modelgym Documentation

Release 0.1.3

anaderi

Apr 13, 2018

Contents

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)

CHAPTER 2

Installation

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 Dependences

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 allright!

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.

CHAPTER 3

Examples

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

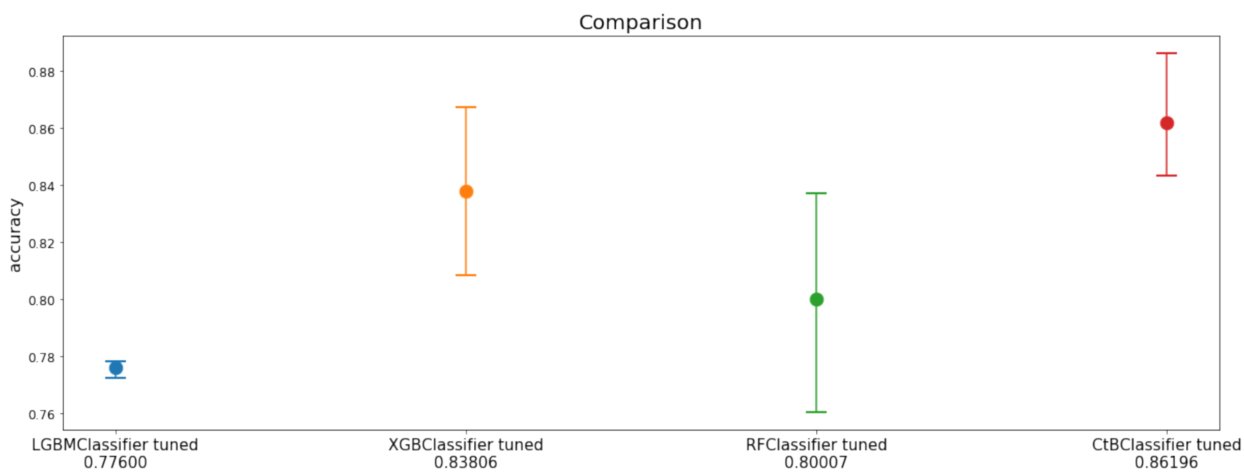
```
reporter.print_all_metric_results()
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~    accuracy    ~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~~~~~~~~~

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

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~    roc_auc    ~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~~~~~~~~~

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

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~    f1_score    ~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~~~~~~~~~

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

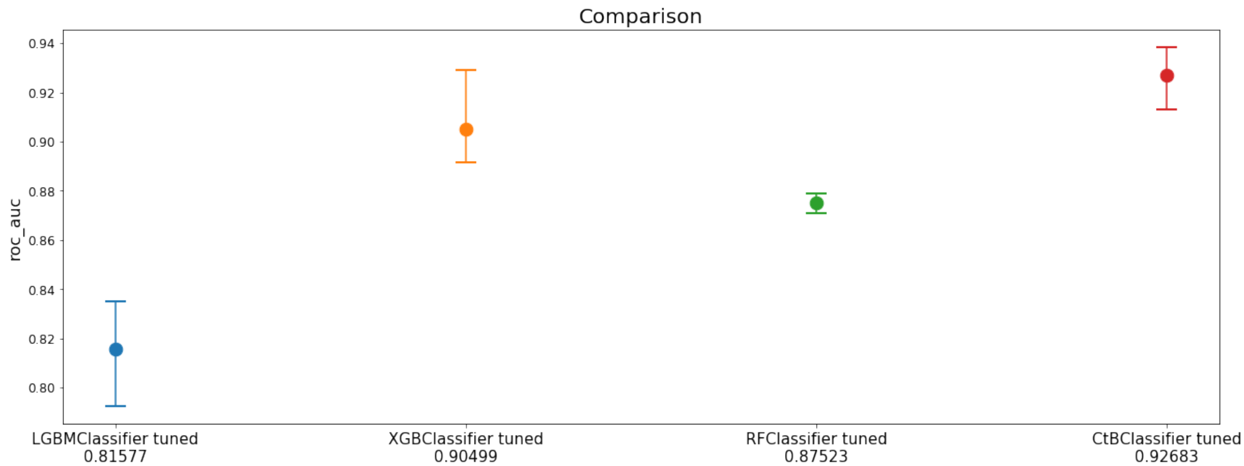
Report plots

```
reporter.plot_all_metrics()
```

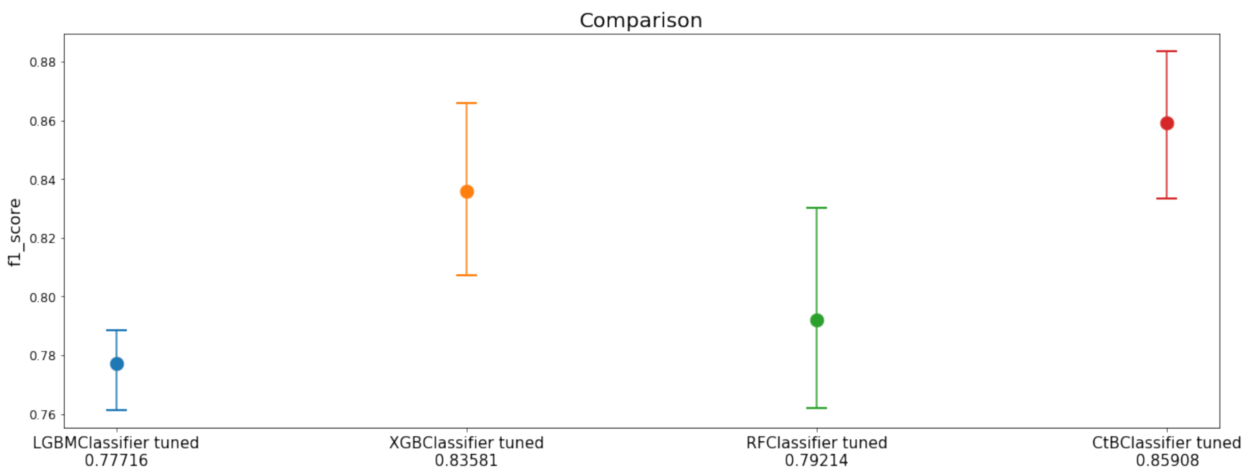
```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~    accuracy    ~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~~~~~~~~~
```



```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~     roc_auc     ~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~~~~~~~~~~
```



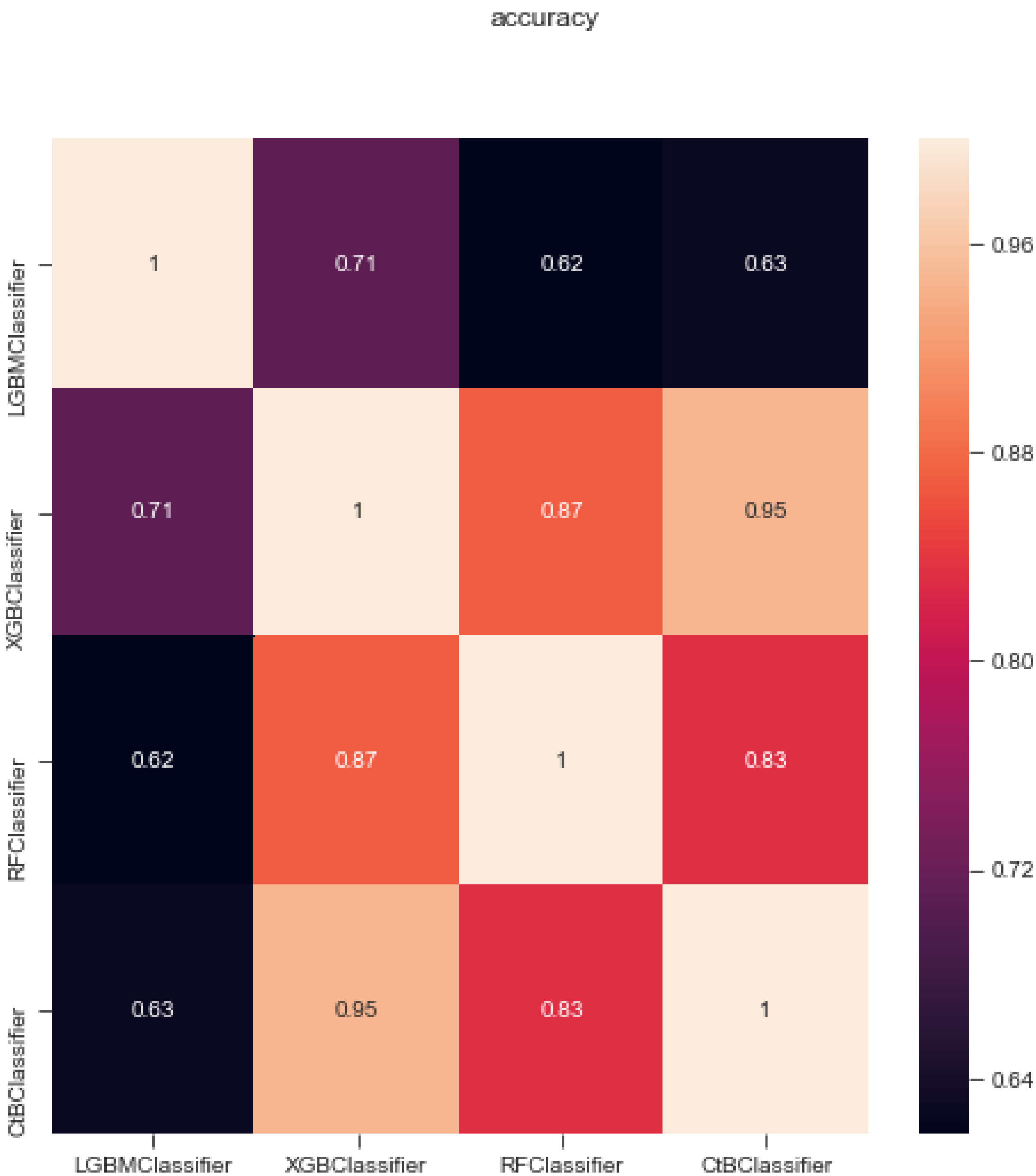

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~     f1_score     ~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~~~~~~~~~~
```



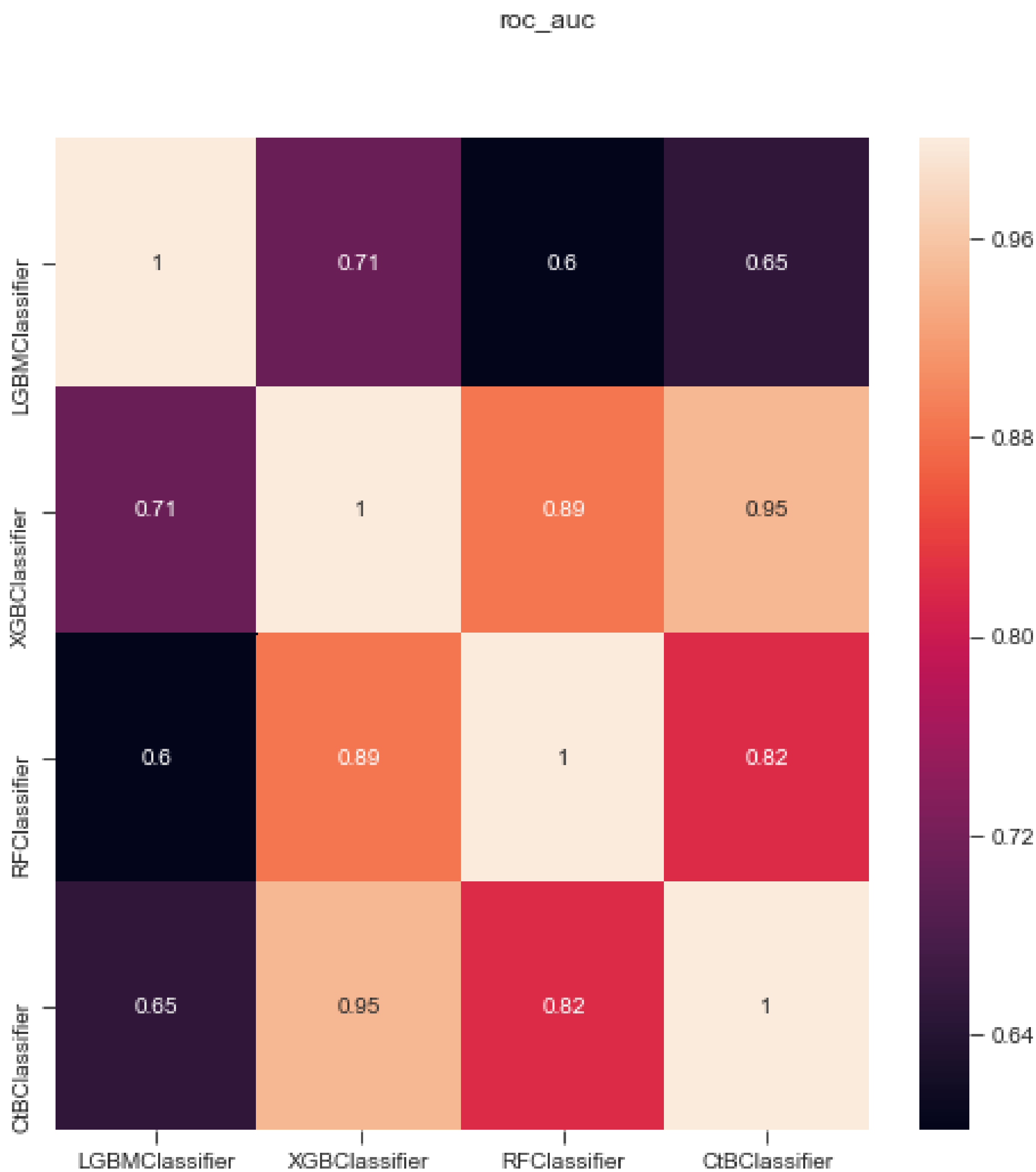
Report heatmaps for each metric

```
reporter.plot_heatmaps()
```

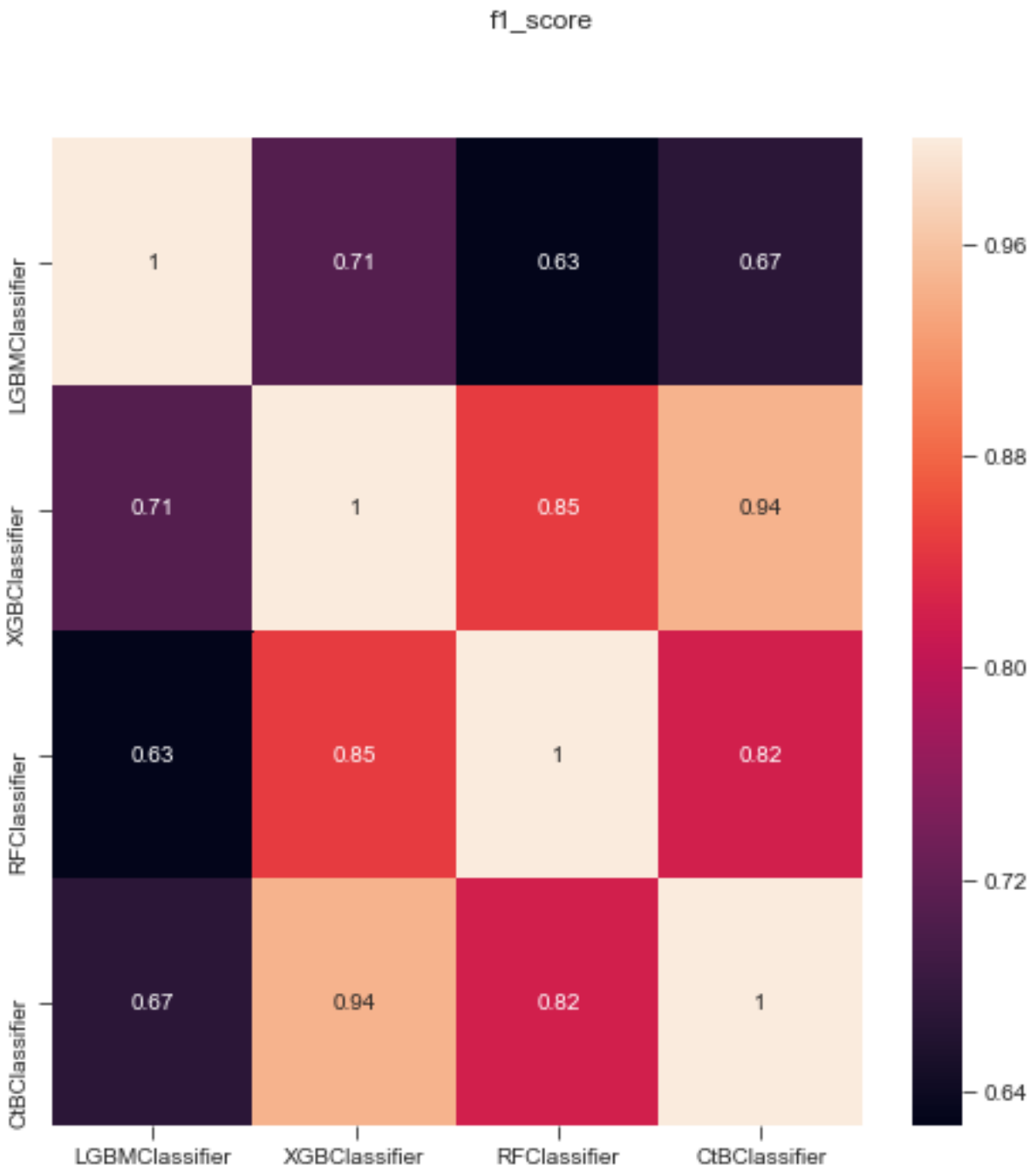
```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~     accuracy     ~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~~~~~~~~~~
```


accuracy
LGBMClassifier
XGBClassifier
RFClassifier
CtBClassifier
1
0.71
0.62
0.63
0.71
1
0.87
0.95
0.62
0.87
1
0.83
0.63
0.95
0.83
1
0.96
0.88
0.80
0.72
0.64
LGBMClassifier
XGBClassifier
RFClassifier
CtBClassifier

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~     roc_auc     ~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~~~~~~~~~~
```


roc_auc
LGBMClassifier
XGBClassifier
RFClassifier
CtBClassifier
1
0.71
0.6
0.65
0.71
1
0.89
0.95
0.6
0.89
1
0.82
0.65
0.95
0.82
1
0.96
0.88
0.80
0.72
0.64

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~    f1_score    ~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~~~~~~~~~~~
```



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 categorial 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_categorial(X)
```

```
Some features are supposed to be categorial. Make sure that all categorial 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

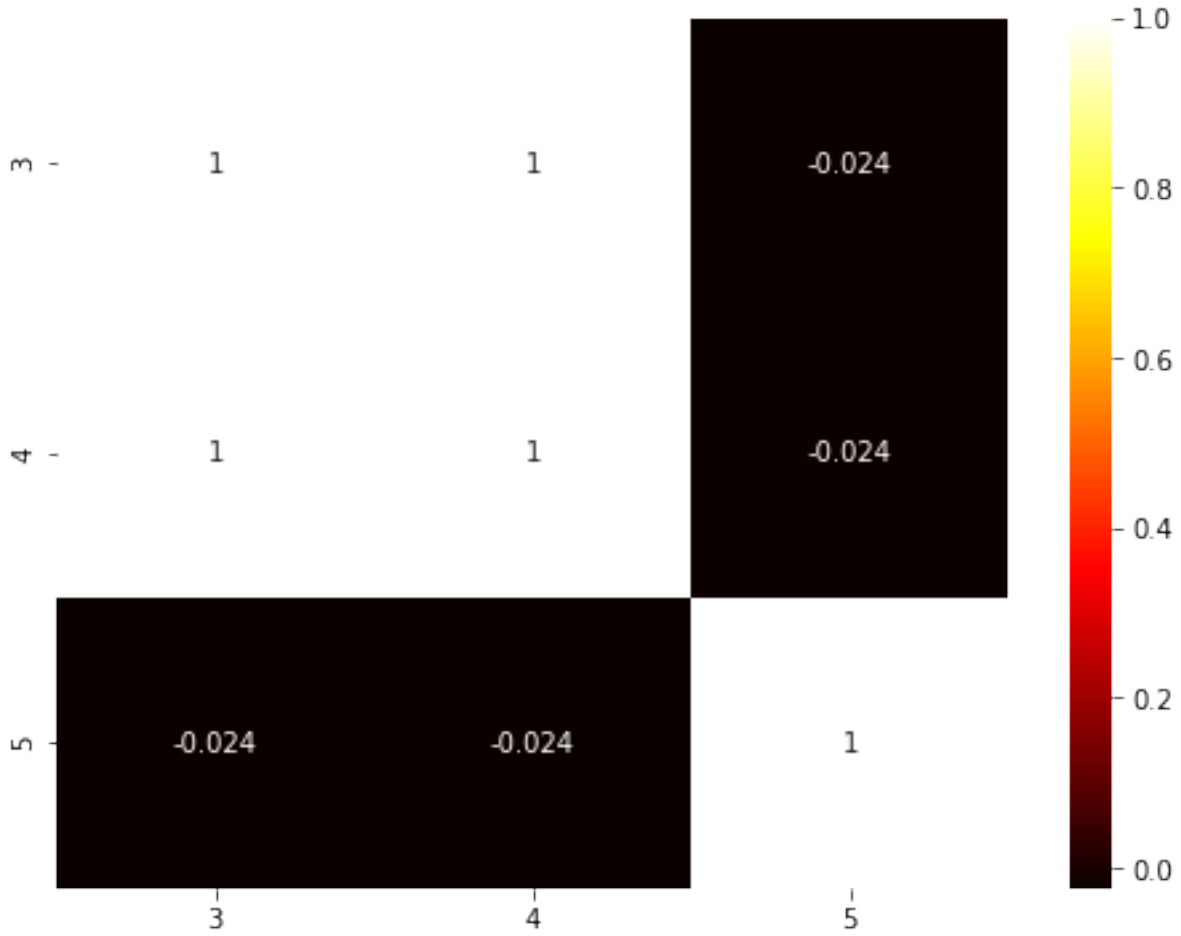
```
guru.check_correlation(X, [3, 4, 5])
```

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

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

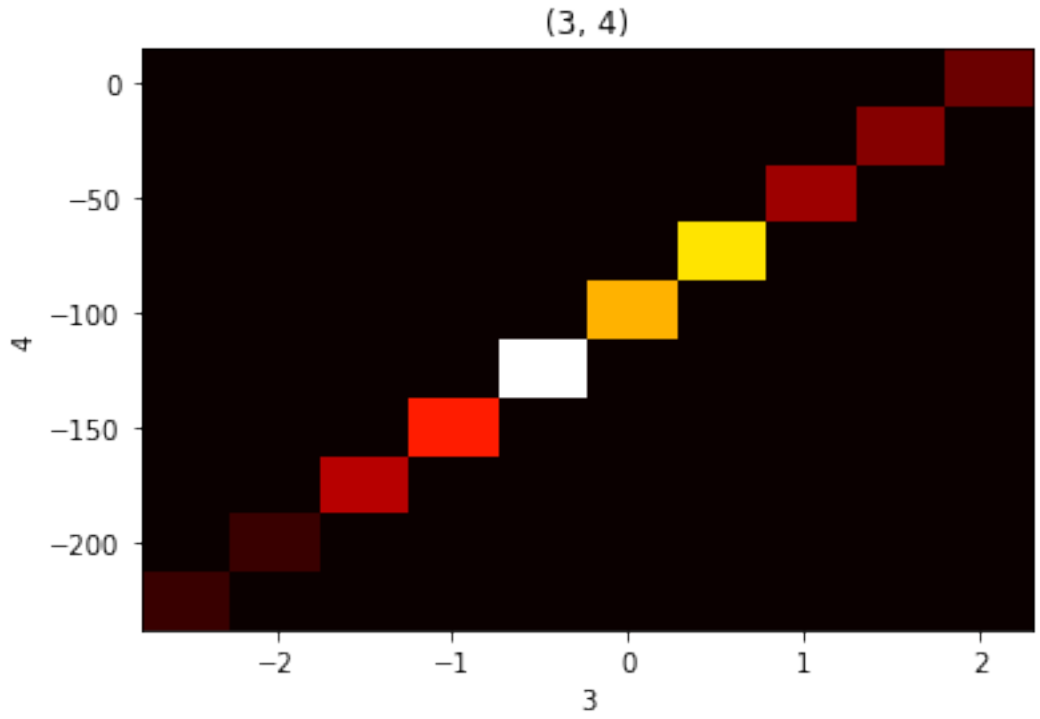
Drawing correlation heatmap for features

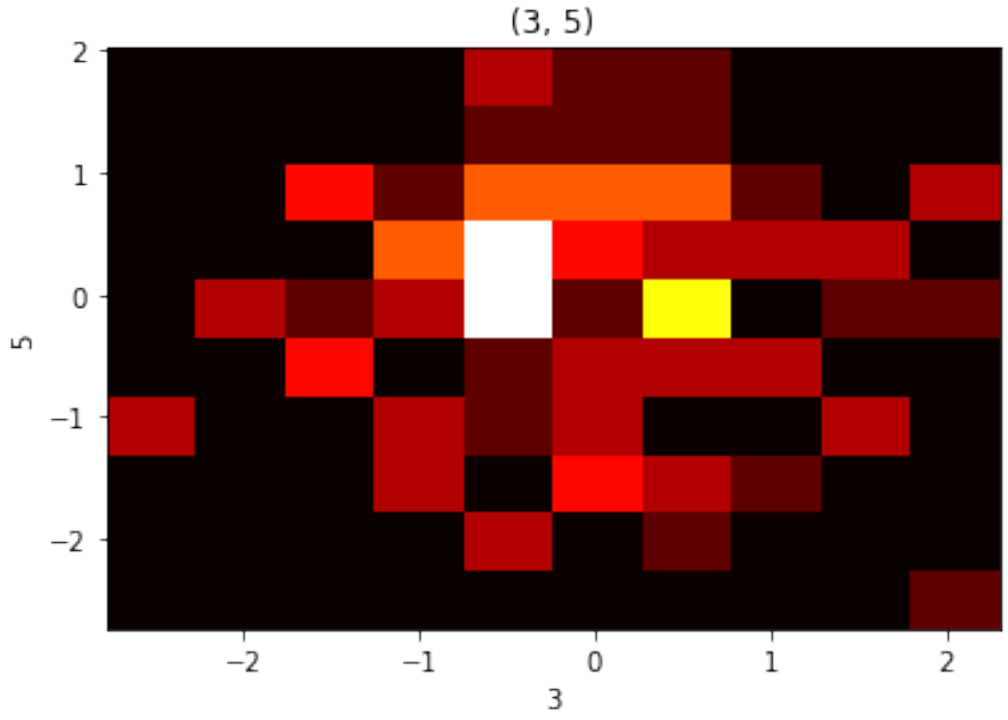
```
guru.draw_correlation_heatmap(X, [3, 4, 5], figsize=(8, 6))
```

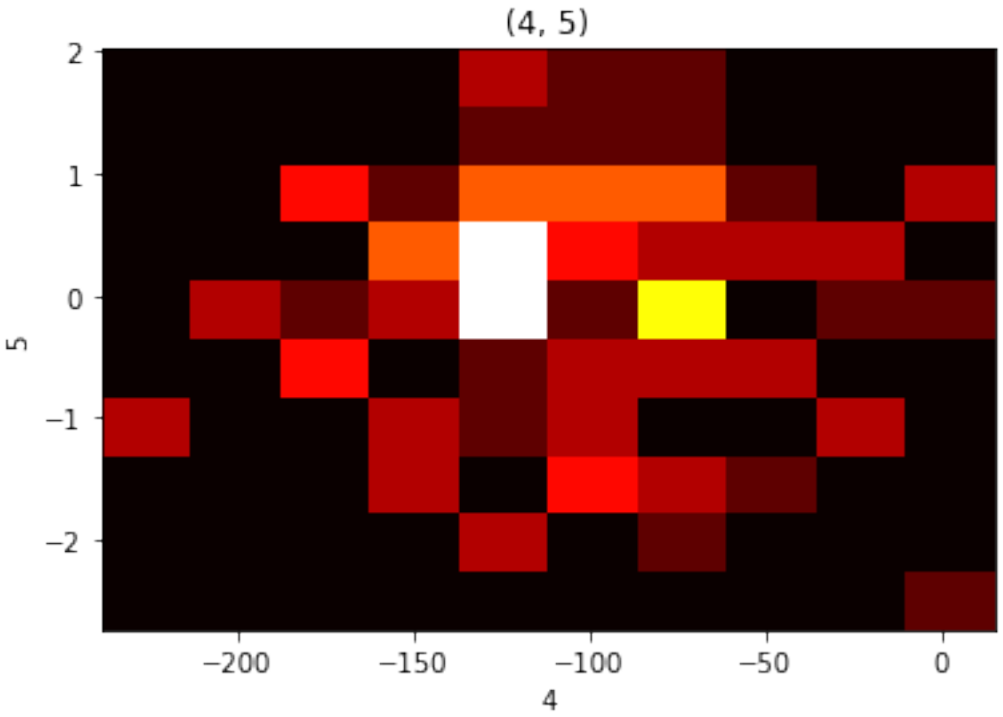


Drawing 2d histograms for features

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


(3, 4)
0
−50
−100
−150
−200
4
−2
−1
0
1
2
3


(3, 5)
2
1
0
−1
−2
5
−2
−1
0
1
2
3



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]})
```

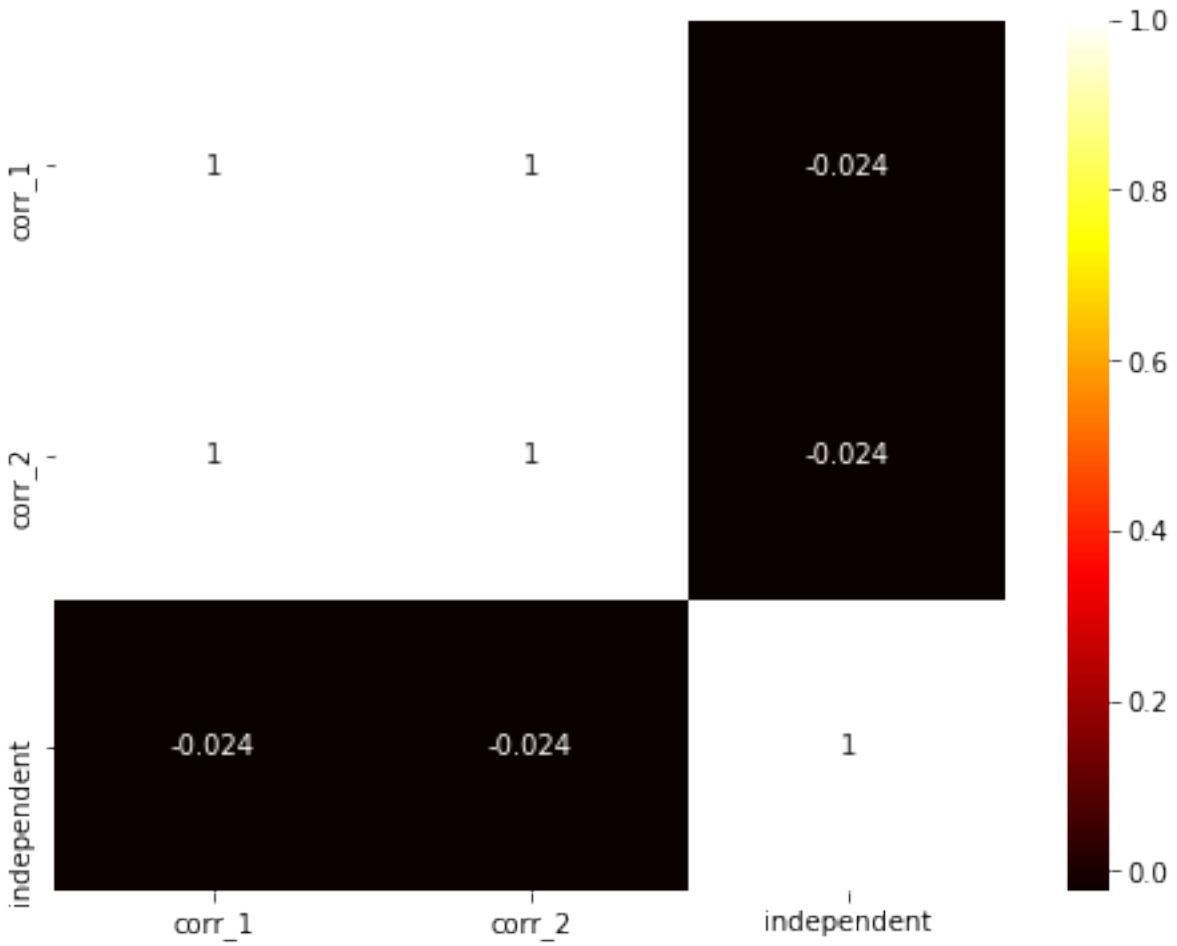
3.2.2 dtype with fields

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'),
                                ('categorial', '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))
```


corr_1
corr_2
independent
1
1
-0.024
1
1
-0.024
-0.024
-0.024
1
corr_1
corr_2
independent
1.0
0.8
0.6
0.4
0.2
0.0

CHAPTER 4

Documentaion

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 < float < 1*) – max number of distinct feature values in sample to assume this feature categorial
- **sparse_qoute** (*0 < float < 1*) – zeros portion in sample required to assume this feature sparse
- **class_disbalance_qoute** (*0 < float < 1*) – class portion should be distant from the mean to assume this class disbalanced

check_categorial(*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

(categorials, sparse, disbalanced, correlated)

- categorials: indexes of features which are supposed to be categorial
- 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_categorials
- 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*

4.4 Trainers

4.4.1 Hyperopt trainers

class modelgym.trainers.hyperopt_trainer.**HyperoptTrainer**(*model_spaces*, *algo=None*, *tracker=None*)

Bases: modelgym.trainers.trainer.Trainer

HyperoptTrainer is a class for models hyperparameter optimization, based on hyperopt 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__
- **algo** (function, e.g *hyperopt.rand.suggest* or *hyperopt.tpe.suggest*) – algorithm to use for optimization
- **tracker** (*modelgym.trackers.Tracker, optional*) – tracker to save (and load, if there was any) optimization progress.

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*, *batch_size=10*, *client=None*, ***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.
- **batch_size** (*int*) – periodicity of saving results to tracker
- **client** –
- ****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,
            "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*

TpeTrainer 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*, *alpha=0.05*, *metric='ROC_AUC'*)

Hypothesis: two models are the same

Python Module Index

m

Index

H

I

L

M

P

R

S

T

X