

### Looking for sparse features

```
guru.check_sparse(X)
```

Consider use hashing trick **for** your sparse features, **if** you haven't already.  
 Following features are supposed to be sparse: [2]

```
[2]
```

### Looking for correlated features

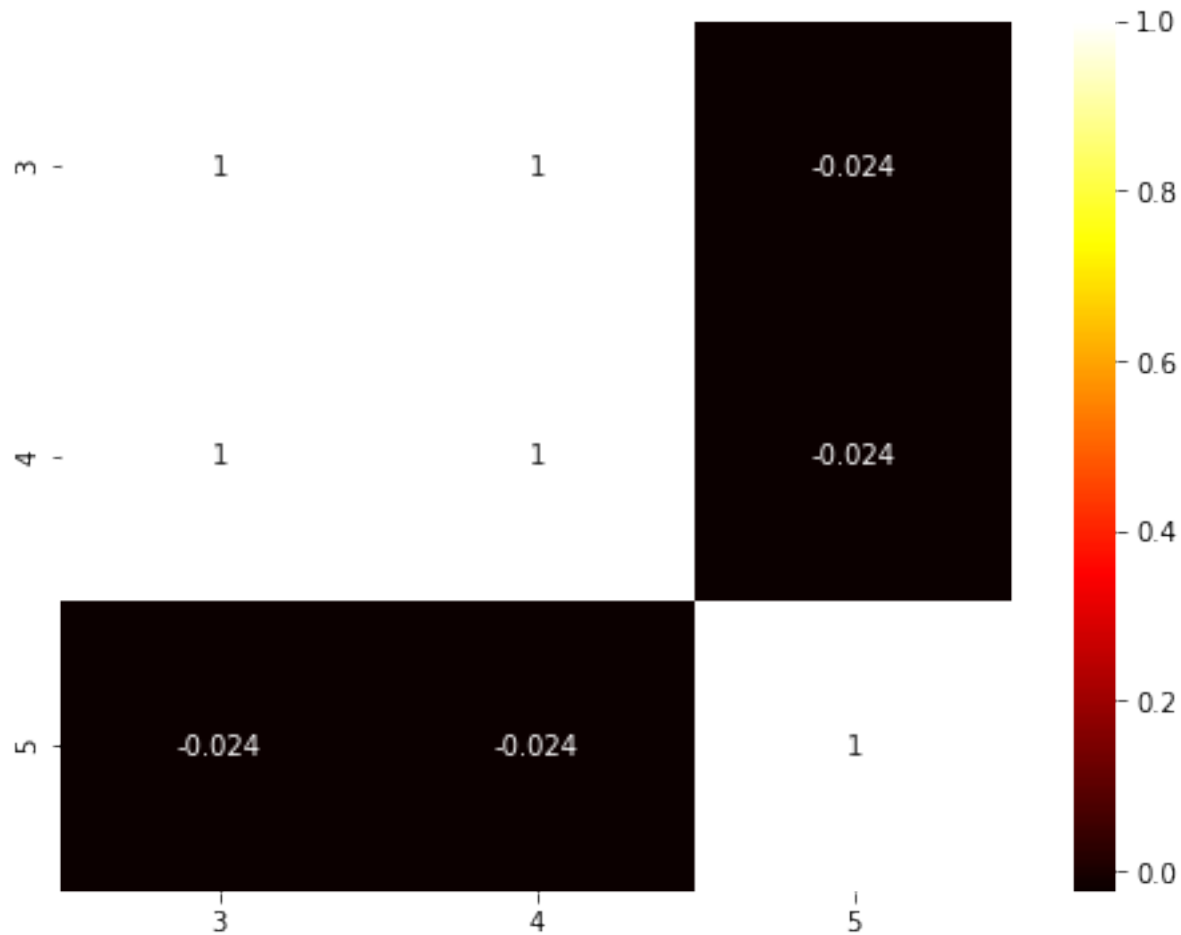
```
guru.check_correlation(X, [3, 4, 5])
```

There are several correlated features. Consider dimension reduction, **for** example you can use PCA. Following pairs of features are supposed to be correlated: [(3, 4)]

```
[(3, 4)]
```

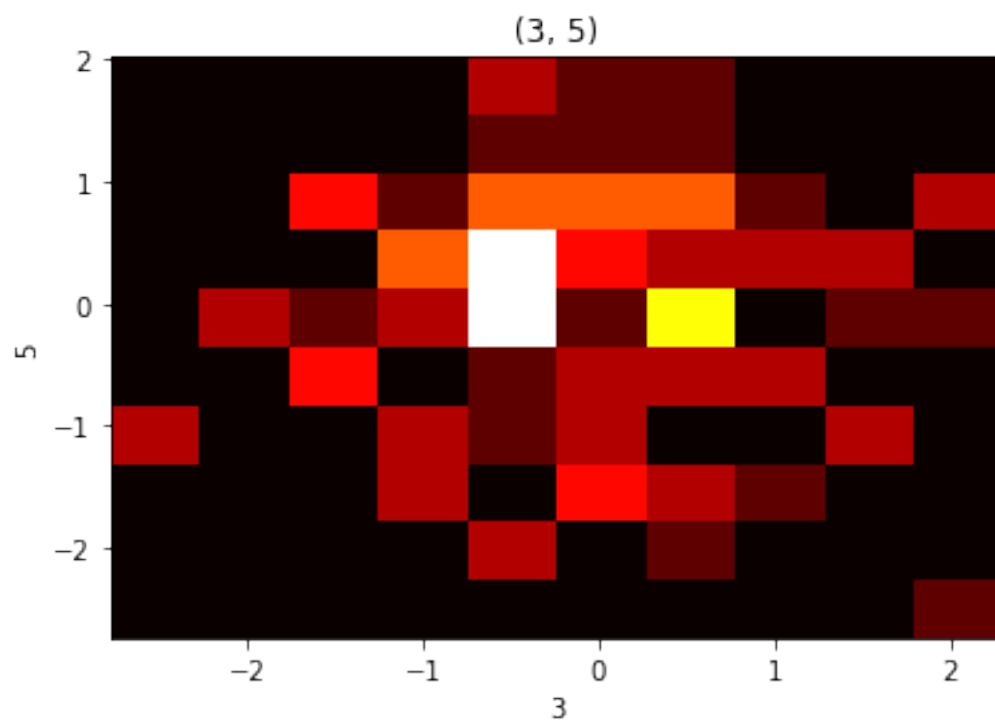
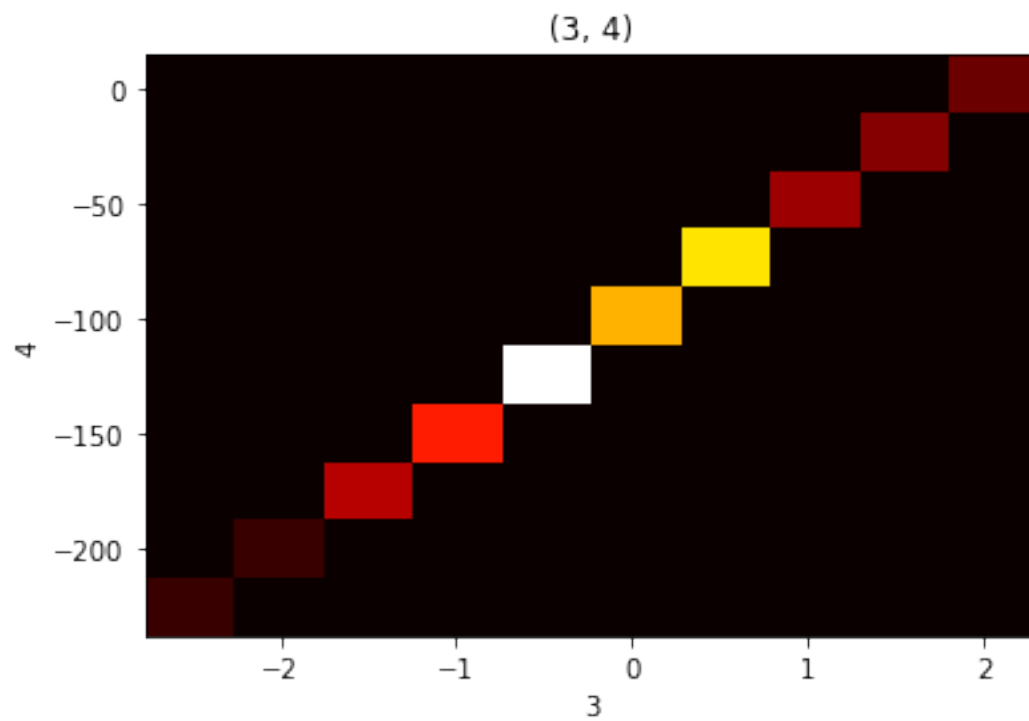
Drawing correlation heatmap for features

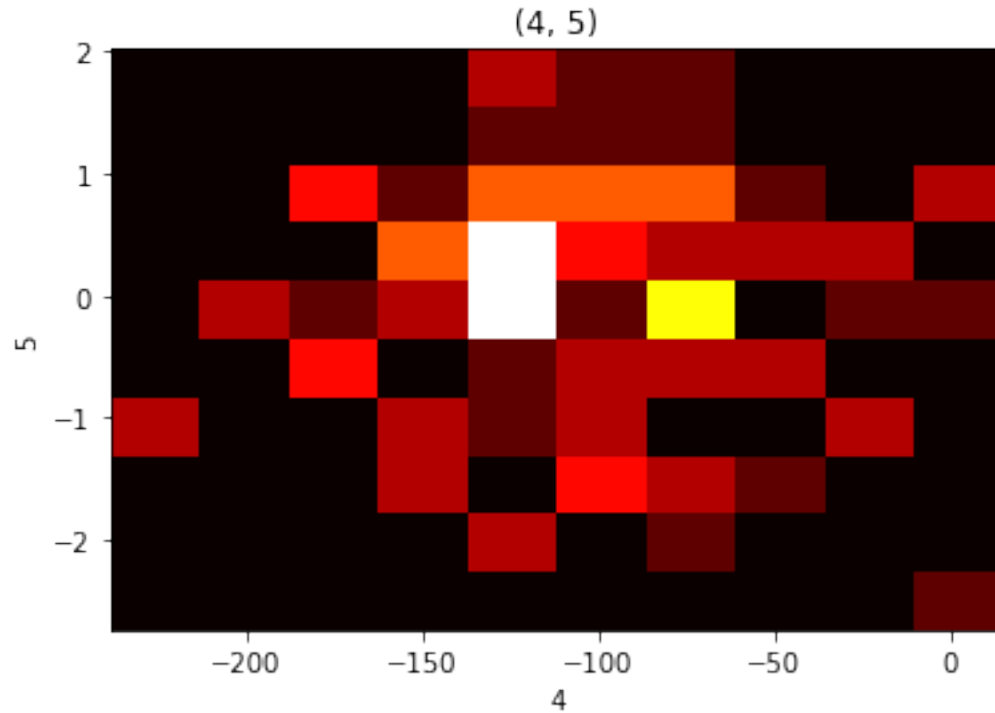
```
guru.draw_correlation_heatmap(X, [3, 4, 5], figsize=(8, 6))
```



Drawing 2d histograms for features

```
guru.draw_2dhist(X, [3, 4, 5])
```





Looking for disbalanced classes

```
guru.check_class_disbalance(y)
```

There **is class disbalance**. Probably, you can solve it by data augmentation.  
Following classes are too common: [3]  
Following classes are too rare: [1, 0]

```
defaultdict(list, {'too common': [3], 'too rare': [1, 0]})
```

### 3.2.2 dtype with fields

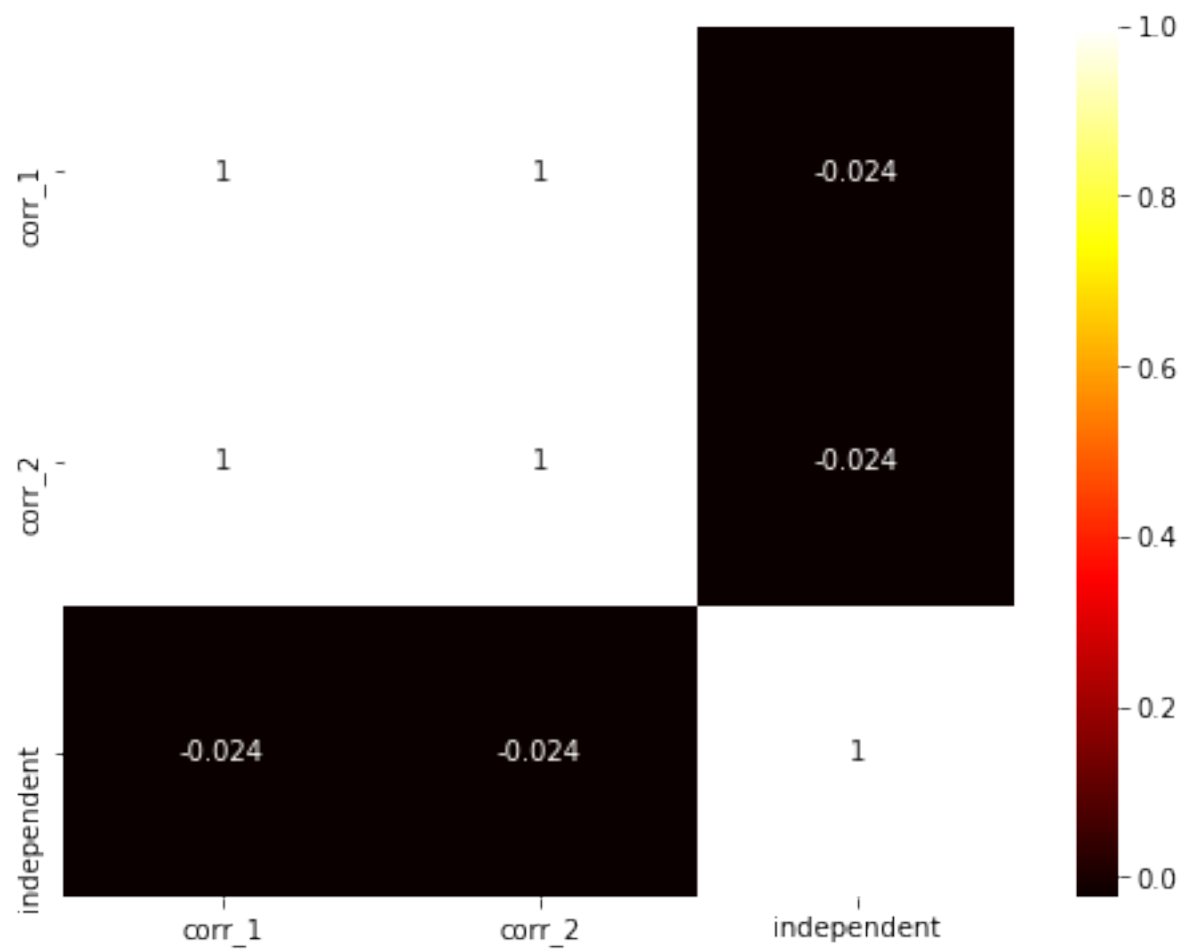
You can also use array with dtype with fields

Let's make another representation of the same data

```
named_X = np.zeros((n,), dtype=[('str', 'U25'),
                                ('categorical', 'int'),
                                ('sparse', float),
                                ('corr_1', float),
                                ('corr_2', float),
                                ('independent', float)])
for i, name in enumerate(named_X.dtype.names):
    named_X[name] = X[:, i]
```

Now we can draw heatmap like this

```
guru.draw_correlation_heatmap(named_X, ['corr_1', 'corr_2', 'independent'],
    figsize=(8, 6))
```



## 4.1 Guru

```
class modelgym.guru.Guru (print_hints=True, sample_size=None, category_qoute=0.2,  
                        sparse_qoute=0.8, class_disbalance_qoute=0.5,  
                        pvalue_boundary=0.05)
```

This class analyze data trying to find some issues.

### Parameters

- **sample\_size** (*int*) – number of objects to be used for category and sparsity diagnostic. If None, whole data will be used.
- **category\_qoute** ( $0 < \text{float} < 1$ ) – max number of distinct feature values in sample to assume this feature categorial
- **sparse\_qoute** ( $0 < \text{float} < 1$ ) – zeros portion in sample required to assume this feature sparse
- **class\_disbalance\_qoute** ( $0 < \text{float} < 1$ ) – class portion should be distant from the mean to assume this class disbalanced

**check\_categorical** (*X*)

Find category features in X

**Parameters** **X** (*array-like with shape (n\_objects, n\_features)*) – features from your dataset

### Returns

dict of shape:

```
{
    'not numeric': list of feature indexes,
    'not variable': list of feature indexes
}
```

**check\_class\_disbalance** (*y*)

Find disbalanced classes in *y*. You should use this method only if you are solving classification task

**Parameters** *y* (*array-like with shape (n\_objects,)*) – target classes in your dataset

**Returns**

dict of shape:

```
{
    'too common': list of classes,
    'too rare': list of classes
}
```

**check\_correlation** (*X, feature\_indexes=None*)

Find correlated features among features with specified indexes from *X*

**Parameters**

- **X** (*array-like with shape (n\_objects x n\_features)*) – features from your dataset
- **feature\_indexes** – list of features which should be checked for correlation. If *None* all features will be checked

**Returns** list of pairs of features which are supposed to be correlated

**check\_everything** (*data*)

Full data check. Find category features, sparse features, correlated features and disbalanced classes.

**Parameters** *data* (*XYCDataSet-like*) – your dataset

**Returns**

(categoricals, sparse, disbalanced, correlated)

- **categoricals**: indexes of features which are supposed to be categorical
- **sparse**: indexes of features which are supposed to be sparse
- **disbalanced**: disbalanced classes
- **correlated**: indexes of features which are supposed to be correlated

For more detailes see methods:

- `check_categoricals`
- `check_sparse`
- `check_class_disbalance`
- `check_correlation`

**check\_sparse** (*X*)

Find sparse features in *X*

**Parameters** *X* (*array-like with shape (n\_objects, n\_features)*) – features from your dataset

**Returns** list of features which are supposed to be sparse

**draw\_2dhist** (*X, feature\_indexes=None, figsize=(6, 4), \*\*hist\_kwargs*)

Draw 2dhist for each pair of features with specified indexes

**Parameters**

- **X** (*array-like with shape (n\_objects x n\_features)*) – features from your dataset
- **feature\_indexes** (*list of int or str*) – features which should be checked for correlation. If None all features will be checked. If it is list of str X should be a np.ndarray and X.dtype should contain fields
- **figsize** (*tuple of int*) – Size of figure with hist2d

**draw\_correlation\_heatmap** (*X, feature\_indexes=None, figsize=(15, 10), \*\*heatmap\_kwargs*)

Draw correlation heatmap between features with specified indexes from X

#### Parameters

- **X** (*array-like with shape (n\_objects x n\_features)*) – features from your dataset
- **feature\_indexes** (*list of int or str*) – features which should be checked for correlation. If None all features will be checked. If it is list of str X should be a np.ndarray and X.dtype should contain fields
- **figsize** (*tuple of int*) – Size of figure with heatmap

## 4.2 Models

In order to use our Trainer you need the wrapper on your model. You can find the required [Model interface](#) below.

We implement wrappers for several models:

- [XGBoost](#)
- [LightGBM](#)
- [RandomForestClassifier](#)
- [Catboost](#)

Also, we implement an [Ensemble Model](#).

### 4.2.1 Model interface

**class** modelgym.models.model.**Model** (*params=None*)

Model is a base class for a specific ML algorithm implementation factory, i.e. it defines algorithm-specific hyperparameter space and generic methods for model training & inference

**Parameters** **params** (*dict or None*) – parameters for model.

**fit** (*dataset, weights=None*)

#### Parameters

- **X** (*np.array, shape (n\_samples, n\_features)*) – the input data
- **y** (*np.array, shape (n\_samples, ) or (n\_samples, n\_outputs)*) – the target data
- **weights** (*np.array, shape (n\_samples, ) or (n\_samples, n\_outputs) or None*) – weights of the data

**Returns** self

**static** **get\_default\_parameter\_space** ()

**Returns** default parameter space

**Return type** dict from parameter name to hyperopt distribution

**static** `get_learning_task()`

**Returns** task

**Return type** modelgym.models.LearningTask

**is\_possible\_predict\_proba()**

**Returns** bool, whether model can predict proba

**static** `load_from_snapshot(filename)`

:snapshot serializable internal model state loads from serializable internal model state snapshot.

**predict** (*dataset*)

**Parameters** **dataset** (*modelgym.utils.XYCDataset*) – the input data, dataset.y may be None

**Returns** predictions

**Return type** np.array, shape (n\_samples, )

**predict\_proba** (*X*)

**Parameters** **dataset** (*np.array, shape (n\_samples, n\_features)*) – the input data

**Returns** predicted probabilities

**Return type** np.array, shape (n\_samples, n\_classes)

**save\_snapshot** (*filename*)

**Returns** serializable internal model state snapshot.

## 4.2.2 XGBoost

**class** modelgym.models.xgboost\_model.XGBClassifier (*params=None*)

Bases: *modelgym.models.model.Model*

**Parameters** **params** (*dict*) – parameters for model.

**fit** (*dataset, weights=None*)

**Parameters**

- **X** (*np.array, shape (n\_samples, n\_features)*) – the input data
- **y** (*np.array, shape (n\_samples, ) or (n\_samples, n\_outputs)*) – the target data
- **weights** (*np.array, shape (n\_samples, ) or (n\_samples, n\_outputs) or None*) – weights of the data

**Returns** self

**static** `get_default_parameter_space()`

**Returns** dict of DistributionWrappers

**static** `get_learning_task()`

**is\_possible\_predict\_proba()**

**Returns** bool, whether model can predict proba

**static load\_from\_snapshot** (*filename*)

:snapshot serializable internal model state loads from serializable internal model state snapshot.

**predict** (*dataset*)

**Parameters** **X** (*np.array, shape (n\_samples, n\_features)*) – the input data

**Returns** np.array, shape (n\_samples, ) or (n\_samples, n\_outputs)

**predict\_proba** (*dataset*)

**Parameters** **X** (*np.array, shape (n\_samples, n\_features)*) – the input data

**Returns** np.array, shape (n\_samples, n\_classes)

**save\_snapshot** (*filename*)

**Returns** serializable internal model state snapshot.

**class** modelgym.models.xgboost\_model.XGBRegressor (*params=None*)

Bases: [modelgym.models.model.Model](#)

**Parameters**

- **params** (*dict or None*) – parameters for model. If None default params are fetched.
- **learning\_task** (*str*) – set type of task(classification, regression, ...)

**fit** (*dataset, weights=None*)

**Parameters**

- **X** (*np.array, shape (n\_samples, n\_features)*) – the input data
- **y** (*np.array, shape (n\_samples, ) or (n\_samples, n\_outputs)*) – the target data
- **weights** (*np.array, shape (n\_samples, ) or (n\_samples, n\_outputs) or None*) – weights of the data

**Returns** self

**static get\_default\_parameter\_space** ()

**Returns** dict of DistributionWrappers

**static get\_learning\_task** ()

**is\_possible\_predict\_proba** ()

**Returns** bool, whether model can predict proba

**static load\_from\_snapshot** (*filename*)

:snapshot serializable internal model state loads from serializable internal model state snapshot.

**predict** (*dataset*)

**Parameters** **X** (*np.array, shape (n\_samples, n\_features)*) – the input data

**Returns** np.array, shape (n\_samples, ) or (n\_samples, n\_outputs)

**predict\_proba** (*dataset*)

**Parameters** **X** (*np.array, shape (n\_samples, n\_features)*) – the input data

**Returns** np.array, shape (n\_samples, n\_classes)

**save\_snapshot** (*filename*)

**Returns** serializable internal model state snapshot.

### 4.2.3 LightGBM

**class** modelgym.models.lightgbm\_model.LGBMClassifier(*params=None*)

Bases: *modelgym.models.model.Model*

#### Parameters

- **params** (*dict* or *None*) – parameters for model. If *None* default params are fetched.
- **learning\_task** (*str*) – set type of task(classification, regression, ...)

**fit** (*dataset*, *weights=None*)

#### Parameters

- **X** (*np.array*, *shape* (*n\_samples*, *n\_features*)) – the input data
- **y** (*np.array*, *shape* (*n\_samples*, ) or (*n\_samples*, *n\_outputs*)) – the target data
- **weights** (*np.array*, *shape* (*n\_samples*, ) or (*n\_samples*, *n\_outputs*) or *None*) – weights of the data

**Returns** *self*

**static** **get\_default\_parameter\_space** ()

**Returns** dict of DistributionWrappers

**static** **get\_learning\_task** ()

**is\_possible\_predict\_proba** ()

**Returns** bool, whether model can predict proba

**static** **load\_from\_snapshot** (*filename*)

:snapshot serializable internal model state loads from serializable internal model state snapshot.

**predict** (*dataset*)

**Parameters** **X** (*np.array*, *shape* (*n\_samples*, *n\_features*)) – the input data

**Returns** *np.array*, *shape* (*n\_samples*, ) or (*n\_samples*, *n\_outputs*)

**predict\_proba** (*dataset*)

**Parameters** **X** (*np.array*, *shape* (*n\_samples*, *n\_features*)) – the input data

**Returns** *np.array*, *shape* (*n\_samples*, *n\_classes*)

**save\_snapshot** (*filename*)

**Returns** serializable internal model state snapshot.

**class** modelgym.models.lightgbm\_model.LGBMRegressor(*params=None*)

Bases: *modelgym.models.model.Model*

#### Parameters

- **params** (*dict* or *None*) – parameters for model. If *None* default params are fetched.
- **learning\_task** (*str*) – set type of task(classification, regression, ...)

**fit** (*dataset*, *weights=None*)

#### Parameters

- **X**(*np.array, shape (n\_samples, n\_features)*) – the input data
- **y**(*np.array, shape (n\_samples, ) or (n\_samples, n\_outputs)*) – the target data
- **weights** (*np.array, shape (n\_samples, ) or (n\_samples, n\_outputs) or None*) – weights of the data

Returns self

**static get\_default\_parameter\_space()**

Returns dict of DistributionWrappers

**static get\_learning\_task()**

**is\_possible\_predict\_proba()**

Returns bool, whether model can predict proba

**static load\_from\_snapshot(filename)**

:snapshot serializable internal model state loads from serializable internal model state snapshot.

**predict(dataset)**

Parameters **X**(*np.array, shape (n\_samples, n\_features)*) – the input data

Returns np.array, shape (n\_samples, ) or (n\_samples, n\_outputs)

**predict\_proba(dataset)**

Parameters **X**(*np.array, shape (n\_samples, n\_features)*) – the input data

Returns np.array, shape (n\_samples, n\_classes)

**save\_snapshot(filename)**

Return: serializable internal model state snapshot.

## 4.2.4 RandomForestClassifier

**class** modelgym.models.rf\_model.**RFClassifier**(*params=None*)

Bases: *modelgym.models.model.Model*

Parameters

- **params**(*dict or None*) – parameters for model. If None default params are fetched.
- **learning\_task**(*str*) – set type of task(classification, regression, ...)

**fit(dataset, weights=None)**

Parameters

- **X**(*np.array, shape (n\_samples, n\_features)*) – the input data
- **y**(*np.array, shape (n\_samples, ) or (n\_samples, n\_outputs)*) – the target data
- **weights** (*np.array, shape (n\_samples, ) or (n\_samples, n\_outputs) or None*) – weights of the data

Returns self

**static get\_default\_parameter\_space()**

Returns dict of DistributionWrappers

```

static get_learning_task()
is_possible_predict_proba()
    Returns bool, whether model can predict proba
static load_from_snapshot(filename)
    :snapshot serializable internal model state loads from serializable internal model state snapshot.
predict(dataset)
    Parameters X (np.array, shape (n_samples, n_features)) – the input data
    Returns np.array, shape (n_samples, ) or (n_samples, n_outputs)
predict_proba(dataset)
    Parameters X (np.array, shape (n_samples, n_features)) – the input data
    Returns np.array, shape (n_samples, n_classes)
save_snapshot(filename)
    Returns serializable internal model state snapshot.

```

## 4.2.5 Catboost

```

class modelgym.models.catboost_model.CtBClassifier(params=None)
    Bases: modelgym.models.model.Model
    Wrapper for CatBoostClassifier

    Parameters params (dict) – parameters for model.

    fit(dataset, weights=None, eval_dataset=None, **kwargs)

        Parameters

        • dataset (XYCDataset) – train

        • y (np.array, shape (n_samples, ) or (n_samples, n_outputs)) – the target data

        • weights (np.array, shape (n_samples, ) or (n_samples, n_outputs) or None) – weights of the data

        • eval_dataset – same as dataset

        • kwargs – CatBoost.Pool kwargs if eval_dataset is None or {'train': train_kwargs, 'eval': eval_kwargs} otherwise

        Returns self

    static get_default_parameter_space()
        Returns dict of DistributionWrappers

    static get_learning_task()

    is_possible_predict_proba()
        Returns bool, whether model can predict proba

    static load_from_snapshot(filename)
        :snapshot serializable internal model state loads from serializable internal model state snapshot.

    predict(dataset, **kwargs)

```

### Parameters

- **X**(*np.array, shape (n\_samples, n\_features)*) – the input data
- **kwargs** – CatBoost.Pool kwargs

**Returns** *np.array, shape (n\_samples, ) or (n\_samples, n\_outputs)*

**predict\_proba** (*dataset, \*\*kwargs*)

### Parameters

- **X**(*np.array, shape (n\_samples, n\_features)*) – the input data
- **kwargs** – CatBoost.Pool kwargs

**Returns** *np.array, shape (n\_samples, n\_classes)*

**save\_snapshot** (*filename*)

**Returns** serializable internal model state snapshot.

**class** *modelgym.models.catboost\_model.CtBRegressor* (*params=None*)

Bases: *modelgym.models.model.Model*

Wrapper for *CatBoostRegressor*

### Parameters

- **params** (*dict or None*) – parameters for model. If None default params are fetched.
- **learning\_task** (*str*) – set type of task(classification, regression, ...)

**fit** (*dataset, weights=None, eval\_dataset=None, \*\*kwargs*)

### Parameters

- **dataset** (*XYCDataset*) –
- **weights** (*np.array, shape (n\_samples, ) or (n\_samples, n\_outputs) or None*) – weights of the data
- **eval\_dataset** – same as dataset
- **kwargs** – CatBoost.Pool kwargs if eval\_dataset is None or {'train': train\_kwargs, 'eval': eval\_kwargs} otherwise

**Returns** *self*

**static get\_default\_parameter\_space** ()

**Returns** dict of DistributionWrappers

**static get\_learning\_task** ()

**is\_possible\_predict\_proba** ()

**Returns** bool, whether model can predict\_proba

**static load\_from\_snapshot** (*filename*)

:snapshot serializable internal model state loads from serializable internal model state snapshot.

**predict** (*dataset, \*\*kwargs*)

### Parameters

- **X**(*np.array, shape (n\_samples, n\_features)*) – the input data
- **kwargs** – CatBoost.Pool kwargs

**Returns** `np.array`, shape `(n_samples, )` or `(n_samples, n_outputs)`

**predict\_proba** (*dataset*, *\*\*kwargs*)

**Parameters**

- **X** (`np.array`, shape `(n_samples, n_features)`) – the input data
- **kwargs** – CatBoost.Pool kwargs

**Returns** `np.array`, shape `(n_samples, n_classes)`

**save\_snapshot** (*filename*)

**Returns** serializable internal model state snapshot.

## 4.2.6 Ensemble Model

**class** `modelgym.models.ensemble_model.EnsembleClassifier` (*params=None*)

Bases: `modelgym.models.model.Model`

**Parameters** **params** (*dict*) – parameters for model.

**fit** (*dataset*, *weights=None*, *\*\*kwargs*)

**Parameters**

- **dataset** (`XYCDataset`) – train
- **y** (`np.array`, shape `(n_samples, )` or `(n_samples, n_outputs)`) – the target data
- **weights** (`np.array`, shape `(n_samples, )` or `(n_samples, n_outputs)` or `None`) – weights of the data
- **eval\_dataset** – same as dataset
- **kwargs** – CatBoost.Pool kwargs if `eval_dataset == None` or `{'train': train_kwargs, 'eval': eval_kwargs}` otherwise

**Returns** `self`

**static** `get_default_parameter_space()`

**Returns** dict of DistributionWrappers

**static** `get_learning_task()`

**static** `get_one_hot(targets, nb_classes)`

**is\_possible\_predict\_proba()**

**Returns** bool, whether model can predict proba

**static** `load_from_snapshot(filename, models)`

**Parameters** **filename** – prefix for models' files

**Returns** EnsembleClassifier

**predict** (*dataset*, *\*\*kwargs*)

**Parameters**

- **X** (`np.array`, shape `(n_samples, n_features)`) – the input data
- **kwargs** – CatBoost.Pool kwargs

**Returns** np.array, shape (n\_samples, ) or (n\_samples, n\_outputs)

**predict\_proba** (dataset, \*\*kwargs)

**Parameters**

- **X** (np.array, shape (n\_samples, n\_features)) – the input data
- **kwargs** – CatBoost.Pool kwargs

**Returns** np.array, shape (n\_samples, n\_classes)

**save\_snapshot** (filename)

**Parameters** **filename** – prefix for models' files

**Returns** serializable internal model state snapshot.

**class** modelgym.models.ensemble\_model.**EnsembleRegressor** (params=None)

Bases: [modelgym.models.model.Model](#)

**Parameters** **params** (dict) – parameters for model

**fit** (dataset, weights=None, \*\*kwargs)

**Parameters**

- **dataset** (XYCDataset) – train
- **y** (np.array, shape (n\_samples, ) or (n\_samples, n\_outputs)) – the target data
- **weights** (np.array, shape (n\_samples, ) or (n\_samples, n\_outputs) or None) – weights of the data
- **eval\_dataset** – same as dataset
- **kwargs** – CatBoost.Pool kwargs if eval\_dataset == None or {'train': train\_kwargs, 'eval': eval\_kwargs} otherwise

**Returns** self

**static** **get\_default\_parameter\_space** ()

**Returns** dict of DistributionWrappers

**static** **get\_learning\_task** ()

**is\_possible\_predict\_proba** ()

**Returns** bool, whether model can predict\_proba

**static** **load\_from\_snapshot** (filename, models)

**Parameters** **filename** – prefix for models' files

**Returns** EnsembleClassifier

**predict** (dataset, \*\*kwargs)

**Parameters**

- **X** (np.array, shape (n\_samples, n\_features)) – the input data
- **kwargs** – CatBoost.Pool kwargs

**Returns** np.array, shape (n\_samples, ) or (n\_samples, n\_outputs)

**predict\_proba** (dataset, \*\*kwargs)

Regressor can't predict\_proba

**save\_snapshot** (*filename*)

**Parameters** **filename** – prefix for models' files

**Returns** serializable internal model state snapshot.

## 4.3 Metrics

In our library you should use metrics inherited from *Base Class*. We have already made some wrappers around *Sklearn Metrics*.

### 4.3.1 Base Class

```
class modelgym.metrics.Metric (scoring_function, requires_proba=False, is_min_optimal=False,  
                                name='default_name')
```

Metric class is a wrapper around sklearn.metrics class, with additional information: when optimizing this metric, should we minimize it (like log\_loss) or maximize (like accuracy), and whether it's calculation requires computed probabilities (like roc\_auc).

Of course, not only sklearn.metrics could be wrapped into this class

**Parameters**

- **scoring\_function** (*types.FunctionType*) – wrapped scoring function
- **requires\_proba** (*bool*) – whether calculation of metric requires computed probabilities
- **is\_min\_optimal** (*bool*) – is the less the better
- **name** (*str*) – name of metric

### 4.3.2 Sklearn Metrics

```
class modelgym.metrics.Accuracy (name='accuracy')  
    Bases: modelgym.metrics.Metric
```

```
class modelgym.metrics.F1 (name='f1_score')  
    Bases: modelgym.metrics.Metric
```

```
class modelgym.metrics.Logloss (name='logloss')  
    Bases: modelgym.metrics.Metric
```

```
class modelgym.metrics.Mse (name='mse')  
    Bases: modelgym.metrics.Metric
```

```
class modelgym.metrics.Precision (name='precision')  
    Bases: modelgym.metrics.Metric
```

```
class modelgym.metrics.Recall (name='recall')  
    Bases: modelgym.metrics.Metric
```

```
class modelgym.metrics.RocAuc (name='roc_auc')  
    Bases: modelgym.metrics.Metric
```



```
{
    name (str): {
        "result": {
            "loss": float,
            "loss_variance": float,
            "status": "ok",
            "metric_cv_results": list,
            "params": dict
        },
        "model_space": modelgym.utils.ModelSpace
    }
}
```

name is a name of corresponding model\_space,

metric\_cv\_results contains dict's from metric names to calculated metric values for each fold in cv\_fold,

params is optimal parameters of corresponding model

model\_space is corresponding model\_space.

**class** modelgym.trainers.hyperopt\_trainer.**RandomTrainer** (*model\_spaces*,  
*tracker=None*)  
 Bases: *modelgym.trainers.hyperopt\_trainer.HyperoptTrainer*

RandomTrainer is a HyperoptTrainer using Random search

**class** modelgym.trainers.hyperopt\_trainer.**TpeTrainer** (*model\_spaces*, *tracker=None*)  
 Bases: *modelgym.trainers.hyperopt\_trainer.HyperoptTrainer*

TpeTrainer is a HyperoptTrainer using Tree-structured Parzen Estimator

## 4.4.2 Skopt trainers

**class** modelgym.trainers.skopt\_trainer.**GPTrainer** (*model\_spaces*, *tracker=None*)  
 Bases: *modelgym.trainers.skopt\_trainer.SkoptTrainer*

GPTrainer is a SkoptTrainer, using Bayesian optimization using Gaussian Processes.

**class** modelgym.trainers.skopt\_trainer.**RFTrainer** (*model\_spaces*, *tracker=None*)  
 Bases: *modelgym.trainers.skopt\_trainer.SkoptTrainer*

RFTrainer is a SkoptTrainer, using Sequential optimisation using decision trees

**class** modelgym.trainers.skopt\_trainer.**SkoptTrainer** (*model\_spaces*, *optimizer*,  
*tracker=None*)  
 Bases: *modelgym.trainers.trainer.Trainer*

SkoptTrainer is a class for models hyperparameter optimization, based on skopt library

### Parameters

- **model\_spaces** (*list of modelgym.models.Model or modelgym.utils.ModelSpaces*) – list of model spaces (model classes and parameter spaces to look in). If some list item is Model, it is converted in ModelSpace with default space and name equal to model class `__name__`
- **(function, e.g forest\_minimize or gp\_minimize (optimizer))** –
- **tracker** (*modelgym.trackers.Tracker, optional*) – ignored

**Raises** ValueError if there are several model\_spaces with similar names

**crossval\_optimize\_params** (*opt\_metric, dataset, cv=3, opt\_evals=50, metrics=None, verbose=False, \*\*kwargs*)

Find optimal hyperparameters for all models

#### Parameters

- **opt\_metric** (*modelgym.metrics.Metric*) – metric to optimize
- **dataset** (*modelgym.utils.XYCDataset or None*) – dataset
- **cv** (*int or list of tuples of (XYCDataset, XYCDataset)*) – if int, then number of cross-validation folds or cross-validation folds themselves otherwise.
- **opt\_evals** (*int*) – number of cross-validation evaluations
- **metrics** (*list of modelgym.metrics.Metric, optional*) – additional metrics to evaluate
- **verbose** (*bool*) – Enable verbose output.
- **\*\*kwargs** – ignored

---

**Note:** if cv is int, than dataset is split into cv parts for cross validation. Otherwise, cv folds are used.

---

**get\_best\_results** ()

When training is complete, return best parameters (and additional information) for each model space

#### Returns

dict of shape:

```
{
    name (str): {
        "result": {
            "loss": float,
            "metric_cv_results": list,
            "params": dict
        },
        "model_space": modelgym.utils.ModelSpace
    }
}
```

name is a name of corresponding model\_space,

metric\_cv\_results contains dict's from metric names to calculated metric values for each fold in cv\_fold,

params is optimal parameters of corresponding model,

model\_space is corresponding model\_space.

## 4.5 Trackers

**class** modelgym.trackers.tracker.**LocalTracker** (*save\_dir, suffix=None*)

**static** **check\_exists** (*directory*)

**load\_state** ()

```
    save_state (state)  
class modelgym.trackers.tracker.TrackerMongo (host, port, db, config_key=None,  
                                              model_name=None)  
  
    load_state (as_list=False)  
    save_state (**kwargs)
```

## 4.6 Compare models

```
modelgym.utils.util.compare_models_different (first_model, second_model, data, al-  
                                              pha=0.05, metric='ROC_AUC')  
    Hypothesis: two models are the same
```

### m

- `modelgym.guru`, [21](#)
- `modelgym.metrics`, [32](#)
- `modelgym.models.catboost_model`, [28](#)
- `modelgym.models.ensemble_model`, [30](#)
- `modelgym.models.lightgbm_model`, [26](#)
- `modelgym.models.model`, [23](#)
- `modelgym.models.rf_model`, [27](#)
- `modelgym.models.xgboost_model`, [24](#)
- `modelgym.trackers.tracker`, [35](#)
- `modelgym.trainers.hyperopt_trainer`, [33](#)
- `modelgym.trainers.skopt_trainer`, [34](#)
- `modelgym.utils.util`, [36](#)



## A

Accuracy (class in `modelgym.metrics`), 32

## C

`check_categorical()` (`modelgym.guru.Guru` method), 21

`check_class_disbalance()` (`modelgym.guru.Guru` method), 21

`check_correlation()` (`modelgym.guru.Guru` method), 22

`check_everything()` (`modelgym.guru.Guru` method), 22

`check_exists()` (`modelgym.trackers.tracker.LocalTracker` static method), 35

`check_sparse()` (`modelgym.guru.Guru` method), 22

`compare_models_different()` (in module `modelgym.utils.util`), 36

`crossval_optimize_params()` (`modelgym.trainers.hyperopt_trainer.HyperoptTrainer` method), 33

`crossval_optimize_params()` (`modelgym.trainers.skopt_trainer.SkoptTrainer` method), 35

`CtBClassifier` (class in `modelgym.models.catboost_model`), 28

`CtBRegressor` (class in `modelgym.models.catboost_model`), 29

## D

`draw_2dhist()` (`modelgym.guru.Guru` method), 22

`draw_correlation_heatmap()` (`modelgym.guru.Guru` method), 23

## E

`EnsembleClassifier` (class in `modelgym.models.ensemble_model`), 30

`EnsembleRegressor` (class in `modelgym.models.ensemble_model`), 31

## F

F1 (class in `modelgym.metrics`), 32

`fit()` (`modelgym.models.catboost_model.CtBClassifier` method), 28

`fit()` (`modelgym.models.catboost_model.CtBRegressor` method), 29

`fit()` (`modelgym.models.ensemble_model.EnsembleClassifier` method), 30

`fit()` (`modelgym.models.ensemble_model.EnsembleRegressor` method), 31

`fit()` (`modelgym.models.lightgbm_model.LGBMClassifier` method), 26

`fit()` (`modelgym.models.lightgbm_model.LGBMRegressor` method), 26

`fit()` (`modelgym.models.model.Model` method), 23

`fit()` (`modelgym.models.rf_model.RFClassifier` method), 27

`fit()` (`modelgym.models.xgboost_model.XGBClassifier` method), 24

`fit()` (`modelgym.models.xgboost_model.XGBRegressor` method), 25

## G

`get_best_results()` (`modelgym.trainers.hyperopt_trainer.HyperoptTrainer` method), 33

`get_best_results()` (`modelgym.trainers.skopt_trainer.SkoptTrainer` method), 35

`get_default_parameter_space()` (`modelgym.models.catboost_model.CtBClassifier` static method), 28

`get_default_parameter_space()` (`modelgym.models.catboost_model.CtBRegressor` static method), 29

`get_default_parameter_space()` (`modelgym.models.ensemble_model.EnsembleClassifier` static method), 30

`get_default_parameter_space()` (`modelgym.models.ensemble_model.EnsembleRegressor` static method), 31

get\_default\_parameter\_space() (modelgym.models.lightgbm\_model.LGBMClassifier static method), 26

get\_default\_parameter\_space() (modelgym.models.lightgbm\_model.LGBMRegressor static method), 27

get\_default\_parameter\_space() (modelgym.models.model.Model static method), 23

get\_default\_parameter\_space() (modelgym.models.rf\_model.RFClassifier static method), 27

get\_default\_parameter\_space() (modelgym.models.xgboost\_model.XGBClassifier static method), 24

get\_default\_parameter\_space() (modelgym.models.xgboost\_model.XGBRegressor static method), 25

get\_learning\_task() (modelgym.models.catboost\_model.CtBClassifier static method), 28

get\_learning\_task() (modelgym.models.catboost\_model.CtBRegressor static method), 29

get\_learning\_task() (modelgym.models.ensemble\_model.EnsembleClassifier static method), 30

get\_learning\_task() (modelgym.models.ensemble\_model.EnsembleRegressor static method), 31

get\_learning\_task() (modelgym.models.lightgbm\_model.LGBMClassifier static method), 26

get\_learning\_task() (modelgym.models.lightgbm\_model.LGBMRegressor static method), 27

get\_learning\_task() (modelgym.models.model.Model static method), 24

get\_learning\_task() (modelgym.models.rf\_model.RFClassifier static method), 27

get\_learning\_task() (modelgym.models.xgboost\_model.XGBClassifier static method), 24

get\_learning\_task() (modelgym.models.xgboost\_model.XGBRegressor static method), 25

get\_one\_hot() (modelgym.models.ensemble\_model.EnsembleClassifier static method), 30

GPTrainer (class in modelgym.trainers.skopt\_trainer), 34

Guru (class in modelgym.guru), 21

## H

HyperoptTrainer (class in modelgym.trainers.hyperopt\_trainer), 33

## I

is\_possible\_predict\_proba() (modelgym.models.catboost\_model.CtBClassifier method), 28

is\_possible\_predict\_proba() (modelgym.models.catboost\_model.CtBRegressor method), 29

is\_possible\_predict\_proba() (modelgym.models.ensemble\_model.EnsembleClassifier method), 30

is\_possible\_predict\_proba() (modelgym.models.ensemble\_model.EnsembleRegressor method), 31

is\_possible\_predict\_proba() (modelgym.models.lightgbm\_model.LGBMClassifier method), 26

is\_possible\_predict\_proba() (modelgym.models.lightgbm\_model.LGBMRegressor method), 27

is\_possible\_predict\_proba() (modelgym.models.model.Model method), 24

is\_possible\_predict\_proba() (modelgym.models.rf\_model.RFClassifier method), 28

is\_possible\_predict\_proba() (modelgym.models.xgboost\_model.XGBClassifier method), 24

is\_possible\_predict\_proba() (modelgym.models.xgboost\_model.XGBRegressor method), 25

## L

LGBMClassifier (class in modelgym.models.lightgbm\_model), 26

LGBMRegressor (class in modelgym.models.lightgbm\_model), 26

load\_from\_snapshot() (modelgym.models.catboost\_model.CtBClassifier static method), 28

load\_from\_snapshot() (modelgym.models.catboost\_model.CtBRegressor static method), 29

load\_from\_snapshot() (modelgym.models.ensemble\_model.EnsembleClassifier static method), 30

load\_from\_snapshot() (modelgym.models.ensemble\_model.EnsembleRegressor static method), 31

load\_from\_snapshot() (modelgym.models.lightgbm\_model.LGBMClassifier static method), 26

load\_from\_snapshot() (modelgym.models.lightgbm\_model.LGBMRegressor static method), 27

load\_from\_snapshot() (modelgym.models.model.Model static method), 24

load\_from\_snapshot() (modelgym.models.rf\_model.RFClassifier static method), 28

load\_from\_snapshot() (modelgym.models.xgboost\_model.XGBClassifier static method), 25

load\_from\_snapshot() (modelgym.models.xgboost\_model.XGBRegressor static method), 25

load\_state() (modelgym.trackers.tracker.LocalTracker method), 35

load\_state() (modelgym.trackers.tracker.TrackerMongo method), 36

LocalTracker (class in modelgym.trackers.tracker), 35

Logloss (class in modelgym.metrics), 32

## M

Metric (class in modelgym.metrics), 32

Model (class in modelgym.models.model), 23

modelgym.guru (module), 21

modelgym.metrics (module), 32

modelgym.models.catboost\_model (module), 28

modelgym.models.ensemble\_model (module), 30

modelgym.models.lightgbm\_model (module), 26

modelgym.models.model (module), 23

modelgym.models.rf\_model (module), 27

modelgym.models.xgboost\_model (module), 24

modelgym.trackers.tracker (module), 35

modelgym.trainers.hyperopt\_trainer (module), 33

modelgym.trainers.skopt\_trainer (module), 34

modelgym.utils.util (module), 36

Mse (class in modelgym.metrics), 32

## P

Precision (class in modelgym.metrics), 32

predict() (modelgym.models.catboost\_model.CtBClassifier method), 28

predict() (modelgym.models.catboost\_model.CtBRegressor method), 29

predict() (modelgym.models.ensemble\_model.EnsembleClassifier method), 30

predict() (modelgym.models.ensemble\_model.EnsembleRegressor method), 31

predict() (modelgym.models.lightgbm\_model.LGBMClassifier method), 26

predict() (modelgym.models.lightgbm\_model.LGBMRegressor method), 27

predict() (modelgym.models.model.Model method), 24

predict() (modelgym.models.rf\_model.RFClassifier method), 28

predict() (modelgym.models.xgboost\_model.XGBClassifier method), 25

predict() (modelgym.models.xgboost\_model.XGBRegressor method), 25

predict\_proba() (modelgym.models.catboost\_model.CtBClassifier method), 29

predict\_proba() (modelgym.models.catboost\_model.CtBRegressor method), 30

predict\_proba() (modelgym.models.ensemble\_model.EnsembleClassifier method), 31

predict\_proba() (modelgym.models.ensemble\_model.EnsembleRegressor method), 31

predict\_proba() (modelgym.models.lightgbm\_model.LGBMClassifier method), 26

predict\_proba() (modelgym.models.lightgbm\_model.LGBMRegressor method), 27

predict\_proba() (modelgym.models.model.Model method), 24

predict\_proba() (modelgym.models.rf\_model.RFClassifier method), 28

predict\_proba() (modelgym.models.xgboost\_model.XGBClassifier method), 25

predict\_proba() (modelgym.models.xgboost\_model.XGBRegressor method), 25

## R

RandomTrainer (class in modelgym.trainers.hyperopt\_trainer), 34

Recall (class in modelgym.metrics), 32

RFClassifier (class in modelgym.models.rf\_model), 27

RFTrainer (class in modelgym.trainers.skopt\_trainer), 34

RocAuc (class in modelgym.metrics), 32

## S

save\_snapshot() (modelgym.models.catboost\_model.CtBClassifier method), 29

save\_snapshot() (modelgym.models.catboost\_model.CtBRegressor method), 30

save\_snapshot() (modelgym.models.ensemble\_model.EnsembleClassifier method), 31

[save\\_snapshot\(\)](#) (modelgym.models.ensemble\_model.EnsembleRegressor method), [31](#)  
[save\\_snapshot\(\)](#) (modelgym.models.lightgbm\_model.LGBMClassifier method), [26](#)  
[save\\_snapshot\(\)](#) (modelgym.models.lightgbm\_model.LGBMRegressor method), [27](#)  
[save\\_snapshot\(\)](#) (modelgym.models.model.Model method), [24](#)  
[save\\_snapshot\(\)](#) (modelgym.models.rf\_model.RFClassifier method), [28](#)  
[save\\_snapshot\(\)](#) (modelgym.models.xgboost\_model.XGBClassifier method), [25](#)  
[save\\_snapshot\(\)](#) (modelgym.models.xgboost\_model.XGBRegressor method), [25](#)  
[save\\_state\(\)](#) (modelgym.trackers.tracker.LocalTracker method), [35](#)  
[save\\_state\(\)](#) (modelgym.trackers.tracker.TrackerMongo method), [36](#)  
[SkoptTrainer](#) (class in modelgym.trainers.skopt\_trainer), [34](#)

## T

[TpeTrainer](#) (class in modelgym.trainers.hyperopt\_trainer), [34](#)  
[TrackerMongo](#) (class in modelgym.trackers.tracker), [36](#)

## X

[XGBClassifier](#) (class in modelgym.models.xgboost\_model), [24](#)  
[XGBRegressor](#) (class in modelgym.models.xgboost\_model), [25](#)