easypheno

Florian Haselbeck, Maura John, Dominik G. Grimm

Mar 23, 2023

CONTENTS

easyPheno

easyPheno is a Python framework that enables the rigorous training, comparison and analysis of phenotype predictions for a variety of different models. easyPheno includes multiple state-of-the-art prediction models. Besides common genomic selection approaches, such as best linear unbiased prediction (BLUP) and models from the Bayesian alphabet, our framework includes several machine learning methods. These range from classical models, such as regularized linear regression over ensemble learners, e.g. XGBoost, to deep learning-based architectures, such as Convolutional Neural Networks (CNN). To enable automatic hyperparameter optimization, we leverage state-of-the-art and efficient Bayesian optimization techniques. In addition, our framework is designed to allow an easy and straightforward integration of further prediction models.

CHAPTER

ONE

CONTRIBUTORS

This pipeline is developed and maintained by members of the Bioinformatics lab lead by Prof. Dr. Dominik Grimm:

- Florian Haselbeck, M.Sc.
- Maura John, M.Sc.

CHAPTER

TWO

CITATION

When using easyPheno, please cite our publication:

easyPheno: An easy-to-use and easy-to-extend Python framework for phenotype prediction using Bayesian optimization.
Florian Haselbeck*, Maura John* and Dominik G Grimm.
Bioinformatics Advances, 2023. doi: 10.1093/bioadv/vbad035
** These authors have contributed equally to this work and share first authorship.*

2.1 Installation Guide

easyPheno offers two ways of using it:

- *Docker workflow*: run the optimization pipeline with only one command in a Docker container
- *pip workflow*: use our framework as a Python library and integrate it into your pipeline

The whole framework was developed and tested using Ubuntu 20.04. Consequently, the subsequent guide is mainly written with regard to Ubuntu 20.04. The framework should work with Windows and Mac as well, but we do not officially provide support for these platforms. If you do not work on Ubuntu, we highly recommend the *Docker workflow*.

Besides the written guides, we also provide some tutorial videos which are embedded in the subpages.

2.1.1 Docker workflow

If you want to do phenotype prediction without the need of integrating parts of your own pipeline, we recommend the *Docker workflow* due to its easy-to-use interface and ready-to-use working environment within a Docker container. Besides the written tutorial, we provide a *Video tutorial: Docker workflow setup* embedded below.

Requirements

For the *Docker workflow*, Docker needs to be installed and running on your machine, see the Installation Guidelines at the Docker website. On Ubuntu, you can use `docker run hello-world` or `docker --version` to check if Docker works (Caution: add sudo if you are not in the docker group).

If you want to use GPU support, you need to install nvidia-docker-2 (see this nvidia-docker Installation Guide) and a version of CUDA >= 11.2 (see this CUDA Installation Guide). To check your CUDA version, just run `nvidia-smi` in a terminal.

Setup

1. Open a Terminal and navigate to the directory where you want to set up the project
2. Clone this repository

   ```
   git clone https://github.com/grimmlab/easyPheno.git
   ```

3. Navigate to *Docker* after cloning the repository

   ```
   cd easyPheno/Docker
   ```

4. Build a Docker image using the provided Dockerfile tagged with the IMAGENAME of your choice

   ```
   docker build -t IMAGENAME .
   ```

5. Run an interactive Docker container based on the created image with a CONTAINERNAME of your choice

   ```
   docker run -it -v /PATH/TO/REPO/FOLDER:/REPO_DIRECTORY/IN/CONTAINER -v /
   ↪PATH/TO/DATA/DIRECTORY:/DATA_DIRECTORY/IN/CONTAINER -v /PATH/TO/RESULTS/
   ↪SAVE/DIRECTORY:/SAVE_DIRECTORY/IN/CONTAINER --name CONTAINERNAME IMAGENAME
   ```

   - Mount the directory where the repository is placed on your machine, the directory where your phenotype and genotype data is stored and the directory where you want to save your results using the option `-v`.
   - You can restrict the number of cpus using the option `cpuset-cpus CPU_INDEX_START-CPU_INDEX_STOP`.
   - Specify a gpu device using `--gpus device=DEVICE_NUMBER` if you want to use GPU support.

   Let's have a look at an example. We assume hat you created a Docker image called `ep-image`, your repository and data is placed in (subfolders of) `/myhome/`, you want to save your results to `/myhome/` (so `/myhome/` is the only directory you need to mount in your container), you only want to use CPUs 0 to 9 and GPU 0 and you want to call your container `ep_container`. Then you have to run the following command:

   ```
   docker run -it -v /myhome/:/myhome_in_my_container/ --cpuset-cpus 0-9 --
   ↪gpus device=0 --name ep_container ep_image
   ```

Your setup is finished! Go to *HowTo: Run easyPheno using Docker* to see how you can now use easyPheno!

Useful Docker commands

The subsequent Docker commands might be useful when using easyPheno. See here for a full guide on the Docker commands.

docker images List all Docker images on your machine

docker ps List all running Docker containers on your machine

docker ps -a List all Docker containers (including stopped ones) on your machine

docker start -i CONTAINERNAME Start a (stopped) Docker container interactively to enter its command line interface

Video tutorial: Docker workflow setup

https://youtu.be/ONC9MZic0n0

2.1.2 pip workflow

easyPheno can be installed via `pip` and used as a common Python library:

```
pip install easypheno
```

Our setup is finished! Go to *HowTo: Use easyPheno as a pip package* to see how you can now use easyPheno!

The pipeline was developed and tested with Python 3.8 and Ubuntu 20.04. The framework should work with Windows and Mac as well, but we do not officially provide support for these platform. We neither officially support other Python versions, however easyPheno should run as well for versions >= 3.8. If these requirements are not fulfilled, we recommend the *Docker workflow*.

2.2 Quickstart Guide

We offer easyPheno both with a command line interface (CLI) and as a pip package.

For a quick start, see the following pages:

2.2.1 Command Line Interface

If you want to use easyPheno's command line interface, we highly recommend the workflow using Docker to ensure a straightforward and proper setup of all dependencies.

To setup easyPheno with Docker, see our Installation Guide: *Docker workflow*

To use easyPheno's command line interface, see *HowTo: Run easyPheno using Docker*

2.2.2 pip package

If you want to use easyPheno as a pip package, e.g. to be able to integrate its functions to your own code, you can easily install it by just running:

```
pip install easypheno
```

More details can be found in the Installation Guide: *pip workflow*

For a walkthrough of easyPheno's main phenotype prediction pipeline using the pip workflow, see the following tutorial: *HowTo: Use easyPheno as a pip package*

2.3 Tutorials

On the following pages, we show several tutorials for the usage of easyPheno - often supported by videos.

2.3.1 HowTo: Run easyPheno using Docker

We assume that you succesfully did all steps described in *Docker workflow* to set up easyPheno using Docker. Besides this written tutorial, we recorded a *Video tutorial: Run easyPheno with Docker* embedded below.

Workflow

You are at the **root directory within your Docker container**, i.e. after step 5 of the setup at *Docker workflow*:.

If you closed the Docker container that you created at the end of the installation, just use `docker start -i CONTAINERNAME` to start it in interactive mode again. If you did not create a container yet, go back to step 5 of the *Docker workflow*.

1. Navigate to the directory where the easyPheno repository is placed within your container

   ```
   cd /REPO_DIRECTORY/IN/CONTAINER/easyPheno
   ```

2. Run easyPheno (as module). By default, easyPheno starts the optimization procedure for 10 trials with XGBoost and a 5-fold nested cross-validation using the data we provide in `docs/source/tutorials/tutorial_data`.

   ```
   python3 -m easypheno.run --save_dir SAVE_DIRECTORY
   ```

   That's it! Very easy! You can now find the results in the save directory you specified.

3. To get an overview of the different options you can set for running easyPheno, just do:

   ```
   python3 -m easypheno.run --help
   ```

Feel free to test easyPheno, e.g. with other prediction models. If you want to start using your own data, please carefully read our *Data Guide* to ensure that your data fulfills all requirements.

Video tutorial: Run easyPheno with Docker

https://youtu.be/uM-yJqzzIPo

2.3.2 HowTo: Use easyPheno as a pip package

In this Jupyter notebook, we show how you can use easyPheno as a pip package and also guide you through the steps that easyPheno is doing when triggering an optimization run.

Please clone the whole GitHub repository if you want to run this tutorial on your own, as we need the tutorial data from our GitHub repository and to make sure that all paths we define are correct: `git clone https://github.com/grimmlab/easyPheno.git`

Then, start a Jupyter notebook server on your machine and open this Jupyter notebook, which is placed at `docs/source/tutorials` in the repository.

However, you could also download the single files and define the paths yourself:

- The Jupyter notebook can be downloaded here: HowTo: Use easyPheno as a pip package.ipynb
- The data we use can be found here: tutorial data

Installation, imports and paths

First, we may need to install easyPheno (uncomment if it is not already installed). Then, we import easyPheno as well as further libraries that we need in this tutorial. In the end, we define some paths and filenames which we will use more often throughout this tutorial. We will save the results in the same directory where this repository is placed.

[1]:
```
# !pip3 install easypheno
```

[6]:
```
import easypheno
import pathlib
import pandas as pd
import datetime
import pprint
```

[7]:
```
# Definition of paths and filenames
cwd = pathlib.Path.cwd()
data_dir = cwd.joinpath('tutorial_data')
save_dir = cwd.parents[3]
genotype_matrix = 'x_matrix.csv'
phenotype_matrix = 'y_matrix.csv'
phenotype = 'continuous_values'
```

Run whole optimization pipeline at once

As shown for the Docker workflow, easyPheno offers a function optim_pipeline.run() that triggers the whole optimization run.

In the definition of `optim_pipeline.run()`, we set several default values. In order to run it using our tutorial data, we just need to define the data and directories we want to use as well as the models we want to optimize. Furthermore, we set values for the `datasplit` and `n_trials` to limit the waiting time for getting the results.

When calling the function, we first see some information regarding the data preprocessing and the configuration of our optimization run, e.g. the data that is used. Then, the current progress of the optuna optimization with results of the individual trials is shown. In the end, we show a summary of the whole optimization run.

[8]:
```
easypheno.optim_pipeline.run(
    data_dir=data_dir, genotype_matrix=genotype_matrix, phenotype_matrix=phenotype_
↪matrix, phenotype=phenotype,
    save_dir=save_dir, models=['xgboost'], n_trials=10, datasplit='cv-test'
)
```

```
Check if all data files have the required format
Genotype file not in required format. Will load genotype matrix and save as .h5 file.␣
↪Will also create required index file.
Load genotype file /home/fhaselbeck/PycharmProjects/easyPheno/docs/source/tutorials/
↪tutorial_data/x_matrix.csv
Save unified genotype file /home/fhaselbeck/PycharmProjects/easyPheno/docs/source/
↪tutorials/tutorial_data/x_matrix.h5
Have genotype matrix. Load phenotype continuous_values from /home/fhaselbeck/
↪PycharmProjects/easyPheno/docs/source/tutorials/tutorial_data/y_matrix.csv
Have phenotype vector. Start matching genotype and phenotype.
Done matching genotype and phenotype. Create index file now.
Done checking data files. All required datasets are available.
----- Starting dataset preparation -----
Load and match raw data
Apply MAF filter
Filter duplicate SNPs
Check if final snp_ids already exist in index_file for used encoding and maf percentage.␣
↪Save them if necessary.
Load datasplit file
Checked datasplit for all folds.
++++++++++ CONFIG INFORMATION ++++++++++
Genotype Matrix: x_matrix.csv
Phenotype Matrix: y_matrix.csv
Phenotype: continuous_values
Encoding: 012
Models: xgboost
Optuna Trials: 10
Datasplit: cv-test (5-20)
MAF: 0
Dataset Infos
- Task detected: regression
- No. of samples: 286, No. of features: 590
- Encoding: 012
- Target variable statistics:
count    286.000000
mean      84.542832
```

(continues on next page)

(continued from previous page)

```
std       20.141244
min       53.000000
25%       69.250000
50%       78.750000
75%       97.750000
max      157.500000
Name: 0, dtype: float64
++++++++++++++++++++++++++++++++++++++++++
### Starting Optuna Optimization for xgboost ###
[I 2022-10-04 16:07:07,575] A new study created in RDB with name: 2022-10-04_16-07-04_x_
↪matrix-y_matrix-continuous_values-MAF0-SPLITcv-test5-20-MODELxgboost-TRIALS10
Params for Trial 0
{'n_estimators': 2500, 'learning_rate': 0.07500000000000001, 'max_depth': 3, 'gamma':␣
↪300, 'subsample': 0.45, 'colsample_bytree': 0.35000000000000003, 'reg_alpha': 290.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
[I 2022-10-04 16:07:15,020] Trial 0 finished with value: 369.22728277444065 and␣
↪parameters: {'n_estimators': 2500, 'learning_rate': 0.07500000000000001, 'max_depth':␣
↪3, 'gamma': 300, 'subsample': 0.45, 'colsample_bytree': 0.35000000000000003, 'reg_alpha
↪': 290.0}. Best is trial 0 with value: 369.22728277444065.
Params for Trial 1
{'n_estimators': 3000, 'learning_rate': 0.3, 'max_depth': 9, 'gamma': 300, 'subsample':␣
↪0.1, 'colsample_bytree': 0.55, 'reg_alpha': 440.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
[I 2022-10-04 16:07:25,321] Trial 1 finished with value: 454.8753047090302 and␣
↪parameters: {'n_estimators': 3000, 'learning_rate': 0.3, 'max_depth': 9, 'gamma': 300,
↪'subsample': 0.1, 'colsample_bytree': 0.55, 'reg_alpha': 440.0}. Best is trial 0 with␣
↪value: 369.22728277444065.
Params for Trial 2
{'n_estimators': 2250, 'learning_rate': 0.2, 'max_depth': 10, 'gamma': 80, 'subsample':␣
↪0.2, 'colsample_bytree': 0.05, 'reg_alpha': 320.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
[I 2022-10-04 16:07:32,860] Trial 2 finished with value: 398.20160514328586 and␣
↪parameters: {'n_estimators': 2250, 'learning_rate': 0.2, 'max_depth': 10, 'gamma': 80,
↪'subsample': 0.2, 'colsample_bytree': 0.05, 'reg_alpha': 320.0}. Best is trial 0 with␣
↪value: 369.22728277444065.
```

```
Params for Trial 3
{'n_estimators': 2000, 'learning_rate': 0.225, 'max_depth': 8, 'gamma': 770, 'subsample':
↪ 0.1, 'colsample_bytree': 0.3, 'reg_alpha': 110.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
```

```
[I 2022-10-04 16:07:41,691] Trial 3 finished with value: 369.6724074334373 and␣
↪parameters: {'n_estimators': 2000, 'learning_rate': 0.225, 'max_depth': 8, 'gamma':␣
↪770, 'subsample': 0.1, 'colsample_bytree': 0.3, 'reg_alpha': 110.0}. Best is trial 0␣
↪with value: 369.22728277444065.
```

```
Params for Trial 4
{'n_estimators': 1750, 'learning_rate': 0.25, 'max_depth': 6, 'gamma': 520, 'subsample':␣
↪0.35000000000000003, 'colsample_bytree': 0.05, 'reg_alpha': 100.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
```

```
[I 2022-10-04 16:07:49,667] Trial 4 finished with value: 364.574017351775 and parameters:
↪ {'n_estimators': 1750, 'learning_rate': 0.25, 'max_depth': 6, 'gamma': 520, 'subsample
↪': 0.35000000000000003, 'colsample_bytree': 0.05, 'reg_alpha': 100.0}. Best is trial 4␣
↪with value: 364.574017351775.
```

```
Params for Trial 5
{'n_estimators': 2750, 'learning_rate': 0.2, 'max_depth': 9, 'gamma': 810, 'subsample':␣
↪0.15000000000000002, 'colsample_bytree': 0.75000000000000001, 'reg_alpha': 540.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
```

```
[I 2022-10-04 16:08:01,195] Trial 5 finished with value: 452.7427816744316 and␣
↪parameters: {'n_estimators': 2750, 'learning_rate': 0.2, 'max_depth': 9, 'gamma': 810,
↪'subsample': 0.15000000000000002, 'colsample_bytree': 0.75000000000000001, 'reg_alpha':␣
↪540.0}. Best is trial 4 with value: 364.574017351775.
```

```
Params for Trial 6
{'n_estimators': 100, 'learning_rate': 0.3, 'max_depth': 4, 'gamma': 520, 'subsample': 0.
↪60000000000000001, 'colsample_bytree': 0.3, 'reg_alpha': 980.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
```

```
[I 2022-10-04 16:08:06,452] Trial 6 finished with value: 447.1631958152293 and␣
↪parameters: {'n_estimators': 100, 'learning_rate': 0.3, 'max_depth': 4, 'gamma': 520,
↪'subsample': 0.60000000000000001, 'colsample_bytree': 0.3, 'reg_alpha': 980.0}. Best is␣
↪trial 4 with value: 364.574017351775.
```

```
Params for Trial 7
{'n_estimators': 50, 'learning_rate': 0.3, 'max_depth': 4, 'gamma': 670, 'subsample': 0.
↪6500000000000001, 'colsample_bytree': 0.2, 'reg_alpha': 730.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
[I 2022-10-04 16:08:10,461] Trial 7 finished with value: 420.049507545245 and parameters:
↪ {'n_estimators': 50, 'learning_rate': 0.3, 'max_depth': 4, 'gamma': 670, 'subsample':␣
↪0.6500000000000001, 'colsample_bytree': 0.2, 'reg_alpha': 730.0}. Best is trial 4 with␣
↪value: 364.574017351775.
Params for Trial 8
{'n_estimators': 1000, 'learning_rate': 0.2, 'max_depth': 3, 'gamma': 690, 'subsample':␣
↪0.35000000000000003, 'colsample_bytree': 0.7500000000000001, 'reg_alpha': 130.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
[I 2022-10-04 16:08:19,589] Trial 8 finished with value: 373.9128450040709 and␣
↪parameters: {'n_estimators': 1000, 'learning_rate': 0.2, 'max_depth': 3, 'gamma': 690,
↪'subsample': 0.35000000000000003, 'colsample_bytree': 0.7500000000000001, 'reg_alpha':␣
↪130.0}. Best is trial 4 with value: 364.574017351775.
Params for Trial 9
{'n_estimators': 250, 'learning_rate': 0.125, 'max_depth': 5, 'gamma': 730, 'subsample':␣
↪0.7500000000000001, 'colsample_bytree': 0.7500000000000001, 'reg_alpha': 780.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
[I 2022-10-04 16:08:28,648] Trial 9 finished with value: 419.9258576836199 and␣
↪parameters: {'n_estimators': 250, 'learning_rate': 0.125, 'max_depth': 5, 'gamma': 730,
↪ 'subsample': 0.7500000000000001, 'colsample_bytree': 0.7500000000000001, 'reg_alpha':␣
↪780.0}. Best is trial 4 with value: 364.574017351775.
## Optuna Study finished ##
Study statistics:
  Finished trials:  10
  Pruned trials:  0
  Completed trials:  10
  Best Trial:  4
  Value:  364.574017351775
  Params:
    colsample_bytree: 0.05
    gamma: 520
    learning_rate: 0.25
    max_depth: 6
    n_estimators: 1750
    reg_alpha: 100.0
```

(continues on next page)

(continued from previous page)

```
    subsample: 0.35000000000000003
## Retrain best model and test ##
## Results on test set ##
{'test_mse': 393.8179197745938, 'test_rmse': 19.84484617664228, 'test_r2_score': 0.
↪05719242798046553, 'test_explained_variance': 0.05724940832753167}
### Finished Optuna Optimization for xgboost ###
# Optimization runs done for models ['xgboost']
Results overview on the test set(s)
{'xgboost': {'Test': {'best_params': {'colsample_bytree': 0.05,
                                      'gamma': 520,
                                      'learning_rate': 0.25,
                                      'max_depth': 6,
                                      'n_estimators': 1750,
                                      'reg_alpha': 100.0,
                                      'subsample': 0.35000000000000003},
                      'eval_metrics': {'test_explained_variance': 0.05724940832753167,
                                       'test_mse': 393.8179197745938,
                                       'test_r2_score': 0.05719242798046553,
                                       'test_rmse': 19.84484617664228},
                      'runtime_metrics': {'process_time_max': 36.719140110000005,
                                          'process_time_mean': 20.092465339700002,
                                          'process_time_min': 1.2973512780000078,
                                          'process_time_std': 13.062248523467469,
                                          'real_time_max': 10.839427947998049,
                                          'real_time_mean': 7.2125649690628055,
                                          'real_time_min': 3.837096929550171,
                                          'real_time_std': 2.0072883008924216}}}}
```

Within the defined `save_dir`, a `results` folder will be created.

Then, easyPheno's default folder structure follows: `name_of_genotype_matrix/name_of_phenotype_matrix/phenotype/`. For instance, all phenotype matrices assigned to the same genotype matrix are gathered in the same subdirectory (`name_of_genotype_matrix/`). The same applies for all phenotypes assigned to the same phenotype matrix (`name_of_genotype_matrix/name_of_phenotype_matrix/`)

We can see this structure below with all optimization results for the defined `phenotype`.

[9]:
```
result_folders = list(save_dir.joinpath('results', genotype_matrix.split('.')[0],␣
↪phenotype_matrix.split('.')[0], phenotype).glob('*'))
for results_dir in result_folders:
    print(results_dir)
```

```
/home/fhaselbeck/PycharmProjects/results/x_matrix/y_matrix/continuous_values/cv-test_5-
↪20_MAF0_xgboost_2022-10-04_16-07-04
```

These result folder names show information on the `datasplit` (type and parameters, e.g. in case of `cv-test` the part `5-20`: 5 relates to 5 folds of the cross-validation and 20 to a test set consisting of 20 percent of the data). Furthermore, we see the maf filter that was applied (`MAF`), the `models` that were optimized and a time stamp.

In the example below, we can see that each result folder contains a `Results_overview_*.csv` as well as detailed results for each of the optimized models. In case of `nested-cv`, this is preceded by a subfolder for each of the outer folds.

[10]:
```
result_elements = list(result_folders[0].glob('*'))
for result_element in result_elements:
    print(result_element)
```

```
/home/fhaselbeck/PycharmProjects/results/x_matrix/y_matrix/continuous_values/cv-test_5-
↪20_MAF0_xgboost_2022-10-04_16-07-04/xgboost
/home/fhaselbeck/PycharmProjects/results/x_matrix/y_matrix/continuous_values/cv-test_5-
↪20_MAF0_xgboost_2022-10-04_16-07-04/Results_overview_xgboost.csv
```

The `Results_overview_*.csv` file contains the best parameters, evaluation as well as runtime metrics for each of the optimized models as we can see in the example below.

[11]:
```
results_overview_file = [overview_file for overview_file in result_elements if 'Results_
↪overview' in str(overview_file)][0]
pd.read_csv(results_overview_file)
```

[11]:
```
  Unnamed: 0                               xgboost___best_params  \
0       Test  [{'colsample_bytree': 0.05, 'gamma': 520, 'lea...

                               xgboost___eval_metrics  \
0  [{'test_mse': 393.8179197745938, 'test_rmse': ...

                           xgboost___runtime_metrics
0  [{'process_time_mean': 20.092465339700002, 'pr...
```

Beyond that, we see below that the detailed results for each optimized model contain validation and test results, saved prediction models, an optuna database, a runtime overview with information for each trial (good for debugging, as pruning reasons are also documented) and for some prediction models also feature importances.

[12]:
```
for subdir in [overview_file for overview_file in result_elements if 'Results_overview'␣
↪not in str(overview_file)][0].rglob('*'):
    print(subdir)
```

```
/home/fhaselbeck/PycharmProjects/results/x_matrix/y_matrix/continuous_values/cv-test_5-
↪20_MAF0_xgboost_2022-10-04_16-07-04/xgboost/Optuna_DB.db
/home/fhaselbeck/PycharmProjects/results/x_matrix/y_matrix/continuous_values/cv-test_5-
↪20_MAF0_xgboost_2022-10-04_16-07-04/xgboost/validation_results_trial4.csv
/home/fhaselbeck/PycharmProjects/results/x_matrix/y_matrix/continuous_values/cv-test_5-
↪20_MAF0_xgboost_2022-10-04_16-07-04/xgboost/xgboost_runtime_overview.csv
/home/fhaselbeck/PycharmProjects/results/x_matrix/y_matrix/continuous_values/cv-test_5-
↪20_MAF0_xgboost_2022-10-04_16-07-04/xgboost/unfitted_model_trial4
/home/fhaselbeck/PycharmProjects/results/x_matrix/y_matrix/continuous_values/cv-test_5-
↪20_MAF0_xgboost_2022-10-04_16-07-04/xgboost/final_model_test_results.csv
/home/fhaselbeck/PycharmProjects/results/x_matrix/y_matrix/continuous_values/cv-test_5-
↪20_MAF0_xgboost_2022-10-04_16-07-04/xgboost/final_model_feature_importances.csv
```

Single elements of the optimization pipeline

For a better understanding of the whole optimization pipeline, we subsequently show some of the single elements which are called within `optim_pipeline.run()`.

First, `optim_pipeline.run()` contains some functions to check the specified arguments, which we will skip for this tutorial. However, we need to define some of the default values and create `pathlib.Path` objects.

[13]:
```
data_dir = pathlib.Path(data_dir)
save_dir = pathlib.Path(save_dir)
datasplit = 'cv-test'
n_innerfolds = 5
test_set_size_percentage=20
maf_percentage = 0
models = ['xgboost']
n_trials = 10
```

The first step of the optimization pipeline is the preparation of the raw data files using `easypheno.preprocess.raw_data_functions.prepare_data_files()`. If the format matches our Data Guide, the raw data files are preprocessed.

The genotype matrix is converted and unified to a `.h5` file and saved with the same name as the raw file, if this genotype matrix is used for the first time.

The phenotype matrix is checked whether the format is fine, but not saved in a different format.

An index file containing indices for filtering the data (e.g. maf or duplicates) and creating the data splits is saved or updated in case it already exists and a datasplit that is currently not present in the file is requested. This ensures reproducibility of the preprocessing and data splits.

[14]:
```
easypheno.preprocess.raw_data_functions.prepare_data_files(
    data_dir=data_dir, genotype_matrix_name=genotype_matrix, phenotype_matrix_
↪name=phenotype_matrix,
    phenotype=phenotype, datasplit=datasplit, n_outerfolds=5, n_innerfolds=n_innerfolds,
    test_set_size_percentage=test_set_size_percentage, val_set_size_percentage=20,
    models=models, user_encoding=None, maf_percentage=maf_percentage
)
```

```
Check if all data files have the required format
Found same file name with ending .h5
Assuming that the raw file was already prepared using our pipepline. Will continue with␣
↪the .h5 file.
Genotype file available in required format, check index file now.
Index file x_matrix-y_matrix-continuous_values.h5 already exists. Will append required␣
↪filters and data splits now.
Done checking data files. All required datasets are available.
```

After setting all seeds for reproducibility using `easypheno.utils.helper_functions.set_all_seeds()`, a model for the current optimization is selected. This information is then used to retrieve its `standard_encoding` if the user did not define an encoding.

With this information, the `easypheno.preprocess.base_dataset.Dataset` object is initialized.

We also print some information regarding the current progress, as loading the data might take some time for bigger datasets.

When running the optimization for multiple models, these are sorted according to their encoding and the dataset is only loaded new if the encoding changes between models.

[15]:
```
easypheno.utils.helper_functions.set_all_seeds()
current_model_name = models[0]
encoding = easypheno.utils.helper_functions.get_mapping_name_to_class()[current_model_
↪name].standard_encoding

dataset = easypheno.preprocess.base_dataset.Dataset(
    data_dir=data_dir, genotype_matrix_name=genotype_matrix, phenotype_matrix_
↪name=phenotype_matrix,
    phenotype=phenotype, datasplit=datasplit, n_outerfolds=5, n_innerfolds=n_innerfolds,
    test_set_size_percentage=test_set_size_percentage, val_set_size_percentage=20,
    encoding=encoding, maf_percentage=maf_percentage
)
```

```
Load and match raw data
Apply MAF filter
Filter duplicate SNPs
Check if final snp_ids already exist in index_file for used encoding and maf percentage.␣
↪Save them if necessary.
Load datasplit file
Checked datasplit for all folds.
```

After retrieving the type of ML `task` using `easypheno.utils.helper_functions.test_likely_categorical()` as well as the time stamp for saving the results, we create an `easypheno.optimization.optuna_optim.OptunaOptim` object. For this purpose, we handover all information that is needed for the hyperparameter search.

[16]:
```
task = 'classification' if easypheno.utils.helper_functions.test_likely_
↪categorical(dataset.y_full) else 'regression'
models_start_time = '+'.join(models) + '_' + datetime.datetime.now().strftime("%Y-%m-%d_
↪%H-%M-%S")

optim_run = easypheno.optimization.optuna_optim.OptunaOptim(
    save_dir=save_dir, genotype_matrix_name=genotype_matrix, phenotype_matrix_
↪name=phenotype_matrix,
    phenotype=phenotype, n_outerfolds=5, n_innerfolds=n_innerfolds,val_set_size_
↪percentage=20,
    test_set_size_percentage=test_set_size_percentage, maf_percentage=maf_percentage, n_
↪trials=n_trials,
    save_final_model=False, batch_size=None, n_epochs=10000, task=task,
    models_start_time=models_start_time, current_model_name=current_model_name,␣
↪dataset=dataset
)
```

Finally, we just need to call the method `run()` of our `easypheno.optimization.optuna_optim.OptunaOptim` object to start the Bayesian hyperparameter search, which will print the current progress and return a dictionary with summary results.

[17]:
```
summary_results = optim_run.run_optuna_optimization()
pprint.PrettyPrinter(depth=4).pprint(summary_results)
```

```
[I 2022-10-04 16:08:39,294] A new study created in RDB with name: 2022-10-04_16-08-33_x_
↪matrix-y_matrix-continuous_values-MAF0-SPLITcv-test5-20-MODELxgboost-TRIALS10
```

```
Params for Trial 0
{'n_estimators': 2500, 'learning_rate': 0.07500000000000001, 'max_depth': 3, 'gamma':␣
↪300, 'subsample': 0.45, 'colsample_bytree': 0.35000000000000003, 'reg_alpha': 290.0}
```

(continues on next page)

(continued from previous page)

```
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
[I 2022-10-04 16:08:46,932] Trial 0 finished with value: 369.22728277444065 and␣
↪parameters: {'n_estimators': 2500, 'learning_rate': 0.07500000000000001, 'max_depth':␣
↪3, 'gamma': 300, 'subsample': 0.45, 'colsample_bytree': 0.35000000000000003, 'reg_alpha
↪': 290.0}. Best is trial 0 with value: 369.22728277444065.
Params for Trial 1
{'n_estimators': 3000, 'learning_rate': 0.3, 'max_depth': 9, 'gamma': 300, 'subsample':␣
↪0.1, 'colsample_bytree': 0.55, 'reg_alpha': 440.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
[I 2022-10-04 16:08:59,323] Trial 1 finished with value: 454.8753047090302 and␣
↪parameters: {'n_estimators': 3000, 'learning_rate': 0.3, 'max_depth': 9, 'gamma': 300,
↪'subsample': 0.1, 'colsample_bytree': 0.55, 'reg_alpha': 440.0}. Best is trial 0 with␣
↪value: 369.22728277444065.
Params for Trial 2
{'n_estimators': 2250, 'learning_rate': 0.2, 'max_depth': 10, 'gamma': 80, 'subsample':␣
↪0.2, 'colsample_bytree': 0.05, 'reg_alpha': 320.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
[I 2022-10-04 16:09:11,364] Trial 2 finished with value: 398.20160514328586 and␣
↪parameters: {'n_estimators': 2250, 'learning_rate': 0.2, 'max_depth': 10, 'gamma': 80,
↪'subsample': 0.2, 'colsample_bytree': 0.05, 'reg_alpha': 320.0}. Best is trial 0 with␣
↪value: 369.22728277444065.
Params for Trial 3
{'n_estimators': 2000, 'learning_rate': 0.225, 'max_depth': 8, 'gamma': 770, 'subsample':
↪ 0.1, 'colsample_bytree': 0.3, 'reg_alpha': 110.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
[I 2022-10-04 16:09:24,604] Trial 3 finished with value: 369.6724074334373 and␣
↪parameters: {'n_estimators': 2000, 'learning_rate': 0.225, 'max_depth': 8, 'gamma':␣
↪770, 'subsample': 0.1, 'colsample_bytree': 0.3, 'reg_alpha': 110.0}. Best is trial 0␣
↪with value: 369.22728277444065.
Params for Trial 4
{'n_estimators': 1750, 'learning_rate': 0.25, 'max_depth': 6, 'gamma': 520, 'subsample':␣
↪0.35000000000000003, 'colsample_bytree': 0.05, 'reg_alpha': 100.0}
```

(continues on next page)

(continued from previous page)

```
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
```

```
[I 2022-10-04 16:09:31,163] Trial 4 finished with value: 364.574017351775 and parameters:
↪ {'n_estimators': 1750, 'learning_rate': 0.25, 'max_depth': 6, 'gamma': 520, 'subsample
↪': 0.35000000000000003, 'colsample_bytree': 0.05, 'reg_alpha': 100.0}. Best is trial 4␣
↪with value: 364.574017351775.
```

```
Params for Trial 5
{'n_estimators': 2750, 'learning_rate': 0.2, 'max_depth': 9, 'gamma': 810, 'subsample':␣
↪0.15000000000000002, 'colsample_bytree': 0.75000000000000001, 'reg_alpha': 540.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
```

```
[I 2022-10-04 16:09:40,337] Trial 5 finished with value: 452.7427816744316 and␣
↪parameters: {'n_estimators': 2750, 'learning_rate': 0.2, 'max_depth': 9, 'gamma': 810,
↪'subsample': 0.15000000000000002, 'colsample_bytree': 0.75000000000000001, 'reg_alpha':␣
↪540.0}. Best is trial 4 with value: 364.574017351775.
```

```
Params for Trial 6
{'n_estimators': 100, 'learning_rate': 0.3, 'max_depth': 4, 'gamma': 520, 'subsample': 0.
↪6000000000000001, 'colsample_bytree': 0.3, 'reg_alpha': 980.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
```

```
[I 2022-10-04 16:09:46,464] Trial 6 finished with value: 447.1631958152293 and␣
↪parameters: {'n_estimators': 100, 'learning_rate': 0.3, 'max_depth': 4, 'gamma': 520,
↪'subsample': 0.6000000000000001, 'colsample_bytree': 0.3, 'reg_alpha': 980.0}. Best is␣
↪trial 4 with value: 364.574017351775.
```

```
Params for Trial 7
{'n_estimators': 50, 'learning_rate': 0.3, 'max_depth': 4, 'gamma': 670, 'subsample': 0.
↪6500000000000001, 'colsample_bytree': 0.2, 'reg_alpha': 730.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
```

```
[I 2022-10-04 16:09:49,729] Trial 7 finished with value: 420.049507545245 and parameters:
↪ {'n_estimators': 50, 'learning_rate': 0.3, 'max_depth': 4, 'gamma': 670, 'subsample':␣
↪0.6500000000000001, 'colsample_bytree': 0.2, 'reg_alpha': 730.0}. Best is trial 4 with␣
↪value: 364.574017351775.
```

```
Params for Trial 8
{'n_estimators': 1000, 'learning_rate': 0.2, 'max_depth': 3, 'gamma': 690, 'subsample':␣
↪0.35000000000000003, 'colsample_bytree': 0.75000000000000001, 'reg_alpha': 130.0}
```

(continues on next page)

(continued from previous page)

```
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
[I 2022-10-04 16:09:56,208] Trial 8 finished with value: 373.9128450040709 and␣
↪parameters: {'n_estimators': 1000, 'learning_rate': 0.2, 'max_depth': 3, 'gamma': 690,
↪'subsample': 0.35000000000000003, 'colsample_bytree': 0.7500000000000001, 'reg_alpha':␣
↪130.0}. Best is trial 4 with value: 364.574017351775.
Params for Trial 9
{'n_estimators': 250, 'learning_rate': 0.125, 'max_depth': 5, 'gamma': 730, 'subsample':␣
↪0.7500000000000001, 'colsample_bytree': 0.7500000000000001, 'reg_alpha': 780.0}
# Processing innerfold_0 #
# Processing innerfold_1 #
# Processing innerfold_2 #
# Processing innerfold_3 #
# Processing innerfold_4 #
[I 2022-10-04 16:10:00,982] Trial 9 finished with value: 419.9258576836199 and␣
↪parameters: {'n_estimators': 250, 'learning_rate': 0.125, 'max_depth': 5, 'gamma': 730,
↪ 'subsample': 0.7500000000000001, 'colsample_bytree': 0.7500000000000001, 'reg_alpha':␣
↪780.0}. Best is trial 4 with value: 364.574017351775.
## Optuna Study finished ##
Study statistics:
  Finished trials:  10
  Pruned trials:  0
  Completed trials:  10
  Best Trial:  4
  Value:  364.574017351775
  Params:
    colsample_bytree: 0.05
    gamma: 520
    learning_rate: 0.25
    max_depth: 6
    n_estimators: 1750
    reg_alpha: 100.0
    subsample: 0.35000000000000003
## Retrain best model and test ##
## Results on test set ##
{'test_mse': 393.8179197745938, 'test_rmse': 19.84484617664228, 'test_r2_score': 0.
↪05719242798046553, 'test_explained_variance': 0.05724940832753167}
{'Test': {'best_params': {'colsample_bytree': 0.05,
                          'gamma': 520,
                          'learning_rate': 0.25,
                          'max_depth': 6,
                          'n_estimators': 1750,
                          'reg_alpha': 100.0,
                          'subsample': 0.35000000000000003},
          'eval_metrics': {'test_explained_variance': 0.05724940832753167,
                           'test_mse': 393.8179197745938,
                           'test_r2_score': 0.05719242798046553,
```

(continues on next page)

(continued from previous page)

```
                        'test_rmse': 19.84484617664228},
         'runtime_metrics': {'process_time_max': 36.19791426699999,
                             'process_time_mean': 20.51563152270001,
                             'process_time_min': 1.4952742340000214,
                             'process_time_std': 13.181602909370394,
                             'real_time_max': 11.987335681915283,
                             'real_time_mean': 7.454001760482788,
                             'real_time_min': 3.102874517440796,
                             'real_time_std': 3.14674639916681}}}
```

Beyond that, `easypheno.optimization.optuna_optim.OptunaOptim` creates and saves the `Results_overview_*.csv` files, which we show above in this tutorial.

Further information

This notebooks shows how the use the easyPheno pip package to run an optimization. Furthermore, we give an overview of the individual steps within `optim_pipeline.run()`.

For more information on specific topcis, see the following links:

- Documentation of the whole package
- easyPheno's GitHub repository
- Prepare your data according to our format: Data Guide
- The Installation Guide as well as basic tutorial for the Docker workflow as an alternative
- Summarize and analyze prediction results with easyPeno: HowTo: Summarize prediction results with easyPheno
- Several advanced topics such as adjusting existing prediction models or creation of new ones

2.3.3 HowTo: Summarize prediction results with easyPheno

In the subpackage postprocess, we included functions to analyze optimization results. We provide scripts to run each of these functions (prefix *run_*) with our *Docker workflow*, on which we will also focus in this tutorial. If you want to use the functions directly (e.g. with the pip installed package), please check the scripts and see which functions are called.

Optimization results in easyPheno are saved using the following directory structure: `user_defined_save_dir/results/name_genotype_matrix/name_phenotyp_matrix/name_phenotype/`. By running `run_summarize_results.py`, you can accumulate all optimization results for a genotype matrix:

```
python3 -m easypheno.postprocess.run_summarize_results -rd path_at_name_
↪genotype_matrix_level
```

This leads to the creation of the summary files described in summarize_results_per_phenotype_and_datasplit().

Using a `Results_summary_all_phenotypes*DATASPLIT-PATTERN*.csv` file created by the command above, we provide scripts to visualize the results of several prediction models on different phenotypes:

```
python3 -m easypheno.postprocess.run_plot_results -rsp path_to_Results_summary_
↪all_phenotypes_XX.csv -sd path_to_save_directory
```

This creates a heatmap plot, which is stored at the specified save directory. Currently, heatmaps are implemented, and we can easily add more plot functions.

Additional analysis for simulated phenotypes

In addition, the subpackage simulate contains results analysis functions, which are only applicable for our simulated phenotypes (see *Synthetic data*).

For simulated phenotypes, we know the ground truth in terms of markers respective features, which influence the phenotypic value. Based on that, we are able to compare these effect sizes with feature importances to analyze how well an algorithm captures the relevant features.

To this end, we conduct a statistical as well as visual analysis, which we further describe in the following publication:

> **A comparison of classical and machine learning-based phenotype prediction methods on simulated data and three plant species**
> Maura John, Florian Haselbeck, Rupashree Dass, Christoph Malisi, Patrizia Ricca, Christian Dreischer, Sebastian J. Schultheiss and Dominik G. Grimm
> *Frontiers in Plant Science, 2022 (currently in press)*

The files to do that can be generated with the following command:

```
python3 -m easypheno.simulate.run_results_analysis_synthetic_data -rd path_at_
↪name_genotype_matrix_level -simd path_to_simulation_configs -sd path_to_save_
↪directory
```

Besides .csv-files with statistical information, a scatter plot visualizing feature importances in comparison with effect sizes is created.

2.3.4 Advanced Topics

The following subpackages contain information on advanced topics:

Code walkthrough video

In the subsequent video, we outline the structure and code of easyPheno:

https://youtu.be/5Yb_FdtU0aU

HowTo: Adjust existing prediction models and their hyperparameters

Every easyPheno prediction model based on BaseModel needs to implement several methods. Most of them are already implemented in SklearnModel, TorchModel and TensorflowModel. So if you make use of these, a prediction model only has to implement define_model() and define_hyperparams_to_tune(). We will therefore focus on these two methods in this tutorial.

If you want to create your own model, see *HowTo: Integrate your own prediction model*.

We already integrated several prediction models (see *Prediction Models*), e.g. LinearRegression and Mlp, which we will use for demonstration purposes in this HowTo.

Besides the written documentation, we recorded the tutorial video shown below with similar content.

Adjust prediction model

If you want to adjust the prediction model itself, you can change its definition in its implementation of `define_model()`. Let's discuss an example using LinearRegression:

```
def define_model(self):
    # Penalty term is fixed to l1, but might also be optimized
    penalty = 'l1'  # self.suggest_hyperparam_to_optuna('penalty')
    if penalty == 'l1':
        l1_ratio = 1
    elif penalty == 'l2':
        l1_ratio = 0
    else:
        l1_ratio = self.suggest_hyperparam_to_optuna('l1_ratio')
    if self.task == 'classification':
        reg_c = self.suggest_hyperparam_to_optuna('C')
        return sklearn.linear_model.LogisticRegression(penalty=penalty, C=reg_
↪c, solver='saga',
                                                       l1_ratio=l1_ratio if␣
↪penalty == 'elasticnet' else None,
                                                       max_iter=10000, random_
↪state=42, n_jobs=-1)
    else:
        alpha = self.suggest_hyperparam_to_optuna('alpha')
        return sklearn.linear_model.ElasticNet(alpha=alpha, l1_ratio=l1_ratio,␣
↪max_iter=10000, random_state=42)
```

You can change the `penalty` term that is actually used by setting the related variable to a fixed value or suggest it as a hyperparameter for tuning (see below for information on how to add or adjust a hyperparameter or its range). The same applies for further hyperparameters such as `reg_c` and `alpha`. Beyond that, you could also adjust currently fixed parameters such as `max_iter`.

Another example can be found in Mlp:

```
def define_model(self) -> torch.nn.Sequential:
    n_layers = self.suggest_hyperparam_to_optuna('n_layers')
    model = []
    act_function = self.get_torch_object_for_string(string_to_get=self.suggest_
↪hyperparam_to_optuna('act_function'))
    in_features = self.n_features
    out_features = int(in_features * self.suggest_hyperparam_to_optuna('n_
↪initial_units_factor'))
    p = self.suggest_hyperparam_to_optuna('dropout')
    perc_decrease = self.suggest_hyperparam_to_optuna('perc_decrease_per_layer
↪')
    for layer in range(n_layers):
        model.append(torch.nn.Linear(in_features=in_features, out_features=out_
↪features))
        model.append(act_function)
        model.append(torch.nn.BatchNorm1d(num_features=out_features))
        model.append(torch.nn.Dropout(p=p))
        in_features = out_features
        out_features = int(in_features * (1-perc_decrease))
    model.append(torch.nn.Linear(in_features=in_features, out_features=self.n_
↪outputs))
```

(continues on next page)

(continued from previous page)

```
    return torch.nn.Sequential(*model)
```

Currently, the model consists of `n_layers` of a sequence of a `Linear()`, `BatchNorm()` and `Dropout()` layer, finally followed by a `Linear()` output layer. You can easily adjust this by e.g. adding further layers or setting `n_layers` to a fixed value. Furthermore, the dropout rate `p` is optimized during hyperparameter search and the same rate is used for each Dropout layer. You could set this to a fixed value or suggest a different value for each `Dropout()` layer (e.g. by suggesting it via `self.suggest_hyperparam_to_optuna('dropout')` within the `for`-loop). Some hyperparameters are already defined in TorchModel.common_hyperparams() , which you can directly use here in its child class. Furthermore, some of them are already suggested in TorchModel.

Beyond that, you can also change the complete architecture of the model if you prefer to do so, e.g. by copying the file and adding your changes there (see also *HowTo: Integrate your own prediction model*).

Adjust hyperparameters

Besides changing the model definition, you can adjust the hyperparameters that are optimized as well as their ranges. To set a hyperparameter to a fixed value, comment its suggestion and directly set a value, as described above. If you want to optimize a hyperparameter which is currently set to a fixed value, do it the other way round. If the hyperparameter is not yet defined in `define_hyperparams_to_tune()` (or `common_hyperparams()` in case of TorchModel and TensorflowModel), you have to add it to `define_hyperparams_to_tune()`.

Let's have a look at an example using Mlp:

```
def define_hyperparams_to_tune(self) -> dict:
    n_layers = {
        'datatype': 'int',
        'lower_bound': 1,
        'upper_bound': 5
    }
    n_initial_units_factor = {
        # Number of units in the first linear layer in relation to the number␣
↪of inputs
        'datatype': 'float',
        'lower_bound': 0.1,
        'upper_bound': 0.7,
        'step': 0.05
    }
    perc_decrease_per_layer = {
        # Percentage decrease of the number of units per layer
        'datatype': 'float',
        'lower_bound': 0.1,
        'upper_bound': 0.5,
        'step': 0.05
    }
    if self.n_features > 20000:
        n_layers = {
            'datatype': 'int',
            'lower_bound': 1,
            'upper_bound': 5
        }
        n_initial_units_factor = {
            # Number of units in the first linear layer in relation to the␣
↪number of inputs
```

(continues on next page)

(continued from previous page)

```
            'datatype': 'float',
            'lower_bound': 0.1,
            'upper_bound': 0.3,
            'step': 0.01
        }
        perc_decrease_per_layer = {
            # Percentage decrease of the number of units per layer
            'datatype': 'float',
            'lower_bound': 0.1,
            'upper_bound': 0.5,
            'step': 0.05
        }
    if self.n_features > 50000:
        n_layers = {
            'datatype': 'int',
            'lower_bound': 1,
            'upper_bound': 3
        }
        n_initial_units_factor = {
            # Number of units in the first linear layer in relation to the␣
↪number of inputs
            'datatype': 'float',
            'lower_bound': 0.01,
            'upper_bound': 0.15,
            'step': 0.01
        }
        perc_decrease_per_layer = {
            # Percentage decrease of the number of units per layer
            'datatype': 'float',
            'lower_bound': 0.2,
            'upper_bound': 0.5,
            'step': 0.05
        }

    return {
        'n_layers': n_layers,
        'n_initial_units_factor': n_initial_units_factor,
        'perc_decrease_per_layer': perc_decrease_per_layer
    }
```

There are multiple options to define a hyperparameter in easyPheno, see define_hyperparams_to_tune() for more information regarding the format. In the example above, three parameters are optimized depending on the number of features, besides the ones which are defined in the parent class TorchModel in common_hyperparams(). The method has to return a dictionary. So if you want to add a further hyperparameter, you need to add it to the dictionary with its name as the key and a dictionary defining its characteristics such as the `datatype` and `lower_bound` in case of a float or int as the value. If you only want to change the range of an existing hyperparameter, you can just change the values in this method.

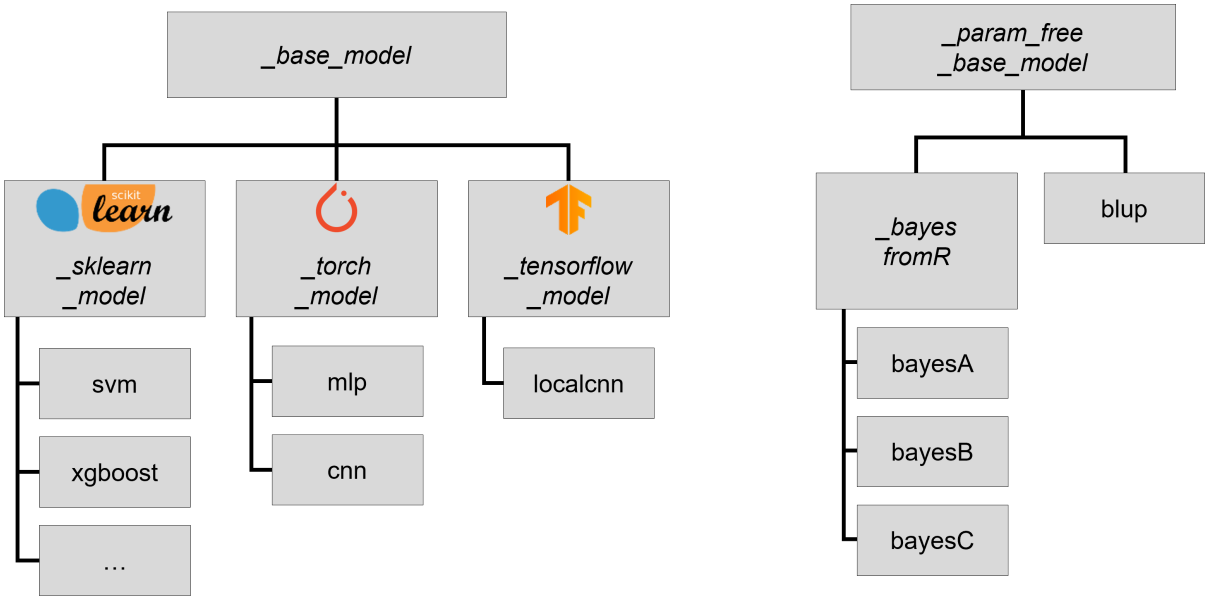
HowTo: Integrate your own prediction model

In this tutorial, we will show you how to integrate your own new prediction model into easyPheno using an example. We recommend to first watch the *Code walkthrough video* for a better understanding of easyPheno's structure.

We further recorded a *Video tutorial: Integrate new model*, which is embedded below .

Overview

The design of the model class makes easyPheno easily extendable with new prediction models. The subsequent figure gives an overview on its structure.



All prediction models are either based on BaseModel or ParamFreeBaseModel in case your model does not contain hyperparameters for optimization (or you do not want to optimize any). These two base models define some methods that are common for all prediction models as well as all methods that each prediction model needs to implement. In this tutorial, we focus on BaseModel. easyPheno already contains child classes of BaseModel implementing some of its obligatory methods for TensorFlow, PyTorch and sklearn. As a consequence, adding a new prediction model based on one of these three very common machine learning frameworks only requires the definition of two attributes and implementation of two methods, which makes easyPheno easy extendable.

An example: Integrating k-nearest-neighbors

We provide template files for all three frameworks and focus on TemplateSklearnModel for the remainder of this tutorial:

```
import sklearn

from . import _sklearn_model


class TemplateSklearnModel(_sklearn_model.SklearnModel):
    """
    Template file for a prediction model based on :obj:`~easypheno.model._
↪sklearn_model.SklearnModel`

    See :obj:`~easypheno.model._base_model.BaseModel` for more information on␣
↪the attributes.

    **Steps you have to do to add your own model:**

        1. Copy this template file and rename it according to your model (will␣
↪be the name to call it later on on the command line)

        2. Rename the class and add it to *easypheno.model.__init__.py*

        3. Adjust the class attributes if necessary

        4. Define your model in *define_model()*

        5. Define the hyperparameters and ranges you want to use for␣
↪optimization in *define_hyperparams_to_tune()*

        6. Test your new prediction model using toy data
    """
    standard_encoding = ...
    possible_encodings = [...]

    def define_model(self):
        """
        Definition of the actual prediction model.

        Use *param = self.suggest_hyperparam_to_optuna(PARAM_NAME_IN_DEFINE_
↪HYPERPARAMS_TO_TUNE)* if you want to use
        the value of a hyperparameter that should be optimized.
        The function needs to return the model object.

        See :obj:`~easypheno.model._base_model.BaseModel` for more information.
        """
        ...

    def define_hyperparams_to_tune(self) -> dict:
        """
        Define the hyperparameters and ranges you want to optimize.
        Caution: they will only be optimized if you add them via *self.suggest_
↪hyperparam_to_optuna(PARAM_NAME)* in *define_model()*
```

(continues on next page)

(continued from previous page)

```
        See :obj:`~easypheno.model._base_model.BaseModel` for more information␣
↪on the format and options.
        """
        return {
            'example_param_1': {
                'datatype': 'categorical',
                'list_of_values': ['cat', 'dog', 'elephant']
            },
            'example_param_2': {
                'datatype': 'float',
                'lower_bound': 0.05,
                'upper_bound': 0.95,
                'step': 0.05
            },
            'example_param_3': {
                'datatype': 'int',
                'lower_bound': 1,
                'upper_bound': 100
            }
        }
```

As an example, we will integrate k-nearest-neighbors (knn) as a new prediction model, both for classification and regression.

First, we copy the template file into the folder containing easyPheno's subpackage *model* and rename it to *knn.py*. Further, we rename the class within the file to `Knn` and add `"knn"` to `__all__` in easypheno.model.__init__.py.

So with updated comments (including `:obj:` references for linking in the auto-generated API documentation), our file now contains the following code:

```
import sklearn

from . import _sklearn_model


class Knn(_sklearn_model.SklearnModel):
    """
    Implementation of a class for k nearest neighbours regressor respective␣
↪classifier.

    See :obj:`~easypheno.model._base_model.BaseModel` for more information on␣
↪the attributes.
    """
    standard_encoding = ...
    possible_encodings = [...]

    def define_model(self):
        """
        Definition of the actual prediction model.

        See :obj:`~easypheno.model._base_model.BaseModel` for more information.
        """
```

(continues on next page)

(continued from previous page)

```
        ...

    def define_hyperparams_to_tune(self) -> dict:
        """
        Definition of hyperparameters and ranges to optimize.

        See :obj:`~easypheno.model._base_model.BaseModel` for more information␣
↪on the format.
        """
        ...
```

Now we need to define the two attributes and implement the two methods. We will use the standard `'012'` encoding in this case (see here for information on the encodings). Further, we optimize the two hyperparameters `n_neighbors` and `weights`. These need to be suggested to Optuna via `self.suggest_hyperparam_to_optuna(PARAM_NAME` in `define_model()` and defined with their ranges in `define_hyperparams_to_tune()` (see here for more information regarding the format and possible options for hyperparameter definition). Finally, we distinguish between `'classification'` and `'regression'` by using the inherited attribute `self.task`.

```
import sklearn

from . import _sklearn_model


class Knn(_sklearn_model.SklearnModel):
    """
    Implementation of a class for k nearest neighbours regressor respective␣
↪classifier.

    See :obj:`~easypheno.model._base_model.BaseModel` for more information on␣
↪the attributes.
    """
    standard_encoding = '012'
    possible_encodings = ['012']

    def define_model(self):
        """
        Definition of the actual prediction model.

        See :obj:`~easypheno.model._base_model.BaseModel` for more information.
        """
        n_neighbors = self.suggest_hyperparam_to_optuna('n_neighbors')
        weights = self.suggest_hyperparam_to_optuna('weights')
        if self.task == 'classification':
            return sklearn.neighbors.KNeighborsClassifier(n_neighbors=n_
↪neighbors, weights=weights)
        else:
            return sklearn.neighbors.KNeighborsRegressor(n_neighbors=n_
↪neighbors, weights=weights)

    def define_hyperparams_to_tune(self) -> dict:
        """
        Definition of hyperparameters and ranges to optimize.
```

(continues on next page)

(continued from previous page)

```
        See :obj:`~easypheno.model._base_model.BaseModel` for more information␣
↪on the format.
        """
        return {
            'n_neighbors': {
                'datatype': 'int',
                'lower_bound': 2,
                'upper_bound': 50,
                'step': 2
            },
            'weights': {
                'datatype': 'categorical',
                'list_of_values': ['uniform', 'distance']
            }
        }
```

Now we are able to test our new prediction model with toy data by calling `python3 -m easypheno.run` with the option `-mod knn` (see *HowTo: Run easyPheno using Docker*).

This example gives an overview on how to integrate your own prediction model. Feel free to get guidance from existing prediction models as well. We are always happy to welcome new contributors and appreciate if you help improving easyPheno by providing your prediction model.

Video tutorial: Integrate new model

https://youtu.be/UjTSKr9bnVc

HowTo: Reuse optimized model

easyPheno enables the reuse of an optimized model for two use cases:

- Run inference using final model on new data (SNP ids of old and new data (after preprocessing, e.g. MAF and duplicate filtering) have to match exactly)
- Retrain on new data using best hyperparameter combination of a previous optimization run

We provide scripts to run these functions (prefix *run_*) with our *Docker workflow*, on which we will also focus in this tutorial. If you want to use the functions directly (e.g. with the pip installed package), please check the scripts and see which functions are called.

Run inference on new data

The main use case for this is that you get new samples for the same phenotype, which come with the same SNP information, and you want to apply a previously optimized model on them. If the final model was not saved, easyPheno will trigger a retraining using the found hyperparameters and old dataset. To apply this prediction model on new data, the SNPs of the old and new dataset need to match exactly!

To apply a final prediction model on new data, you have to run the following command:

```
python3 -m easypheno.postprocess.run_inference -rd full_path_to_model_results -
↪odd path_to_old_data -ndd path_to_new_data -ngm name_new_genotype_matrix -
↪npm name_new_phenotype_matrix -sd path_to_save_directory
```

(continues on next page)

(continued from previous page)

By doing so, a .csv file containing the predictions on the whole new dataset will be created by applying the final prediction model, eventually after retraining on the old dataset if the final model was not saved.

Retrain on new data

If you want to train a new prediction model with the same hyperparameters found for another dataset, you can call easypheno.postprocess.run_retrain_on_new_data. As this script contains many parameters, we refer to the API documentation respective source code for more information. Similar to applying a final model on new samples, a .csv file with the predictions on that new data will be created.

2.4 Data Guide

To run easyPheno on your data, you need to provide a fully-imputed genotype file and a corresponding phenotype file both stored in the same data directory. easyPheno is designed to work with several genotype and phenotype file types.

2.4.1 Genotype files

Independent of the original file type, when loading it the first time, the genotype data will be saved to the data directory in a unified H5 file with the same prefix as the original genotype file to simplify further processing and future runs. easyPheno accepts the following genotype file types:

HDF5 / H5 / H5PY

The file has to contain the following keys:

- **X_raw:** genotype matrix in IUPAC nucleotide code (i.e. 'A', 'C', 'G', 'T', 'M', 'R', 'W', 'S', 'Y', 'K') with *samples* as *rows* and *markers* as *columns*
- **sample_ids:** vector containing corresponding sample ids in the same order as the rows of the genotype matrix
- **snp_ids:** vector containing the identifiers of all SNPs in the same order as the columns of the genotype matrix

CSV

The *first column* must contain the unique **sample id** for each sample. The *column names* should be the **SNP identifiers**. The *values* should be the **genotype matrix** in IUPAC nucleotide code (i.e. 'A', 'C', 'G', 'T', 'M', 'R', 'W', 'S', 'Y', 'K'), with *samples* as *rows* and *markers* as *columns*

PLINK

To use PLINK data, a **.map** and **.ped** file with the same prefix need to be in the data directory. To run the framework with PLINK files, you can use PREFIX.map or PREFIX.ped as option for the genotype file. (See PLINK for more info on the file type)

binary PLINK

To use binary PLINK data, a **.bed**, **.bim** and **.fam** file with the same prefix need to be in the data directory. To run the framework with binary PLINK files, you can use PREFIX.bed, PREFIX.bim or PREFIX.fam as option for the genotype file. (See PLINK for more info on the file type)

2.4.2 Phenotype file

easyPheno currently only accepts **.csv**, **.pheno** and **.txt** files for the phenotype. For .txt and .pheno files it is assumed that the values are separated by a single space. A phenotype file can contain several phenotypes. The *first column* must always contain the **sample ids** corresponding to the genotype matrix (need not be in the same order). The remaining columns should contain the **phenotype values** with the **phenotype name** as *column name*.

2.4.3 Preprocessing

For each genotype-phenotype combination a separate index file will be created. This file contains the sample indices to quickly re-match the genotype and phenotype matrices as well as datasets with indices for different data splits and minor-allele-frequency filters. This way the data splits are the same for all models. Additionally, the sample ids and minor allele frequencies for all SNPs are stored to easily create new MAF filters and data splits and append to the index file. To test the model on new unseen data, the index file also contains the final SNP ids used by each model, sorted by used encoding and minor-allele-frequency. When first creating the index file, some standard values for the data splits and MAF filters will be used additionally to the values specified by the user. The index file has the following format:

```
'matched_data': {
    'y': matched phenotypic values,
    'matched_sample_ids': sample ids of matched genotype/phenotype,
    'X_index': indices of genotype matrix to redo matching,
    'y_index': indices of phenotype vector to redo matching,
    'ma_frequency': minor allele frequency of each SNP in genotype file
    'final_snp_ids':{
        '{encoding}':{
            'maf_{maf_percentage}_snp_ids'
            }
        }
    }
'maf_filter': {
    'maf_{maf_percentage}': indices of SNPs to delete (with MAF < maf_
↪percentage),
    ...
    }
'datasplits': {
    'nested_cv': {
        '#outerfolds-#innerfolds': {
            'outerfold_0': {
                'innerfold_0': {'train': indices_train, 'val': indices_val},
```

(continues on next page)

(continued from previous page)

```
                ...
                'innerfold_n': {'train': indices_train, 'val': indices_val},
                'test': test_indices
                },
            ...
            'outerfold_m': {
                'innerfold_0': {'train': indices_train, 'val': indices_val},
                ...
                'innerfold_n': {'train': indices_train, 'val': indices_val},
                'test': test_indices
                }
            },
            ...
        }
    'cv-test': {
        '#folds-test_percentage': {
            'outerfold_0': {
                'innerfold_0': {'train': indices_train, 'val': indices_val},
                ...
                'innerfold_n': {'train': indices_train, 'val': indices_val},
                'test': test_indices
                }
            },
            ...
        }
    'train-val-test': {
        'train_percentage-val_percentage-test_percentage': {
            'outerfold_0': {
                'innerfold_0': {'train': indices_train, 'val': indices_val},
                'test': test_indices
                }
            },
            ...
        }
    }
```

2.5 Prediction Models

easyPheno includes various phenotype prediction models, both classical genomic selection approaches as well as machine and deep learning-based methods. In the following pages, we will give some details for all of the currently implemented models. We further included a subpage explaining the Bayesian optimization that we use for our automatic hyperparameter search.

We provide both a workflow running easyPheno with a command line interface using Docker and as a pip package, see the following tutorials for more details:

- *HowTo: Run easyPheno using Docker*
- *HowTo: Use easyPheno as a pip package*

In both cases, you need to select the prediction model you want to run - or also multiple ones within the same optimization run. A specific prediction model can be selected by giving the name of the *.py* file in which it is implemented

(without the *.py* suffix). For instance, if you want to run a Support Vector Machine implemented in *svm.py*, you need to specify *svm*.

easyPheno automatically chooses based on the selected phenotype whether to use the implementation for a classification (discrete trait) or regression (continuous trait) task. All models except the classical genomic selection approaches (RR-BLUP and Bayesian alphabet models) provide an implementation for both cases.

In the following table, we give the keys for all prediction models as well as links to detailed descriptions and the source code:

Table 1: Phenotype Prediction Models

Model	Key in easyPheno	Description	Source Code	Notes
Ridge Regression BLUP	blup	*RR-BLUP*	blup.py	
Bayes A	bayesAfromR	*Bayesian alphabet*	bayesAfromR.py	requires Docker workflow
Bayes B	bayesBfromR	*Bayesian alphabet*	bayesBfromR.py	requires Docker workflow
Bayes C	bayesCfromR	*Bayesian alphabet*	bayesCfromR.py	requires Docker workflow
L1-regularized Linear / Logistic Regression	linearregression	*Linear and Logistic Regression*	linearregression.py	Regularization type in source code adjustable
Elastic Net-regularized Linear / Logistic Regression	elasticnet	*Linear and Logistic Regression*	elasticnet.py	
Support Vector Machine / Regression	svm	*Support Vector Machine / Regression*	svm.py	
Random Forest	randomforest	*Random Forest*	randomforest.py	
XGBoost	xgboost	*XGBoost*	xgboost.py	
Multilayer Perceptron	mlp	*Multilayer Perceptron*	mlp.py	
Convolutional Neural Network	cnn	*Convolutional Neural Network*	cnn.py	
Local Convolutional Neural Network	localcnn	*Local Convolutional Neural Network*	localcnn.py	

If you are interested in adjusting an existing model or its hyperparameters: *HowTo: Adjust existing prediction models and their hyperparameters*.

If you want to integrate your own prediction model: *HowTo: Integrate your own prediction model*.

2.5.1 RR-BLUP

Subsequently, we give details on our implementation of Ridge Regression Best Linear Unbiased Predictor (RR-BLUP), which is a classical genomic selection approach. References for a more detailed theoretical background can be found at the end of this page, which were also used for writing this text.

RR-BLUP is based on a linear mixed model, for which phenotype values $\mathbf{y}$ can be calculated as

$$\mathbf{y} = \beta 1 + \mathbf{X}\mathbf{u} + \epsilon$$

with the overall mean β, the genotype matrix $\mathbf{X}$ with corresponding marker effects $\mathbf{u}$ and the residuals vector ϵ. When fitting the model to the training data, $\beta and mathbf{u}` are determined.

In easyPheno, RR-BLUP is implemented as a child class of ParamFreeBaseModel and is named Blup. As you can see there, the `fit()` method contains the fitting of the model to match the training data. In its current implementation, RR-BLUP can only be used for continuous traits.

References

1. Meuwissen, T. H., Hayes, B. J., & Goddard, M. E. (2001). Prediction of total genetic value using genome-wide dense marker maps. Genetics, 157(4), 1819–1829.

2.5.2 Bayesian alphabet

Subsequently, we give details on our implementation of models from the Bayesian alphabet, namely Bayes A, Bayes B and Bayes C. These are classical genomic selection approaches that are closely related to each other. References for a more detailed theoretical background can be found at the end of this page, which were also used for writing this text.

For our implementation, we used the R package BGLR. By doing so, we apply their efficient implementation and show the integration of an R package using rpy2. As a limitation, the Bayesian alphabet models are only available in the Docker workflow, as the R environment also needs to be set up, which cannot be guaranteed for the Python package workflow. Furthermore, in the current implementation, they can only be used for continuous traits.

Similar to RR-BLUP, phenotype values $\mathbf{y}$ for Bayesian alphabet models are defined according to the following linear mixed model:

$$\mathbf{y} = \beta 1 + \mathbf{X}\mathbf{u} + \epsilon$$

with the overall mean β, the genotype matrix $\mathbf{X}$ with corresponding marker effects $\mathbf{u}$ and the residuals vector ϵ.

In contrast to RR-BLUP, the variance of the residuals is commonly assigned a scaled-inverse Chi-squared distribution. The difference between the models from the Bayesian alphabet is the prior distribution of the marker effects $\mathbf{u}$:

- Bayes A uses a scaled-*t* distribution.
- Bayes B uses a mixture of two scaled-*t* distributions: one with a point of mass at zero and one with a large variance.
- Bayes C uses a mixture of two normal distributions: one with a point of mass at zero and one with a large variance.

As mentioned above, we use the R package BGLR for our implementation. To this end, we created a parent class Bayes_R for all models from the Bayesian alphabet containing the integration of the R package. In the code block below, we show the whole `Bayes_R` class.

```
class Bayes_R(_param_free_base_model.ParamFreeBaseModel):
    standard_encoding = '012'
    possible_encodings = ['101']

    def __init__(self, task: str, model_name: str, encoding: str = None, n_
↪iter: int = 6000, burn_in: int = 1000):
        super().__init__(task=task, encoding=encoding)
        self.model_name = model_name
        self.n_iter = n_iter
        self.burn_in = burn_in
        self.mu = None
```

(continues on next page)

(continued from previous page)

```
        self.beta = None

    def fit(self, X: np.array, y: np.array) -> np.array:
        # import necessary R packages
        base = importr('base')
        BGLR = importr('BGLR')

        # create R objects for X and y
        R_X = robjects.r['matrix'](X, nrow=X.shape[0], ncol=X.shape[1])
        R_y = robjects.FloatVector(y)

        # run BGLR for BayesB
        ETA = base.list(base.list(X=R_X, model=self.model_name))
        fmBB = BGLR.BGLR(y=R_y, ETA=ETA, verbose=True, nIter=self.n_iter,␣
↪burnIn=self.burn_in)

        # save results as numpy arrays
        self.beta = np.asarray(fmBB.rx2('ETA').rx2(1).rx2('b'))
        self.mu = fmBB.rx2('mu')
        return self.predict(X_in=X)

    def predict(self, X_in: np.array) -> np.array:
        return self.mu + np.matmul(X_in, self.beta)
```

The constructor has a parameter `model_name`, which we then use for switching between Bayes A, Bayes B and Bayes C. Furthermore, it contains `n_iter` and `burn_in`, so the number of total and burn in iterations. The whole model fitting can be found in the `fit()` method. There, we first import R and the BGLR package. Then, we call the functions from BGLR to fit the model and return the predicted values. For more information on the specific functions, we refer to the documentation of BGLR.

The implementation of the Bayes A, Bayes B and Bayes C models is then straightforward. In the code block below, we examplary show `BayesA`. As you can see, the class is based on `Bayes_R`. To select a specific method, we only need to call the constructor of the parent class. All other methods are inherited from `Bayes_R` .

```
class BayesA(_bayesfromR.Bayes_R):
    def __init__(self, task: str, encoding: str = None):
        super().__init__(task=task, model_name='BayesA', encoding=encoding)
```

References

1. Meuwissen, T. H., Hayes, B. J., & Goddard, M. E. (2001). Prediction of total genetic value using genome-wide dense marker maps. Genetics, 157(4), 1819–1829.
2. Habier, D., Fernando, R.L., Kizilkaya, K. et al. Extension of the bayesian alphabet for genomic selection. BMC Bioinformatics 12, 186 (2011)
3. Gianola D. (2013). Priors in whole-genome regression: the bayesian alphabet returns. Genetics, 194(3), 573–596.

2.5.3 Linear and Logistic Regression

Subsequently, we give details on the regularized linear respective logistic regression approaches that are integrated in easyPheno. First, we outline the regularized linear regression models, which can be used for predicting continuous traits. Then, we describe the closely related logistic regression approaches suitable for discrete phenotypes. References for a more detailed theoretical background can be found at the end of this page, which were also used for writing this text. For our implementation, we use the machine learning framework scikit-learn, which also provides a user guide for these models.

Regularized linear regression for continuous traits

With respect to regularized linear regressions models, the model weights can be optimized by minimizing the deviation between predicted and true phenotypic values, often with considering an additive penalty term for regularization:

$$\text{argmin}_{\mathbf{w}} \frac{1}{2}|\mathbf{y} - \mathbf{X}^*\mathbf{w}|_2^2 + \alpha\Omega(\mathbf{w})$$

In case of the Least Absolute Shrinkage and Selection Operator, usually abbreviated with LASSO, the L1-norm, so the sum of the absolute value of the weights, is used for regularization. This constraint usually leads to sparse solutions forcing unimportant weights to zero. Intuitively speaking, this can be seen as an automatic feature selection. The L2-norm, also known as the Euclidean norm, is defined as the square root of the summed up quadratic weights. Regularized linear regression using the L2-norm is called Ridge Regression. This penalty term has the effect of grouping correlated features. Elastic Net combines both the L1- and the L2-norm, introducing a further hyperparameter controlling the influence of each of the two parts.

All these three approaches - LASSO, Ridge and Elastic Net Regression - are currently implemented in easyPheno. However, as the feature selection effect of LASSO seems to be profitable considering the often large genotype matrices, LASSO and Elastic Net Regression are configured by a variable for selecting the penalty term.

The following code block shows the implementation of LASSO in linearregression.py.

```
class LinearRegression(_sklearn_model.SklearnModel):
"""
Implementation of a class for Linear respective Logistic Regression.

See :obj:`~easypheno.model._base_model.BaseModel` for more information on the␣
↪attributes.
"""
standard_encoding = '012'
possible_encodings = ['012']

def define_model(self):
    """
    Definition of the actual prediction model.

    See :obj:`~easypheno.model._base_model.BaseModel` for more information.
    """
    # Penalty term is fixed to l1, but might also be optimized
    penalty = 'l1'  # self.suggest_hyperparam_to_optuna('penalty')
    if penalty == 'l1':
        l1_ratio = 1
    elif penalty == 'l2':
        l1_ratio = 0
    else:
```

(continues on next page)

(continued from previous page)

```
        l1_ratio = self.suggest_hyperparam_to_optuna('l1_ratio')
    if self.task == 'classification':
        reg_c = self.suggest_hyperparam_to_optuna('C')
        return sklearn.linear_model.LogisticRegression(penalty=penalty, C=reg_
↪c, solver='saga',
                                                       l1_ratio=l1_ratio if␣
↪penalty == 'elasticnet' else None,
                                                       max_iter=10000, random_
↪state=42, n_jobs=-1)
    else:
        alpha = self.suggest_hyperparam_to_optuna('alpha')
        return sklearn.linear_model.ElasticNet(alpha=alpha, l1_ratio=l1_ratio,␣
↪max_iter=10000, random_state=42)

def define_hyperparams_to_tune(self) -> dict:
    """
    See :obj:`~easypheno.model._base_model.BaseModel` for more information on␣
↪the format.
    """
    return {
        'penalty': {
            'datatype': 'categorical',
            'list_of_values': ['l1', 'l2', 'elasticnet']
        },
        'l1_ratio': {
            'datatype': 'float',
            'lower_bound': 0.05,
            'upper_bound': 0.95,
            'step': 0.05
        },
        'alpha': {
            'datatype': 'float',
            'lower_bound': 10**-3,
            'upper_bound': 10**3,
            'log': True
        },
        'C': {
            'datatype': 'float',
            'lower_bound': 10**-3,
            'upper_bound': 10**3,
            'log': True
        }
    }
```

Currently, we set `penalty='l1'` in `define_model()`, to get the implementation of LASSO (or L1-regularized Logistic Regression). But one could also choose another penalty term or treat its selection as a hyperparameter, see *HowTo: Adjust existing prediction models and their hyperparameters*.

Furthermore, Elastic Net is implemented in a separate file containing very similar code to enable a comparison of Elastic Net and LASSO regression. Its implementation can be found in elasticnet.py.

Regularized logistic regression for discrete traits

In contrast to linear regression that is applied for regression tasks (continuous traits), logistic regression is used for

classification (discrete traits). Logistic regression applies the logistic function to the linear combination of the features and weights to get probability scores and assign a discrete label. The same penalty terms as for regularized linear regression (L1, L2 and Elastic Net) are often included in the cost function that is optimized during training, with similar effects as described above.

We implemented regularized logistic regression in the same classes as linear regression (see linearregression.py and elasticnet.py) and switch between both based on the machine learning task that was detected by easyPheno (see `if self.task == 'classification': ... else: ...` in the code block above).

References

1. Hastie, T., Tibshirani, R., & Friedman, J. H. (2009). The elements of statistical learning: data mining, inference, and prediction. 2nd ed. New York, Springer.
2. Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society: Series B (Methodological), 58(1), 267–288.
3. Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. Journal of the Royal Statistical Society, Series B, 67, 301–320.
4. Pedregosa, F. et al. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825–2830.

2.5.4 Support Vector Machine / Regression

Subsequently, we give details on the Support Vector Machine (SVM) respective Support Vector Regression (SVR) integrated in easyPheno. Depending on the machine learning task that was detected (`'classification'` or `'regression'`), easyPheno automatically switches between SVM and SVR. References for a more detailed theoretical background can be found at the end of this page, which were also used for writing this text. For our implementation, we use the machine learning framework scikit-learn, which also provides a user guide for these models.

Support Vector Machine

A SVM aims to find a hyperplane that optimally separates samples belonging to different classes, i.e. a hyperplane reflecting the maximum margin between the classes. To address the non-linear-separable case, SVM introduces two of its main concepts, namely a soft margin and the kernel trick.

With a so-called soft margin, mis-classifications are tolerated up to a certain degree. Mathematically, this is realized by introducing a penalty term consisting of the so-called slack variables, which give the distance to the corresponding class margin and are set to zero in case of a correct classification. The influence of this penalty term - and consequently "the degree of tolerance" - is controlled by a weighting factor usually called C, which is an important hyperparameter. Larger values for C lead to a stronger penalization of model errors, which might lead to a more accurate model, but is also more prone to overfitting. In contrast, smaller C values put less emphasis on wrong predictions. This has a regularizing effect lowering the risk of overfitting, but could also cause underfitting. The so-called kernel trick enables a transformation via kernel functions into a higher-dimensional space and consequently to find a solution for the separating hyperplane there. Hence, the selection of the kernel function to use as well as the values of the related hyperparameters are important during model optimization.

In the code bock below, you can see our implementation for the SVM respective SVR, which are included in the same class and chosen based on the determined task (`if self.task == 'classification': ... else: ...`). Furthermore, one can see that depending on the suggested `kernel`, we are deciding which further hyperparameters need to be suggested.

```
class SupportVectorMachine(_sklearn_model.SklearnModel):
    """
    Implementation of a class for Support Vector Machine respective Regression.
```

(continues on next page)

(continued from previous page)

```
    See :obj:`~easypheno.model._base_model.BaseModel` for more information on␣
↪the attributes.
    """
    standard_encoding = '012'
    possible_encodings = ['012']

    def define_model(self):
        """
        Definition of the actual prediction model.

        See :obj:`~easypheno.model._base_model.BaseModel` for more information.
        """
        kernel = self.suggest_hyperparam_to_optuna('kernel')
        reg_c = self.suggest_hyperparam_to_optuna('C')
        if kernel == 'poly':
            degree = self.suggest_hyperparam_to_optuna('degree')
            gamma = self.suggest_hyperparam_to_optuna('gamma')
        elif kernel in ['rbf', 'sigmoid']:
            degree = 42  # default
            gamma = self.suggest_hyperparam_to_optuna('gamma')
        elif kernel == 'linear':
            degree = 42  # default
            gamma = 42  # default
        if self.task == 'classification':
            return sklearn.svm.SVC(kernel=kernel, C=reg_c, degree=degree,␣
↪gamma=gamma, random_state=42,
                                   max_iter=1000000)
        else:
            return sklearn.svm.SVR(kernel=kernel, C=reg_c, degree=degree,␣
↪gamma=gamma, max_iter=1000000)

    def define_hyperparams_to_tune(self) -> dict:
        """
        See :obj:`~easypheno.model._base_model.BaseModel` for more information␣
↪on the format.
        """
        return {
            'kernel': {
                'datatype': 'categorical',
                'list_of_values': ['linear', 'poly', 'rbf'],
            },
            'degree': {
                'datatype': 'int',
                'lower_bound': 1,
                'upper_bound': 5
            },
            'gamma': {
                'datatype': 'float',
                'lower_bound': 10**-3,
                'upper_bound': 10**3,
                'log': True
            },
```

(continues on next page)

(continued from previous page)

```
            'C': {
                'datatype': 'float',
                'lower_bound': 10**-3,
                'upper_bound': 10**3,
                'log': True
            }
        }
```

Support Vector Regression

The concept of SVR is pretty similar, but instead of optimizing for a separating hyperplane, the goal is to find a function that is within a certain threshold around the target values of the training samples. Apart from that, similar concepts such as the kernel-trick are used, leading to the same hyperparameters that need to be optimized.

As already mentioned, you can find our implementation in the code block above, as SVM and SVR are integrated in the same class.

References

1. Bishop, Christopher M. (2006). Pattern recognition and machine learning. New York, Springer.
2. Smola, A. J. and Schölkopf, B. (2004). A tutorial on support vector regression. Statistics and computing 14, 199–222.
3. Drucker, H., Chris, Kaufman, B. L., Smola, A., and Vapnik, V. (1997). Support vector regression machines. In Advances in Neural Information Processing Systems 9. vol. 9, 155–161.

2.5.5 Random Forest

Subsequently, we give details on our implementation of Random Forest. Depending on the machine learning task that was detected (`'classification'` or `'regression'`), easyPheno automatically switches between both implementations. References for a more detailed theoretical background can be found at the end of this page, which were also used for writing this text. For our implementation, we use the machine learning framework scikit-learn, which also provides a user guide for Random Forests.

A Random Forest is a so-called ensemble learner, which uses multiple weak learners - in this case Decision Trees - to derive a final prediction. Random Forests use a technique called Bootstrap aggregating, usually abbreviated with Bagging. With this technique, a random subsample with replacement (bootstrap samples) of the whole training data is used to construct each Decision Tree, and finally the predictions of these are aggregated. Beyond that, the algorithm was extended by not using all features for training each weak learner, but also a random subset. The goal of both techniques is to prevent overfitting, which Decision Trees tend to, by decreasing variance of the ensemble.

It is worth to mention that the individual weak learners are trained independent from each other, so their construction can also be parallelized. In the end, the individual predictions need to be combined, for which several approaches exist. For regression tasks, our implementation averages the predictions of all Decision Trees in the ensemble. In case of a classification, the predictions of each weak learner are weighted by the probability estimates and then averaged across the whole ensemble. Finally, the class with the largest averaged probability is predicted.

As you can see in the code block below, we use `RandomForestClassifier` respective `RandomForestRegressor` from scikit-learn. Besides the strategy for determining the `max_features` per Decision Tree, we optimize hyperparameters such as the number of trees in the whole ensemble (`n_estimators`).

```
class RandomForest(_sklearn_model.SklearnModel):
    """
    Implementation of a class for Random Forest.
```

(continues on next page)

(continued from previous page)

```
    See :obj:`~easypheno.model._base_model.BaseModel` for more information on␣
↪the attributes.
    """
    standard_encoding = '012'
    possible_encodings = ['012']

    def define_model(self):
        """
        Definition of the actual prediction model.

        See :obj:`~easypheno.model._base_model.BaseModel` for more information.
        """
        # all hyperparameters defined are suggested for optimization
        params = self.suggest_all_hyperparams_to_optuna()
        # add random_state for reproducibility and n_jobs for multiprocessing
        params.update({'random_state': 42, 'n_jobs': -1})
        if self.task == 'classification':
            return sklearn.ensemble.RandomForestClassifier(**params)
        else:
            return sklearn.ensemble.RandomForestRegressor(**params)

    def define_hyperparams_to_tune(self) -> dict:
        """
        See :obj:`~easypheno.model._base_model.BaseModel` for more information␣
↪on the format.
        """
        return {
            'n_estimators': {
                'datatype': 'categorical',
                'list_of_values': [50, 100, 250, 500, 750, 1000, 1250, 1500,␣
↪1750, 2000, 2250, 2500, 2750, 3000,
                                   3500, 4000, 4500, 5000]
            },
            'min_samples_split': {
                'datatype': 'float',
                'lower_bound': 0.005,
                'upper_bound': 0.2,
                'step': 0.005
            },
            'max_depth': {
                'datatype': 'int',
                'lower_bound': 2,
                'upper_bound': 50,
                'step': 2
            },
            'min_samples_leaf': {
                'datatype': 'float',
                'lower_bound': 0.005,
                'upper_bound': 0.2,
                'step': 0.005
            },
            'max_features': {
```

(continues on next page)

(continued from previous page)

```
                'datatype': 'categorical',
                'list_of_values': ['sqrt', 'log2']
            }
        }
```

For further information on all hyperparameters, we refer to the documentation of scikit-learn: Random Forest Classifier and Random Forest Regressor.

References

1. Breiman, L. (2001). Random forests. Machine Learning 45, 5–32.
2. Hastie, T., Tibshirani, R., & Friedman, J. H. (2009). The elements of statistical learning: data mining, inference, and prediction. 2nd ed. New York, Springer.

2.5.6 XGBoost

Subsequently, we give details on our implementation of extreme gradient Boosting, usually abbreviated with XGBoost. Depending on the machine learning task that was detected (`'classification'` or `'regression'`), easyPheno automatically switches between both implementations. References for a more detailed theoretical background can be found at the end of this page, which were also used for writing this text. For our implementation, we use the library xgboost, which also provides a user guide.

XGBoost applies a technique called Boosting. Similar to Random Forest, XGBoost is also an ensemble learner, i.e. trying to build a strong prediction model based on multiple weak learners. But as a conceptual difference, weak learners in XGBoost are not independent. Instead, they are constructed sequentially, with putting more focus on the errors of the current ensemble for the training of a new weak learner. With Gradient Boosting, the sequential construction of the ensemble is formalized as a gradient descent algorithm on a loss function that needs to be minimized.

In comparison with Bagging, which is employed in Random Forest, Boosting aims to reduce bias instead of variance. This might lead to overfitting, which is aimed to be prevented by certain measures. One example is constraining the weak learners, so e.g. limiting the number of estimators or the depth of the Decision Trees. Further methods against overfitting are similar to concepts of bagging, e.g. using random subsets of samples and features for the training of each weak learner. Besides this, for XGBoost a learning rate shrinking the weights update for correcting ensemble errors during the learning process is typically used.

XGBoost is an efficient implementation that leverages Gradient Boosting, for which further details can be found in the original paper. It has proven its predictive power in many application areas, e.g. in Kaggle competitions.

For XGBoost, we use a specific library that is also available as a Python package. In the code block below, you can see our implementation. In `define_model()`, we distinguish between the `'classification'` and `'regression'` case. Furthermore, we optimize several hyperparameters, such as the number of weak learners (`n_estimators`) or the `learning_rate`. A full explanation of all XGBoost parameters can be found in their documentation: XGBoost Parameter Guide

```
class XgBoost(_sklearn_model.SklearnModel):
    """
    Implementation of a class for XGBoost.

    See :obj:`~easypheno.model._base_model.BaseModel` for more information on␣
↪the attributes.
    """
    standard_encoding = '012'
    possible_encodings = ['012']
```

(continues on next page)

(continued from previous page)

```
    def define_model(self) -> xgboost.XGBModel:
        """
        Definition of the actual prediction model.

        See :obj:`~easypheno.model._base_model.BaseModel` for more information.
        """
        # all hyperparameters defined for XGBoost are suggested for optimization
        params = self.suggest_all_hyperparams_to_optuna()
        # add random_state for reproducibility
        params.update({'random_state': 42, 'reg_lambda': 0})
        if self.task == 'classification':
            # set some parameters to prevent warnings
            params.update({'use_label_encoder': False})
            eval_metric = 'mlogloss' if self.n_outputs > 2 else 'logloss'
            params.update({'eval_metric': eval_metric})
            return xgboost.XGBClassifier(**params)
        else:
            return xgboost.XGBRegressor(**params)

    def define_hyperparams_to_tune(self) -> dict:
        """
        See :obj:`~easypheno.model._base_model.BaseModel` for more information on the format.

        Further params that potentially can be optimized

                'reg_lambda': {
                    'datatype': 'float',
                    'lower_bound': 0,
                    'upper_bound': 1000,
                    'step': 10
                }

        """
        return {
            'n_estimators': {
                'datatype': 'categorical',
                'list_of_values': [50, 100, 250, 500, 750, 1000, 1250, 1500, 1750, 2000, 2250, 2500, 2750, 3000]
            },
            'learning_rate': {
                    'datatype': 'float',
                    'lower_bound': 0.025,
                    'upper_bound': 0.3,
                    'step': 0.025
            },
            'max_depth': {
                'datatype': 'int',
                'lower_bound': 2,
                'upper_bound': 10
            },
```

(continues on next page)

(continued from previous page)

```
            'gamma': {
                'datatype': 'int',
                'lower_bound': 0,
                'upper_bound': 1000,
                'step': 10
            },
            'subsample': {
                'datatype': 'float',
                'lower_bound': 0.05,
                'upper_bound': 0.8,
                'step': 0.05
            },
            'colsample_bytree': {
                'datatype': 'float',
                'lower_bound': 0.05,
                'upper_bound': 0.8,
                'step': 0.05
            },
            'reg_alpha': {
                'datatype': 'float',
                'lower_bound': 0,
                'upper_bound': 1000,
                'step': 10
            }
        }
```

References

1. Chen, T., & Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 785–794). New York, NY, USA: ACM.

2.5.7 Multilayer Perceptron

Subsequently, we give details on our implementation of a Multilayer Perceptron (MLP, also known as feedforward neural network). References for a more detailed theoretical background can be found at the end of this page, which were also used for writing this text. We use PyTorch for our implementation. For more information on specific PyTorch objects that we use, e.g. layers, see the PyTorch documentation.

Some of the methods and attributes relevant for the MLP are already defined in its parent class TorchModel. There, you can e.g. find the epoch- and batch-wise training loop. In the code block below, we show the constructor of TorchModel.

```
class TorchModel(_base_model.BaseModel, abc.ABC):
    def __init__(self, task: str, optuna_trial: optuna.trial.Trial, encoding:␣
↪str = None, n_outputs: int = 1,
                 n_features: int = None, width_onehot: int = None, batch_size:␣
↪int = None, n_epochs: int = None,
                 early_stopping_point: int = None):
        self.all_hyperparams = self.common_hyperparams()  # add␣
↪hyperparameters commonly optimized for all torch models
        self.n_features = n_features
        self.width_onehot = width_onehot  # relevant for models using onehot␣
↪encoding e.g. CNNs
```

(continues on next page)

(continued from previous page)

```
        super().__init__(task=task, optuna_trial=optuna_trial,␣
↪encoding=encoding, n_outputs=n_outputs)
        self.batch_size = \
            batch_size if batch_size is not None else self.suggest_hyperparam_
↪to_optuna('batch_size')
        self.n_epochs = n_epochs if n_epochs is not None else self.suggest_
↪hyperparam_to_optuna('n_epochs')
        self.optimizer = torch.optim.Adam(params=self.model.parameters(),
                                          lr=self.suggest_hyperparam_to_optuna(
↪'learning_rate'))
        self.loss_fn = torch.nn.CrossEntropyLoss() if task == 'classification'␣
↪else torch.nn.MSELoss()
        # self.l1_factor = self.suggest_hyperparam_to_optuna('l1_factor')
        # early stopping if there is no improvement on validation loss for a␣
↪certain number of epochs
        self.early_stopping_patience = self.suggest_hyperparam_to_optuna(
↪'early_stopping_patience')
        self.early_stopping_point = early_stopping_point
        self.device = torch.device('cuda:0' if torch.cuda.is_available() else
↪'cpu')
```

We define attributes and suggest hyperparameters that are relevant for all neural network implementations, e.g. the `optimizer` to use and the `learning_rate` to apply. Some attributes are also set to fixed values, for instance the loss function (`self.loss_fn`) depending on the detected machine learning task. Furthermore, early stopping is parametrized, which we use as a measure to prevent overfitting. With early stopping, the validation loss is monitored and if it does not improve for a certain number of epochs (`self.early_stopping_patience`), the training process is stopped. When working with our MLP implementation, it is important to keep in mind that some relevant code and hyperparameters can also be found in TorchModel.

The definition of the MLP model itself as well as of some specific hyperparameters and ranges can be found in the Mlp class. In the code block below, we show its `define_model()` method. Our MLP model consists of `n_layers` of blocks, which include a `Linear()`, `BatchNorm()` and `Dropout` layer. The last of these blocks is followed by a `Linear()` output layer. The number of outputs in the first layers is defined by a hyperparameter (`n_initial_units_factor`), that is multiplied with the number of inputs. Then, with each of the above-mentioned blocks, the number of outputs decreases by a percentage parameter `perc_decrease`. Further, we use `Dropout` for regularization and define the dropout rate as the hyperparameter `p`. Finally, we transform the list to which we added all network layers into a `torch.nn.Sequential()` object.

```
def define_model(self) -> torch.nn.Sequential:
    """
    Definition of an MLP network.

    Architecture:

        - N_LAYERS of (Linear + BatchNorm + Dropout)
        - Linear output layer

    Number of units in the first linear layer and percentage decrease after␣
↪each may be fixed or optimized.
    """
    n_layers = self.suggest_hyperparam_to_optuna('n_layers')
    model = []
    act_function = self.get_torch_object_for_string(string_to_get=self.suggest_
↪hyperparam_to_optuna('act_function'))
```

(continues on next page)

(continued from previous page)

```
    in_features = self.n_features
    out_features = int(in_features * self.suggest_hyperparam_to_optuna('n_
↪initial_units_factor'))
    p = self.suggest_hyperparam_to_optuna('dropout')
    perc_decrease = self.suggest_hyperparam_to_optuna('perc_decrease_per_layer
↪')
    for layer in range(n_layers):
        model.append(torch.nn.Linear(in_features=in_features, out_features=out_
↪features))
        model.append(act_function)
        model.append(torch.nn.BatchNorm1d(num_features=out_features))
        model.append(torch.nn.Dropout(p=p))
        in_features = out_features
        out_features = int(in_features * (1-perc_decrease))
    model.append(torch.nn.Linear(in_features=in_features, out_features=self.n_
↪outputs))
    return torch.nn.Sequential(*model)
```

The implementations for `'classification'` and `'regression'` just differ by the `out_features` of the output layer (and loss function as you can see in the first code block). `self.n_outputs` is inherited from `BaseModel`, where it is set to 1 for `regression` (one continuous output) or to the number of different classes for `classification`.

References

1. Bishop, Christopher M. (2006). Pattern recognition and machine learning. New York, Springer.
2. Goodfellow, I., Bengio, Y.,, Courville, A. (2016). Deep Learning. MIT Press. Available at https://www.deeplearningbook.org/

2.5.8 Convolutional Neural Network

Subsequently, we give details on our implementation of a Convolutional Neural Network (CNN). References for a more detailed theoretical background can be found at the end of this page, which were also used for writing this text. We use PyTorch for our implementation. For more information on specific PyTorch objects that we use, e.g. layers, see the PyTorch documentation.

For CNN, we one-hot encoded the data, as this data can be easily processed by a CNN. This type of encoding preserves the whole nucleotide information and might thus lead to a smaller information loss than other encodings.

Some of the methods and attributes relevant for the CNN are already defined in its parent class TorchModel. There, you can e.g. find the epoch- and batch-wise training loop. In the code block below, we show the constructor of TorchModel.

```
class TorchModel(_base_model.BaseModel, abc.ABC):
    def __init__(self, task: str, optuna_trial: optuna.trial.Trial, encoding:␣
↪str = None, n_outputs: int = 1,
                 n_features: int = None, width_onehot: int = None, batch_size:␣
↪int = None, n_epochs: int = None,
                 early_stopping_point: int = None):
        self.all_hyperparams = self.common_hyperparams()  # add␣
↪hyperparameters commonly optimized for all torch models
        self.n_features = n_features
        self.width_onehot = width_onehot  # relevant for models using onehot␣
↪encoding e.g. CNNs
        super().__init__(task=task, optuna_trial=optuna_trial,␣
↪encoding=encoding, n_outputs=n_outputs)
```

(continues on next page)

(continued from previous page)

```
        self.batch_size = \
            batch_size if batch_size is not None else self.suggest_hyperparam_
↪to_optuna('batch_size')
        self.n_epochs = n_epochs if n_epochs is not None else self.suggest_
↪hyperparam_to_optuna('n_epochs')
        self.optimizer = torch.optim.Adam(params=self.model.parameters(),
                                          lr=self.suggest_hyperparam_to_optuna(
↪'learning_rate'))
        self.loss_fn = torch.nn.CrossEntropyLoss() if task == 'classification'␣
↪else torch.nn.MSELoss()
        # self.l1_factor = self.suggest_hyperparam_to_optuna('l1_factor')
        # early stopping if there is no improvement on validation loss for a␣
↪certain number of epochs
        self.early_stopping_patience = self.suggest_hyperparam_to_optuna(
↪'early_stopping_patience')
        self.early_stopping_point = early_stopping_point
        self.device = torch.device('cuda:0' if torch.cuda.is_available() else
↪'cpu')
```

We define attributes and suggest hyperparameters that are relevant for all neural network implementations, e.g. the `optimizer` to use and the `learning_rate` to apply. Some attributes are also set to fixed values, for instance the loss function (`self.loss_fn`) depending on the detected machine learning task. Furthermore, early stopping is parametrized, which we use as a measure to prevent overfitting. With early stopping, the validation loss is monitored and if it does not improve for a certain number of epochs (`self.early_stopping_patience`), the training process is stopped. When working with our CNN implementation, it is important to keep in mind that some relevant code and hyperparameters can also be found in TorchModel.

The definition of the CNN model itself as well as of some specific hyperparameters and ranges can be found in the Cnn class. In the code block below, we show its `define_model()` method. Our CNN model consists of `n_layers` of blocks, which include a `Conv1d()`, `BatchNorm()` and `Dropout` layer. The last of these blocks is followed by a `MaxPool1d` and `Flatten` layer. This output is further processed by a `Linear`, `BatchNorm` and `Dropout` layer, before the final `Linear()` output layer. The `kernel_size` and `stride` of the convolutional layers are two important hyperparameters that are optimized. The number of output channels of the first layer (`out_channels`) and the frequency of doubling the number of output channels (`frequency_out_channels_doubling`) are currently set, but can also be defined as an hyperparameter. The number of outputs in the first linear layer after the convolutional blocks is defined by a hyperparameter (`n_initial_units_factor`), that is multiplied with current dimensionality. Further, we use `Dropout` for regularization and define the dropout rate as the hyperparameter `p`. Finally, we transform the list to which we added all network layers into a `torch.nn.Sequential()` object.

```
def define_model(self) -> torch.nn.Sequential:
    """
    Definition of a CNN network.

    Architecture:

        - N_LAYERS of (Conv1d + BatchNorm + Dropout)
        - MaxPool1d, Flatten, Linear, BatchNorm, Dropout
        - Linear output layer

    Kernel sizes for convolutional and max pooling layers may be fixed or␣
↪optimized.
    Same applies for strides, number of output channels of the first␣
↪convolutional layer, dropout rate,
```

(continues on next page)

(continued from previous page)

```
    frequency of a doubling of the output channels and number of units in the␣
↪first linear layer.
    """
    n_layers = self.suggest_hyperparam_to_optuna('n_layers')
    model = []
    act_function = self.get_torch_object_for_string(string_to_get=self.suggest_
↪hyperparam_to_optuna('act_function'))
    in_channels = self.width_onehot
    kernel_size = self.suggest_hyperparam_to_optuna('kernel_size')
    stride = max(1, int(kernel_size * self.suggest_hyperparam_to_optuna(
↪'stride_perc_of_kernel_size')))
    out_channels = 2 ** 2  # self.suggest_hyperparam_to_optuna('initial_out_
↪channels_exp')
    frequency_out_channels_doubling = 2  # self.suggest_hyperparam_to_optuna(
↪'frequency_out_channels_doubling')
    p = self.suggest_hyperparam_to_optuna('dropout')
    for layer in range(n_layers):
        model.append(torch.nn.Conv1d(in_channels=in_channels, out_channels=out_
↪channels,
                                     kernel_size=kernel_size, stride=stride))
        model.append(act_function)
        model.append(torch.nn.BatchNorm1d(num_features=out_channels))
        model.append(torch.nn.Dropout(p))
        in_channels = out_channels
        if ((layer+1) % frequency_out_channels_doubling) == 0:
            out_channels *= 2
    model.append(torch.nn.MaxPool1d(kernel_size=kernel_size))
    model.append(torch.nn.Flatten())
    in_features = torch.nn.Sequential(*model)(torch.zeros(size=(1, self.width_
↪onehot, self.n_features))).shape[1]
    out_features = int(in_features * self.suggest_hyperparam_to_optuna('n_
↪units_factor_linear_layer'))
    model.append(torch.nn.Linear(in_features=in_features, out_features=out_
↪features))
    model.append(act_function)
    model.append(torch.nn.BatchNorm1d(num_features=out_features))
    model.append(torch.nn.Dropout(p))
    model.append(torch.nn.Linear(in_features=out_features, out_features=self.n_
↪outputs))
    return torch.nn.Sequential(*model)
```

The implementations for `'classification'` and `'regression'` just differ by the `out_features` of the output layer (and loss function as you can see in the first code block). `self.n_outputs` is inherited from `BaseModel`, where it is set to 1 for `regression` (one continuous output) or to the number of different classes for `classification`.

References

1. Bishop, Christopher M. (2006). Pattern recognition and machine learning. New York, Springer.
2. Goodfellow, I., Bengio, Y.,, Courville, A. (2016). Deep Learning. MIT Press. Available at https://www.deeplearningbook.org/

2.5.9 Local Convolutional Neural Network

Subsequently, we give details on our implementation of a Local Convolutional Neural Network (LCNN). References for a more detailed theoretical background can be found at the end of this page, which were also used for writing this text. We use TensorFlow for our implementation. For more information on specific TensorFlow objects that we use, e.g. layers, see the TensorFlow documentation.

In contrast to normal convolutional layers, local convolutional layers have region-specific filters with individual weights. In the context of phenotype prediction, marker variants in different regions in the genome might have a completely different influence on the phenotype. The hope is to capture these different effects via the region-specific filters of the local convolutional layers.

For LCNN, we one-hot encoded the data, as this data can be easily processed by a LCNN. This type of encoding preserves the whole nucleotide information and might thus lead to a smaller information loss than other encodings.

Some of the methods and attributes relevant for the LCNN are already defined in its parent class TensorflowModel. There, you can e.g. find the epoch- and batch-wise training loop. In the code block below, we show the constructor of `TensorflowModel`.

```
class TensorflowModel(_base_model.BaseModel, abc.ABC):
    def __init__(self, task: str, optuna_trial: optuna.trial.Trial, encoding:␣
↪str = None, n_outputs: int = 1,
                 n_features: int = None, width_onehot: int = None, batch_size:␣
↪int = None, n_epochs: int = None,
                 early_stopping_point: int = None):
        self.all_hyperparams = self.common_hyperparams()  # add␣
↪hyperparameters commonly optimized for all torch models
        self.n_features = n_features
        self.width_onehot = width_onehot  # relevant for models using onehot␣
↪encoding e.g. CNNs
        super().__init__(task=task, optuna_trial=optuna_trial,␣
↪encoding=encoding, n_outputs=n_outputs)
        self.batch_size = \
            batch_size if batch_size is not None else self.suggest_hyperparam_
↪to_optuna('batch_size')
        self.n_epochs = n_epochs if n_epochs is not None else self.suggest_
↪hyperparam_to_optuna('n_epochs')
        self.optimizer = tf.keras.optimizers.Adam(learning_rate=self.suggest_
↪hyperparam_to_optuna('learning_rate'))
        self.loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_
↪logits=True) if task == 'classification' \
            else tf.keras.losses.MeanSquaredError()
        # early stopping if there is no improvement on validation loss for a␣
↪certain number of epochs
        self.early_stopping_patience = self.suggest_hyperparam_to_optuna(
↪'early_stopping_patience')
        self.early_stopping_point = early_stopping_point
        self.early_stopping_callback = tf.keras.callbacks.EarlyStopping(
            monitor='val_loss', patience=self.early_stopping_patience, mode=
↪'min', restore_best_weights=True,
            min_delta=0.1
        )
        self.model.compile(self.optimizer, loss=self.loss_fn)
```

We define attributes and suggest hyperparameters that are relevant for all neural network implementations, e.g. the `optimizer` to use and the `learning_rate` to apply. Some attributes are also set to fixed values, for instance the

loss function (`self.loss_fn`) depending on the detected machine learning task. Furthermore, early stopping is parametrized, which we use as a measure to prevent overfitting. With early stopping, the validation loss is monitored and if it does not improve for a certain number of epochs (`self.early_stopping_patience`), the training process is stopped. When working with our LCNN implementation, it is important to keep in mind that some relevant code and hyperparameters can also be found in `TensorflowModel`.

The definition of the LCNN model itself as well as of some specific hyperparameters and ranges can be found in the LocalCnn class. In the code block below, we show its `define_model()` method. The architecture of our LCNN model starts with a `LocallyConnected1D` layer, for which the `kernel_size` and `stride` are optimized during hyperparameter search. This layer is followed by a `BatchNormalization`, `Dropout`, `MaxPool` and `Flatten` layer. This output is forwarded to `n_layers` of blocks, which include a `Dense()`, `BatchNormalization()` and `Dropout` layer. The last of these blocks is followed by a `Dense` output layer. The number of outputs in the first `Dense` layer is defined by a hyperparameter (`n_initial_units_factor`), that is multiplied with the number of inputs. Then, with each of the above-mentioned blocks, the number of outputs decreases by a percentage parameter `perc_decrease`. Further, we use `Dropout` for regularization and define the dropout rate as the hyperparameter `p`.

```
def define_model(self) -> tf.keras.Sequential:
    """
    Definition of a LocalCNN network.

    Architecture:

        - LocallyConnected1D, BatchNorm, Dropout, MaxPool1D, Flatten
        - N_LAYERS of (Dense + BatchNorm + Dropout)
        - Dense output layer

    Kernel size for LocallyConnectedLayer and max pooling layer may be fixed␣
↪or optimized.
    Same applies for stride, number of units in the first dense layer and␣
↪percentage decrease after each layer.
    """
    n_layers = self.suggest_hyperparam_to_optuna('n_layers')
    model = tf.keras.Sequential()
    act_function = tf.keras.layers.Activation(self.suggest_hyperparam_to_
↪optuna('act_function'))
    l1_regularizer = None  # tf.keras.regularizers.L1(l1=self.suggest_
↪hyperparam_to_optuna('l1_factor'))
    in_channels = self.width_onehot
    width = self.n_features
    model.add(tf.keras.Input(shape=(width, in_channels)))
    n_filters = 1
    kernel_size = int(2 ** self.suggest_hyperparam_to_optuna('kernel_size_exp
↪'))
    stride = max(1, int(kernel_size * self.suggest_hyperparam_to_optuna(
↪'stride_perc_of_kernel_size')))
    model.add(tf.keras.layers.LocallyConnected1D(filters=n_filters, kernel_
↪size=kernel_size,
                                                 strides=stride,␣
↪activation=None,
                                                 kernel_regularizer=l1_
↪regularizer))
    model.add(act_function)
    model.add(tf.keras.layers.BatchNormalization())
    p = self.suggest_hyperparam_to_optuna('dropout')
```

(continues on next page)

(continued from previous page)

```
    model.add(tf.keras.layers.Dropout(rate=p, seed=42))
    kernel_size_max_pool = 2 ** 4  # self.suggest_hyperparam_to_optuna('maxpool_
↪kernel_size_exp')
    model.add(tf.keras.layers.MaxPool1D(pool_size=kernel_size_max_pool))
    model.add(tf.keras.layers.Flatten())
    n_units = int(model.output_shape[1] * self.suggest_hyperparam_to_optuna('n_
↪initial_units_factor'))
    perc_decrease = self.suggest_hyperparam_to_optuna('perc_decrease_per_layer
↪')
    for layer in range(n_layers):
        model.add(tf.keras.layers.Dense(units=n_units, activation=None,
                                        kernel_regularizer=l1_regularizer))
        model.add(act_function)
        model.add(tf.keras.layers.BatchNormalization())
        model.add(tf.keras.layers.Dropout(rate=p))
        n_units = int(n_units * (1-perc_decrease))
    model.add(tf.keras.layers.Dense(units=self.n_outputs))
    return model
```

The implementations for `'classification'` and `'regression'` just differ by the `units` of the output layer (and loss function as you can see in the first code block). `self.n_outputs` is inherited from `BaseModel`, where it is set to 1 for `regression` (one continuous output) or to the number of different classes for `classification`.

References

1. Bishop, Christopher M. (2006). Pattern recognition and machine learning. New York, Springer.
2. Goodfellow, I., Bengio, Y.,, Courville, A. (2016). Deep Learning. MIT Press. Available at https://www.deeplearningbook.org/
3. Pook, T., Freudenthal, J.A., Korte, A., & Simianer, H. (2020). Using Local Convolutional Neural Networks for Genomic Prediction. Frontiers in Genetics, 11.

2.5.10 Hyperparameter optimization

Subsequently, we give details on our implementation of Bayesian optimization for the automatic hyperparameter search. References for a more detailed theoretical background can be found at the end of this page, which were also used for writing this text. For our implementation, we use the optimization framework Optuna, for which its developers also provide a comprehensive online documentation.

Common hyperparameter optimization methods, e.g. Grid Search or Random Search, do not make use of information gained during the optimization process. However, Bayesian optimization uses this knowledge and tries to direct the hyperparameter search towards more promising parameter candidates. The search is guided by a so-called objective value, which is in the machine learning context usually the performance on validation data. With this objective value, a probability model mapping from parameter candidates to a probability of an objective value can be defined. As a result, the most promising parameters can be selected for further trials. This trial-wise optimization with using existing knowledge makes Bayesian optimization potentially more efficient than Grid or Random Search, despite the computational resources needed for the selection of parameter candidates.

Our implementation can be found in the class OptunaOptim. Besides results saving, the main part of this class can be found in the `objective()` method. This method is called for each new trial. At the beginning of a new trial, a prediction model using the suggested parameter set is defined. Then, we loop over the whole training and validation data to retrieve the objective value. In case of multiple validation sets, we take the mean value.

To improve efficiency, we implemented pruning based on intermediate results - so results on validation sets within the

cross-validation - and stop a trial if the intermediate result is worse than the 80th percentile of previous ones at the same time. The probability that such a parameter set would let to better results in the end is pretty low. Furthermore, we set the number of finished trials before we start with pruning to 20. Besides that, we check for parameter set duplicates, as the implementation of Optuna does not prevent to suggest the same parameters again (if they are most likely the best ones to suggest in the current state). The whole optimization process is saved in a database for debugging purposes.

For more detailed information regarding the objects and functions we use from Optuna, see the Optuna documentation.

References

1. Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A Next-generation Hyperparameter Optimization Framework.
2. Bergstra, J., Bardenet, Ré., Bengio, Y. & Kégl, B. (2011). Algorithms for Hyper-parameter Optimization.
3. Snoek, J., Larochelle, H. & Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms.

2.6 Synthetic data

In this tutorial we will show you how to use easyPheno to create synthetic phenotypes for real genotypes.

Besides the written tutorial, we recorded a *Video tutorial: Synthetic data generation*, which is embedded below.

2.6.1 Additive model

To create synthetic phenotypes, easyPheno uses an additive model

$$\mathbf{y} = \mathbf{X}\beta + \mathbf{Z}\gamma + \epsilon$$

where the phenotype $\mathbf{y}$ is given as the sum of one or more causal markers $\mathbf{X}$ with effect sizes β; random effects $\mathbf{Z}$ with small effect sizes γ drawn from a Gaussian distribution, which simulate the polygenic background; and some noise ϵ.

The noise can either follow a Gaussian distribution or, for skewed phenotypes, a gamma distribution. Additionally, the number of causal markers and of markers used to simulate the polygenic background, as well as the number of samples used for the simulation are adjustable. Further, the heritability, i.e. the amount of variance that can be explained by the polygenic background, and the variance explained by the causal markers can both be altered by the user.

2.6.2 Create synthetic data in easyPheno

To create a synthetic phenotype using the command line, all you need is the path to the folder where your data is stored (`data_dir`) and the name of your genotype matrix (`name_of_genotype_matrix`). Please read our *Data Guide* for more information on the data structure of the genotype matrix.

```
python3 -m easypheno.simulate.run_synthetic_phenotypes --data_dir data_dir --
↪genotype_matrix name_of_genotype_matrix
```

This will create a subfolder `name_of_genotype_matrix` within the `data_dir` and save two files, where each simulation gets a unique number or ID (sim_id) to distinguish them from each other:

Simulation_{sim_id}.csv Contains the sample IDs corresponding to the genotype matrix, a column for the simulated phenotype (e.g. `sim1`) and one column with the same phenotype but shifted to get rid of negative values (`sim1_shift`)

Simulations_Overview.csv Contains the sim_id and additional information such as number of samples, number of causal SNPs, etc. for each simulation

And within another subfolder `sim_configs` three files containing additional information:

simulation_config_{sim_id}.csv Contains detailed information of the phenotype such as the SNP ID and effect size of causal markers.

background_{sim_id}.csv Contains all SNP IDs of the used background markers

betas_background_{sim_id}.csv Contains the effect size for each background marker in the same order as the background SNPs

Per default easyPheno creates synthetic phenotypes with 1000 samples, and 1000 markers to simulate the polygenic background with a heritability of 70%, i.e. such that the background accounts for 70% of the phenotypic variance. To change that you can specify the number of samples (`--number_of_samples`), number of background markers (`--number_background_snps`) and heritability (`--heritability`). For example

```
python3 -m easypheno.simulate.run_synthetic_phenotypes --data_dir data_dir --
↪genotype_matrix name_of_genotype_matrix --number_of_samples 100 --number_
↪background_snps 200 --heritability 50
```

will create a phenotype with 100 samples and use 200 markers to simulate the background with a heritability of 50%.

easyPheno will use one causal marker for the synthetic phenotypes that explains 30% of the total variance. You can adjust that by specifying the number of causal markers (`--number_causal_snps`) and the explained variance (`--explained_variance`). For example

```
python3 -m easypheno.simulate.run_synthetic_phenotypes --data_dir data_dir --
↪genotype_matrix name_of_genotype_matrix --number_causal_snps 5 --explained_
↪variance 20
```

will create a phenotype with 5 causal markers that together explain around 20% of the total phenotypic variance.

It is also possible to simulate phenotypes with a skewed distribution by using the flag `--distribution 'gamma'`. If you use a gamma distribution you can additionally adjust the shape parameter with `-shape`.

If you want to create several phenotypes with the same specifications at once, you can specify the number of simulations with `--number_of_simulations`. Then the corresponding sim_id will contain the number of the first and last simulation, e.g. '10-15' for the six simulations '10', '11', '12', '13', '14', '15'.

To get an overview over the other options you can adjust when creating synthetic phenotypes with easyPheno, just use:

```
python3 -m easypheno.simulate.run_synthetic_phenotypes --help
```

2.6.3 Video tutorial: Synthetic data generation

https://youtu.be/pMLzgD1hJnM

2.7 easypheno

2.7.1 Subpackages

easypheno.evaluation

Submodules

easypheno.evaluation.eval_metrics

Module Contents

Functions

get_evaluation_report(y_pred, y_true, task, prefix = '')	Get values for common evaluation metrics

easypheno.evaluation.eval_metrics.**get_evaluation_report**(*y_pred*, *y_true*, *task*, *prefix=''*)

Get values for common evaluation metrics

Parameters

- **y_pred** (*numpy.array*) – predicted values
- **y_true** (*numpy.array*) – true values
- **task** (*str*) – ML task to solve
- **prefix** (*str*) – prefix to be added to the key if multiple eval metrics are collected

Returns dictionary with common metrics

Return type dict

easypheno.model

Submodules

easypheno.model._base_model

Module Contents

Classes

BaseModel	BaseModel parent class for all models that can be used within the framework.

class easypheno.model._base_model.**BaseModel**(*task*, *optuna_trial*, *encoding=None*, *n_outputs=1*)

Bases: abc.ABC

BaseModel parent class for all models that can be used within the framework.

Every model must be based on *BaseModel* directly or BaseModel's child classes, e.g. *SklearnModel* or *TorchModel*

Please add `super().__init__(PARAMS)` to the constructor in case you override it in a child class

Attributes

Class attributes

- standard_encoding (*str*): the standard encoding for this model
- possible_encodings (*List<str>*): a list of all encodings that are possible according to the model definition

Instance attributes

- task (*str*): ML task ('regression' or 'classification') depending on target variable
- optuna_trial (*optuna.trial.Trial*): trial of optuna for optimization
- encoding (*str*): the encoding to use (standard encoding or user-defined)
- n_outputs (*int*): number of outputs of the prediction model
- all_hyperparams (*dict*): dictionary with all hyperparameters with related info that can be tuned (structure see *define_hyperparams_to_tune*)
- model: model object

Parameters

- **task** (*str*) – ML task (regression or classification) depending on target variable
- **optuna_trial** (*optuna.trial.Trial*) – Trial of optuna for optimization
- **encoding** (*str*) – the encoding to use (standard encoding or user-defined)
- **n_outputs** (*int*) – number of outputs of the prediction model

property standard_encoding(*cls*)

the standard encoding for this model

property possible_encodings(*cls*)

a list of all encodings that are possible according to the model definition

abstract define_model(*self*)

Method that defines the model that needs to be optimized. Hyperparams to tune have to be specified in all_hyperparams and suggested via suggest_hyperparam_to_optuna(). The hyperparameters have to be included directly in the model definiton to be optimized. e.g. if you want to optimize the number of layers, do something like

```
n_layers = self.suggest_hyperparam_to_optuna('n_layers') # same name in␣
↪define_hyperparams_to_tune()
for layer in n_layers:
    do something
```

Then the number of layers will be optimized by optuna.

abstract define_hyperparams_to_tune(*self*)

Method that defines the hyperparameters that should be tuned during optimization and their ranges. Required format is a dictionary with:

```
{
    'name_hyperparam_1':
        {
        # MANDATORY ITEMS
        'datatype': 'float' | 'int' | 'categorical',
        FOR DATATYPE 'categorical':
            'list_of_values': []  # List of all possible values
        FOR DATATYPE ['float', 'int']:
            'lower_bound': value_lower_bound,
            'upper_bound': value_upper_bound,
            # OPTIONAL ITEMS (only for ['float', 'int']):
            'log': True | False  # sample value from log domain or not
            'step': step_size # step of discretization.
                              # Caution: cannot be combined with log=True
                                            # - in case of 'float' in␣
↪general and
                                            # - for step!=1 in case of 'int'
        },
    'name_hyperparam_2':
        {
        ...
        },
    ...
    'name_hyperparam_k':
        {
        ...
        }
}
```

If you want to use a similar hyperparameter multiple times (e.g. Dropout after several layers), you only need to specify the hyperparameter once. Individual parameters for every suggestion will be created.

Return type dict

abstract retrain(*self*, *X_retrain*, *y_retrain*)

Method that runs the retraining of the model

Parameters

- **X_retrain** (*numpy.array*) – feature matrix for retraining
- **y_retrain** (*numpy.array*) – target vector for retraining

abstract predict(*self*, *X_in*)

Method that predicts target values based on the input X_in

Parameters **X_in** (*numpy.array*) – feature matrix as input

Returns numpy array with the predicted values

Return type numpy.array

abstract train_val_loop(*self*, *X_train*, *y_train*, *X_val*, *y_val*)

Method that runs the whole training and validation loop

Parameters

- **X_train** (*numpy.array*) – feature matrix for the training
- **y_train** (*numpy.array*) – target vector for training
- **X_val** (*numpy.array*) – feature matrix for validation
- **y_val** (*numpy.array*) – target vector for validation

Returns predictions on validation set

Return type numpy.array

suggest_hyperparam_to_optuna(*self*, *hyperparam_name*)

Suggest a hyperparameter of hyperparam_dict to the optuna trial to optimize it.

If you want to add a parameter to your model / in your pipeline to be optimized, you need to call this method

Parameters **hyperparam_name** (*str*) – name of the hyperparameter to be tuned (see *define_hyperparams_to_tune*)

Returns suggested value

suggest_all_hyperparams_to_optuna(*self*)

Some models accept a dictionary with the model parameters. This method suggests all hyperparameters in all_hyperparams and gives back a dictionary containing them.

Returns dictionary with suggested hyperparameters

Return type dict

save_model(*self*, *path*, *filename*)

Persist the whole model object on a hard drive (can be loaded with *load_model*)

Parameters

- **path** (*pathlib.Path*) – path where the model will be saved
- **filename** (*str*) – filename of the model

easypheno.model._bayesfromR

Module Contents

Classes

Bayes_R	Implementation of a class for Bayesian alphabet.

class easypheno.model._bayesfromR.**Bayes_R**(*task*, *model_name*, *encoding=None*, *n_iter=6000*, *burn_in=1000*)

Bases: *easypheno.model._param_free_base_model.ParamFreeBaseModel*

Implementation of a class for Bayesian alphabet.

Attributes

Inherited attributes

See *ParamFreeBaseModel* for more information on the attributes.

Additional attributes

- mu (*np.array*): intercept
- beta (*np.array*): effect size
- model_name (*str*): model to use (BayesA, BayesB or BayesC)
- n_iter (*int*): iterations for sampling
- burn_in (*int*): warmup/burnin for sampling

Parameters

- **task** (*str*) –
- **model_name** (*str*) –
- **encoding** (*str*) –
- **n_iter** (*int*) –
- **burn_in** (*int*) –

standard_encoding = 012

possible_encodings = ['101']

fit(*self*, *X*, *y*)

Implementation of fit function for Bayesian alphabet imported from R.

See *ParamFreeBaseModel* for more information.

Parameters

- **X** (*numpy.array*) –
- **y** (*numpy.array*) –

Return type numpy.array

predict(*self*, *X_in*)

Implementation of predict function for Bayesian alphabet model imported from R.

See *ParamFreeBaseModel* for more information.

Parameters **X_in** (*numpy.array*) –

Return type numpy.array

easypheno.model._bayesian_linreg

Module Contents

Classes

Bayes	Implementation of a class for Bayesian linear regression.

class easypheno.model._bayesian_linreg.**Bayes**(*task*, *encoding=None*, *iterations=100*, *warmup=10*)

Bases: *easypheno.model._param_free_base_model.ParamFreeBaseModel*

Implementation of a class for Bayesian linear regression.

Attributes

Inherited attributes

See *ParamFreeBaseModel* for more information on the attributes.

Additional attributes

- mu (*np.array*): intercept
- beta (*np.array*): effect size
- iterations (*int*): MCMC sampler iterations
- warmup (*int*): number of discarded MCMC warmup iterations

Parameters

- **task** (*str*) –
- **encoding** (*str*) –
- **iterations** (*int*) –
- **warmup** (*int*) –

standard_encoding = 012

possible_encodings = ['101']

abstract probability_model(*self*, *X*, *y*)

Probability model that needs to be implemented by each child model.

Parameters

- **X** (*torch.Tensor*) – feature matrix
- **y** (*torch.Tensor*) – target vector

fit(*self*, *X*, *y*)

Implementation of fit function for Bayesian linear regression.

See *ParamFreeBaseModel* for more information.

Parameters

- **X** (*numpy.array*) –
- **y** (*numpy.array*) –

Return type numpy.array

predict(*self*, *X_in*)

Implementation of predict function for Bayesian linear regression.

See *ParamFreeBaseModel* for more information.

Parameters **X_in** (*numpy.array*) –

Return type numpy.array

easypheno.model._model_functions

Module Contents

Functions

load_retrain_model(path, filename, X_retrain, y_retrain, early_stopping_point = None)	Load and retrain persisted model
retrain_model_with_results_file(results_file_path, model_name, datasplit, outerfold_number, dataset, saved_outerfold_number = None, saved_datasplit = None)	Retrain a model based on information saved in a results overview file.
load_model(path, filename)	Load persisted model

easypheno.model._model_functions.**load_retrain_model**(*path*, *filename*, *X_retrain*, *y_retrain*, *early_stopping_point=None*)

Load and retrain persisted model

Parameters

- **path** (*pathlib.Path*) – path where the model is saved
- **filename** (*str*) – filename of the model
- **X_retrain** (*numpy.array*) – feature matrix for retraining
- **y_retrain** (*numpy.array*) – target vector for retraining
- **early_stopping_point** (*int*) – optional early stopping point relevant for some models

Returns model instance

Return type *easypheno.model._base_model.BaseModel*

easypheno.model._model_functions.**retrain_model_with_results_file**(*results_file_path*, *model_name*, *datasplit*, *outerfold_number*, *dataset*, *saved_outerfold_number=None*, *saved_datasplit=None*)

Retrain a model based on information saved in a results overview file.

Parameters

- **results_file_path** (*pathlib.Path*) – path to the results overview file containing the information for retraining
- **model_name** (*str*) – name of the model to retrain
- **datasplit** (*str*) – datasplit to use
- **outerfold_number** (*int*) – outerfold number to use
- **dataset** (easypheno.preprocess.base_dataset.Dataset) – dataset for training
- **saved_outerfold_number** (*int*) – outerfold number of the results file in case the model is retrained with fixed hyperparameters of one outerfold
- **saved_datasplit** (*str*) – outerfold number of the results file in case the model is retrained on another datasplit

Returns retrained model

Return type *easypheno.model._base_model.BaseModel*

easypheno.model._model_functions.**load_model**(*path*, *filename*)

Load persisted model

Parameters

- **path** (*pathlib.Path*) – path where the model is saved
- **filename** (*str*) – filename of the model

Returns model instance

Return type *easypheno.model._base_model.BaseModel*

easypheno.model._param_free_base_model

Module Contents

Classes

ParamFreeBaseModel	BaseModel parent class for all models that do not have hyperparameters, e.g. BLUP.

class easypheno.model._param_free_base_model.**ParamFreeBaseModel**(*task*, *encoding=None*)

Bases: abc.ABC

BaseModel parent class for all models that do not have hyperparameters, e.g. BLUP.

Every model must be based on ParamFreeBaseModel directly or ParamFreeBaseModel's child classes.

Please add super().__init__(PARAMS) to the constructor in case you override it in a child class

Attributes

Class attributes

- standard_encoding (*str*): the standard encoding for this model
- possible_encodings (*List<str>*): a list of all encodings that are possible according to the model definition

Instance attributes

- task (*str*): ML task ('regression' or 'classification') depending on target variable
- encoding (*str*): the encoding to use (standard encoding or user-defined)

Parameters

- **task** (*str*) – ML task (regression or classification) depending on target variable
- **encoding** (*str*) – the encoding to use (standard encoding or user-defined)

property standard_encoding(*cls*)

the standard encoding for this model

property possible_encodings(*cls*)

a list of all encodings that are possible according to the model definition

abstract fit(*self*, *X*, *y*)

Method that fits the model based on features X and targets y

Parameters

- **X** (*numpy.array*) – feature matrix for retraining
- **y** (*numpy.array*) – target vector

Returns numpy array with values predicted for X

Return type numpy.array

abstract predict(*self*, *X_in*)

Method that predicts target values based on the input X_in

Parameters **X_in** (*numpy.array*) – feature matrix as input

Returns numpy array with the predicted values

Return type numpy.array

save_model(*self*, *path*, *filename*)

Persist the whole model object on a hard drive (can be loaded with *load_model*)

Parameters

- **path** (*pathlib.Path*) – path where the model will be saved
- **filename** (*str*) – filename of the model

easypheno.model._sklearn_model

Module Contents

Classes

SklearnModel	Parent class based on *BaseModel* for all models with a sklearn-like API to share

class easypheno.model._sklearn_model.**SklearnModel**(*task*, *optuna_trial*, *encoding=None*, *n_outputs=1*)

Bases: *easypheno.model._base_model.BaseModel*, abc.ABC

Parent class based on *BaseModel* for all models with a sklearn-like API to share functionalities. See *BaseModel* for more information.

Attributes

Inherited attributes

See *BaseModel*

Parameters

- **task** (*str*) –
- **optuna_trial** (*optuna.trial.Trial*) –

- **encoding** (*str*) –
- **n_outputs** (*int*) –

retrain(*self*, *X_retrain*, *y_retrain*)

Implementation of the retraining for models with sklearn-like API. See *BaseModel* for more information.

Parameters

- **X_retrain** (*numpy.array*) –
- **y_retrain** (*numpy.array*) –

predict(*self*, *X_in*)

Implementation of a prediction based on input features for models with sklearn-like API. See *BaseModel* for more information.

Parameters **X_in** (*numpy.array*) –

Return type numpy.array

train_val_loop(*self*, *X_train*, *y_train*, *X_val*, *y_val*)

Implementation of a train and validation loop for models with sklearn-like API. See *BaseModel* for more information.

Parameters

- **X_train** (*numpy.array*) –
- **y_train** (*numpy.array*) –
- **X_val** (*numpy.array*) –
- **y_val** (*numpy.array*) –

Return type numpy.array

easypheno.model._template_sklearn_model

Module Contents

Classes

TemplateSklearnModel	Template file for a prediction model based on *SklearnModel*

class easypheno.model._template_sklearn_model.**TemplateSklearnModel**(*task*, *optuna_trial*, *encoding=None*, *n_outputs=1*)

Bases: *easypheno.model._sklearn_model.SklearnModel*

Template file for a prediction model based on *SklearnModel*

See *BaseModel* for more information on the attributes.

Steps you have to do to add your own model:

1. Copy this template file and rename it according to your model (will be the name to call it later on on the command line)

2. Rename the class and add it to *easypheno.model.__init__.py*
3. Adjust the class attributes if necessary
4. Define your model in *define_model()*
5. Define the hyperparameters and ranges you want to use for optimization in *define_hyperparams_to_tune()*
6. Test your new prediction model using toy data

Parameters

- **task** (*str*) –
- **optuna_trial** (*optuna.trial.Trial*) –
- **encoding** (*str*) –
- **n_outputs** (*int*) –

standard_encoding = Ellipsis

possible_encodings = [Ellipsis]

define_model(*self*)

Definition of the actual prediction model.

Use *param = self.suggest_hyperparam_to_optuna(PARAM_NAME_IN_DEFINE_HYPERPARAMS_TO_TUNE)* if you want to use the value of a hyperparameter that should be optimized. The function needs to return the model object.

See *BaseModel* for more information.

define_hyperparams_to_tune(*self*)

Define the hyperparameters and ranges you want to optimize. Caution: they will only be optimized if you add them via *self.suggest_hyperparam_to_optuna(PARAM_NAME)* in *define_model()*

See *BaseModel* for more information on the format and options.

Return type dict

easypheno.model._template_tensorflow_model

Module Contents

Classes

TemplateTensorflowModel	Template file for a prediction model based on *TensorflowModel*

class easypheno.model._template_tensorflow_model.**TemplateTensorflowModel**(*task*, *optuna_trial*, *encoding=None*, *n_outputs=1*, *n_features=None*, *width_onehot=None*, *batch_size=None*, *n_epochs=None*, *early_stopping_point=None*)

Bases: *easypheno.model._tensorflow_model.TensorflowModel*

Template file for a prediction model based on *TensorflowModel*

See *BaseModel* and *TensorflowModel* for more information on the attributes.

Steps you have to do to add your own model:

1. Copy this template file and rename it according to your model (will be the name to call it later on on the command line)
2. Rename the class and add it to *easypheno.model.__init__.py*
3. Adjust the class attributes if necessary
4. Define your model in *define_model()*
5. Define the hyperparameters and ranges you want to use for optimization in *define_hyperparams_to_tune()*.

   CAUTION: Some hyperparameters are already defined in *common_hyperparams()*, which you can directly use here. Some of them are already suggested in `TensorflowModel`.
6. Test your new prediction model using toy data

Parameters

- **task** (*str*) –
- **optuna_trial** (*optuna.trial.Trial*) –
- **encoding** (*str*) –
- **n_outputs** (*int*) –
- **n_features** (*int*) –
- **width_onehot** (*int*) –
- **batch_size** (*int*) –
- **n_epochs** (*int*) –
- **early_stopping_point** (*int*) –

standard_encoding = Ellipsis

possible_encodings = [Ellipsis]

define_model(*self*)

Definition of the actual prediction model.

Use *param = self.suggest_hyperparam_to_optuna(PARAM_NAME_IN_DEFINE_HYPERPARAMS_TO_TUNE)* if you want to use the value of a hyperparameter that should be optimized. The function needs to return the model object.

See *BaseModel* for more information.

define_hyperparams_to_tune(*self*)

Define the hyperparameters and ranges you want to optimize. Caution: they will only be optimized if you add them via *self.suggest_hyperparam_to_optuna(PARAM_NAME)* in *define_model()*

See *BaseModel* for more information on the format and options.

Check *TensorflowModel* for already defined (and for some cases also suggested) hyperparameters.

Return type dict

easypheno.model._template_torch_model

Module Contents

Classes

TemplateTorchModel	Template file for a prediction model based on *TorchModel*

class easypheno.model._template_torch_model.**TemplateTorchModel**(*task*, *optuna_trial*, *encoding=None*, *n_outputs=1*, *n_features=None*, *width_onehot=None*, *batch_size=None*, *n_epochs=None*, *early_stopping_point=None*)

Bases: *easypheno.model._torch_model.TorchModel*

Template file for a prediction model based on *TorchModel*

See *BaseModel* and *TorchModel* for more information on the attributes.

Steps you have to do to add your own model:

1. Copy this template file and rename it according to your model (will be the name to call it later on on the command line)
2. Rename the class and add it to *easypheno.model.__init__.py*
3. Adjust the class attributes if necessary
4. Define your model in *define_model()*
5. Define the hyperparameters and ranges you want to use for optimization in *define_hyperparams_to_tune()*.

   CAUTION: Some hyperparameters are already defined in *common_hyperparams()*, which you can directly use here. Some of them are already suggested in *TorchModel*.
6. Test your new prediction model using toy data

Parameters

- **task** (*str*) –
- **optuna_trial** (*optuna.trial.Trial*) –
- **encoding** (*str*) –
- **n_outputs** (*int*) –
- **n_features** (*int*) –
- **width_onehot** (*int*) –
- **batch_size** (*int*) –
- **n_epochs** (*int*) –
- **early_stopping_point** (*int*) –

standard_encoding = Ellipsis

possible_encodings = [Ellipsis]

define_model(*self*)

Definition of the actual prediction model.

Use *param = self.suggest_hyperparam_to_optuna(PARAM_NAME_IN_DEFINE_HYPERPARAMS_TO_TUNE)* if you want to use the value of a hyperparameter that should be optimized. The function needs to return the model object.

See *BaseModel* for more information.

define_hyperparams_to_tune(*self*)

Define the hyperparameters and ranges you want to optimize. Caution: they will only be optimized if you add them via *self.suggest_hyperparam_to_optuna(PARAM_NAME)* in *define_model()*

See *BaseModel* for more information on the format and options.

Check *TorchModel* for already defined (and for some cases also suggested) hyperparameters.

Return type dict

easypheno.model._tensorflow_model

Module Contents

Classes

TensorflowModel	Parent class based on *BaseModel* for all TensorFlow models to share functionalities.

class easypheno.model._tensorflow_model.**TensorflowModel**(*task*, *optuna_trial*, *encoding=None*, *n_outputs=1*, *n_features=None*, *width_onehot=None*, *batch_size=None*, *n_epochs=None*, *early_stopping_point=None*)

Bases: *easypheno.model._base_model.BaseModel*, abc.ABC

Parent class based on *BaseModel* for all TensorFlow models to share functionalities. See *BaseModel* for more information.

Attributes

Inherited attributes

See *BaseModel*.

Additional attributes

- n_features (*int*): Number of input features to the model
- width_onehot (*int*): Number of input channels in case of onehot encoding
- batch_size (*int*): Batch size for batch-based training
- n_epochs (*int*): Number of epochs for optimization
- optimizer (*tf.keras.optimizers.Optimizer*): optimizer for model fitting

- loss_fn: loss function for model fitting
- early_stopping_patience (*int*): epochs without improvement before early stopping
- early_stopping_point (*int*): epoch at which early stopping occured
- early_stopping_callback (*tf.keras.callbacks.EarlyStopping*): callback for early stopping

Parameters

- **task** (*str*) – ML task (regression or classification) depending on target variable
- **optuna_trial** (*optuna.trial.Trial*) – optuna.trial.Trial : trial of optuna for optimization
- **encoding** (*str*) – the encoding to use (standard encoding or user-defined)
- **n_features** (*int*) – Number of input features to the model
- **width_onehot** (*int*) – Number of input channels in case of onehot encoding
- **batch_size** (*int*) – Batch size for batch-based training
- **n_epochs** (*int*) – Number of epochs for optimization
- **early_stopping_point** (*int*) – Stop training at defined epoch
- **n_outputs** (*int*) –

train_val_loop(*self*, *X_train*, *y_train*, *X_val*, *y_val*)

Implementation of a train and validation loop for TensorFlow models. See *BaseModel* for more information

Parameters

- **X_train** (*numpy.array*) –
- **y_train** (*numpy.array*) –
- **X_val** (*numpy.array*) –
- **y_val** (*numpy.array*) –

Return type numpy.array

retrain(*self*, *X_retrain*, *y_retrain*)

Implementation of the retraining for PyTorch models. See *BaseModel* for more information

Parameters

- **X_retrain** (*numpy.array*) –
- **y_retrain** (*numpy.array*) –

predict(*self*, *X_in*)

Implementation of a prediction based on input features for PyTorch models. See *BaseModel* for more information

Parameters **X_in** (*numpy.array*) –

Return type numpy.array

get_dataloader(*self*, *X*, *y=None*, *shuffle=True*)

Get a dataloader using the specified data and batch size

Parameters

- **X** (*numpy.array*) – feature matrix to use

- **y** (*numpy.array*) – optional target vector to use
- **shuffle** (*bool*) – shuffle parameter for DataLoader

Returns batched dataset

Return type tensorflow.data.Dataset

static common_hyperparams()

Add hyperparameters that are common for PyTorch models. Do not need to be included in optimization for every child model. Also See *BaseModel* for more information

save_model(*self*, *path*, *filename*)

Method to persist the whole model object on a hard drive (can be loaded with *load_model*)

Parameters

- **path** (*pathlib.Path*) – path where the model will be saved
- **filename** (*str*) – filename of the model

easypheno.model._torch_model

Module Contents

Classes

TorchModel	Parent class based on *BaseModel* for all PyTorch models to share functionalities.

class easypheno.model._torch_model.**TorchModel**(*task*, *optuna_trial*, *encoding=None*, *n_outputs=1*, *n_features=None*, *width_onehot=None*, *batch_size=None*, *n_epochs=None*, *early_stopping_point=None*)

Bases: *easypheno.model._base_model.BaseModel*, abc.ABC

Parent class based on *BaseModel* for all PyTorch models to share functionalities. See *BaseModel* for more information.

Attributes

Inherited attributes

See *BaseModel*.

Additional attributes

- n_features (*int*): Number of input features to the model
- width_onehot (*int*): Number of input channels in case of onehot encoding
- batch_size (*int*): Batch size for batch-based training
- n_epochs (*int*): Number of epochs for optimization
- optimizer (*torch.optim.optimizer.Optimizer*): optimizer for model fitting
- loss_fn: loss function for model fitting
- early_stopping_patience (*int*): epochs without improvement before early stopping

- early_stopping_point (*int*): epoch at which early stopping occured
- device (*torch.device*): device to use, e.g. GPU

Parameters

- **task** (*str*) – ML task (regression or classification) depending on target variable
- **optuna_trial** (*optuna.trial.Trial*) – optuna.trial.Trial : trial of optuna for optimization
- **encoding** (*str*) – the encoding to use (standard encoding or user-defined)
- **n_outputs** (*int*) – Number of outputs of the model
- **n_features** (*int*) – Number of input features to the model
- **width_onehot** (*int*) – Number of input channels in case of onehot encoding
- **batch_size** (*int*) – Batch size for batch-based training
- **n_epochs** (*int*) – Number of epochs for optimization
- **early_stopping_point** (*int*) – Stop training at defined epoch

train_val_loop(*self*, *X_train*, *y_train*, *X_val*, *y_val*)

Implementation of a train and validation loop for PyTorch models. See *BaseModel* for more information

Parameters

- **X_train** (*numpy.array*) –
- **y_train** (*numpy.array*) –
- **X_val** (*numpy.array*) –
- **y_val** (*numpy.array*) –

Return type numpy.array

train_one_epoch(*self*, *train_loader*)

Train one epoch

Parameters **train_loader** (*torch.utils.data.DataLoader*) – DataLoader with training data

validate_one_epoch(*self*, *val_loader*)

Validate one epoch

Parameters **val_loader** (*torch.utils.data.DataLoader*) – DataLoader with validation data

Returns loss based on loss-criterion

Return type float

retrain(*self*, *X_retrain*, *y_retrain*)

Implementation of the retraining for PyTorch models. See *BaseModel* for more information

Parameters

- **X_retrain** (*numpy.array*) –
- **y_retrain** (*numpy.array*) –

predict(*self*, *X_in*)

Implementation of a prediction based on input features for PyTorch models. See *BaseModel* for more information

Parameters **X_in** (*numpy.array*) –

Return type numpy.array

get_loss(*self*, *outputs*, *targets*)

Calculate the loss based on the outputs and targets

Parameters

- **outputs** (*torch.Tensor*) – outputs of the model
- **targets** (*torch.Tensor*) – targets of the dataset

Returns loss

Return type torch.Tensor

get_dataloader(*self*, *X*, *y=None*, *shuffle=True*)

Get a Pytorch DataLoader using the specified data and batch size

Parameters

- **X** (*numpy.array*) – feature matrix to use
- **y** (*numpy.array*) – optional target vector to use
- **shuffle** (*bool*) – shuffle parameter for DataLoader

Returns Pytorch DataLoader

Return type torch.utils.data.DataLoader

static common_hyperparams()

Add hyperparameters that are common for PyTorch models. Do not need to be included in optimization for every child model. Also See *BaseModel* for more information

static get_torch_object_for_string(*string_to_get*)

Get the torch object for a specific string, e.g. when suggesting to optuna as hyperparameter

Parameters **string_to_get** (*str*) – string to retrieve the torch object

Returns torch object

easypheno.model.bayesAfromR

Module Contents

Classes

BayesA	Implementation of a class for Bayes A.

class easypheno.model.bayesAfromR.**BayesA**(*task*, *encoding=None*)

Bases: *easypheno.model._bayesfromR.Bayes_R*

Implementation of a class for Bayes A.

Attributes

Inherited attributes

See *Bayes_R* for more information on the attributes.

Parameters

- **task** (*str*) –
- **encoding** (*str*) –

easypheno.model.bayesBfromR

Module Contents

Classes

BayesB	Implementation of a class for Bayes B.

class easypheno.model.bayesBfromR.**BayesB**(*task*, *encoding=None*)

Bases: *easypheno.model._bayesfromR.Bayes_R*

Implementation of a class for Bayes B.

Attributes

Inherited attributes

See *Bayes_R* for more information on the attributes.

Parameters

- **task** (*str*) –
- **encoding** (*str*) –

easypheno.model.bayesCfromR

Module Contents

Classes

BayesC	Implementation of a class for Bayes A.

class easypheno.model.bayesCfromR.**BayesC**(*task*, *encoding=None*)

Bases: *easypheno.model._bayesfromR.Bayes_R*

Implementation of a class for Bayes A.

Attributes

Inherited attributes

See *Bayes_R* for more information on the attributes.

Parameters

- **task** (*str*) –
- **encoding** (*str*) –

easypheno.model.bayes_ridge

Module Contents

Classes

BayesRidge	Implementation of a class for Bayesian Ridge Regression. R implementaiton of Bayes A, B and C is availabile via Docker workflow and weigh faster.

class easypheno.model.bayes_ridge.**BayesRidge**(*task*, *encoding=None*, *iterations=100*, *warmup=10*)

Bases: *easypheno.model._bayesian_linreg.Bayes*

Implementation of a class for Bayesian Ridge Regression. R implementaiton of Bayes A, B and C is availabile via Docker workflow and weigh faster.

Attributes

Inherited attributes

See *Bayes* for more information on the attributes.

Parameters

- **task** (*str*) –
- **encoding** (*str*) –
- **iterations** (*int*) –
- **warmup** (*int*) –

probability_model(*self*, *X*, *y*)

Implementation of probability model for Bayesian ridge regression

See *Bayes* for more information on the attributes.

Parameters

- **X** (*torch.tensor*) –
- **y** (*torch.Tensor*) –

easypheno.model.blup

Module Contents

Classes

Blup	Implementation of a class for BLUP.

class easypheno.model.blup.**Blup**(*task*, *encoding=None*)

Bases: *easypheno.model._param_free_base_model.ParamFreeBaseModel*

Implementation of a class for BLUP.

Attributes

Inherited attributes

See *ParamFreeBaseModel* for more information on the attributes.

Additional attributes

- beta (*np.array*): best linear unbiased estimate (BLUE) of the fixed effects
- u (*np.array*): best linear unbiased prediction (BLUP) of the random effects

Parameters

- **task** (*str*) –
- **encoding** (*str*) –

standard_encoding = 101

possible_encodings = ['101']

static reml(*delta*, *n*, *eigenvalues*, *omega2*)

Function to compute the restricted maximum likelihood

Parameters

- **delta** (*float*) – variance component
- **n** (*int*) – number of samples
- **eigenvalues** (*numpy.array*) – eigenvalues of SHS
- **omega2** (*numpy.array*) – point-wise product of V_SHS*y with itself

Returns restricted maximum likelihood

fit(*self*, *X*, *y*)

Implementation of fit function for BLUP.

See *ParamFreeBaseModel* for more information.

Parameters

- **X** (*numpy.array*) –
- **y** (*numpy.array*) –

Return type numpy.array

predict(*self*, *X_in*)

Implementation of predict function for BLUP.

See *ParamFreeBaseModel* for more information.

Parameters **X_in** (*numpy.array*) –

Return type numpy.array

easypheno.model.cnn

Module Contents

Classes

Cnn	Implementation of a class for a Convolutional Neural Network (CNN).

class easypheno.model.cnn.**Cnn**(*task*, *optuna_trial*, *encoding=None*, *n_outputs=1*, *n_features=None*, *width_onehot=None*, *batch_size=None*, *n_epochs=None*, *early_stopping_point=None*)

Bases: *easypheno.model._torch_model.TorchModel*

Implementation of a class for a Convolutional Neural Network (CNN).

See *BaseModel* and *TorchModel* for more information on the attributes.

Parameters

- **task** (*str*) –
- **optuna_trial** (*optuna.trial.Trial*) –
- **encoding** (*str*) –
- **n_outputs** (*int*) –
- **n_features** (*int*) –
- **width_onehot** (*int*) –
- **batch_size** (*int*) –
- **n_epochs** (*int*) –
- **early_stopping_point** (*int*) –

standard_encoding = onehot

possible_encodings = ['onehot']

define_model(*self*)

Definition of a CNN network.

Architecture:

- N_LAYERS of (Conv1d + BatchNorm + Dropout)
- MaxPool1d, Flatten, Linear, BatchNorm, Dropout
- Linear output layer

Kernel sizes for convolutional and max pooling layers may be fixed or optimized. Same applies for strides, number of output channels of the first convolutional layer, dropout rate, frequency of a doubling of the output channels and number of units in the first linear layer.

Return type torch.nn.Sequential

define_hyperparams_to_tune(*self*)

See *BaseModel* for more information on the format.

See *TorchModel* for more information on hyperparameters common for all torch models.

Return type dict

easypheno.model.elasticnet

Module Contents

Classes

ElasticNet	Implementation of a class for Linear respective Logistic Regression using ElasticNet penalty.

class easypheno.model.elasticnet.**ElasticNet**(*task*, *optuna_trial*, *encoding=None*, *n_outputs=1*)

Bases: *easypheno.model._sklearn_model.SklearnModel*

Implementation of a class for Linear respective Logistic Regression using ElasticNet penalty.

See *BaseModel* for more information on the attributes.

Parameters

- **task** (*str*) –
- **optuna_trial** (*optuna.trial.Trial*) –
- **encoding** (*str*) –
- **n_outputs** (*int*) –

standard_encoding = 012

possible_encodings = ['012']

define_model(*self*)

Definition of the actual prediction model.

See *BaseModel* for more information.

define_hyperparams_to_tune(*self*)

See *BaseModel* for more information on the format.

Return type dict

easypheno.model.linearregression

Module Contents

Classes

LinearRegression	Implementation of a class for Linear respective Logistic Regression.

class easypheno.model.linearregression.**LinearRegression**(*task*, *optuna_trial*, *encoding=None*, *n_outputs=1*)

Bases: *easypheno.model._sklearn_model.SklearnModel*

Implementation of a class for Linear respective Logistic Regression.

See *BaseModel* for more information on the attributes.

Parameters

- **task** (*str*) –
- **optuna_trial** (*optuna.trial.Trial*) –
- **encoding** (*str*) –
- **n_outputs** (*int*) –

standard_encoding = 012

possible_encodings = ['012']

define_model(*self*)

Definition of the actual prediction model.

See *BaseModel* for more information.

define_hyperparams_to_tune(*self*)

See *BaseModel* for more information on the format.

Return type dict

easypheno.model.localcnn

Module Contents

Classes

LocalCnn	Implementation of a class for a Locally-connected Convolutional Neural Network (LocalCNN).

class easypheno.model.localcnn.**LocalCnn**(*task*, *optuna_trial*, *encoding=None*, *n_outputs=1*, *n_features=None*, *width_onehot=None*, *batch_size=None*, *n_epochs=None*, *early_stopping_point=None*)

Bases: *easypheno.model._tensorflow_model.TensorflowModel*

Implementation of a class for a Locally-connected Convolutional Neural Network (LocalCNN).

See *BaseModel* and *TensorflowModel* for more information on the attributes.

Parameters

- **task** (*str*) –
- **optuna_trial** (*optuna.trial.Trial*) –
- **encoding** (*str*) –
- **n_outputs** (*int*) –
- **n_features** (*int*) –
- **width_onehot** (*int*) –
- **batch_size** (*int*) –
- **n_epochs** (*int*) –
- **early_stopping_point** (*int*) –

standard_encoding = onehot

possible_encodings = ['onehot']

define_model(*self*)

Definition of a LocalCNN network.

Architecture:

- LocallyConnected1D, BatchNorm, Dropout, MaxPool1D, Flatten
- N_LAYERS of (Dense + BatchNorm + Dropout)
- Dense output layer

Kernel size for LocallyConnectedLayer and max pooling layer may be fixed or optimized. Same applies for stride, number of units in the first dense layer and percentage decrease after each layer.

Return type tensorflow.keras.Sequential

define_hyperparams_to_tune(*self*)

See *BaseModel* for more information on the format.

See TensorflowModel for more information on hyperparameters common for all tensorflow models.

Return type dict

easypheno.model.mlp

Module Contents

Classes

Mlp	Implementation of a class for a feedforward Multilayer Perceptron (MLP).

class easypheno.model.mlp.**Mlp**(*task*, *optuna_trial*, *encoding=None*, *n_outputs=1*, *n_features=None*, *width_onehot=None*, *batch_size=None*, *n_epochs=None*, *early_stopping_point=None*)

Bases: *easypheno.model._torch_model.TorchModel*

Implementation of a class for a feedforward Multilayer Perceptron (MLP).

See *BaseModel* and *TorchModel* for more information on the attributes.

Parameters

- **task** (*str*) –
- **optuna_trial** (*optuna.trial.Trial*) –
- **encoding** (*str*) –
- **n_outputs** (*int*) –
- **n_features** (*int*) –
- **width_onehot** (*int*) –
- **batch_size** (*int*) –
- **n_epochs** (*int*) –
- **early_stopping_point** (*int*) –

standard_encoding = 012

possible_encodings = ['012']

define_model(*self*)

Definition of an MLP network.

Architecture:

- N_LAYERS of (Linear + BatchNorm + Dropout)
- Linear output layer

Number of units in the first linear layer and percentage decrease after each may be fixed or optimized.

Return type torch.nn.Sequential

define_hyperparams_to_tune(*self*)

See *BaseModel* for more information on the format.

See *TorchModel* for more information on hyperparameters common for all torch models.

Return type dict

easypheno.model.randomforest

Module Contents

Classes

RandomForest	Implementation of a class for Random Forest.

class easypheno.model.randomforest.**RandomForest**(*task*, *optuna_trial*, *encoding=None*, *n_outputs=1*)

Bases: *easypheno.model._sklearn_model.SklearnModel*

Implementation of a class for Random Forest.

See *BaseModel* for more information on the attributes.

Parameters

- **task** (*str*) –
- **optuna_trial** (*optuna.trial.Trial*) –
- **encoding** (*str*) –
- **n_outputs** (*int*) –

standard_encoding = 012

possible_encodings = ['012']

define_model(*self*)

Definition of the actual prediction model.

See *BaseModel* for more information.

define_hyperparams_to_tune(*self*)

See *BaseModel* for more information on the format.

Return type dict

easypheno.model.svm

Module Contents

Classes

SupportVectorMachine	Implementation of a class for Support Vector Machine respective Regression.

class easypheno.model.svm.**SupportVectorMachine**(*task*, *optuna_trial*, *encoding=None*, *n_outputs=1*)

Bases: *easypheno.model._sklearn_model.SklearnModel*

Implementation of a class for Support Vector Machine respective Regression.

See *BaseModel* for more information on the attributes.

Parameters

- **task** (*str*) –
- **optuna_trial** (*optuna.trial.Trial*) –
- **encoding** (*str*) –
- **n_outputs** (*int*) –

standard_encoding = 012

possible_encodings = ['012']

define_model(*self*)

Definition of the actual prediction model.

See *BaseModel* for more information.

define_hyperparams_to_tune(*self*)

See *BaseModel* for more information on the format.

Return type dict

easypheno.model.xgboost

Module Contents

Classes

XgBoost	Implementation of a class for XGBoost.

class easypheno.model.xgboost.**XgBoost**(*task*, *optuna_trial*, *encoding=None*, *n_outputs=1*)

Bases: *easypheno.model._sklearn_model.SklearnModel*

Implementation of a class for XGBoost.

See *BaseModel* for more information on the attributes.

Parameters

- **task** (*str*) –
- **optuna_trial** (*optuna.trial.Trial*) –
- **encoding** (*str*) –
- **n_outputs** (*int*) –

standard_encoding = 012

possible_encodings = ['012']

define_model(*self*)

Definition of the actual prediction model.

See *BaseModel* for more information.

Return type xgboost.XGBModel

define_hyperparams_to_tune(*self*)

See *BaseModel* for more information on the format.

Further params that potentially can be optimized

```
'reg_lambda': {
    'datatype': 'float',
    'lower_bound': 0,
    'upper_bound': 1000,
    'step': 10
}
```

Return type dict

easypheno.optimization

Submodules

easypheno.optimization.optuna_optim

Module Contents

Classes

OptunaOptim	Class that contains all info for the whole optimization using optuna for one model and dataset.

class easypheno.optimization.optuna_optim.**OptunaOptim**(*save_dir*, *genotype_matrix_name*, *phenotype_matrix_name*, *phenotype*, *n_outerfolds*, *n_innerfolds*, *val_set_size_percentage*, *test_set_size_percentage*, *maf_percentage*, *n_trials*, *save_final_model*, *batch_size*, *n_epochs*, *task*, *current_model_name*, *dataset*, *models_start_time*, *intermediate_results_interval=50*, *outerfold_number_to_run=None*)

Class that contains all info for the whole optimization using optuna for one model and dataset.

Attributes

- task (*str*): ML task (regression or classification) depending on target variable
- current_model_name (*str*): name of the current model according to naming of .py file in package model
- dataset (*Dataset*): dataset to use for optimization run
- datasplit_subpath (*str*): subpath with datasplit info relevant for saving / naming
- base_path (*str*): base_path for save_path
- save_path (*str*): path for model and results storing
- study (*optuna.study.Study*): optuna study for optimization run
- current_best_val_result (*float*): the best validation result so far
- early_stopping_point (*int*): point at which early stopping occured (relevant for some models)
- user_input_params (*dict*): all params handed over to the constructor that are needed in the whole class

Parameters

- **save_dir** (*pathlib.Path*) – directory for saving the results.
- **genotype_matrix_name** (*str*) – name of the genotype matrix including datatype ending
- **phenotype_matrix_name** (*str*) – name of the phenotype matrix including datatype ending
- **phenotype** (*str*) – name of the phenotype to predict
- **n_outerfolds** (*int*) – number of outerfolds relevant for nested-cv
- **n_innerfolds** (*int*) – number of folds relevant for nested-cv and cv-test

- **test_set_size_percentage** (*int*) – size of the test set relevant for cv-test and train-val-test
- **val_set_size_percentage** (*int*) – size of the validation set relevant for train-val-test
- **maf_percentage** (*int*) – threshold for MAF filter as percentage value
- **n_trials** (*int*) – number of trials for optuna
- **save_final_model** (*bool*) – specify if the final model should be saved
- **batch_size** (*int*) – batch size for neural network models
- **n_epochs** (*int*) – number of epochs for neural network models
- **task** (*str*) – ML task (regression or classification) depending on target variable
- **current_model_name** (*str*) – name of the current model according to naming of .py file in package model
- **dataset** (easypheno.preprocess.base_dataset.Dataset) – dataset to use for optimization run
- **models_start_time** (*str*) – optimized models and starting time of the optimization run for saving purposes
- **intermediate_results_interval** (*int*) – number of trials after which intermediate results will be saved
- **outerfold_number_to_run** (*int*) – outerfold to run in case you do not want to run all

create_new_study(*self*)

Create a new optuna study.

Returns a new optuna study instance

Return type optuna.study.Study

objective(*self*, *trial*, *train_val_indices*)

Objective function for optuna optimization that returns a score

Parameters

- **trial** (*optuna.trial.Trial*) – trial of optuna for optimization
- **train_val_indices** (*dict*) – indices of train and validation sets

Returns score of the current hyperparameter config

Return type float

clean_up_after_exception(*self*, *trial_number*, *trial_params*, *reason*)

Clean up things after an exception: delete unfitted model if it exists and update runtime csv

Parameters

- **trial_number** (*int*) – number of the trial
- **trial_params** (*dict*) – parameters of the trial
- **reason** (*str*) – hint for the reason of the Exception

write_runtime_csv(*self*, *dict_runtime*)

Write runtime info to runtime csv file

Parameters **dict_runtime** (*dict*) – dictionary with runtime information

calc_runtime_stats(*self*)

Calculate runtime stats for saved csv file.

Returns dict with runtime info enhanced with runtime stats

Return type dict

check_params_for_duplicate(*self*, *current_params*)

Check if params were already suggested which might happen by design of TPE sampler.

Parameters **current_params** (*dict*) – dictionar with current parameters

Returns bool reflecting if current params were already used in the same study

Return type bool

generate_results_on_test(*self*, *outerfold_info*)

Generate the results on the testing data

Parameters **outerfold_info** (*dict*) – dictionary with outerfold datasplit indices

Returns evaluation metrics dictionary

Return type dict

get_feature_importance(*self*, *model*, *X*, *y*, *top_n=1000*, *include_perm_importance=False*)

Get feature importances for models that possess such a feature, e.g. XGBoost

Parameters

- **model** (easypheno.model._base_model.BaseModel) – model to analyze
- **X** (*numpy.array*) – feature matrix for permutation
- **y** (*numpy.array*) – target vector for permutation
- **top_n** (*int*) – top n features to select
- **include_perm_importance** (*bool*) – include permutation based feature importance or not

Returns DataFrame with feature importance information

Return type pandas.DataFrame

run_optuna_optimization(*self*)

Run whole optuna optimization for one model, dataset and datasplit.

Returns dictionary with results overview

Return type dict

easypheno.optimization.paramfree_fitting

Module Contents

Classes

ParamFreeFitting	Class that contains all info for the whole optimization using optuna for one model and dataset.

class easypheno.optimization.paramfree_fitting.**ParamFreeFitting**(*save_dir*, *genotype_matrix_name*, *phenotype_matrix_name*, *phenotype*, *n_outerfolds*, *n_innerfolds*, *val_set_size_percentage*, *test_set_size_percentage*, *maf_percentage*, *save_final_model*, *task*, *current_model_name*, *dataset*, *models_start_time*)

Class that contains all info for the whole optimization using optuna for one model and dataset.

Attributes

- task (*str*): ML task (regression or classification) depending on target variable
- current_model_name (*str*): name of the current model according to naming of .py file in package model
- dataset (*Dataset*): dataset to use for optimization run
- datasplit_subpath (*str*): subpath with datasplit info relevant for saving / naming
- base_path (*str*): base_path for save_path
- save_path (*str*): path for model and results storing
- user_input_params (*dict*): all params handed over to the constructor that are needed in the whole class

Parameters

- **save_dir** (*pathlib.Path*) – directory for saving the results.
- **genotype_matrix_name** (*str*) – name of the genotype matrix including datatype ending
- **phenotype_matrix_name** (*str*) – name of the phenotype matrix including datatype ending
- **phenotype** (*str*) – name of the phenotype to predict
- **n_outerfolds** (*int*) – number of outerfolds relevant for nested-cv
- **n_innerfolds** (*int*) – number of folds relevant for nested-cv and cv-test
- **test_set_size_percentage** (*int*) – size of the test set relevant for cv-test and train-val-test
- **val_set_size_percentage** (*int*) – size of the validation set relevant for train-val-test
- **maf_percentage** (*int*) – threshold for MAF filter as percentage value
- **save_final_model** (*bool*) – specify if the final model should be saved
- **task** (*str*) – ML task (regression or classification) depending on target variable
- **current_model_name** (*str*) – name of the current model according to naming of .py file in package model
- **dataset** (easypheno.preprocess.base_dataset.Dataset) – dataset to use for optimization run
- **models_start_time** (*str*) – optimized models and starting time of the optimization run for saving purposes

run_fitting(*self*)

Run fitting of parameter-free models

Returns dictionary with results overview

write_runtime_csv(*self*, *dict_runtime*)

Write runtime info to runtime csv file

Parameters **dict_runtime** (*dict*) – dictionary with runtime information

get_feature_importance(*self*, *model*, *top_n=1000*)

Get feature importances for models that possess such a feature, e.g. BLUP

Parameters

- **model** (easypheno.model._param_free_base_model.ParamFreeBaseModel) – model to analyze
- **top_n** (*int*) – top n features to select

Returns DataFrame with feature importance information

Return type pandas.DataFrame

easypheno.postprocess

Submodules

easypheno.postprocess.feat_importance

Module Contents

Functions

post_generate_feature_importances(results_directory_genotype_level, data_dir)	Post-generate the feature importances for several models for all sub-folders of the specified directory of already optimized models.

easypheno.postprocess.feat_importance.**post_generate_feature_importances**(*results_directory_genotype_level*, *data_dir*)

Post-generate the feature importances for several models for all sub-folders of the specified directory of already optimized models. Only needed in case you e.g. forgot to implement the saving of the feature importances.

Parameters

- **results_directory_genotype_level** (*str*) – results directory at the level of the name of the genotype matrix
- **data_dir** (*str*) – data directory where the phenotype and genotype matrix as well as index file are stored

easypheno.postprocess.model_reuse

Module Contents

Functions

apply_final_model(results_directory_model, old_data_dir, new_data_dir, new_genotype_matrix, new_phenotype_matrix, save_dir = None)	Apply a final model on a new dataset. It will be applied to the whole dataset.
retrain_on_new_data(results_directory_model, data_dir, genotype_matrix, phenotype_matrix, phenotype, encoding = None, maf_percentage = 0, save_dir = None, datasplit = 'nested-cv', n_outerfolds = 5, n_innerfolds = 5, test_set_size_percentage = 20, val_set_size_percentage = 20, save_final_model = True)	Train a model on a new dataset using the hyperparameters that worked best for the specified model results.

easypheno.postprocess.model_reuse.**apply_final_model**(*results_directory_model*, *old_data_dir*, *new_data_dir*, *new_genotype_matrix*, *new_phenotype_matrix*, *save_dir=None*)

Apply a final model on a new dataset. It will be applied to the whole dataset. So the main purpose of this function is, if you get new samples you want to predict on. If the final model was saved, this will be used for inference on the new dataset. Otherwise, it will be retrained on the initial dataset and then used for inference on the new dataset.

The new dataset will be filtered for the SNP ids that the model was initially trained on.

CAUTION: the SNPs of the old and the new dataset have to be the same!

Parameters

- **results_directory_model** (*str*) – directory that contains the model results that you want to use
- **old_data_dir** (*str*) – directory that contains the data that the model was trained on
- **new_data_dir** (*str*) – directory that contains the new genotype and phenotype matrix
- **new_genotype_matrix** (*str*) – new genotype matrix (incl. file suffix)
- **new_phenotype_matrix** (*str*) – new phenotype matrix (incl. file suffix)
- **save_dir** (*str*) – directory to store the results

easypheno.postprocess.model_reuse.**retrain_on_new_data**(*results_directory_model*, *data_dir*, *genotype_matrix*, *phenotype_matrix*, *phenotype*, *encoding=None*, *maf_percentage=0*, *save_dir=None*, *datasplit='nested-cv'*, *n_outerfolds=5*, *n_innerfolds=5*, *test_set_size_percentage=20*, *val_set_size_percentage=20*, *save_final_model=True*)

Train a model on a new dataset using the hyperparameters that worked best for the specified model results.

Parameters

- **data_dir** (*str*) – data directory where the phenotype and genotype matrix are stored

- **genotype_matrix** (*str*) – name of the genotype matrix including datatype ending
- **phenotype_matrix** (*str*) – name of the phenotype matrix including datatype ending
- **phenotype** (*str*) – name of the phenotype to predict
- **encoding** (*str*) – encoding to use. Default is None, so standard encoding of each model will be used. Options are: '012', 'onehot', 'raw'
- **maf_percentage** (*int*) – threshold for MAF filter as percentage value. Default is 0, so no MAF filtering
- **save_dir** (*str*) – directory for saving the results. Default is None, so same directory as data_dir
- **datasplit** (*str*) – datasplit to use. Options are: nested-cv, cv-test, train-val-test
- **n_outerfolds** (*int*) – number of outerfolds relevant for nested-cv
- **n_innerfolds** (*int*) – number of folds relevant for nested-cv and cv-test
- **test_set_size_percentage** (*int*) – size of the test set relevant for cv-test and train-val-test
- **val_set_size_percentage** (*int*) – size of the validation set relevant for train-val-test
- **save_final_model** (*bool*) – specify if the final model should be saved
- **results_directory_model** (*str*) –

easypheno.postprocess.results_analysis

Module Contents

Functions

summarize_results_per_phenotype_and_datasplit(results_directory_genotype, eval_metric = None)	Summarize the results for each phenotype and datasplit for all models and save in a file.
result_string_to_dictionary(result_string)	Convert result string saved in a .csv file to a dictionary
plot_heatmap_results(path_to_results_summary_csv, save_dir)	Generate a heatmap based on the results summary .csv file

easypheno.postprocess.results_analysis.**summarize_results_per_phenotype_and_datasplit**(*results_directory_genotype*, *eval_metric=None*)

Summarize the results for each phenotype and datasplit for all models and save in a file.

The following files will be created:

- at phenotype-folder level within results directories:
  - Detailed_results_summary_*PHENOTYPE*DATASPLIT-PATTERN*.xlsx: .xlsx-file containing detailed results for each phenotype and datasplit-maf pattern (e.g. with all runtime results etc.)
  - Results_summary**PHENOTYPE*DATASPLIT-PATTERN*.csv: .csv-file containing an overview of the performance of each model for a phenotype and datasplit-maf pattern combination
- at genotype-folder level within results directories (the one that was specified):

- Results_summary_all_phenotypes*DATASPLIT-PATTERN*.xlsx: .xlsx-file containing an overview of the performance of each model on each phenotype used for this genotype matrix with the specified datasplit-maf pattern
- Results_summary_all_phenotypes*EVALMETRIC*DATASPLIT-PATTERN*.csv: only overview sheet of Results_summary*DATASPLIT-PATTERN*.xlsx

Parameters

- **results_directory_genotype_level** (*str*) – results directory at the level of the name of the genotype matrix
- **eval_metric** (*str*) – eval metric to use for summary. Options (default given first): Regression: explained_variance, r2_score, rmse, mse. Classification: mcc, f1_score, accuracy, precision, recall.

easypheno.postprocess.results_analysis.**result_string_to_dictionary**(*result_string*)

Convert result string saved in a .csv file to a dictionary

Parameters **result_string** (*str*) – string from .csv file

Returns dictionary with info from .csv file

Return type dict

easypheno.postprocess.results_analysis.**plot_heatmap_results**(*path_to_results_summary_csv*, *save_dir*)

Generate a heatmap based on the results summary .csv file

Parameters

- **path_to_results_summary_csv** (*str*) – path to the results summary .csv file
- **save_dir** (*str*) – directory to save the plots

easypheno.preprocess

Submodules

easypheno.preprocess.base_dataset

Module Contents

Classes

Dataset	Class containing dataset ready for optimization (e.g. geno/phenotype matched).

class easypheno.preprocess.base_dataset.**Dataset**(*data_dir*, *genotype_matrix_name*, *phenotype_matrix_name*, *phenotype*, *datasplit*, *n_outerfolds*, *n_innerfolds*, *test_set_size_percentage*, *val_set_size_percentage*, *encoding*, *maf_percentage*, *do_snp_filters=True*)

Class containing dataset ready for optimization (e.g. geno/phenotype matched).

Attributes

- encoding (*str*): the encoding to use (standard encoding or user-defined)
- X_full (*numpy.array*): all (matched, maf- and duplicated-filtered) SNPs
- y_full (*numpy.array*): all target values
- sample_ids_full (*numpy.array*):all sample ids
- snp_ids (*numpy.array*): SNP ids
- datasplit (*str*): datasplit to use
- datasplit_indices (*dict*): dictionary containing all indices for the specified datasplit

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored
- **genotype_matrix_name** (*str*) – name of the genotype matrix including datatype ending
- **phenotype_matrix_name** (*str*) – name of the phenotype matrix including datatype ending
- **phenotype** (*str*) – name of the phenotype to predict
- **datasplit** (*str*) – datasplit to use. Options are: nested-cv, cv-test, train-val-test
- **n_outerfolds** (*int*) – number of outerfolds relevant for nested-cv
- **n_innerfolds** (*int*) – number of folds relevant for nested-cv and cv-test
- **test_set_size_percentage** (*int*) – size of the test set relevant for cv-test and train-val-test
- **val_set_size_percentage** (*int*) – size of the validation set relevant for train-val-test
- **maf_percentage** (*int*) – threshold for MAF filter as percentage value
- **encoding** (*str*) – the encoding to use (standard encoding or user-defined)
- **do_snp_filters** (*bool*) – specify if SNP filters (e.g. duplicates, maf etc.) should be applied

load_match_raw_data(*self*, *data_dir*, *genotype_matrix_name*, *do_snp_filters=True*)

Load the full genotype and phenotype matrices specified and match them

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored
- **genotype_matrix_name** (*str*) – name of the genotype matrix including datatype ending
- **do_snp_filters** (*bool*) – specify if SNP filters (e.g. duplicates, maf etc.) should be applied

Returns matched genotype, phenotype and sample ids

Return type (numpy.ndarray, numpy.ndarray, numpy.ndarray)

maf_filter_raw_data(*self*, *data_dir*, *maf_percentage*)

Apply maf filter to full raw data, if maf=0 only non-informative SNPs will be removed

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored

- **maf_percentage** (*int*) – threshold for MAF filter as percentage value

filter_duplicate_snps(*self*)

Remove duplicate SNPs, i.e. SNPs that are completely the same for all samples and therefore do not add information.

check_and_save_filtered_snp_ids(*self*, *data_dir*, *maf_percentage*)

Check if snp_ids for specific maf percentage and encoding are saved in index_file. If not, save them in 'matched_data/final_snp_ids/{encoding}/maf_{maf_percentage}_snp_ids'

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored
- **maf_percentage** (*int*) – threshold for MAF filter as percentage value

load_datasplit_indices(*self*, *data_dir*, *n_outerfolds*, *n_innerfolds*, *test_set_size_percentage*, *val_set_size_percentage*)

Load the datasplit indices saved during file unification.

Structure:

```
{
    'outerfold_0': {
        'innerfold_0': {'train': indices_train, 'val': indices_val},
        'innerfold_1': {'train': indices_train, 'val': indices_val},
        ...
        'innerfold_n': {'train': indices_train, 'val': indices_val},
        'test': test_indices
        },
    ...
    'outerfold_m': {
        'innerfold_0': {'train': indices_train, 'val': indices_val},
        'innerfold_1': {'train': indices_train, 'val': indices_val},
        ...
        'innerfold_n': {'train': indices_train, 'val': indices_val},
        'test': test_indices
        }
}
```

Caution: The actual structure depends on the datasplit specified by the user, e.g. for a train-val-test split only 'outerfold_0' and its subelements 'innerfold_0' and 'test' exist.

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored
- **n_outerfolds** (*int*) – number of outerfolds relevant for nested-cv
- **n_innerfolds** (*int*) – number of folds relevant for nested-cv and cv-test
- **test_set_size_percentage** (*int*) – size of the test set relevant for cv-test and train-val-test
- **val_set_size_percentage** (*int*) – size of the validation set relevant for train-val-test

Returns dictionary with the above-described structure containing all indices for the specified data split

Return type dict

check_datasplit(*self*, *n_outerfolds*, *n_innerfolds*)

Check if the datasplit is valid. Raise Exceptions if train, val or test sets contain same samples.

Parameters

- **n_outerfolds** (*int*) – number of outerfolds in datasplit_indices dictionary
- **n_innerfolds** (*int*) – number of folds in datasplit_indices dictionary

static get_index_file_name(*genotype_matrix_name*, *phenotype_matrix_name*, *phenotype*)

Get the name of the file containing the indices for maf filters and data splits

Parameters

- **genotype_matrix_name** (*str*) – name of the genotype matrix including datatype ending
- **phenotype_matrix_name** (*str*) – name of the phenotype matrix including datatype ending
- **phenotype** (*str*) – name of the phenotype to predict

Returns name of index file

Return type str

easypheno.preprocess.encoding_functions

Module Contents

Functions

get_encoding(models, user_encoding)	Get a list of all required encodings.
get_list_of_encodings()	Get a list of all implemented encodings.
get_base_encoding(encoding)	Check which base encoding is needed to create required encoding.
check_encoding_of_genotype(X)	Check the encoding of the genotype matrix
encode_genotype(X, required_encoding)	Compute the required encoding of the genotype matrix
get_additive_encoding(X, style = '012')	Generate genotype matrix in additive encoding:
get_onehot_encoding(X)	Generate genotype matrix in onehot encoding. If genotype matrix is homozygous, create 3d torch tensor with

easypheno.preprocess.encoding_functions.**get_encoding**(*models*, *user_encoding*)

Get a list of all required encodings.

Parameters

- **models** – models to consider
- **user_encoding** (*str*) – encoding specified by the user

Returns list of encodings

Return type list

easypheno.preprocess.encoding_functions.**get_list_of_encodings**()

Get a list of all implemented encodings.

! Adapt if new encoding is added !

Returns List of all possible encodings

Return type list

easypheno.preprocess.encoding_functions.**get_base_encoding**(*encoding*)

Check which base encoding is needed to create required encoding.

! Adapt if new encoding is added !

Parameters **encoding** (*str*) – required encoding

Returns base encoding

Return type str

easypheno.preprocess.encoding_functions.**check_encoding_of_genotype**(*X*)

Check the encoding of the genotype matrix

! Adapt if new encoding is added !

Parameters **X** (*numpy.array*) – genotype matrix

Returns encoding of the genotype matrix

Return type str

easypheno.preprocess.encoding_functions.**encode_genotype**(*X*, *required_encoding*)

Compute the required encoding of the genotype matrix

! Adapt if new encoding is added !

Parameters

- **X** (*numpy.array*) – genotype matrix
- **required_encoding** (*str*) – encoding of genotype matrix to create

Returns X in new encoding

Return type numpy.array

easypheno.preprocess.encoding_functions.**get_additive_encoding**(*X*, *style='012'*)

Generate genotype matrix in additive encoding:

- 0: homozygous major allele,
- 1: heterozygous
- 2: homozygous minor allele

for style=012 - 1: homozygous major allele, - 0: heterozygous - -1: homozygous minor allele

Parameters

- **X** (*numpy.array*) – genotype matrix in raw encoding, i.e. containing the alleles
- **style** (*str*) – encoding style, '012' or '101' default is '012'

Returns genotype matrix in additive encoding (X_012)

Return type numpy.array

easypheno.preprocess.encoding_functions.**get_onehot_encoding**(*X*)

Generate genotype matrix in onehot encoding. If genotype matrix is homozygous, create 3d torch tensor with (samples, SNPs, 4), with 4 as the onehot encoding

- A : [1,0,0,0]
- C : [0,1,0,0]
- G : [0,0,1,0]
- T : [0,0,0,1]

If genotype matrix is heterozygous, create 3d torch tensor with (samples, SNPs, 10), with 10 as the onehot encoding

- A : [1,0,0,0,0,0,0,0,0,0]
- C : [0,1,0,0,0,0,0,0,0,0]
- G : [0,0,1,0,0,0,0,0,0,0]
- K : [0,0,0,1,0,0,0,0,0,0]
- M : [0,0,0,0,1,0,0,0,0,0]
- R : [0,0,0,0,0,1,0,0,0,0]
- S : [0,0,0,0,0,0,1,0,0,0]
- T : [0,0,0,0,0,0,0,1,0,0]
- W : [0,0,0,0,0,0,0,0,1,0]
- Y : [0,0,0,0,0,0,0,0,0,1]

Parameters **X** (*numpy.array*) – genotype matrix in raw encoding, i.e. containing the alleles

Returns genotype matrix in onehot encoding (X_onehot)

Return type numpy.array

easypheno.preprocess.raw_data_functions

Module Contents

Functions

prepare_data_files(data_dir, genotype_matrix_name, phenotype_matrix_name, phenotype, datasplit, n_outerfolds, n_innerfolds, test_set_size_percentage, val_set_size_percentage, models, user_encoding, maf_percentage)	Prepare all data files for a common format: genotype matrix, phenotype matrix and index file.
check_genotype_h5_file(data_dir, genotype_matrix_name, encodings)	Check .h5 genotype file. Should contain:
check_index_file(data_dir, genotype_matrix_name, phenotype_matrix_name, phenotype)	Check if index file is available and if the datasets 'y', 'matched_sample_ids', 'X_index', 'y_index' and
save_all_data_files(data_dir, genotype_matrix_name, phenotype_matrix_name, phenotype, models, user_encoding, maf_percentage, datasplit, n_outerfolds, n_innerfolds, test_set_size_percentage, val_set_size_percentage)	Prepare and save all required data files:
check_transform_format_genotype_matrix(data_dir, genotype_matrix_name, models, user_encoding, save_h5 = True)	Check the format of the specified genotype matrix.
check_genotype_csv_file(data_dir, genotype_matrix_name, encodings)	Load .csv genotype file. File must have the following structure:
check_genotype_binary_plink_file(data_dir, genotype_matrix_name)	Load binary PLINK file, .bim, .fam, .bed files with same prefix need to be in same folder.
check_genotype_plink_file(data_dir, genotype_matrix_name)	Load PLINK files, .map and .ped file with same prefix need to be in same folder.
check_duplicate_samples(sample_ids)	check if genotype matrix contain duplicate samples
check_genotype_shape(X, sample_ids, snp_ids)	Check if number of samples in sample_ids and genotype matrix match
create_genotype_h5_file(data_dir, genotype_matrix_name, sample_ids, snp_ids, X)	Save genotype matrix in unified .h5 file.
check_and_load_phenotype_matrix(data_dir, phenotype_matrix_name, phenotype)	Check and load the specified phenotype matrix. Only accept .csv, .pheno, .txt files.
genotype_phenotype_matching(X, X_ids, y)	Match the handed over genotype and phenotype matrix for the phenotype specified by the user, i.e. compare sample ids
get_matched_data(data, index)	Get elements of data specified in index array
append_index_file(data_dir, genotype_matrix_name, phenotype_matrix_name, phenotype, datasplit, n_outerfolds, n_innerfolds, test_set_size_percentage, val_set_size_percentage, maf_percentage)	Check index file, described in create_index_file(), and append datasets if necessary
create_index_file(data_dir, genotype_matrix_name, phenotype_matrix_name, phenotype, datasplit, n_outerfolds, n_innerfolds, test_set_size_percentage, val_set_size_percentage, maf_percentage, X, y, sample_ids, X_index, y_index)	Create the .h5 index file containing the maf filters and data splits for the combination of genotype matrix,
filter_non_informative_snps(X)	Remove non-informative SNPs, i.e. SNPs that are constant
get_minor_allele_freq(X)	Compute minor allele frequencies of genotype matrix
create_maf_filter(maf, freq)	Create minor allele frequency filter
check_datasplit_user_input(user_datasplit, user_n_outerfolds, user_n_innerfolds, user_test_set_size_percentage, user_val_set_size_percentage, datasplit, param_to_check)	Check if user input of data split parameters differs from standard values.
check_train_test_splits(y, datasplit, datasplit_params)	Create stratified train-test splits. Continuous values will be grouped into bins and stratified according to those.
make_bins(y, datasplit, datasplit_params)	Create bins of continuous values for stratification.

easypheno.preprocess.raw_data_functions.**prepare_data_files**(*data_dir*, *genotype_matrix_name*, *phenotype_matrix_name*, *phenotype*, *datasplit*, *n_outerfolds*, *n_innerfolds*, *test_set_size_percentage*, *val_set_size_percentage*, *models*, *user_encoding*, *maf_percentage*)

Prepare all data files for a common format: genotype matrix, phenotype matrix and index file.

First check if genotype file is .h5 file (standard format of this framework):

- YES: First check if all required information is present in the file, raise Exception if not. Then check if index file exists:
  - NO: Load genotype and create all required index files
  - YES: Append all required data splits and maf-filters to index file
- NO: Load genotype and create all required files

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored
- **genotype_matrix_name** (*str*) – name of the genotype matrix including datatype ending
- **phenotype_matrix_name** (*str*) – name of the phenotype matrix including datatype ending
- **phenotype** (*str*) – name of the phenotype to predict
- **datasplit** (*str*) – datasplit to use. Options are: nested-cv, cv-test, train-val-test
- **n_outerfolds** (*int*) – number of outerfolds relevant for nested-cv
- **n_innerfolds** (*int*) – number of folds relevant for nested-cv and cv-test
- **test_set_size_percentage** (*int*) – size of the test set relevant for cv-test and train-val-test
- **val_set_size_percentage** (*int*) – size of the validation set relevant for train-val-test
- **models** – models to consider
- **user_encoding** (*str*) – encoding specified by the user
- **maf_percentage** (*int*) – threshold for MAF filter as percentage value

easypheno.preprocess.raw_data_functions.**check_genotype_h5_file**(*data_dir*, *genotype_matrix_name*, *encodings*)

Check .h5 genotype file. Should contain:

- sample_ids: vector with sample names of genotype matrix,
- snp_ids: vector with SNP identifiers of genotype matrix,
- X_{enc}: (samples x SNPs)-genotype matrix in enc encoding, where enc might refer to:
  - '012': additive (number of minor alleles)
  - 'raw': raw (alleles)

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored

- **genotype_matrix_name** (*str*) – name of the phenotype matrix including datatype ending
- **encodings** (*list*) – list of needed encodings

easypheno.preprocess.raw_data_functions.**check_index_file**(*data_dir*, *genotype_matrix_name*, *phenotype_matrix_name*, *phenotype*)

Check if index file is available and if the datasets 'y', 'matched_sample_ids', 'X_index', 'y_index' and 'ma_frequency' exist.

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored
- **genotype_matrix_name** (*str*) – name of the genotype matrix including datatype ending
- **phenotype_matrix_name** (*str*) – name of the phenotype matrix including datatype ending
- **phenotype** (*str*) – name of the phenotype to predict

Returns bool reflecting check result

Return type bool

easypheno.preprocess.raw_data_functions.**save_all_data_files**(*data_dir*, *genotype_matrix_name*, *phenotype_matrix_name*, *phenotype*, *models*, *user_encoding*, *maf_percentage*, *datasplit*, *n_outerfolds*, *n_innerfolds*, *test_set_size_percentage*, *val_set_size_percentage*)

Prepare and save all required data files:

- genotype matrix in unified format as .h5 file with,
- phenotype matrix in unified format as .csv file,
- file containing maf filter and data split indices as .h5

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored
- **genotype_matrix_name** (*str*) – name of the genotype matrix including datatype ending
- **phenotype_matrix_name** (*str*) – name of the phenotype matrix including datatype ending
- **phenotype** (*str*) – name of the phenotype to predict
- **models** – models to consider
- **user_encoding** (*str*) – encoding specified by the user
- **maf_percentage** (*int*) – threshold for MAF filter as percentage value
- **datasplit** (*str*) – datasplit to use. Options are: nested-cv, cv-test, train-val-test
- **n_outerfolds** (*int*) – number of outerfolds relevant for nested-cv
- **n_innerfolds** (*int*) – number of folds relevant for nested-cv and cv-test
- **test_set_size_percentage** (*int*) – size of the test set relevant for cv-test and train-val-test

- **val_set_size_percentage** (*int*) – size of the validation set relevant for train-val-test

easypheno.preprocess.raw_data_functions.**check_transform_format_genotype_matrix**(*data_dir*, *genotype_matrix_name*, *models*, *user_encoding*, *save_h5=True*)

Check the format of the specified genotype matrix.

Unified genotype matrix will be saved in subdirectory data and named NAME_OF_GENOTYPE_MATRIX.h5

Unified format of the .h5 file of the genotype matrix required for the further processes:

- mandatory:
  - sample_ids: vector with sample names of genotype matrix,
  - SNP_ids: vector with SNP identifiers of genotype matrix,
  - X_{enc}: (samples x SNPs)-genotype matrix in enc encoding, where enc might refer to:
    - '012': additive (number of minor alleles)
    - 'raw': raw (alleles)
- optional: genotype in additional encodings

Accepts .h5, .hdf5, .h5py, .csv, PLINK binary and PLINK files. .h5, .hdf5, .h5py files must satisfy the unified format. If the genotype matrix contains constant SNPs, those will be removed and a new file will be saved. Will open .csv, PLINK and binary PLINK files and generate required .h5 format.

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored
- **genotype_matrix_name** (*str*) – name of the genotype matrix including datatype ending
- **models** – models to consider
- **user_encoding** (*str*) – encoding specified by the user
- **save_h5** (*bool*) – save genotype in unified h5 format if True, default is True

Returns genotype matrix (raw encoded if present, 012 encoded otherwise), sample ids and SNP ids

Return type (numpy.array, numpy.array, numpy.array)

easypheno.preprocess.raw_data_functions.**check_genotype_csv_file**(*data_dir*, *genotype_matrix_name*, *encodings*)

Load .csv genotype file. File must have the following structure: First column must contain the sample ids, the column names should be the SNP ids. The values should be the genotype matrix either in additive encoding or in raw encoding. If the name of the first column is 'MarkerID' it is assumed that the rows contain the markers and the column contain the samples and the genotype matrix will be transposed. If the csv file contains the genotype in biallelic notation (i.e. 'AA', 'AT', ...), this function generates a genotype matrix in iupac notation (i.e. 'A', 'W', ...).

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored

- **genotype_matrix_name** (*str*) – name of the genotype matrix including datatype ending
- **encodings** (*list*) – list of needed encodings

Returns sample ids, SNP ids and genotype in additive / raw encoding

Return type (numpy.array, numpy.array, numpy.array)

easypheno.preprocess.raw_data_functions.**check_genotype_binary_plink_file**(*data_dir*, *genotype_matrix_name*)

Load binary PLINK file, .bim, .fam, .bed files with same prefix need to be in same folder. Compute additive and raw encoding of genotype

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored
- **genotype_matrix_name** (*str*) – name of the genotype matrix including datatype ending

Returns sample ids, SNP ids and genotype in raw encoding

Return type (numpy.array, numpy.array, numpy.array)

easypheno.preprocess.raw_data_functions.**check_genotype_plink_file**(*data_dir*, *genotype_matrix_name*)

Load PLINK files, .map and .ped file with same prefix need to be in same folder. Accepts GENOTYPENAME.ped and GENOTYPENAME.map as input

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored
- **genotype_matrix_name** (*str*) – name of the genotype matrix including datatype ending

Returns sample ids, SNP ids and genotype in raw encoding

Return type (numpy.array, numpy.array, numpy.array)

easypheno.preprocess.raw_data_functions.**check_duplicate_samples**(*sample_ids*)

check if genotype matrix contain duplicate samples

Parameters **sample_ids** (*numpy.array*) – sample ids of genotype matrix

Returns True if duplicates are present, False if not

Return type bool

easypheno.preprocess.raw_data_functions.**check_genotype_shape**(*X*, *sample_ids*, *snp_ids*)

Check if number of samples in sample_ids and genotype matrix match and if number of markers in snp_ids and genotype matrix match.

Parameters

- **X** (*numpy.array*) – genotype matrix
- **sample_ids** (*numpy.array*) – vector containing sample ids of genotype
- **snp_ids** (*numpy.array*) – vector containing SNP ids of genotype

easypheno.preprocess.raw_data_functions.**create_genotype_h5_file**(*data_dir*, *genotype_matrix_name*, *sample_ids*, *snp_ids*, *X*)

Save genotype matrix in unified .h5 file.

Structure:

- sample_ids
- snp_ids
- X_raw (or X_012 if X_raw not available)

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored
- **genotype_matrix_name** (*str*) – name of the genotype matrix including datatype ending
- **sample_ids** (*numpy.array*) – array containing sample ids of genotype data
- **snp_ids** (*numpy.array*) – array containing snp ids of genotype data
- **X** (*numpy.array*) – matrix containing genotype either in raw or in additive encoding

easypheno.preprocess.raw_data_functions.**check_and_load_phenotype_matrix**(*data_dir*, *phenotype_matrix_name*, *phenotype*)

Check and load the specified phenotype matrix. Only accept .csv, .pheno, .txt files. Sample ids need to be in first column, remaining columns should contain phenotypic values with phenotype name as column name

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored
- **phenotype_matrix_name** (*str*) – name of the phenotype matrix including datatype ending
- **phenotype** (*str*) – name of the phenotype to predict

Returns DataFrame with sample_ids as index and phenotype values as single column without NAN values

Return type pandas.DataFrame

easypheno.preprocess.raw_data_functions.**genotype_phenotype_matching**(*X*, *X_ids*, *y*)

Match the handed over genotype and phenotype matrix for the phenotype specified by the user, i.e. compare sample ids

Parameters

- **X** (*numpy.array*) – genotype matrix in additive encoding
- **X_ids** (*numpy.array*) – sample ids of genotype matrix
- **y** (*pandas.DataFrame*) – pd.DataFrame containing sample ids of phenotype as index and phenotype values as single column

Returns matched genotype matrix, matched sample ids, index arrays for genotype and phenotype to redo matching

Return type tuple

easypheno.preprocess.raw_data_functions.**get_matched_data**(*data*, *index*)

Get elements of data specified in index array

Parameters

- **data** (*numpy.array*) – matrix or array
- **index** (*numpy.array*) – index array

Returns data at selected indices

Return type numpy.array

easypheno.preprocess.raw_data_functions.**append_index_file**(*data_dir*, *genotype_matrix_name*, *phenotype_matrix_name*, *phenotype*, *datasplit*, *n_outerfolds*, *n_innerfolds*, *test_set_size_percentage*, *val_set_size_percentage*, *maf_percentage*)

Check index file, described in create_index_file(), and append datasets if necessary

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored
- **genotype_matrix_name** (*str*) – name of the genotype matrix including datatype ending
- **phenotype_matrix_name** (*str*) – name of the phenotype matrix including datatype ending
- **phenotype** (*str*) – name of the phenotype to predict
- **maf_percentage** (*int*) – threshold for MAF filter as percentage value
- **datasplit** (*str*) – datasplit to use. Options are: nested-cv, cv-test, train-val-test
- **n_outerfolds** (*int*) – number of outerfolds relevant for nested-cv
- **n_innerfolds** (*int*) – number of folds relevant for nested-cv and cv-test
- **test_set_size_percentage** (*int*) – size of the test set relevant for cv-test and train-val-test
- **val_set_size_percentage** (*int*) – size of the validation set relevant for train-val-test

easypheno.preprocess.raw_data_functions.**create_index_file**(*data_dir*, *genotype_matrix_name*, *phenotype_matrix_name*, *phenotype*, *datasplit*, *n_outerfolds*, *n_innerfolds*, *test_set_size_percentage*, *val_set_size_percentage*, *maf_percentage*, *X*, *y*, *sample_ids*, *X_index*, *y_index*)

Create the .h5 index file containing the maf filters and data splits for the combination of genotype matrix, phenotype matrix and phenotype. It will be created using standard values additionally to user inputs for the maf filters and data splits.

Unified format of .h5 file containing the maf filters and data splits:

```
'matched_data': {
        'y': matched phenotypic values,
        'matched_sample_ids': sample ids of matched genotype/phenotype,
        'X_index': indices of genotype matrix to redo matching,
```

(continues on next page)

(continued from previous page)

```
        'y_index': indices of phenotype vector to redo matching,
        'ma_frequency': minor allele frequency of each SNP of genotype file to␣
↪create new MAF filters
        }
'maf_filter': {
        'maf_{maf_percentage}': indices of SNPs to delete  # (with MAF < maf_
↪percentage),
        ...
        }
'datasplits': {
        'nested_cv': {
                '#outerfolds-#innerfolds': {
                        'outerfold_0': {
                            'innerfold_0': {'train': indices_train, 'val': indices_
↪val},
                            ...
                            'innerfold_n': {'train': indices_train, 'val': indices_
↪val},
                            'test': test_indices
                            },
                        ...
                        'outerfold_m': {
                            'innerfold_0': {'train': indices_train, 'val': indices_
↪val},
                            ...
                            'innerfold_n': {'train': indices_train, 'val': indices_
↪val},
                            'test': test_indices
                            }
                        },
                ...
                }
        'cv-test': {
                '#folds-test_percentage': {
                        'outerfold_0': {
                            'innerfold_0': {'train': indices_train, 'val': indices_
↪val},
                            ...
                            'innerfold_n': {'train': indices_train, 'val': indices_
↪val},
                            'test': test_indices
                            }
                        },
                ...
                }
        'train-val-test': {
                'train_percentage-val_percentage-test_percentage': {
                        'outerfold_0': {
                            'innerfold_0': {'train': indices_train, 'val': indices_
↪val},
                            'test': test_indices
                            }
```

(continues on next page)

(continued from previous page)

```
                    },
                ...
                }
        }
```

Standard values for the maf filters and data splits:

- maf thresholds: 1, 3, 5
- folds (inner-/outerfolds for 'nested-cv' and folds for 'cv-test'): 5
- test percentage (for 'cv-test' and 'train-val-test'): 20
- val percentage (for 'train-val-test'): 20

Parameters

- **data_dir** (*pathlib.Path*) – data directory where the phenotype and genotype matrix are stored
- **genotype_matrix_name** (*str*) – name of the genotype matrix including datatype ending
- **phenotype_matrix_name** (*str*) – name of the phenotype matrix including datatype ending
- **phenotype** (*str*) – name of the phenotype to predict
- **maf_percentage** (*int*) – threshold for MAF filter as percentage value
- **datasplit** (*str*) – datasplit to use. Options are: nested-cv, cv-test, train-val-test
- **n_outerfolds** (*int*) – number of outerfolds relevant for nested-cv
- **n_innerfolds** (*int*) – number of folds relevant for nested-cv and cv-test
- **test_set_size_percentage** (*int*) – size of the test set relevant for cv-test and train-val-test
- **val_set_size_percentage** (*int*) – size of the validation set relevant for train-val-test
- **X** (*numpy.array*) – genotype in additive encoding to create ma-frequencies
- **y** (*numpy.array*) – matched phenotype values
- **sample_ids** (*numpy.array*) – matched sample ids of genotype/phenotype
- **X_index** (*numpy.array*) – index file of genotype to redo matching
- **y_index** (*numpy.array*) – index file of phenotype to redo matching

easypheno.preprocess.raw_data_functions.**filter_non_informative_snps**(*X*)

Remove non-informative SNPs, i.e. SNPs that are constant

Parameters **X** (*numpy.array*) – genotype matrix in raw or additive encoding

Returns filtered genotype matrix and filter-vector

Return type (numpy.array, numpy.array)

easypheno.preprocess.raw_data_functions.**get_minor_allele_freq**(*X*)

Compute minor allele frequencies of genotype matrix

Parameters **X** (*numpy.array*) – genotype matrix in additive encoding

Returns array with frequencies

Return type numpy.array

easypheno.preprocess.raw_data_functions.**create_maf_filter**(*maf*, *freq*)

Create minor allele frequency filter

Parameters

- **maf** (*int*) – maf threshold as percentage value
- **freq** (*numpy.array*) – array containing minor allele frequencies as decimal value

Returns array containing indices of SNPs with MAF smaller than specified threshold, i.e. SNPs to delete

Return type numpy.array

easypheno.preprocess.raw_data_functions.**check_datasplit_user_input**(*user_datasplit*, *user_n_outerfolds*, *user_n_innerfolds*, *user_test_set_size_percentage*, *user_val_set_size_percentage*, *datasplit*, *param_to_check*)

Check if user input of data split parameters differs from standard values. If it does, add input to list of parameters

Parameters

- **user_datasplit** (*str*) – datasplit specified by the user
- **user_n_outerfolds** (*int*) – number of outerfolds relevant for nested-cv specified by the user
- **user_n_innerfolds** (*int*) – number of folds relevant for nested-cv and cv-test specified by the user
- **user_test_set_size_percentage** (*int*) – size of the test set relevant for cv-test and train-val-test specified by the user
- **user_val_set_size_percentage** (*int*) – size of the validation set relevant for train-val-test specified by the user
- **datasplit** (*str*) – type of data split
- **param_to_check** (*list*) – standard parameters to compare to

Returns adapted list of parameters

Return type list

easypheno.preprocess.raw_data_functions.**check_train_test_splits**(*y*, *datasplit*, *datasplit_params*)

Create stratified train-test splits. Continuous values will be grouped into bins and stratified according to those.

Datasplit parameters:

- nested-cv: [n_outerfolds, n_innerfolds]
- cv-test: [n_innerfolds, test_set_size_percentage]
- train-val-test: [val_set_size_percentage, train_set_size_percentage]

Parameters

- **datasplit** (*str*) – type of datasplit ('nested-cv', 'cv-test', 'train-val-test')
- **y** (*numpy.array*) – array with phenotypic values for stratification
- **datasplit_params** (*list*) – parameters to use for split

Returns dictionary respectively arrays with indices

easypheno.preprocess.raw_data_functions.**make_bins**(*y*, *datasplit*, *datasplit_params*)

Create bins of continuous values for stratification.

Datasplit parameters:

- nested-cv: [n_outerfolds, n_innerfolds]
- cv-test: [n_innerfolds, test_set_size_percentage]
- train-val-test: [val_set_size_percentage, train_set_size_percentage]

Parameters

- **y** (*numpy.array*) – array containing phenotypic values
- **datasplit** (*str*) – train test split to use
- **datasplit_params** (*list*) – parameters to use for split

Returns binned array

Return type numpy.array

easypheno.preprocess.raw_data_functions.**make_nested_cv**(*y*, *outerfolds*, *innerfolds*)

Create index dictionary for stratified nested cross validation with the following structure:

```
{
    'outerfold_0_test': test_indices,
    'outerfold_0': {
        'fold_0_train': innerfold_0_train_indices,
        'fold_0_test': innerfold_0_test_indices,
        ...
        'fold_n_train': innerfold_n_train_indices,
        'fold_n_test': innerfold_n_test_indices
    },
    ...
    'outerfold_m_test': test_indices,
    'outerfold_m': {
        'fold_0_train': innerfold_0_train_indices,
        'fold_0_test': innerfold_0_test_indices,
        ...
        'fold_n_train': innerfold_n_train_indices,
        'fold_n_test': innerfold_n_test_indices
    }
}
```

Parameters

- **y** (*numpy.array*) – target values grouped in bins for stratification
- **outerfolds** (*int*) – number of outer folds
- **innerfolds** (*int*) – number of inner folds

Returns index dictionary

Return type dict

easypheno.preprocess.raw_data_functions.**make_stratified_cv**(*x*, *y*, *split_number*)

Create index dictionary for stratified cross-validation with following structure:

```
{
    'fold_0_train': fold_0_train_indices,
    'fold_0_test': fold_0_test_indices,
    ...
    'fold_n_train': fold_n_train_indices,
    'fold_n_test': fold_n_test_indices
}
```

Parameters

- **x** (*numpy.array*) – whole train indices
- **y** (*numpy.array*) – target values binned in groups for stratification
- **split_number** (*int*) – number of folds

Returns dictionary containing train and validation indices for each fold

Return type dict

easypheno.preprocess.raw_data_functions.**make_train_test_split**(*y*, *test_size*, *val_size=None*, *val=False*, *random=42*)

Create index arrays for stratified train-test, respectively train-val-test splits.

Parameters

- **y** (*numpy.array*) – target values grouped in bins for stratification
- **test_size** (*int*) – size of test set as percentage value
- **val_size** – size of validation set as percentage value
- **val** – if True, function returns validation set additionally to train and test set
- **random** – controls shuffling of data

Returns either train, val and test index arrays or train and test index arrays and corresponding binned target values

Return type (numpy.array, numpy.array, numpy.array)

easypheno.simulate

Submodules

easypheno.simulate.results_analysis_synthetic_data

Module Contents

Functions

gather_sim_configs(sim_config_dir, save_dir)	Collect the information on the simulation configurations for all within the specified directory
gather_feature_importances(results_dir, save_dir, datasplit_maf_pattern)	Collect the information on the feature importances for all models within the specified directory and for the specified datasplit maf pattern
get_statistics_featimps_vs_simulation(all_sim_configs, all_feat_imps, min_perc_threshold = 0.01)	Get statistics on feature importances compared to effect sizes on synthetic data, e.g. on how many background SNPs were detected
generate_scatterplots_featimps_vs_simulation(all_feat_imps, all_sim_configs, save_dir, datasplit_maf_pattern)	Generate scatterplots based on feature importances and effect sizes on synthetic data. One plot containing all models for the specified datasplit maf pattern as well as single plots for each model will be generated and saved.
featimps_vs_simulation(results_directory_genotype_level, sim_config_dir, save_dir)	Analyze feature importances versus effect sizes on synthetic data, both by retrieving stastistics and generating plots

easypheno.simulate.results_analysis_synthetic_data.**gather_sim_configs**(*sim_config_dir*, *save_dir*)

Collect the information on the simulation configurations for all within the specified directory

Parameters

- **sim_config_dir** (*pathlib.Path*) – directory which contains the sim config files
- **save_dir** (*pathlib.Path*) – directory to save the collected sim config info

easypheno.simulate.results_analysis_synthetic_data.**gather_feature_importances**(*results_dir*, *save_dir*, *datasplit_maf_pattern*)

Collect the information on the feature importances for all models within the specified directory and for the specified datasplit maf pattern

Parameters

- **results_dir** (*pathlib.Path*) – results directory at the level of the name of the genotype matrix
- **save_dir** (*pathlib.Path*) – directory to save the collected info
- **datasplit_maf_pattern** (*str*) – datasplit maf pattern to search on

easypheno.simulate.results_analysis_synthetic_data.**get_statistics_featimps_vs_simulation**(*all_sim_configs*, *all_feat_imps*, *min_perc_threshol*

Get statistics on feature importances compared to effect sizes on synthetic data, e.g. on how many background SNPs were detected

Parameters

- **all_sim_configs** (*pandas.DataFrame*) – simulation configs to consider
- **all_feat_imps** (*pandas.DataFrame*) – feature importances to consider
- **min_perc_threshold** (*float*) – threshold for minimum feature importance in relation to maximum feature importance for a specific model

Returns statistics for a comparison between feature importances and effect sizes in a DataFrame

Return type pandas.DataFrame

easypheno.simulate.results_analysis_synthetic_data.**generate_scatterplots_featimps_vs_simulation**(*all_feat_i*
all_sim_c
save_dir,
datas-
*plit_maf_*

Generate scatterplots based on feature importances and effect sizes on synthetic data. One plot containing all models for the specified datasplit maf pattern as well as single plots for each model will be generated and saved.

Parameters

- **all_sim_configs** (*pandas.DataFrame*) – simulation configs to consider
- **all_feat_imps** (*pandas.DataFrame*) – feature importances to consider
- **save_dir** (*pathlib.Path*) – directory to save the plots
- **datasplit_maf_pattern** (*str*) – datasplit maf pattern to search on

easypheno.simulate.results_analysis_synthetic_data.**featimps_vs_simulation**(*results_directory_genotype_level*,
sim_config_dir,
save_dir)

Analyze feature importances versus effect sizes on synthetic data, both by retrieving stastistics and generating plots

Parameters

- **results_directory_genotype_level** (*str*) – results directory at the level of the name of the genotype matrix
- **sim_config_dir** (*str*) – directory which contains the sim config files
- **save_dir** (*str*) – directory to save the results

easypheno.simulate.synthetic_phenotypes

Module Contents

Functions

filter_duplicates(X, snp_ids)	Remove duplicate SNPs, i.e. SNPs that are completely the same for all samples and therefore do not add information.
get_simulation(X, sample_ids, snp_ids, number_of_samples, number_causal_snps, explained_variance, maf, heritability, seed, number_background_snps, distribution, shape)	Simulate phenotypes based on (real) genotypes in an additive setting with normally or gamma distributed noise and
check_sim_id(sim_dir)	Check which ids were already used for simulations.
save_sim_overview(save_dir, sim_names, number_of_samples, number_causal_snps, explained_variance, maf, heritability, seeds, number_background_snps, distribution, shape)	save overview file for all simulations; append new simulations if file already exists
save_simulation(save_dir, genotype_matrix_name, number_of_sim, X, sample_ids, snp_ids, number_of_samples, number_causal_snps, explained_variance, maf, heritability, seed, number_background_snps, distribution, shape)	Set all variables and generate one or more simulations with same configurations.

easypheno.simulate.synthetic_phenotypes.**filter_duplicates**(*X*, *snp_ids*)

Remove duplicate SNPs, i.e. SNPs that are completely the same for all samples and therefore do not add information.

Parameters

- **X** (*numpy.array*) – genotype matrix to be filtered
- **snp_ids** (*numpy.array*) – vector containing corresponding SNP ids

Returns filtered genotype matrix and filtered SNP ids

Return type (numpy.array, numpy.array)

easypheno.simulate.synthetic_phenotypes.**get_simulation**(*X*, *sample_ids*, *snp_ids*, *number_of_samples*, *number_causal_snps*, *explained_variance*, *maf*, *heritability*, *seed*, *number_background_snps*, *distribution*, *shape*)

Simulate phenotypes based on (real) genotypes in an additive setting with normally or gamma distributed noise and normally distributed effect sizes of causal SNPs.

Parameters

- **X** (*numpy.array*) – genotype matrix in additive encoding
- **sample_ids** (*numpy.array*) – sample ids of genotype matrix
- **snp_ids** (*numpy.array*) – SNP ids of genotype matrix
- **number_of_samples** (*int*) – number of samples of synthetic phenotype
- **number_causal_snps** (*int*) – number of SNPs used as causal markers in simulation
- **explained_variance** (*int*) – percentage value of how much of the total variance the causal SNPs should explain
- **maf** (*int*) – percentage value used for maf filtering of genotype matrix

- **heritability** (*int*) – percentage value of how much of the variance should be explained by polygenic background
- **seed** (*int*) – seed for random sampling
- **number_background_snps** (*int*) – number of randomly selected SNPs to simulate the polygenic background
- **distribution** (*str*) – probability distribution used to draw random noise can be 'normal' or 'gamma'
- **shape** (*float*) – only needed if distribution is 'gamma'

Returns simulated phenotype with corresponding sample ids, SNP ids of causal SNPs, SNP ids of background SNPs,

Return type (numpy.array, numpy.array, numpy.array, numpy.array, numpy.array, numpy.array, numpy.array)

effect sizes of background, effect sizes of causal SNPs, used explained variance for each causal SNP

easypheno.simulate.synthetic_phenotypes.**check_sim_id**(*sim_dir*)

Check which ids were already used for simulations.

Parameters **sim_dir** (*pathlib.Path*) – directory containing simulations to check

Returns last simulation number + 1

Return type int

easypheno.simulate.synthetic_phenotypes.**save_sim_overview**(*save_dir*, *sim_names*, *number_of_samples*, *number_causal_snps*, *explained_variance*, *maf*, *heritability*, *seeds*, *number_background_snps*, *distribution*, *shape*)

save overview file for all simulations; append new simulations if file already exists

Parameters

- **save_dir** (*pathlib.Path*) – directory to save overview file to
- **sim_names** (*list*) – list containing simulation name for each simulation
- **number_of_samples** (*list*) – list containing number of samples for each simulation
- **number_causal_snps** (*list*) – list containing number of causal SNPS for each simulation
- **explained_variance** (*list*) – list containing total explained variance of causal SNPs for each simulation
- **maf** (*list*) – list containing used maf frequency for each simulation
- **heritability** (*list*) – list containing used heritability for each simulation
- **seeds** (*list*) – list containing used seed for each simulation
- **number_background_snps** (*list*) – list containing number of background SNPs for each simulation
- **distribution** (*list*) – list containing used distribution of random noise for each simulation
- **shape** (*list*) – list containing shape of gamma distribution, resp. None for normal distribution for each simulation

easypheno.simulate.synthetic_phenotypes.**save_simulation**(*save_dir*, *genotype_matrix_name*, *number_of_sim*, *X*, *sample_ids*, *snp_ids*, *number_of_samples*, *number_causal_snps*, *explained_variance*, *maf*, *heritability*, *seed*, *number_background_snps*, *distribution*, *shape*)

Set all variables and generate one or more simulations with same configurations. Save overview file and simulated phenotypes to subfolder 'genotype_matrix_name' in save_dir as 'Simulations_Overview.csv' and Simulation_{sim_id}.csv. Save SNP ids of background SNPs, effect sizes/betas of background SNPs and configuration infos containing SNP ids and betas of causal SNPs to subfolder sim_configs in 'genotype_matrix_name' as 'background_{sim_id}.csv', 'betas_background_{sim_id}.csv' and 'simulation_config_{sim_id}.csv'. If only one phenoype is simulated, the sim_id consists of a single number. If several phenotypes are simulated with the same configurations, then the sim_id is the number of the first simulation '-' number of last simulation, e.g. '10-15'

Parameters

- **save_dir** (*str*) – directory to save simulations to
- **genotype_matrix_name** (*str*) – name of genotype matrix to be used for simulations, needed to create subfolder in save_dir
- **number_of_sim** (*int*) – number of simulations to create with same configurations
- **X** (*numpy.array*) – genotype matrix in additive encoding
- **sample_ids** (*numpy.array*) – sample ids of genotype matrix
- **snp_ids** (*numpy.array*) – SNP ids of genotype matrix
- **number_of_samples** (*int*) – number of samples of synthetic phenotype
- **number_causal_snps** (*int*) – number of SNPs used as causal markers in simulation
- **explained_variance** (*int*) – percentage value of how much of the total variance the causal SNPs should explain
- **maf** (*int*) – percentage value used for maf filtering of genotype matrix
- **heritability** (*int*) – percentage value of how much of the variance should be explained by polygenic background
- **seed** (*int*) – seed for random sampling
- **number_background_snps** (*int*) – number of randomly selected SNPs to simulate the polygenic background
- **distribution** (*str*) – probability distribution used to draw random noise can be 'normal' or 'gamma'
- **shape** (*float*) – only needed if distribution is 'gamma'

easypheno.utils

Submodules

easypheno.utils.check_functions

Module Contents

Functions

check_all_specified_arguments(arguments)	Check all specified arguments for plausibility
check_exist_directories(list_of_dirs, create_if_not_exist = False)	Check if each directory within a list exists
check_exist_files(list_of_files)	Check if each file within a list exists
compare_snp_id_vectors(snp_id_vector_small_equal, snp_id_vector_big_equal)	Compare two SNP id vectors if they contain the same ids

easypheno.utils.check_functions.**check_all_specified_arguments**(*arguments*)

Check all specified arguments for plausibility

Parameters **arguments** (*dict*) – all arguments provided by the user

easypheno.utils.check_functions.**check_exist_directories**(*list_of_dirs*, *create_if_not_exist=False*)

Check if each directory within a list exists

Parameters

- **list_of_dirs** (*list*) – list with directories as pathlib.Path
- **create_if_not_exist** (*bool*) – bool if non-existing directories should be created

Returns True if all exist, False otherwise

Return type bool

easypheno.utils.check_functions.**check_exist_files**(*list_of_files*)

Check if each file within a list exists

Parameters **list_of_files** (*list*) – list with files as pathlib.Path

Returns True if all exist, False otherwise

Return type bool

easypheno.utils.check_functions.**compare_snp_id_vectors**(*snp_id_vector_small_equal*, *snp_id_vector_big_equal*)

Compare two SNP id vectors if they contain the same ids

Parameters

- **snp_id_vector_small_equal** (*numpy.array*) – vector 1 with SNP ids, can be a (smaller) subset of snp_id_vector_big_equal
- **snp_id_vector_big_equal** (*numpy.array*) – vector 2 with SNP ids, can contain more SNP ids than snp_id_vector_small_equal

Returns True if snp_id_vector_small_equal is a subset of the other vector

Return type bool

easypheno.utils.helper_functions

Module Contents

Functions

get_list_of_implemented_models()	Create a list of all implemented models based on files existing in 'model' subdirectory of the repository.
test_likely_categorical(vector_to_test, abs_unique_threshold = 20)	Test whether a vector is most likely categorical.
get_mapping_name_to_class()	Get a mapping from model name (naming in package model without .py) to class name.
set_all_seeds(seed = 42)	Set all seeds of libs with a specific function for reproducibility of results
get_subpath_for_datasplit(datasplit, datasplit_params)	Construct the subpath according to the datasplit.
save_model_overview_dict(model_overview, save_path)	Structure and save results of a whole optimization run for multiple models in one csv file
sort_models_by_encoding(models_list)	Sort models by the encoding that will be used
get_all_subdirectories_non_recursive(path)	Get all non-recursive subdirectories of path
get_all_files(path)	Get all non-recursive files of path
get_all_files_with_suffix(path, suffix)	Get all non-recursive files of path
get_datasplit_config_info_for_resultfolder(resultfolder)	Get datasplit info for a result folder

easypheno.utils.helper_functions.**get_list_of_implemented_models**()

Create a list of all implemented models based on files existing in 'model' subdirectory of the repository.

Return type list

easypheno.utils.helper_functions.**test_likely_categorical**(*vector_to_test*, *abs_unique_threshold=20*)

Test whether a vector is most likely categorical. Simple heuristics: checking if the number of unique values exceeds a specified threshold

Parameters

- **vector_to_test** (*list*) – vector that is tested if it is most likely categorical
- **abs_unique_threshold** (*int*) – threshold of unique values' ratio to declare vector categorical

Returns True if the vector is most likely categorical, False otherwise

Return type bool

easypheno.utils.helper_functions.**get_mapping_name_to_class**()

Get a mapping from model name (naming in package model without .py) to class name.

Returns dictionary with mapping model name to class name

Return type dict

easypheno.utils.helper_functions.**set_all_seeds**(*seed=42*)

Set all seeds of libs with a specific function for reproducibility of results

Parameters **seed** (*int*) – seed to use

easypheno.utils.helper_functions.**get_subpath_for_datasplit**(*datasplit*, *datasplit_params*)

Construct the subpath according to the datasplit.

Datasplit parameters:

- nested-cv: [n_outerfolds, n_innerfolds]
- cv-test: [n_innerfolds, test_set_size_percentage]
- train-val-test: [val_set_size_percentage, test_set_size_percentage]

Parameters

- **datasplit** (*str*) – datasplit to retrieve
- **datasplit_params** (*list*) – parameters to use for the specific datasplit

Returns string with the subpath

Return type str

easypheno.utils.helper_functions.**save_model_overview_dict**(*model_overview*, *save_path*)

Structure and save results of a whole optimization run for multiple models in one csv file

Parameters

- **model_overview** (*dict*) – dictionary with results overview
- **save_path** (*str*) – filepath for saving the results overview file

easypheno.utils.helper_functions.**sort_models_by_encoding**(*models_list*)

Sort models by the encoding that will be used

Parameters **models_list** (*list*) – unsorted list of models

Returns list of models sorted by encoding

Return type list

easypheno.utils.helper_functions.**get_all_subdirectories_non_recursive**(*path*)

Get all non-recursive subdirectories of path

Parameters **path** (*pathlib.Path*) – path to search

Returns list with all non-recursive subdirs

Return type list

easypheno.utils.helper_functions.**get_all_files**(*path*)

Get all non-recursive files of path

Parameters **path** (*pathlib.Path*) – path to search

Returns list with all non-recursive files

Return type list

easypheno.utils.helper_functions.**get_all_files_with_suffix**(*path*, *suffix*)

Get all non-recursive files of path

Parameters

- **path** (*pathlib.Path*) – path to search
- **suffix** (*str*) – suffix to search

Returns list with all non-recursive files

Return type list

easypheno.utils.helper_functions.**get_datasplit_config_info_for_resultfolder**(*resultfolder*)

Get all datasplit info for a result folder

Parameters **resultfolder** (*str*) – path to retrieve info

Returns datasplit info with datasplit, n_outerfolds, n_innerfolds, val_set_size_percentage, test_set_size_percentage, maf_percentage

Return type tuple

easypheno.utils.print_functions

Module Contents

Functions

print_config_info(arguments, dataset, task)	Print info of current configuration.

easypheno.utils.print_functions.**print_config_info**(*arguments*, *dataset*, *task*)

Print info of current configuration.

Parameters

- **arguments** (*dict*) – arguments specified by the user
- **dataset** (easypheno.preprocess.base_dataset.Dataset) – dataset used for optimization
- **task** (*str*) – task that was detected

2.7.2 Submodules

easypheno.optim_pipeline

Module Contents

Functions

run(data_dir, genotype_matrix, phenotype_matrix, phenotype, encoding = None, maf_percentage = 0, save_dir = None, datasplit = 'nested-cv', n_outerfolds = 5, n_innerfolds = 5, test_set_size_percentage = 20, val_set_size_percentage = 20, models = None, n_trials = 100, save_final_model = False, batch_size = 32, n_epochs = 100000, outerfold_number_to_run = None)	Run the whole optimization pipeline

easypheno.optim_pipeline.**run**(*data_dir*, *genotype_matrix*, *phenotype_matrix*, *phenotype*, *encoding=None*, *maf_percentage=0*, *save_dir=None*, *datasplit='nested-cv'*, *n_outerfolds=5*, *n_innerfolds=5*, *test_set_size_percentage=20*, *val_set_size_percentage=20*, *models=None*, *n_trials=100*, *save_final_model=False*, *batch_size=32*, *n_epochs=100000*, *outerfold_number_to_run=None*)

Run the whole optimization pipeline

Parameters

- **data_dir** (*str*) – data directory where the phenotype and genotype matrix are stored
- **genotype_matrix** (*str*) – name of the genotype matrix including datatype ending
- **phenotype_matrix** (*str*) – name of the phenotype matrix including datatype ending
- **phenotype** (*str*) – name of the phenotype to predict
- **encoding** (*str*) – encoding to use. Default is None, so standard encoding of each model will be used. Options are: '012', 'onehot', 'raw'
- **maf_percentage** (*int*) – threshold for MAF filter as percentage value. Default is 0, so no MAF filtering
- **save_dir** (*str*) – directory for saving the results. Default is None, so same directory as data_dir
- **datasplit** (*str*) – datasplit to use. Options are: nested-cv, cv-test, train-val-test
- **n_outerfolds** (*int*) – number of outerfolds relevant for nested-cv
- **n_innerfolds** (*int*) – number of folds relevant for nested-cv and cv-test
- **test_set_size_percentage** (*int*) – size of the test set relevant for cv-test and train-val-test
- **val_set_size_percentage** (*int*) – size of the validation set relevant for train-val-test
- **models** (*list*) – list of models that should be optimized
- **n_trials** (*int*) – number of trials for optuna
- **save_final_model** (*bool*) – specify if the final model should be saved
- **batch_size** (*int*) – batch size for neural network models
- **n_epochs** (*int*) – number of epochs for neural network models
- **outerfold_number_to_run** (*int*) – outerfold to run in case you do not want to run all

2.7.3 Package Contents

easypheno.**__version__ = 0.0.3**

easypheno.**__author__ = Florian Haselbeck, Maura John, Dominik G. Grimm**

easypheno.**__credits__ = GrimmLab @ TUM Campus Straubing (https://bit.cs.tum.de/)**

PYTHON MODULE INDEX

e

INDEX

Symbols

A

B

C

D

E

F

G

L

M

O

P

R

S

T

V

W

X