DeepProg Documentation

Olivier Poirion

Jul 22, 2021

CONTENTS

1 Installation **1**
- 1.1 Requirements . . . 1
- 1.2 Tested python package versions . . . 1
- 1.3 Alternative deep-Learning packages installation . . . 2
- 1.4 Alternative support for CNTK / theano / tensorflow . . . 3
- 1.5 Visualisation module (Experimental) . . . 3
- 1.6 Usage . . . 4
- 1.7 Example scripts . . . 4

2 Tutorial: Simple DeepProg model **5**
- 2.1 Input parameters . . . 5
- 2.2 Input matrices . . . 5
- 2.3 Creating a simple DeepProg model with one autoencoder for each omic . . . 7

3 Tutorial: Ensemble of DeepProg model **11**
- 3.1 Instanciation . . . 11
- 3.2 Fitting . . . 12
- 3.3 Evaluate the models . . . 15
- 3.4 Predicting on test dataset . . . 15
- 3.5 Distributed computation . . . 17
- 3.6 More examples . . . 18

4 Tutorial: Advanced usage of DeepProg model **19**
- 4.1 Visualisation . . . 19
- 4.2 Hyperparameters . . . 22
- 4.3 Usage of metadata associated with patients . . . 25
- 4.4 Computing cluster-specific feature signatures . . . 26
- 4.5 R installation (Alternative to Python lifelines) . . . 27
- 4.6 Save / load models . . . 27

5 Tutorial: use DeepProg from the docker image **29**
- 5.1 Installation with docker or harmony . . . 29
- 5.2 Alternative Image with R libraries . . . 29
- 5.3 Usage (Docker) . . . 30
- 5.4 Example . . . 30
- 5.5 Usage (Singularity) . . . 33

6 Case study: Analyzing TCGA HCC dataset **35**
- 6.1 Dataset preparation . . . 35
- 6.2 Model fitting . . . 36
- 6.3 Visualisation and analysis . . . 37

7 Tutorial: Tuning DeepProg **41**
7.1 A first example . . . 41
7.2 Tuning using one or multiple test datasets . . . 43
7.3 Results . . . 44
7.4 Recommendation . . . 44

8 License **45**

9 simdeep package **47**
9.1 Submodules . . . 47
9.2 simdeep.config module . . . 47
9.3 simdeep.coxph_from_r module . . . 47
9.4 simdeep.deepmodel_base module . . . 48
9.5 simdeep.extract_data module . . . 48
9.6 simdeep.plot_utils module . . . 49
9.7 simdeep.simdeep_analysis module . . . 49
9.8 simdeep.simdeep_boosting module . . . 49
9.9 simdeep.simdeep_distributed module . . . 49
9.10 simdeep.simdeep_multiple_dataset module . . . 49
9.11 simdeep.simdeep_utils module . . . 49
9.12 simdeep.survival_utils module . . . 50
9.13 Module contents . . . 51

10 Introduction **53**

11 Access **55**

12 Contribute **57**

13 Support **59**

14 Data availability **61**

15 Citation **63**

16 License **65**

Python Module Index **67**

Index **69**

CHAPTER

ONE

INSTALLATION

Here we describe how to install the DeepProg package. We assume that the installation will be done locally, using the `--user` flag from pip. Alternatively, the package can be installed using a virtual environment or globally with sudo. Both python2.7 or python3.6 (or higher) can be used. We tested the installation on a linux, OSX and Windows environment.

1.1 Requirements

- Python 2 or 3 (Python3 is recommended)
- Either theano, tensorflow or CNTK (tensorflow is recommended)
- theano (the used version for the manuscript was 0.8.2)
- tensorflow as a more robust alternative to theano
- cntk CNTK is anoter DL library that can present some advantages compared to tensorflow or theano. See https://docs.microsoft.com/en-us/cognitive-toolkit/
- scikit-learn (>=0.18)
- numpy, scipy
- lifelines
- (if using python3) scikit-survival
- (For distributed computing) ray (ray >= 0.8.4) framework
- (For hyperparameter tuning) scikit-optimize

1.2 Tested python package versions

Python 3.8 (tested for Linux and OSX. For Windows Visual C++ is required and LongPathsEnabled shoud be set to 1 in windows registry)

- tensorflow == 2.4.1 (2.4.1 currently doesn't seem to work with python3.9)
- keras == 2.4.3
- ray == 0.8.4
- scikit-learn == 0.23.2
- scikit-survival == 0.14.0 (currently doesn't seem to work with python3.9)
- lifelines == 0.25.5

- scikit-optimize == 0.8.1 (currently doesn't seem to work with python3.9)
- mpld3 == 0.5.1

Since ray and tensorflow are rapidly evolving libraries, newest versions might unfortunatly break DeepProg's API. To avoid any dependencies issues, we recommand working inside a Python 3 virtual environement (`virtualenv`) and install the tested packages.

1.2.1 installation (local)

```
# The downloading can take few minutes due to the size of th git project
git clone https://github.com/lanagarmire/DeepProg.git
cd DeepProg

# (RECOMMENDED) to install the tested python library versions
pip install -e . -r requirements_tested.txt

# Basic installation (under python3/pip3)
pip3 install -e . -r requirements.txt
# To intall the distributed frameworks
pip3 install -e . -r requirements_distributed.txt
# Installing scikit-survival (python3 only)
pip3 install -r requirements_pip3.txt
# Install ALL required dependencies with the most up to date packages
pip install -e . -r requirements_all.txt


# **Ignore this if you are working under python3**
# python 3 is highly preferred, but DeepProg working with python2/pip2, however there is␣
↪no support for scikit-survival in python2
pip2 install -e . -r requirements.txt
pip2 install -e . -r requirements_distributed.txt
```

1.2.2 Installation with docker

We have created a docker image (`opoirion/deepprog_docker:v1`) with all the dependencies already installed. For the docker (and singularity) instruction, please refer to the docker tutorial.

1.3 Alternative deep-Learning packages installation

The required python packages can be installed using pip:

```
pip install theano --user # Original backend used OR
pip install tensorflow --user # Alternative backend for keras and default
pip install keras --user
```

1.4 Alternative support for CNTK / theano / tensorflow

We originally used Keras with theano as backend plateform. However, Tensorflow (currently the defaut background DL framework) or CNTK are more recent DL framework that can be faster or more stable than theano. Because keras supports these 3 backends, it is possible to use them as alternative. To install CNTK, please refer to the official guidelines . To change backend, please configure the `$HOME/.keras/keras.json` file. (See official instruction here).

The default configuration file: `~/.keras/keras.json` looks like this:

```
{
    "image_data_format": "channels_last",
    "epsilon": 1e-07,
    "floatx": "float32",
    "backend": "tensorflow"
}
```

1.4.1 R installation (Alternative to Python lifelines)

In his first implementation, DeepProg used the R survival toolkits to fit the survival functions (cox-PH models) and compute the concordance indexes. These functions have been replaced with the python toolkits lifelines and scikit-survival for more convenience and avoid any compatibility issue. However, differences exists regarding the computation of the c-indexes using either python or R libraries. To use the original R functions, it is necessary to install the following R libraries.

- R
- the R "survival" package installed.
- rpy2 3.4.4 (for python2 rpy2 can be install with: pip install rpy2==2.8.6, for python3 pip3 install rpy2==2.8.6).

```
install.packages("survival")
install.packages("glmnet")
if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
BiocManager::install("survcomp")
```

Then, when instantiating a `SimDeep` or a `SimDeepBoosting` object, the option `use_r_packages` needs to be set to `True`.

1.5 Visualisation module (Experimental)

To visualise test sets projected into the multi-omic survival space, it is required to install `mpld3` module. Note that the pip version of mpld3 installed with pip on my computer presented a bug: `TypeError: array([1.]) is not JSON serializable`. However, the newest version of the mpld3 available from the github solved this issue. Rather than executing `pip install mpld3 --user` It is therefore recommended to install the newest version to avoid this issue directly from the github repository:

```
git clone https://github.com/mpld3/mpld3
cd mpld3
pip install -e . --user
```

1.5.1 Distributed computation

- It is possible to use the python ray framework https://github.com/ray-project/ray to control the parallel computation of the multiple models. To use this framework, it is required to install it: `pip install ray`
- Alternatively, it is also possible to create the model one by one without the need of the ray framework

1.5.2 Visualisation module (Experimental)

- To visualise test sets projected into the multi-omic survival space, it is required to install `mpld3` module: `pip install mpld3`
- Note that the pip version of mpld3 installed on my computer presented a bug: `TypeError: array([1.]) is not JSON serializable`. However, the newest version of the mpld3 available from the github solved this issue. It is therefore recommended to install the newest version to avoid this issue.

1.6 Usage

- test if simdeep is functional (all the software are correctly installed): go to main folder (./DeepProg/) and run the following

```
python3 test/test_simdeep.py -v #
```

- All the default parameters are defined in the config file: `./simdeep/config.py` but can be passed dynamically. Three types of parameters must be defined:
  - The training dataset (omics + survival input files)
    - In addition, the parameters of the test set, i.e. the omic dataset and the survival file
  - The parameters of the autoencoder (the default parameters works but it might be fine-tuned.
  - The parameters of the classification procedures (default are still good)

1.7 Example scripts

Example scripts are availables in ./examples/ which will assist you to build a model from scratch with test and real data:

```
examples
├── example_hyperparameters_tuning.py
├── example_hyperparameters_tuning_with_test_dataset.py
├── example_with_dummy_data_distributed.py
├── example_with_dummy_data.py
└── load_3_omics_model.py
```

CHAPTER

TWO

TUTORIAL: SIMPLE DEEPPROG MODEL

The principle of DeepProg can be summarized as follow:

- Loading of multiple samples x OMIC matrices
- Preprocessing ,normalisation, and sub-sampling of the input matrices
- Matrix transformation using autoencoder
- Detection of survival features
- Survival feature agglomeration and clustering
- Creation of supervised models to predict the output of new samples

2.1 Input parameters

All the default parameters are defined in the config file: `./simdeep/config.py` but can be passed dynamically. Three types of parameters must be defined:

- The training dataset (omics + survival input files)
  - In addition, the parameters of the test set, i.e. the omic dataset and the survival file
- The parameters of the autoencoder (the default parameters works but it might be fine-tuned.
- The parameters of the classification procedures (default are still good)

2.2 Input matrices

As examples, we included two datasets:

- A dummy example dataset in the `example/data/` folder:

```
examples
├── data
│   ├── meth_dummy.tsv
│   ├── mir_dummy.tsv
│   ├── rna_dummy.tsv
│   ├── rna_test_dummy.tsv
│   ├── survival_dummy.tsv
│   └── survival_test_dummy.tsv
```

- And a real dataset in the `data` folder. This dataset derives from the TCGA HCC cancer dataset. This dataset needs to be decompressed before processing:

```
data
├── meth.tsv.gz
├── mir.tsv.gz
├── rna.tsv.gz
└── survival.tsv
```

An input matrix file should follow this format:

```
head mir_dummy.tsv

Samples        dummy_mir_0     dummy_mir_1     dummy_mir_2     dummy_mir_3 ...
sample_test_0  0.469656032287  0.347987447237  0.706633335508  0.440068758445 ...
sample_test_1  0.0453108219657 0.0234642968791 0.593393816691  0.981872970341 ...
sample_test_2  0.908784043793  0.854397550009  0.575879144667  0.553333958713 ...
...
```

Also, if multiple matrices are used as input, they must keep the sample order. For example:

```
head rna_dummy.tsv

Samples        dummy_gene_0     dummy_gene_1     dummy_gene_2     dummy_gene_3 ...
sample_test_0  0.69656032287  0.47987447237  0.06633335508  0.40068758445 ...
sample_test_1  0.53108219657 0.234642968791 0.93393816691  0.81872970341 ...
sample_test_2  0.8784043793  0.54397550009  0.75879144667  0.53333958713 ...
...
```

The arguments `training_tsv` and `path_data` from the `extract_data` module are used to defined the input matrices.

```
# The keys/values of this dict represent the name of the omic and the corresponding␣
↪input matrix
training_tsv = {
    'GE': 'rna_dummy.tsv',
    'MIR': 'mir_dummy.tsv',
    'METH': 'meth_dummy.tsv',
}
```

a survival file must have this format:

```
head survival_dummy.tsv

barcode        days recurrence
sample_test_0  134  1
sample_test_1  291  0
sample_test_2  125  1
sample_test_3  43   0
...
```

In addition, the fields corresponding to the patient IDs, the survival time, and the event should be defined using the `survival_flag` argument:

```
#Default value
survival_flag = {'patient_id': 'barcode',
```

(continues on next page)

(continued from previous page)

```
                'survival': 'days',
               'event': 'recurrence'}
```

2.3 Creating a simple DeepProg model with one autoencoder for each omic

First, we will build a model using the example dataset from `./examples/data/` (These example files are set as default in the config.py file). We will use them to show how to construct a single DeepProg model inferring a autoencoder for each omic

```
# SimDeep class can be used to build one model with one autoencoder for each omic
from simdeep.simdeep_analysis import SimDeep
from simdeep.extract_data import LoadData

help(SimDeep) # to see all the functions
help(LoadData) # to see all the functions related to loading datasets

# Defining training datasets
from simdeep.config import TRAINING_TSV
from simdeep.config import SURVIVAL_TSV
# Location of the input matrices and survival file
from simdeep.config import PATH_DATA

dataset = LoadData(training_tsv=TRAINING_TSV,
        survival_tsv=SURVIVAL_TSV,
        path_data=PATH_DATA)

# Defining the result path in which will be created an output folder
PATH_RESULTS = "./TEST_DUMMY/"

# instantiate the model with the dummy example training dataset defined in the config␣
↪file
simDeep = SimDeep(
        dataset=dataset,
        path_results=PATH_RESULTS,
        path_to_save_model=PATH_RESULTS, # This result path can be used to save the␣
↪autoencoder
        )

simDeep.load_training_dataset() # load the training dataset
simDeep.fit() # fit the model
```

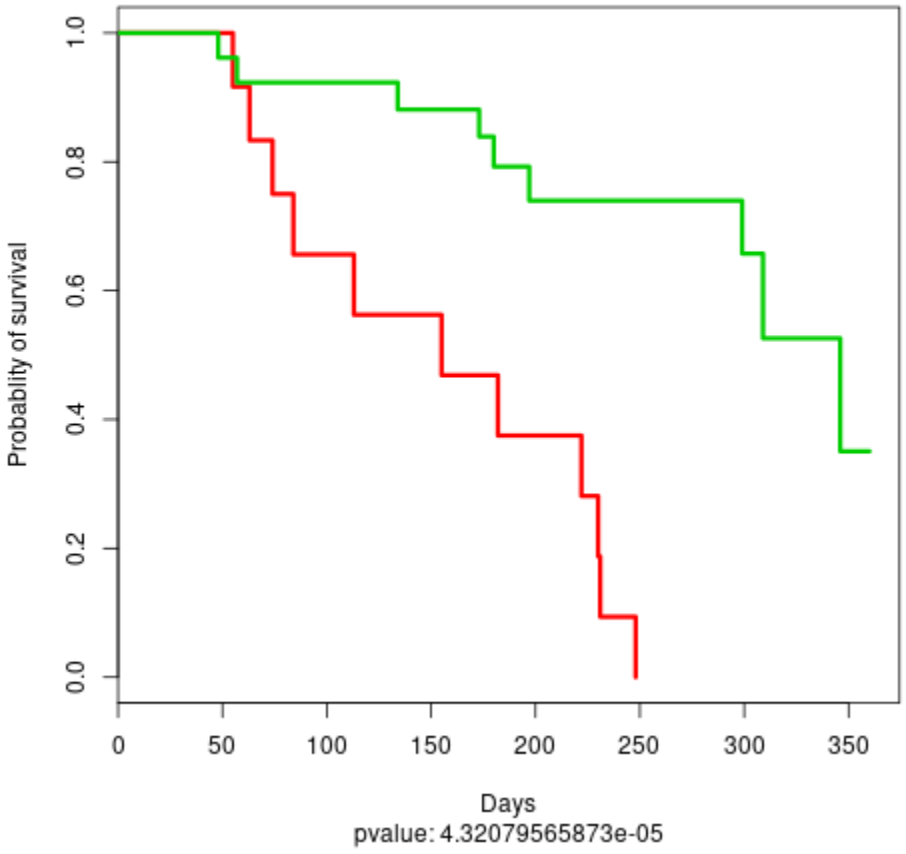
At that point, the model is fitted and some output files are available in the output folder:

```
TEST_DUMMY
├── test_dummy_dataset_KM_plot_training_dataset.png
└── test_dummy_dataset_training_set_labels.tsv
```

The tsv file contains the label and the label probability for each sample:

```
sample_test_0   1       7.22678272919e-12
sample_test_1   1       4.48594196888e-09
sample_test_4   1       1.53363205571e-06
sample_test_5   1       6.72170409655e-08
sample_test_6   0       0.9996581662
sample_test_7   1       3.38139255666e-08
```

And we also have the visualisation of a Kaplan-Meier Curve:



KM plot

Now we are ready to use a test dataset and to infer the class label for the test samples. The test dataset do not need to have the same input omic matrices than the training dataset and not even the sample features for a given omic. However, it needs to have at least some features in common.

```
# Defining test datasets
from simdeep.config import TEST_TSV
from simdeep.config import SURVIVAL_TSV_TEST

simDeep.load_new_test_dataset(
    TEST_TSV,
    fname_key='dummy',
    path_survival_file=SURVIVAL_TSV_TEST, # [OPTIONAL] test survival file useful to␣
↪compute accuracy of test dataset
```

(continues on next page)

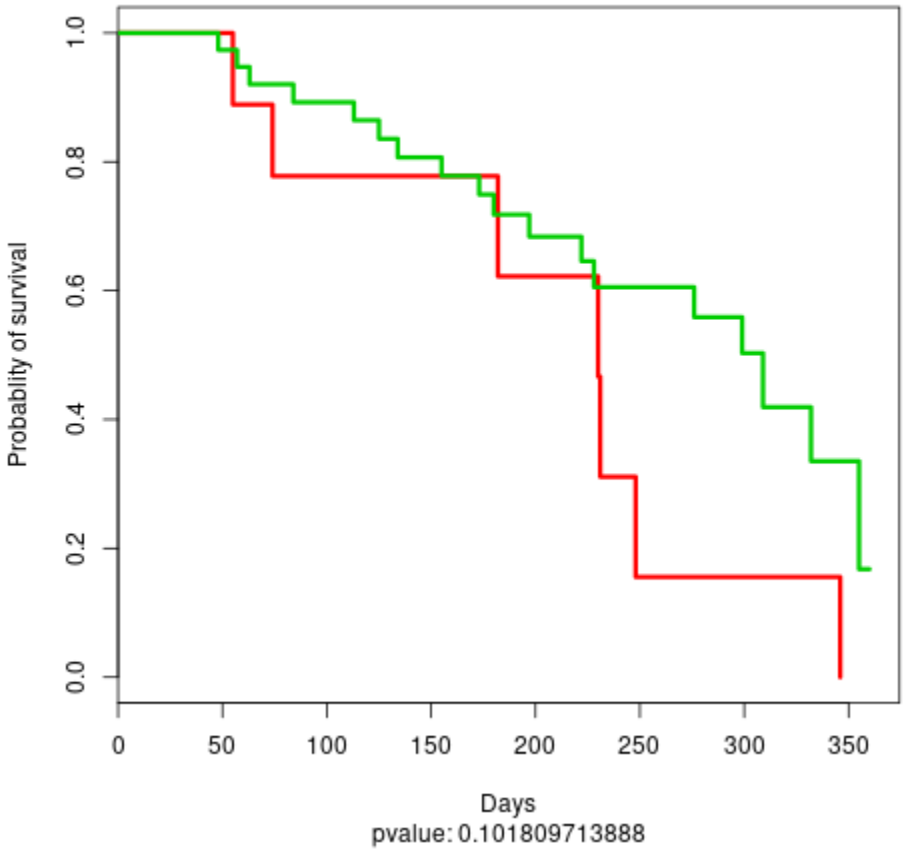
(continued from previous page)

```
    )

# The test set is a dummy rna expression (generated randomly)
print(simDeep.dataset.test_tsv) # Defined in the config file
# The data type of the test set is also defined to match an existing type
print(simDeep.dataset.data_type) # Defined in the config file
simDeep.predict_labels_on_test_dataset() # Perform the classification analysis and label␣
↪the set dataset

print(simDeep.test_labels)
print(simDeep.test_labels_proba)
```

The assigned class and class probabilities for the test samples are now available in the output folder:

```
TEST_DUMMY
├── test_dummy_dataset_dummy_KM_plot_test.png
├── test_dummy_dataset_dummy_test_labels.tsv
├── test_dummy_dataset_KM_plot_training_dataset.png
└── test_dummy_dataset_training_set_labels.tsv

head test_dummy_dataset_training_set_labels.tsv
```

And a KM plot is also constructed using the test labels



KM plot test

Finally, it is possible to save the keras model:

```
simDeep.save_encoders('dummy_encoder.h5')
```

CHAPTER

THREE

TUTORIAL: ENSEMBLE OF DEEPPROG MODEL

Secondly, we will build a more complex DeepProg model constituted of an ensemble of sub-models, each originated from a subset of the data. For that purpose, we need to use the `SimDeepBoosting` class:

```
from simdeep.simdeep_boosting import SimDeepBoosting

help(SimDeepBoosting)
```

Similarly, to the SimDeep class, we define our training dataset

```
# Location of the input matrices and survival file
from simdeep.config import PATH_DATA

from collections import OrderedDict

# Example tsv files
tsv_files = OrderedDict([
          ('MIR', 'mir_dummy.tsv'),
          ('METH', 'meth_dummy.tsv'),
          ('RNA', 'rna_dummy.tsv'),
])

# File with survival event
survival_tsv = 'survival_dummy.tsv'
```

3.1 Instanciation

Then, we define arguments specific to DeepProg and instanciate an instance of the class

```
project_name = 'stacked_TestProject'
epochs = 10 # Autoencoder epochs. Other hyperparameters can be fine-tuned. See the␣
↪example files
seed = 3 # random seed used for reproducibility
nb_it = 5 # This is the number of models to be fitted using only a subset of the␣
↪training data
nb_threads = 2 # These treads define the number of threads to be used to compute␣
↪survival function
PATH_RESULTS = "./"

boosting = SimDeepBoosting(
```

(continues on next page)

(continued from previous page)

```
    nb_threads=nb_threads,
    nb_it=nb_it,
    split_n_fold=3,
    survival_tsv=survival_tsv,
    training_tsv=tsv_files,
    path_data=PATH_DATA,
    project_name=project_name,
    path_results=PATH_RESULTS,
    epochs=epochs,
    seed=seed)
```

Here, we define a DeepProg model that will create 5 SimDeep instances each based on a subset of the original training dataset.the number of instance is defined by he `nb_it` argument. Other arguments related to the autoencoders construction can be defined during the class instanciation, such as `epochs`.

3.2 Fitting

Once the model is defined we can fit it

```
# Fit the model
boosting.fit()
# Predict and write the labels
boosting.predict_labels_on_full_dataset()
```

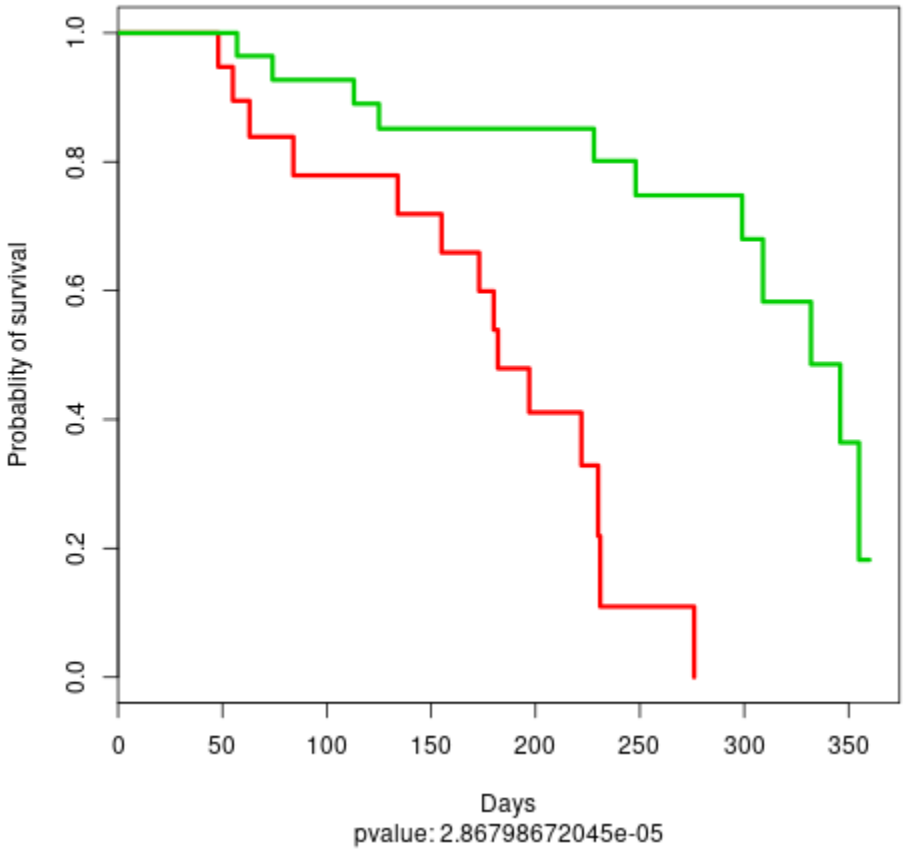
Some output files are generated in the output folder:

```
stacked_TestProject
├── stacked_TestProject_full_labels.tsv
├── stacked_TestProject_KM_plot_boosting_full.png
├── stacked_TestProject_proba_KM_plot_boosting_full.png
├── stacked_TestProject_test_fold_labels.tsv
└── stacked_TestProject_training_set_labels.tsv
```

The inferred labels, labels probability, survival time, and event are written in the `stacked_TestProject_full_labels.tsv` file:

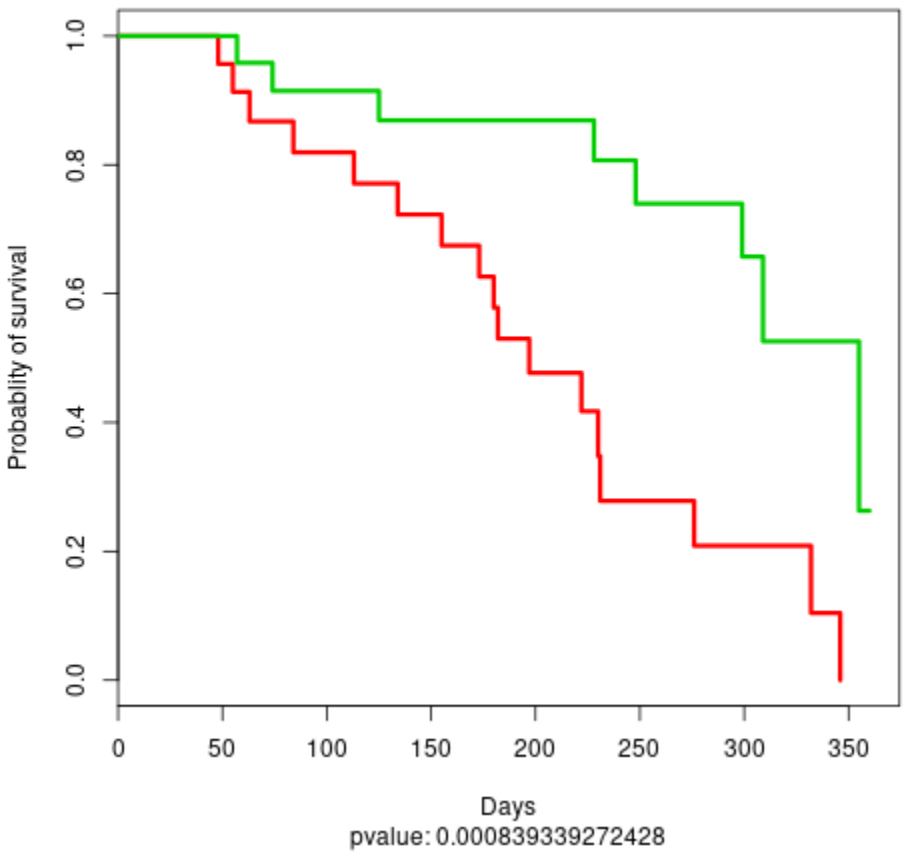
```
sample_test_48  1       0.474781026865  332.0   1.0
sample_test_49  1       0.142554926379  120.0   0.0
sample_test_46  1       0.355333486034  355.0   1.0
sample_test_47  0       0.618825352398  48.0    1.0
sample_test_44  1       0.346797097671  179.0   0.0
sample_test_45  1       0.0254692404734 360.0   0.0
sample_test_42  1       0.441997226254  346.0   1.0
sample_test_43  1       0.0783603292911 335.0   0.0
sample_test_40  1       0.380182410315  149.0   0.0
sample_test_41  0       0.953659261728  155.0   1.0
```

Note that the label probablity corresponds to the probability to belongs to the subtype with the lowest survival rate. Two KM plots are also generated, one using the cluster labels:



KM plot 3

and one using the cluster label probability dichotomized:



KM plot 4

We can also compute the feature importance per cluster:

```
# oOmpute the feature importance
boosting.compute_feature_scores_per_cluster()
# Write the feature importance
boosting.write_feature_score_per_cluster()
```

The results are updated in the output folder:

```
stacked_TestProject
├── stacked_TestProject_features_anticorrelated_scores_per_clusters.tsv
├── stacked_TestProject_features_scores_per_clusters.tsv
├── stacked_TestProject_full_labels.tsv
├── stacked_TestProject_KM_plot_boosting_full.png
├── stacked_TestProject_proba_KM_plot_boosting_full.png
├── stacked_TestProject_test_fold_labels.tsv
└── stacked_TestProject_training_set_labels.tsv
```

3.3 Evaluate the models

DeepProg allows to compute specific metrics relative to the ensemble of models:

```
# Compute internal metrics
boosting.compute_clusters_consistency_for_full_labels()

# Collect c-index
boosting.compute_c_indexes_for_full_dataset()
# Evaluate cluster performance
boosting.evalutate_cluster_performance()
# Collect more c-indexes
boosting.collect_cindex_for_test_fold()
boosting.collect_cindex_for_full_dataset()
boosting.collect_cindex_for_training_dataset()

# See Ave. number of significant features per omic across OMIC models
boosting.collect_number_of_features_per_omic()
```

3.4 Predicting on test dataset

We can then load and evaluate a first test dataset

```
boosting.load_new_test_dataset(
    {'RNA': 'rna_dummy.tsv'}, # OMIC file of the test set. It doesnt have to be the same␣
↪as for training
    'TEST_DATA_1', # Name of the test test to be used
    'survival_dummy.tsv', # [OPTIONAL] Survival file of the test set. USeful to compute␣
↪accuracy metrics on the test dataset
)

# Predict the labels on the test dataset
boosting.predict_labels_on_test_dataset()
# Compute C-index
boosting.compute_c_indexes_for_test_dataset()
# See cluster consistency
boosting.compute_clusters_consistency_for_test_labels()
```

We can load an evaluate a second test dataset

```
boosting.load_new_test_dataset(
    {'MIR': 'mir_dummy.tsv'}, # OMIC file of the test set. It doesnt have to be the same␣
↪as for training
    'TEST_DATA_2', # Name of the test test to be used
    'survival_dummy.tsv', # Survival file of the test set
)

# Predict the labels on the test dataset
boosting.predict_labels_on_test_dataset()
# Compute C-index
boosting.compute_c_indexes_for_test_dataset()
```

(continues on next page)

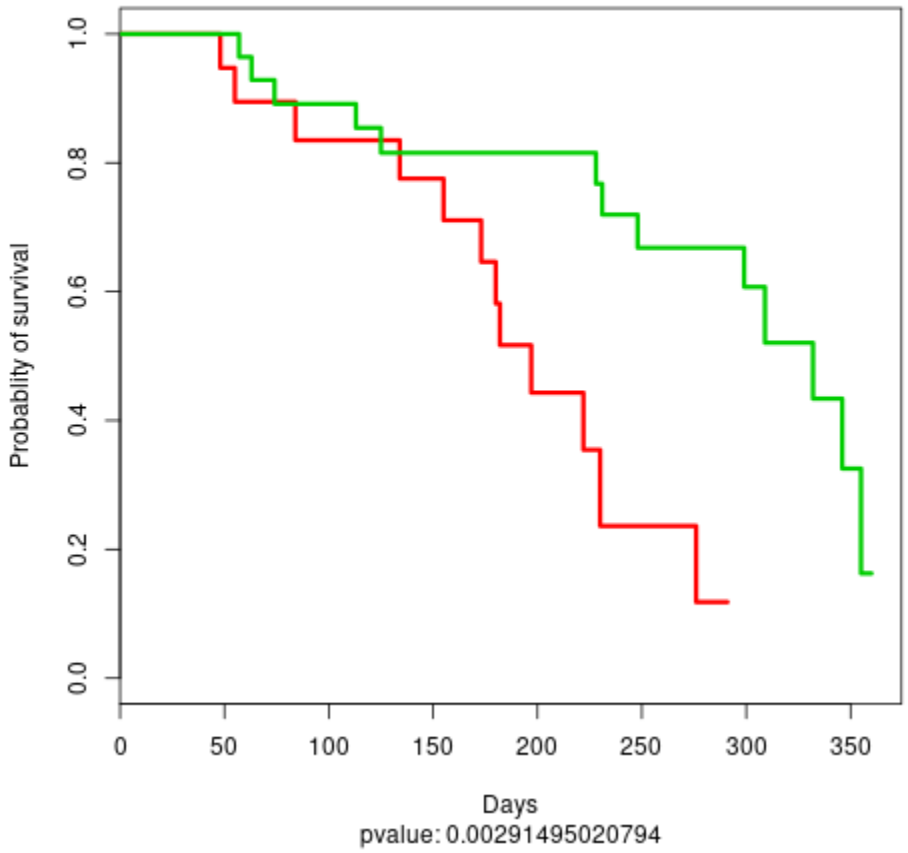
(continued from previous page)

```
# See cluster consistency
boosting.compute_clusters_consistency_for_test_labels()
```

The output folder is updated with the new output files

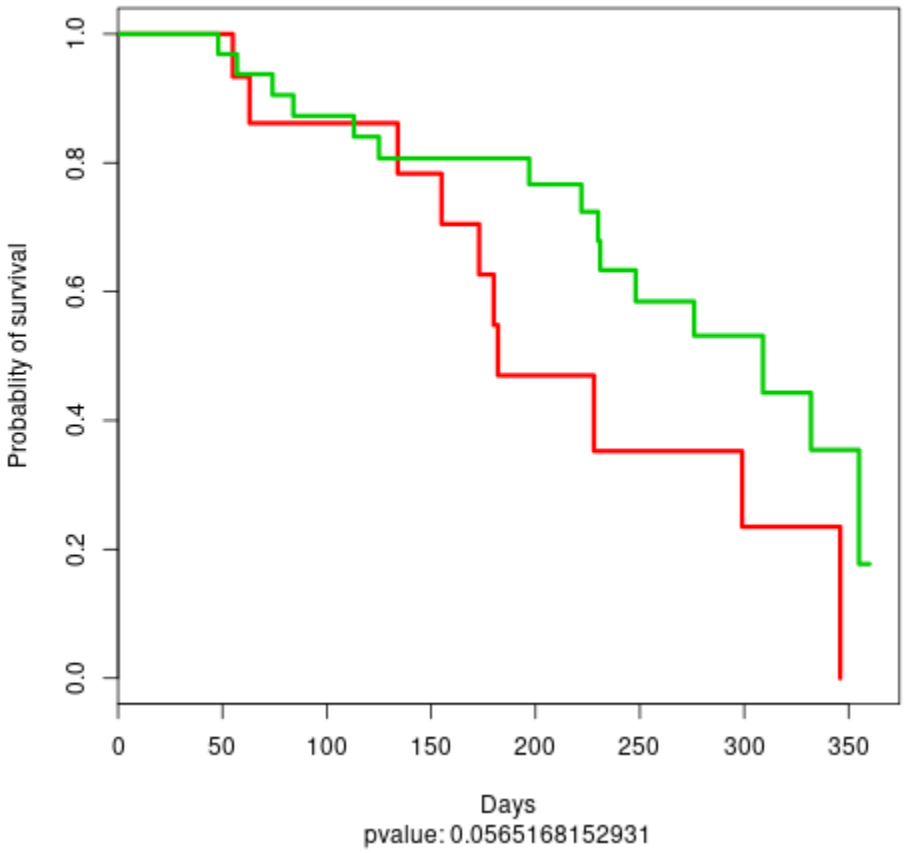
```
stacked_TestProject
├── stacked_TestProject_features_anticorrelated_scores_per_clusters.tsv
├── stacked_TestProject_features_scores_per_clusters.tsv
├── stacked_TestProject_full_labels.tsv
├── stacked_TestProject_KM_plot_boosting_full.png
├── stacked_TestProject_proba_KM_plot_boosting_full.png
├── stacked_TestProject_TEST_DATA_1_KM_plot_boosting_test.png
├── stacked_TestProject_TEST_DATA_1_proba_KM_plot_boosting_test.png
├── stacked_TestProject_TEST_DATA_1_test_labels.tsv
├── stacked_TestProject_TEST_DATA_2_KM_plot_boosting_test.png
├── stacked_TestProject_TEST_DATA_2_proba_KM_plot_boosting_test.png
├── stacked_TestProject_TEST_DATA_2_test_labels.tsv
├── stacked_TestProject_test_fold_labels.tsv
├── stacked_TestProject_test_labels.tsv
└── stacked_TestProject_training_set_labels.tsv
```

file: stacked_TestProject_TEST_DATA_1_KM_plot_boosting_test.png



test KM plot 1

file: stacked_TestProject_TEST_DATA_2_KM_plot_boosting_test.png



test KM plot 2

3.5 Distributed computation

Because SimDeepBoosting constructs an ensemble of models, it is well suited to distribute the individual construction of each SimDeep instance. To do such a task, we implemented the use of the ray framework that allow DeepProg to distribute the creation of each submodel on different clusters/nodes/CPUs. The configuration of the nodes / clusters, or local CPUs to be used needs to be done when instanciating a new ray object with the ray API. It is however quite straightforward to define the number of instances launched on a local machine such as in the example below in which 3 instances are used.

```
# Instanciate a ray object that will create multiple workers
import ray
ray.init(webui_host='0.0.0.0', num_cpus=3)
# More options can be used (e.g. remote clusters, AWS, memory,...etc...)
# ray can be used locally to maximize the use of CPUs on the local machine
# See ray API: https://ray.readthedocs.io/en/latest/index.html

boosting = SimDeepBoosting(
    ...
    distribute=True, # Additional option to use ray cluster scheduler
```

(continues on next page)

(continued from previous page)

```
    ...
)
...
# Processing
...

# Close clusters and free memory
ray.shutdown()
```

3.6 More examples

More example scripts are availables in ./examples/ which will assist you to build a model from scratch with test and real data:

```
examples
├── create_autoencoder_from_scratch.py # Construct a simple deeprog model on the dummy␣
↪example dataset
├── example_with_dummy_data_distributed.py # Process the dummy example dataset using ray
├── example_with_dummy_data.py # Process the dummy example dataset
└── load_3_omics_model.py # Process the example HCC dataset
```

CHAPTER

FOUR

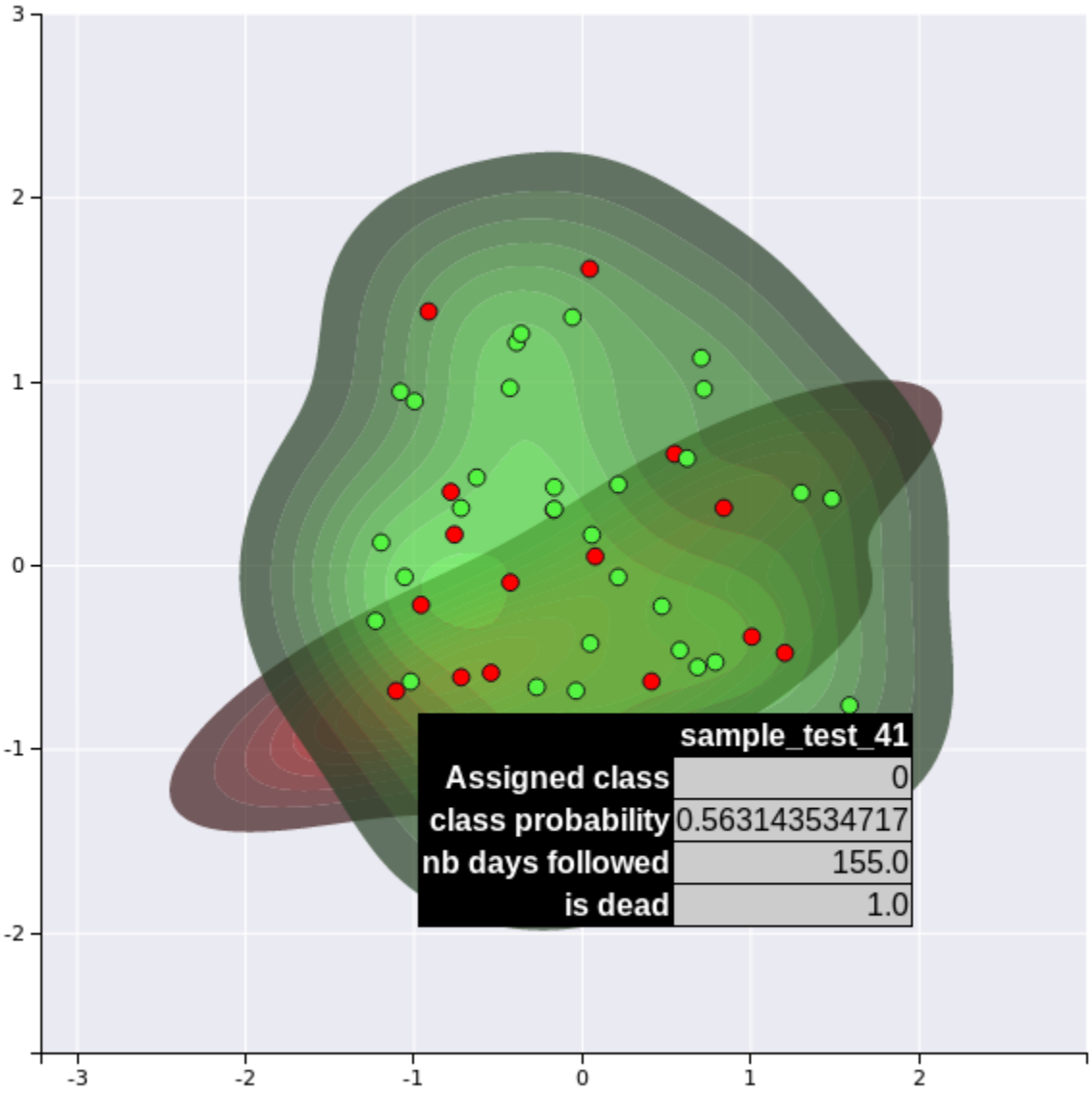
TUTORIAL: ADVANCED USAGE OF DEEPPROG MODEL

4.1 Visualisation

Once a DeepProg model is fitted, it might be interessant to obtain different visualisations of the samples for the training or the test sets, based on new survival features inferred by the autoencoders.For that purpose, we developped two methods to project the samples into a 2D space that can be called once a `SimDeepBoosting` or a `simDeep` is fitted.

```
# boosting class instance fitted using the ensemble tutorial
boosting.plot_supervised_predicted_labels_for_test_sets()
```

The first method transforms the OMIC matrix activities into the new survival feature space inferred by the autoencoders and projects the samples into a 2D space using PCA analysis. The figure creates a kernel density for each cluster and project the labels of the test set.

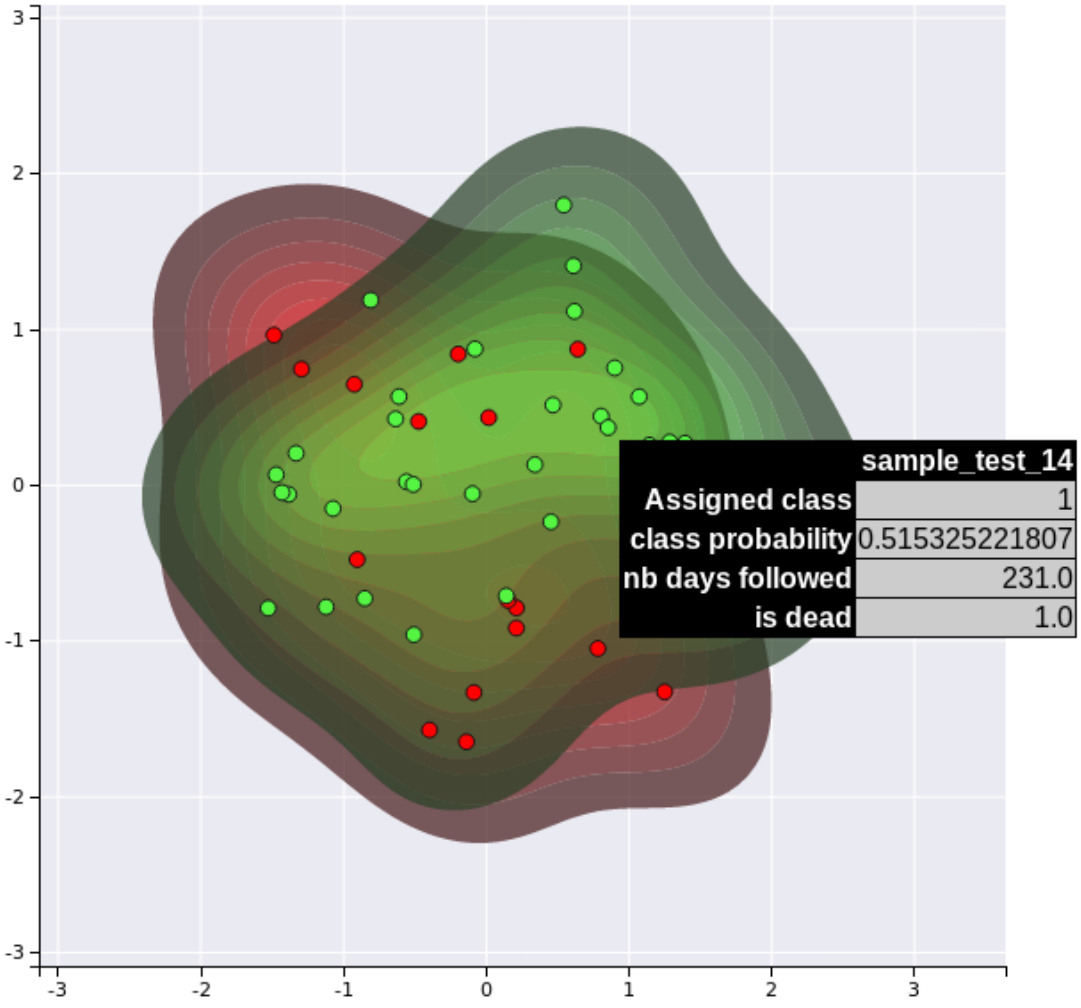


kdplot

1

A second more sophisticated method uses the new features inferred by the autoencoders to compute new features by constructing a supervised network targetting the inferred subtype labels. The new set of features are then projected into a 2D space using PCA analysis. This second method might present more efficient visualisations of the different clusters since it is uses a supervised algorithm.

```
boosting.plot_supervised_kernel_for_test_sets()
```



kdplot

2

Note that these visualisation are not very efficient in that example dataset, since we have only a limited number of samples (40) and features. However, they might become more useful for real datasets.

4.2 Hyperparameters

Hyperparameters can have a considerable influence on the accuracy of DeepProgs models. We set up the default hyperparameters to be used on a maximum of different datasets. However, specific datasets might require additional optimizations. Below, we are listing

4.2.1 Normalisation

DeepProg uses by default a four-step normalisation for both training and test datasets:

1. Selection of the top 100 features according to the variances
2. Rank normalisation per sample
3. Sample-sample Correlation similarity transformation
4. Rank normalisation

```
default_normalisation =  {
    'NB_FEATURES_TO_KEEP': 100,
    'TRAIN_RANK_NORM': True,
    'TRAIN_CORR_REDUCTION': True,
    'TRAIN_CORR_RANK_NORM': True,
}

boosting = SimDeepBoosting(
        normalization=default_normalisation
    )
```

However, it is possible to use other normalisation using external python classes that have `fit` and `fit_transform` methods.

```
from sklearn.preprocessing import RobustScaler

custom_norm =  {
    'CUSTOM': RobustScaler,
}

boosting = SimDeepBoosting(
        normalization=custom_norm
    )

    ```

Finally, more alternative normalisations are proposed in the config file.

Number of clusters

The parameters `nb_clusters` is used to define the number of partitions to produce

```python
#Example
boosting = SimDeepBoosting(
```

(continues on next page)

(continued from previous page)

```
    nb_clusters=3)
boosting.fit()
```

4.2.2 Clustering algorithm

By default, DeepProg is using a gaussian mixture model from the scikit-learn library to perform clustering. The hyperparameter of the model are customisable using the `mixture_params` parameter:

```
# Default params from the config file:

MIXTURE_PARAMS = {
    'covariance_type': 'diag',
    'max_iter': 1000,
    'n_init': 100
    }

boosting = SimDeepBoosting(
    mixture_params=MIXTURE_PARAMS,
    nb_clusters=3,
    cluster_method='mixture' # Default
    )
```

In addition to the gaussian mixture model, three alternative clustering approaches are available: a) `kmeans`, which refers to the scikit-learn KMeans class, b) `coxPH` which fits a L1 penalized multi-dimensional Cox-PH model and then dichotomize the samples into K groups using the predicted suvival times, and c) `coxPHMixture` which fit a Mixture model on the predicted survival time from the L1 penalized Cox-PH model. The L1 penalised Cox-PH model is fitted using scikit-survival `CoxnetSurvivalAnalysis`class for python3 so it cannot be computed when using python 2. Finally, external clustering class instances can be used as long as they have a `fit_predict` method returning an array of labels, and accepting a `nb_clusters` parameter.

```
# External clustering class having fit_predict method
from sklearn.cluster.hierarchical import AgglomerativeClustering

boostingH = SimDeepBoosting(
        nb_clusters=3,
        cluster_method=AgglomerativeClustering # Default
    )


class DummyClustering:
    self __init__(self, nb_clusters):
        """ """
        self.nb_clusters

    def fit_predict(M):
        """ """
        import numpy as np
        return np.random.randint(0, self.nb_clusters, M.shape[0])


boostingDummy = SimDeepBoosting(
```

(continues on next page)

(continued from previous page)

```
        nb_clusters=3,
        cluster_method=DummyClustering # Default
    )
```

4.2.3 Embedding and survival features selection

after each omic matrix is normalised, DeepProg transforms each feature matrix using by default an autoencoder network as embedding algorithm and then select the transformed features linked to survival using univariate Cox-PH models. Alternatively, DeepProg can accept any external embedding algorithm having a `fit` and transform `method`, following the scikit-learn nomenclature. For instance, `PCA` and `fastICA` classes of the scikit-learn package can be used as replacement for the autoencoder.

```
# Example using PCA as alternative embedding.


from scklearn.decomposition import PCA


boosting = SimDeepBoosting(
        nb_clusters=3,
        alternative_embedding=PCA,
    )
```

Another example is the use of the MAUI multi-omic method instead of the autoencoder

```
class MauiFitting():

    def __init__(self, **kwargs):
        """
        """
        self._kwargs = kwargs
        self.model = Maui(**kwargs)


    def fit(self, matrix):
        """ """
        self.model.fit({'cat': pd.DataFrame(matrix).T})

    def transform(self, matrix):
        """ """
        res = self.model.transform({'cat': pd.DataFrame(matrix).T})

        return np.asarray(res)

    boosting = SimDeepBoosting(
        nb_clusters=3,
        alternative_embedding=MauiFitting,
        ...
    )
```

After the embedding step, DeepProg is computing by default the individual feature contribution toward survival using univariate Cox-PH model (`feature_selection_usage='individual'`). Alterna-

tively, DeepProg can select features linked to survival using a l1-penalized multivariate Cox-PH model (`feature_selection_usage={'individual', 'lasso'}`). Finally if the option `feature_surv_analysis` is parsed as False, DeepProg will skip the survival feature selection step.

```
# Example using l1-penalized Cox-PH for selecting new survival features.


from scklearn.decomposition import PCA


boosting = SimDeepBoosting(
        nb_clusters=3,
        feature_selection_usage='individual'lasso',
        # feature_surv_analysis=False # Not using feature selection step
        ...
    )
```

4.2.4 Number of models and random splitting seed

A DeepProg model is constructed using an ensemble of submodels following the Bagging methodology. Each submodel is created from a random split of the input dataset. Three parameters control the creation of the random splits:

- `-nb_it <int>` which defines the number of sub-models to create
- and `-split_n_fold` which controls how the dataset will be splitted for each submodel. If `-split_n_fold=2`, the input dataset will be splitted in 2 using the `KFold` class instance from sciki-learn and the training /test set size ratio will be 0.5. If `-split_n_fold=3` the training /test set size ratio will be 3 / 2 and so on.
- The `-seed` parameter ensures to obtain the same random splitting for `split_n_fold` and `nb_it` constant for different DeepProg instances. Different seed values can produce different performances since it creates different training datasets and is especially true when using low `nb_it` (below 50). Unfortunalley, using large `nb_it` such as 100 can be very computationally intensive, especially when tuning the models with other hyperparameters. However, tuning the model with small `nb_it` is also OK to achieve good to optimal performances (see next section).

4.3 Usage of metadata associated with patients

DeepProg can accept an additional metadata file characterizing the individual sample (patient). These metdata can optionally be used as covariates when constructing the DeepProg models or inferring the features associated with each inferred subtypes. The metadata file should be a samples x features table with the first line as header with variable name and the first column the sample IDs. Also, the metadata file can be used to filter a subset of samples.

```
# See the example metadata table from the file: examples/data/metadata_dummy.tsv:

head examples/data/metadata_dummy.tsv

barcode sex     stage
sample_test_0   M       I
sample_test_1   M       I
sample_test_2   M       I
sample_test_3   M       I
```

(continues on next page)

(continued from previous page)

```
sample_test_4   M       I
sample_test_5   M       I
```

Each of the column features containing only numeric values will be scaled using the sklearn `RobustScaler` method. Each of the column having string values will be one-hot encoded using all the possible values of the given feature and stacked together.

The metadata file and the metadata usage should be configured at the instantiation of a new `DeepProg` instance.

```
    # metadata file
    OPTIONAL_METADATA = 'examples/data/metadata_dummy.tsv'
    # dictionary used to filter samples based on their metadata values
    # Multiple fields can be used
    SUBSET_TRAINING_WITH_META = {'stage': ['I', 'II', 'III']}

    boosting = SimDeepBoosting(
        survival_tsv=SURVIVAL_TSV,
        training_tsv=TRAINING_TSV,
        metadata_tsv=OPTIONAL_METADATA,
        metadata_usage='all',
        subset_training_with_meta=SUBSET_TRAINING_WITH_META,
        ...
        )
```

`metadata_usage` can have different values:

- `None` or `False`: the metadata will not be used for constructing DeepProg models or computing significant features
- `'"labels"'`: The metadata matrix will only be used as covariates when inferring the survival models from the infered clustering labels.
- `"new-features"`: The metadata matrix will only be used as covariates when computing the survival models to infer new features linked to survival
- `"test-labels"`: The metadata matrix will only be used as covariates when inferring the survival models from the labels obtained for the test datasets
- `"all"`, `True`: use the metadata matrix for all the usages described above.

4.4 Computing cluster-specific feature signatures

Once a DeepProg model is fitted, two functions can be used to infer the features signature of each subtype:

- `compute_feature_scores_per_cluster`: Perform a mann-Withney test between the expression of each feature within and without the subtype
- `compute_survival_feature_scores_per_cluster`: This function computes the Log-rank p-value after fitting an individual Cox-PH model for each of the significant features inferred by `compute_feature_scores_per_cluster`.

4.5 R installation (Alternative to Python lifelines)

In his first implementation, DeepProg used the R survival toolkits to fit the survival functions (cox-PH models) and compute the concordance indexes. These functions have been replaced with the python toolkits lifelines and scikit-survival for more convenience and avoid any compatibility issue. However, differences exists regarding the computation of the c-indexes using either python or R libraries. To use the original R functions, it is necessary to install the following R libraries.

- R
- the R "survival" package installed.
- rpy2 3.4.4 (for python2 rpy2 can be install with: pip install rpy2==2.8.6, for python3 pip3 install rpy2==2.8.6).

```
install.packages("survival")
install.packages("glmnet")
if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
BiocManager::install("survcomp")
```

Then, when instantiating a `SimDeep` or a `SimDeepBoosting` object, the option `use_r_packages` needs to be set to `True`.

```
    boosting = SimDeepBoosting(
    ...
        use_r_packages=True,
    ...
)
```

4.6 Save / load models

4.6.1 Save /load the entire model

Despite dealing with very voluminous data files, Two mechanisms exist to save and load dataset. First the models can be entirely saved and loaded using `dill` (pickle like) libraries.

```
from simdeep.simdeep_utils import save_model
from simdeep.simdeep_utils import load_model

# Save previous boosting model
save_model(boosting, "./test_saved_model")

# Delete previous model
del boosting

# Load model
boosting = load_model("TestProject", "./test_saved_model")
boosting.predict_labels_on_full_dataset()
```

See an example of saving/loading model in the example file: `load_and_save_models.py`

4.6.2 Save / load models from precomputed sample labels

However, this mechanism presents a huge drawback since the models saved can be very large (all the hyperparameters/matrices… etc… are saved). Also, the equivalent dependencies and DL libraries need to be installed in both the machine computing the models and the machine used to load them which can lead to various errors.

A second solution is to save only the labels inferred for each submodel instance. These label files can then be loaded into a new DeepProg instance that will be used as reference for building the classifier.

```
# Fitting a model
boosting.fit()
# Saving individual labels
boosting.save_test_models_classes(
    path_results=PATH_PRECOMPUTED_LABELS # Where to save the labels
    )

boostingNew = SimDeepBoosting(
        survival_tsv=SURVIVAL_TSV, # Same reference training set for `boosting` model
        training_tsv=TRAINING_TSV, # Same reference training set for `boosting` model
        path_data=PATH_DATA,
        project_name=PROJECT_NAME,
        path_results=PATH_DATA,
        distribute=False, # Option to use ray cluster scheduler (True or False)
    )

boostingNew.fit_on_pretrained_label_file(
    labels_files_folder=PATH_PRECOMPUTED_LABELS,
    file_name_regex="*.tsv")

boostingNew.predict_labels_on_full_dataset()
```

CHAPTER

FIVE

TUTORIAL: USE DEEPPROG FROM THE DOCKER IMAGE

We created a docker image with deepprog python dependencies installed. The docker image (`opoirion/deepprog_docker:v1`) can be downloaded using `docker pull` and used to analyse a multi-omic dataset. Alternatively, DeepProg image can be installed using the Singularity container engine.

5.1 Installation with docker or harmony

Docker or Singularity needs to be installed first.

```
# Using docker
docker pull opoirion/deepprog_docker:v1

# Using Singularity
singularity pull docker://opoirion/deepprog_docker:v1
# After singularity finishing pulling he image,  A SIF image (deeprog_docker_v1.sif)␣
↪should have been created within the local folder
```

The version of the package installed correspond to the versions described in the `requirements_tested.txt`. Thus, they are NOT the most up to date python packages, especially regarding the `ray` installed package (installed version is 0.8.4). Since ray is used to configure the nodes, memories, CPUs when distributing DeepProg in a cluster, the API to use might differ with the most up-to-date ray API.

5.2 Alternative Image with R libraries

We also created a docker image containing R and the survival R dependencies (`survival`, `survcomp`, and `glmnet`) installed. This image can be used with the option `use_r_packages=True`. However, this version is signficantly larger (1.3GiG) to install.

```
# Using alternative docker image with R libraries installed
docker pull opoirion/deepprog_docker:RVersion1

# Using Singularity
singularity pull docker://opoirion/deepprog_docker:RVersion1
```

5.3 Usage (Docker)

the docker container needs to have access to three folders:

1. the input folder containing the matrices and the survival data
2. the output folder where will be generated the output file
3. the folder containing the DeepProg python code to launch

```
docker run \
    -v <ABSOLUTE PATH FOR INPUT DATA>:/input \
     -v <ABSOLUTE PATH FOR OUTPUT DATA>:/output \
     -v <ABSOLUTE PATH FOR THE SCRIPT>:/code \
     --rm \ # remove the container once the computation is finished
     --name greedy_beaver \ # Name of the temporary docker process to create
     deepprog_docker \ # name of the DeepProg docker image to invoke
     python3.8 /code/<NAME OF THE PYTHON SCRIPT FILE>
```

5.4 Example

1. Create three folders for input, output, and scripts

```
cd $HOME
mkdir local_input
mkdir local_output
mkdir local_code
```

1. Go to `local_input` and download the matrices and survival data from the STAD cancer here `http://ns102669.ip-147-135-37.us/DeepProg/matrices/STAD/`

```
cd local_input
wget http://ns102669.ip-147-135-37.us/DeepProg/matrices/STAD/meth_mapped_STAD.tsv
wget http://ns102669.ip-147-135-37.us/DeepProg/matrices/STAD/mir_mapped_STAD.tsv
wget http://ns102669.ip-147-135-37.us/DeepProg/matrices/STAD/rna_mapped_STAD.tsv
wget http://ns102669.ip-147-135-37.us/DeepProg/matrices/STAD/surv_mapped_STAD.tsv
```

1. Go to `local_code` and open a text editor to create the following script named `processing_STAD.py`

```
### script: processing_STAD.py

# Import DeepProg class
from simdeep.simdeep_boosting import SimDeepBoosting

# Defining global variables for input and output paths the mounted folder from the␣
↪docker image
PATH_DATA = '/input/' # virtual folder. If using Singularity, This  should be the␣
↪existing path on the machine
PATH_RESULTS = '/output/' # virtual folder If using Singularity, This  should be the␣
↪existing path on the machine


# Defining a main function
```

(continues on next page)

(continued from previous page)

```
def main():
    """
    processing of STAD multiomic cancer
    """

    #Downloaded matrix files
    TRAINING_TSV = {
        'RNA': 'rna_mapped_STAD.tsv',
        'METH': 'meth_mapped_STAD.tsv',
        'MIR': 'mir_mapped_STAD.tsv'
        }

    #survival file
    SURVIVAL_TSV = 'surv_mapped_STAD.tsv'

    # survival flag
    survival_flag = {'patient_id': 'SampleID', 'survival': 'time','event': 'event'}

    # output folder name
    OUTPUT_NAME = 'STAD_docker'
    PROJECT_NAME = 'STAD_docker'

    # Import ray, the library that will distribute our model computation accros␣
↪different nodes
    import ray

    ray.init(
        webui_host='127.0.0.1', # This option is required when using ray from the docker␣
↪image
        num_cpus=10 #
        )

    # Random seed defining how the input dataset will be split
    SEED = 3
    # Number of DeepProg submodels to create
    nb_it = 10
    EPOCHS = 10

    boosting = SimDeepBoosting(
        nb_it=nb_it,
        split_n_fold=3,
        survival_flag=survival_flag,
        survival_tsv=SURVIVAL_TSV,
        training_tsv=TRAINING_TSV,
        path_data=PATH_DATA,
        project_name=PROJECT_NAME,
        path_results=PATH_RESULTS,
        epochs=EPOCHS,
        distribute=True, # Option to use ray cluster scheduler
        seed=SEED)

    # Fit the model
```

(continues on next page)

(continued from previous page)

```
    boosting.fit()
    # Save the labels of each submodels
    boosting.save_models_classes()
    boosting.save_cv_models_classes()

    # Predict labels on the full (trainings + cv splits) datasets
    boosting.predict_labels_on_full_dataset()

    # Compute consistency
    boosting.compute_clusters_consistency_for_full_labels()
    # Performance indexes
    boosting.evalutate_cluster_performance()
    boosting.collect_cindex_for_test_fold()
    boosting.collect_cindex_for_full_dataset()

    # Feature scores
    boosting.compute_feature_scores_per_cluster()
    boosting.collect_number_of_features_per_omic()

    boosting.write_feature_score_per_cluster()

    # Close clusters and free memory
    ray.shutdown()


# Excecute main function if this file is launched as a script
if __name__ == '__main__':
    main()
```

1. After saving this script, we are now ready to launch DeepProg using the docker image:

```
docker run \
    -v ~/local_input:/input \
     -v ~/local_output:/output \
     -v ~/local_code:/code \
     --rm \
     --name greedy_beaver \
     deepprog_docker \
     python3.8 /code/processing_STAD.py
```

1. After the execution, a new output folder inside `~/local_output` should have been created

```
ls ~/local_output/'STAD_docker

# Output
-rw-r--r-- 1 root root  22K Mar 30 07:37 STAD_docker_KM_plot_boosting_full.pdf
-rw-r--r-- 1 root root 830K Mar 30 07:37 STAD_docker_features_anticorrelated_scores_per_
↪clusters.tsv
-rw-r--r-- 1 root root 812K Mar 30 07:37 STAD_docker_features_scores_per_clusters.tsv
-rw-r--r-- 1 root root  16K Mar 30 07:37 STAD_docker_full_labels.tsv
-rw-r--r-- 1 root root  22K Mar 30 07:37 STAD_docker_proba_KM_plot_boosting_full.pdf
drwxr-xr-x 2 root root 4.0K Mar 30 07:37 saved_models_classes
drwxr-xr-x 2 root root 4.0K Mar 30 07:37 saved_models_cv_classes
```

5.5 Usage (Singularity)

Contrary to Docker, Singularity does not require to mount a specific volume for data sharing and

Then, the DeepProg docker can be invoked using the following command

```
singularity run \
     deepprog_docker_v1.sif \ # Path toward the downlaoded Singularity SIF image
     python3.8 <PYTHON SCRIPT>
```

If we want to use the example script `processing_STAD.py` described above with singularity, we just need to replace `PATH_DATA` and `PATH_RESULTS` with the paths on the machine.

1. the same methodology should be followed for adding more analyses, such as predicting a test dataset, embedding, or perform a hyperparameter tuning. Also, a better description of DeepProg different options is available in the other section of this tutorial

CHAPTER

SIX

CASE STUDY: ANALYZING TCGA HCC DATASET

In this example, we will use the RNA-Seq, miRNA, and DNA Methylation datsets from the TCGA HCC cancer dataset to perform subtype detection, identify subtype specific features, and fit supervised model that we will use to project the HCC samples using only the RNA-Seq OMIC layer. This real case dataset is available directly inside the `data` folder from the package.

6.1 Dataset preparation

First, locate the data folder and the compressed matrices:

```
data
├── meth.tsv.gz
├── mir.tsv.gz
├── rna.tsv.gz
└── survival.tsv
```

Go to that folder (**cd ./data/**) and decompress these files using `gzip -d *.gz`. Now, go back to the main folder (**cd ../**), and we are ready to instanciate a DeepProg instance.

```
from simdeep.simdeep_boosting import SimDeepBoosting
from simdeep.config import PATH_THIS_FILE

from collections import OrderedDict
from os.path import isfile

# specify your data path
path_data = './data/'

assert(isfile(path_data + "/meth.tsv"))
assert(isfile(path_data + "/rna.tsv"))
assert(isfile(path_data + "/mir.tsv"))

tsv_files = OrderedDict([
    ('MIR', 'mir.tsv'),
    ('METH', 'meth.tsv'),
    ('RNA', 'rna.tsv'),
])

# The survival file located also in the same folder
survival_tsv = 'survival.tsv'
```

(continues on next page)

(continued from previous page)

```
assert(isfile(path_data + "survival.tsv"))

# More attributes
PROJECT_NAME = 'HCC_dataset' # Name
EPOCHS = 10 # autoencoder fitting epoch
SEED = 10045 # random seed
nb_it = 10 # Number of submodels to be fitted
nb_threads = 2 # Number of python threads used to fit survival model
```

We need also to specify the columns to use from the survival file:

```
head data/survival.tsv

Samples days    event
TCGA.2V.A95S.01 0       0
TCGA.2Y.A9GS.01 724     1
TCGA.2Y.A9GT.01 1624    1
TCGA.2Y.A9GU.01 1939    0
TCGA.2Y.A9GV.01 2532    1
TCGA.2Y.A9GW.01 1271    1
TCGA.2Y.A9GX.01 2442    0
TCGA.2Y.A9GY.01 757     1
TCGA.2Y.A9GZ.01 848     1
```

```
survival_flag = {
    'patient_id': 'Samples',
    'survival': 'days',
    'event': 'event'}
```

Now we define a ray instance to distribute the fitting of the submodels

```
import ray
ray.init(webui_host='0.0.0.0', num_cpus=3)
```

6.2 Model fitting

We are now ready to instanciate a DeepProg instance and to fit a model

```
# Instanciate a DeepProg instance
boosting = SimDeepBoosting(
    nb_threads=nb_threads,
    nb_it=nb_it,
    split_n_fold=3,
    survival_tsv=survival_tsv,
    training_tsv=tsv_files,
    path_data=path_data,
    project_name=PROJECT_NAME,
    path_results=path_data,
```

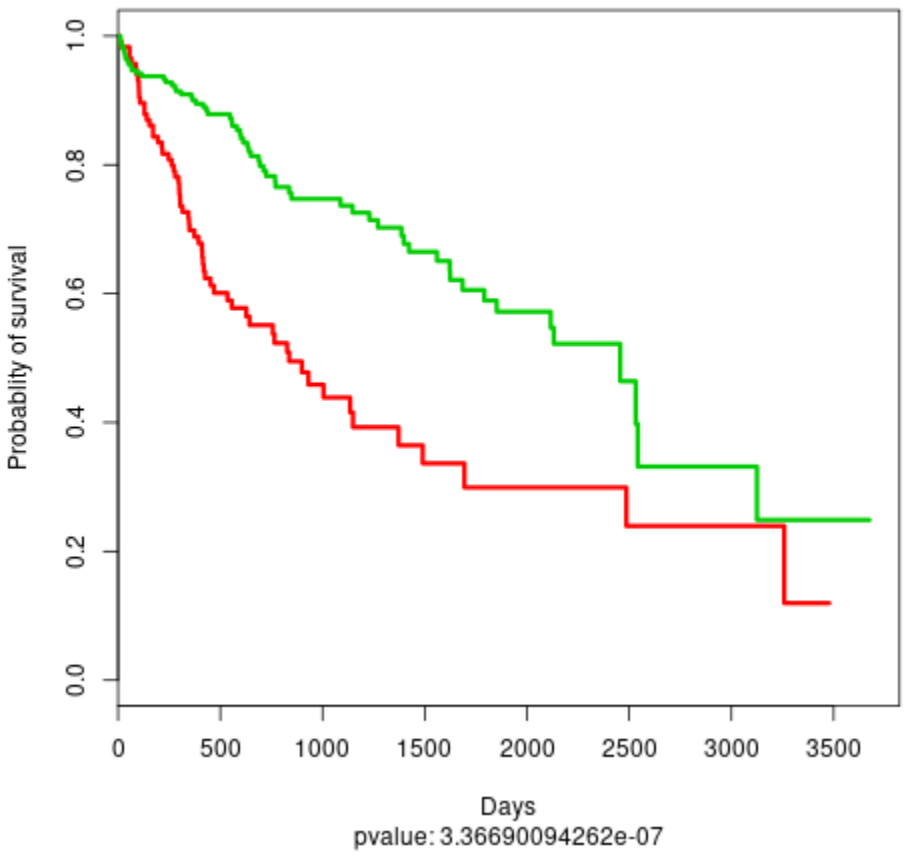
(continues on next page)

(continued from previous page)

```
    epochs=EPOCHS,
    survival_flag=survival_flag,
    distribute=True,
    seed=SEED)

boosting.fit()

# predict labels of the training

boosting.predict_labels_on_full_dataset()
boosting.compute_clusters_consistency_for_full_labels()
boosting.evalutate_cluster_performance()
boosting.collect_cindex_for_test_fold()
boosting.collect_cindex_for_full_dataset()

boosting.compute_feature_scores_per_cluster()
boosting.write_feature_score_per_cluster()
```

6.3 Visualisation and analysis

We should obtain subtypes with very significant survival differences, as we can see in the results located in the results folder



HCC KM plot

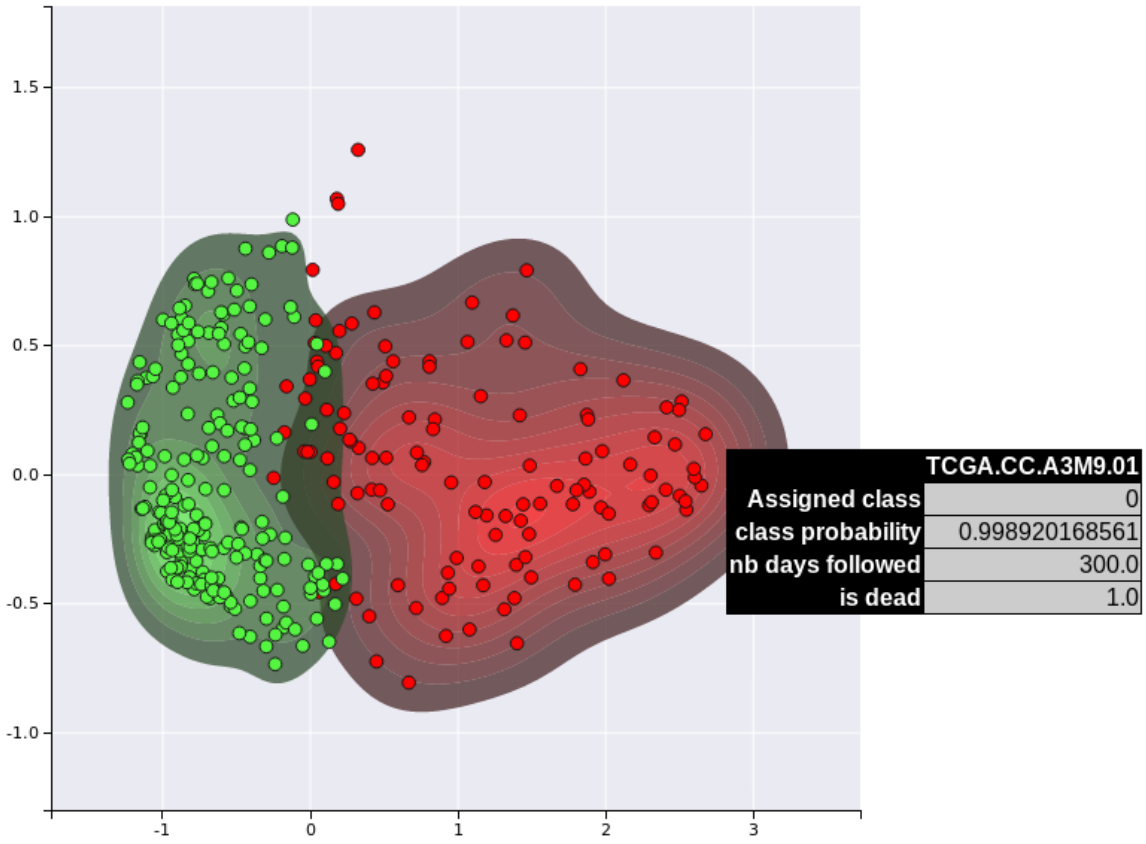
Now we might want to project the training samples using only the RNA-Seq layer

```
boosting.load_new_test_dataset(
    {'RNA': 'rna.tsv'},
    'test_RNA_only',
    survival_tsv,
)

boosting.predict_labels_on_test_dataset()
boosting.compute_c_indexes_for_test_dataset()
boosting.compute_clusters_consistency_for_test_labels()
```

We can use the visualisation functions to project our samples into a 2D space

```
# Experimental method to plot the test dataset amongst the class kernel densities
boosting.plot_supervised_kernel_for_test_sets()
boosting.plot_supervised_predicted_labels_for_test_sets()
```

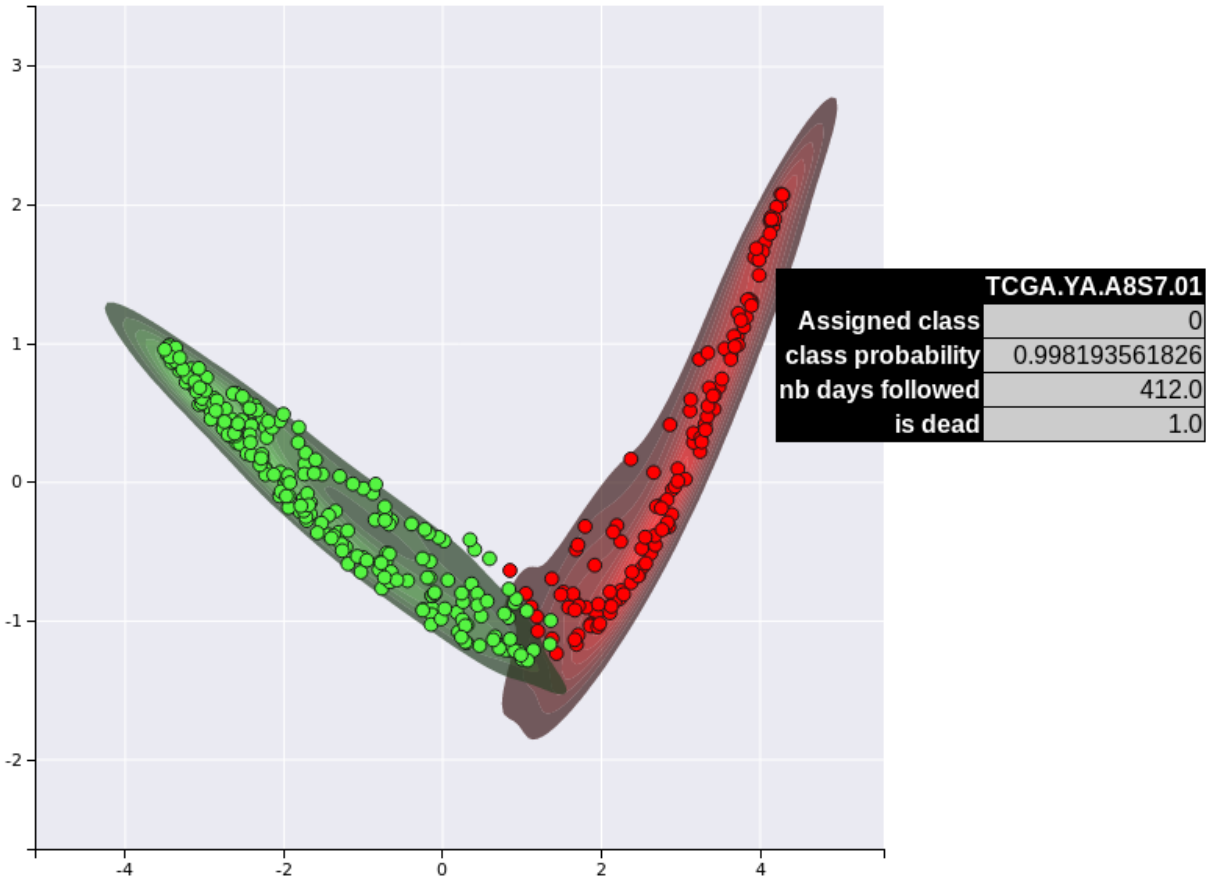
Results for unsupervised projection



Unsupervised

KDE plot

Results for supervised projection



Supervised KDE plot

CHAPTER

SEVEN

TUTORIAL: TUNING DEEPPROG

DeepProg can accept various alernative hyperparameters to fit a model, including alterative clustering, normalisation, embedding, choice of autoencoder hyperparameters, use/restrict embedding and survival selection, size of holdout samples, ensemble model merging criterion. Furthermore it can accept external methods to perform clustering / normalisation or embedding. To help ones to find the optimal combinaisons of hyperparameter for a given dataset, we implemented an optional hyperparameter search module based on sequencial-model optimisation search and relying on the tune and scikit-optimize python libraries. The optional hyperparameter tuning will perform a non-random itertive grid search and will select each new set of hyperparameters based on the performance of the past iterations. The computation can be entirely distributed thanks to the ray interace (see above).

A DeepProg instance depends on a lot of hyperparameters. Most important hyperparameters to tune are:

- The combination of `-nb_it` (Number of sumbmodels), `-split_n_fold`(How each submodel is randomly constructed) and `-seed` (random seed).
- The number of clusters `-nb_clusters`
- The clustering algorithm (implemented: `kmeans`, `mixture`, `coxPH`, `coxPHMixture`)
- The preprocessing normalization (`-normalization` option, see `Tutorial: Advanced usage of DeepProg model`)
- The embedding used (`alternative_embedding` option)
- The way of creating the new survival features (`-feature_selection_usage` option)

7.1 A first example

A first example of tuning is available in the example folder (example_hyperparameters_tuning.py). The first part of the script defines the array of hyperparameters to screen. An instance of `SimdeepTuning` is created in which the output folder and the project name are defined.

```
from simdeep.simdeep_tuning import SimDeepTuning

# AgglomerativeClustering is an external class that can be used as
# a clustering algorithm since it has a fit_predict method
from sklearn.cluster.hierarchical import AgglomerativeClustering

# Array of hyperparameters
args_to_optimize = {
    'seed': [100, 200, 300, 400],
    'nb_clusters': [2, 3, 4, 5],
```

(continues on next page)

(continued from previous page)

```
    'cluster_method': ['mixture', 'coxPH', 'coxPHMixture',
                       AgglomerativeClustering],
    'use_autoencoders': (True, False),
    'class_selection': ('mean', 'max'),
}

tuning = SimDeepTuning(
    args_to_optimize=args_to_optimize,
    nb_threads=nb_threads,
    survival_tsv=SURVIVAL_TSV,
    training_tsv=TRAINING_TSV,
    path_data=PATH_DATA,
    project_name=PROJECT_NAME,
    path_results=PATH_DATA,
)
```

The SimDeepTuning module requires the use of the `ray` and `tune` python modules.

```
ray.init(webui_host='0.0.0.0', )
```

7.1.1 SimDeepTuning hyperparameters

- `num_samples` is the number of experiements
- `distribute_deepprog` is used to further distribute each DeepProg instance into the ray framework. If set to True, be sure to either have a large number of CPUs to use and/or to use a small number of `max_concurrent` (which is the number of concurrent experiments run in parallel). `iterations` is the number of iterations to run for each experiment (results will be averaged).

DeepProg can be tuned using different objective metrics:

- `"log_test_fold_pvalue"`: uses the stacked *out of bags* samples (survival and labels) to predict the -log10(log-rank Cox-PH pvalue)
- `"log_full_pvalue"`: minimizes the Cox-PH log-rank pvalue of the model (This metric can lead to overfitting since it relies on all the samples included in the model)
- `"test_fold_cindex"`: Maximizes the mean c-index of the test folds.
- `"cluster_consistency"`: Maximizes the adjusted Rand scores computed for all the model pairs. (Higher values imply stable clusters)

```
tuning.fit(
    # We will use the holdout samples Cox-PH pvalue as objective
    metric='log_test_fold_pvalue',
    num_samples=35,
    # Experiment run concurently using ray as dispatcher
    max_concurrent=2,
    # In addition, each deeprog model will be distributed
    distribute_deepprog=True,
    iterations=1)

# We recommend using large `max_concurrent` and distribute_deepprog=True
```

(continues on next page)

(continued from previous page)

```
# when a large number CPUs and large RAMs are availables

# Results
table = tuning.get_results_table()
print(table)
```

7.2 Tuning using one or multiple test datasets

The computation of labels and the associate metrics from external test datasets can be included in the tuning workflowand be used as objective metrics. Please refers to example folder (see example_hyperparameters_tuning_with_dataset.py).

Let's define two dummy test datasets:

```
    # We will use the methylation and the RNA value as test datasets
    test_datasets = {
        'testdataset1': ({'METH': 'meth_dummy.tsv'}, 'survival_dummy.tsv')
        'testdataset2': ({RNA: rna_dummy.tsv'}, 'survival_dummy.tsv')
    }
```

We then include these two datasets when instanciating the `SimDeepTuning` instance:

```
    tuning = SimDeepTuning(
        args_to_optimize=args_to_optimize,
        test_datasets=test_datasets,
        survival_tsv=SURVIVAL_TSV,
        training_tsv=TRAINING_TSV,
        path_data=PATH_DATA,
        project_name=PROJECT_NAME,
        path_results=PATH_DATA,
    )
```

and Finally fits the model using a objective metric accounting for the test datasets:

- `"log_test_pval"` maximizes the sum of the -log10(log-rank Cox-PH pvalue) for each test dataset
- `"test_cindex"` maximizes the mean on the test C-indexes
- `"sum_log_pval"` maximizes the sum of the model -log10(log-rank Cox-PH pvalue) with all the test datasets p-value
- `"mix_score"`: maximizes the product of `"sum_log_pval"`, `"cluster_consistency"`, `"test_fold_cindex"`

```
    tuning.fit(
        metric='log_test_pval',
        num_samples=10,
        distribute_deepprog=True,
        max_concurrent=2,
        # iterations is usefull to take into account the DL parameter fitting variations
        iterations=1,
    )
```

(continues on next page)

(continued from previous page)

```
table = tuning.get_results_table()
tuning.save_results_table()
```

7.3 Results

The results will be generated in the `path_results` folder and one results folder per experiement will be generated. The report of all the experiements and metrics will be written in the result tables generated in the `path_results` folder. Once a model achieve satisfactory performance, it is possible to directly use the model by loading the generated labels with the `fit_on_pretrained_label_file` API (see the section `Save / load models from precomputed sample labels`)

7.4 Recommendation

- According to the number of the size N of the hyperparameter array(e.g. the number of combination), it is recommanded to perform at least more than sqrt(N) experiment but a higher N will always allow to explore a higher hyperparameter space and increase the performance.
- `seed` is definitively a hyperparameter to screen, especially for small number of models `nb_its` (less than 50). It is recommanded to at least screen for 8-10 different seed when using `nb_it` < 20
- Please, test you configuration using a small `num_samples` first

CHAPTER

EIGHT

LICENSE

PolyForm Perimeter License 1.0.0

Copyright (c) 2020 Poirion

<https://polyformproject.org/licenses/perimeter/1.0.0>

Acceptance

In order to get any license under these terms, you must agree to them as both strict obligations and conditions to all your licenses.

Copyright License

The licensor grants you a copyright license for the software to do everything you might do with the software that would otherwise infringe the licensor's copyright in it for any permitted purpose. However, you may only distribute the software according to [Distribution License](#distribution-license) and make changes or new works based on the software according to [Changes and New Works License](#changes-and-new-works-license).

Distribution License

The licensor grants you an additional copyright license to distribute copies of the software. Your license to distribute covers distributing the software with changes and new works permitted by [Changes and New Works License](#changes-and-new-works-license).

Notices

You must ensure that anyone who gets a copy of any part of the software from you also gets a copy of these terms or the URL for them above, as well as copies of any plain-text lines beginning with *Required Notice:* that the licensor provided with the software. For example:

> Required Notice: Copyright Yoyodyne, Inc. (http://example.com)

Changes and New Works License

The licensor grants you an additional copyright license to make changes and new works based on the software for any permitted purpose.

Patent License

The licensor grants you a patent license for the software that covers patent claims the licensor can license, or becomes able to license, that you would infringe by using the software.

Noncompete

Any purpose is a permitted purpose, except for providing to others any product that competes with the software.

Competition

If you use this software to market a product as a substitute for the functionality or value of the software, it competes with the software. A product may compete regardless how it is designed or deployed. For example, a product may

compete even if it provides its functionality via any kind of interface (including services, libraries or plug-ins), even if it is ported to a different platforms or programming languages, and even if it is provided free of charge.

Fair Use

You may have “fair use” rights for the software under the law. These terms do not limit them.

No Other Rights

These terms do not allow you to sublicense or transfer any of your licenses to anyone else, or prevent the licensor from granting licenses to anyone else. These terms do not imply any other licenses.

Patent Defense

If you make any written claim that the software infringes or contributes to infringement of any patent, your patent license for the software granted under these terms ends immediately. If your company makes such a claim, your patent license ends immediately for work on behalf of your company.

Violations

The first time you are notified in writing that you have violated any of these terms, or done anything with the software not covered by your licenses, your licenses can nonetheless continue if you come into full compliance with these terms, and take practical steps to correct past violations, within 32 days of receiving notice. Otherwise, all your licenses end immediately.

No Liability

As far as the law allows, the software comes as is, without any warranty or condition, and the licensor will not be liable to you for any damages arising out of these terms or the use or nature of the software, under any kind of legal claim.

Definitions

The **licensor** is the individual or entity offering these terms, and the **software** is the software the licensor makes available under these terms.

A **product** can be a good or service, or a combination of them.

You refers to the individual or entity agreeing to these terms.

Your company is any legal entity, sole proprietorship, or other kind of organization that you work for, plus all organizations that have control over, are under the control of, or are under common control with that organization. **Control** means ownership of substantially all the assets of an entity, or the power to direct its management and policies by vote, contract, or otherwise. Control can be direct or indirect.

Your licenses are all the licenses granted to you for the software under these terms.

Use means anything you do with the software requiring one of your licenses.

CHAPTER

NINE

SIMDEEP PACKAGE

9.1 Submodules

9.2 simdeep.config module

9.3 simdeep.coxph_from_r module

simdeep.coxph_from_r.**c_index**(*values*, *isdead*, *nbdays*, *values_test*, *isdead_test*, *nbdays_test*, *isfactor=False*, *use_r_packages=False*, *seed=None*)

simdeep.coxph_from_r.**c_index_from_python**(*values*, *isdead*, *nbdays*, *values_test*, *isdead_test*, *nbdays_test*, *isfactor=False*)

simdeep.coxph_from_r.**c_index_from_r**(*values*, *isdead*, *nbdays*, *values_test*, *isdead_test*, *nbdays_test*, *isfactor=False*)

simdeep.coxph_from_r.**c_index_multiple**(*values*, *isdead*, *nbdays*, *values_test*, *isdead_test*, *nbdays_test*, *isfactor=False*, *use_r_packages=False*, *seed=None*)

simdeep.coxph_from_r.**c_index_multiple_from_python**(*matrix*, *isdead*, *nbdays*, *matrix_test*, *isdead_test*, *nbdays_test*, *isfactor=False*)

simdeep.coxph_from_r.**c_index_multiple_from_r**(*matrix*, *isdead*, *nbdays*, *matrix_test*, *isdead_test*, *nbdays_test*, *lambda_val=None*, *isfactor=False*)

simdeep.coxph_from_r.**convert_to_rmatrix**(*data*)

simdeep.coxph_from_r.**coxph**(*values*, *isdead*, *nbdays*, *do_KM_plot=False*, *metadata_mat=None*, *png_path='./'*, *dichotomize_afterward=False*, *fig_name='KM_plot.png'*, *isfactor=False*, *use_r_packages=False*, *seed=None*)

simdeep.coxph_from_r.**coxph_from_python**(*values*, *isdead*, *nbdays*, *do_KM_plot=False*, *png_path='./'*, *metadata_mat=None*, *dichotomize_afterward=False*, *fig_name='KM_plot.pdf'*, *penalizer=0.01*, *l1_ratio=0.0*, *isfactor=False*)

simdeep.coxph_from_r.**coxph_from_r**(*values*, *isdead*, *nbdays*, *do_KM_plot=False*, *metadata_mat=None*, *png_path='./'*, *dichotomize_afterward=False*, *fig_name='KM_plot.png'*, *isfactor=False*)

input:

values array values of activities

isdead array <binary> Event occured int boolean: 0/1

nbdays array <int>

return: pvalues from wald test

simdeep.coxph_from_r.**main**()
DEBUG

simdeep.coxph_from_r.**predict_with_coxph_glmnet**(*matrix*, *isdead*, *nbdays*, *matrix_test*, *alpha=0.5*, *lambda_val=None*)

simdeep.coxph_from_r.**surv_mean**(*isdead*, *nbdays*, *use_r_packages=False*)

simdeep.coxph_from_r.**surv_mean_from_python**(*isdead*, *nbdays*)

simdeep.coxph_from_r.**surv_mean_from_r**(*isdead*, *nbdays*)

simdeep.coxph_from_r.**surv_median**(*isdead*, *nbdays*, *use_r_packages=False*)

simdeep.coxph_from_r.**surv_median_from_python**(*isdead*, *nbdays*)

simdeep.coxph_from_r.**surv_median_from_r**(*isdead*, *nbdays*)

9.4 simdeep.deepmodel_base module

9.5 simdeep.extract_data module

class simdeep.extract_data.**LoadData**(*path_data='/home/docs/checkouts/readthedocs.org/user_builds/deepprog-garmires-lab/checkouts/stable/simdeep/../examples/data/'*, *training_tsv={'GE': 'rna_dummy.tsv', 'METH': 'meth_dummy.tsv', 'MIR': 'mir_dummy.tsv'}*, *survival_tsv='survival_dummy.tsv'*, *metadata_tsv=None*, *metadata_test_tsv=None*, *test_tsv={'MIR': 'mir_test_dummy.tsv'}*, *survival_tsv_test='survival_test_dummy.tsv'*, *cross_validation_instance=KFold(n_splits=5, random_state=1, shuffle=True)*, *test_fold=0*, *stack_multi_omic=False*, *fill_unkown_feature_with_0=True*, *normalization={'NB_FEATURES_TO_KEEP': 100, 'TRAIN_CORR_RANK_NORM': True, 'TRAIN_CORR_REDUCTION': True, 'TRAIN_MAD_SCALE': False, 'TRAIN_MIN_MAX': False, 'TRAIN_NORM_SCALE': False, 'TRAIN_QUANTILE_TRANSFORM': False, 'TRAIN_RANK_NORM': True, 'TRAIN_ROBUST_SCALE': False, 'TRAIN_ROBUST_SCALE_TWO_WAY': False}*, *survival_flag={'event': 'recurrence', 'patient_id': 'barcode', 'survival': 'days'}*, *subset_training_with_meta={}*, *_autoencoder_parameters={}*, *verbose=True*)

Bases: object

create_a_cv_split()

load_array()

load_matrix_full()

load_matrix_test(*normalization=None*)

load_matrix_test_fold()

load_meta_data(*sep='\t'*)

load_meta_data_test(*metadata_file=''*, *sep='\t'*)

load_new_test_dataset(*tsv_dict*, *path_survival_file=None*, *survival_flag=None*, *normalization=None*, *metadata_file=None*)

load_survival()

load_survival_test(*survival_flag=None*)

normalize_training_array()

reorder_matrix_array(*new_sample_ids*)

save_ref_matrix(*path_folder*, *project_name*)

subset_training_sets(*change_cv=False*)

transform_matrices(*matrix_ref*, *matrix*, *key*, *normalization=None*)

9.6 simdeep.plot_utils module

class simdeep.plot_utils.**SampleHTML**(*name*, *label*, *proba*, *survival*)
Bases: object

simdeep.plot_utils.**make_color_dict**(*id_list*)
According to an id_list define a color gradient return {id:color}

simdeep.plot_utils.**make_color_dict_from_r**(*labels*)

simdeep.plot_utils.**make_color_list**(*id_list*)
According to an id_list define a color gradient return {id:color}

simdeep.plot_utils.**plot_kernel_plots**(*test_labels*, *test_labels_proba*, *labels*, *activities*, *activities_test*, *dataset*, *path_html*, *metadata_frame=None*)
perform a html kernel plot

9.7 simdeep.simdeep_analysis module

9.8 simdeep.simdeep_boosting module

9.9 simdeep.simdeep_distributed module

9.10 simdeep.simdeep_multiple_dataset module

9.11 simdeep.simdeep_utils module

simdeep.simdeep_utils.**feature_selection_usage_type**(*value*)

simdeep.simdeep_utils.**load_labels_file**(*path_labels*, *sep='\t'*)

simdeep.simdeep_utils.**load_model**(*project_name*, *path_model='./'*)

simdeep.simdeep_utils.**metadata_usage_type**(*value*)

simdeep.simdeep_utils.**save_model**(*boosting*, *path_to_save_model='./'*)

9.12 simdeep.survival_utils module

class simdeep.survival_utils.**CorrelationReducer**(*distance='correlation'*, *threshold=None*)
Bases: object

fit(*dataset*)

fit_transform(*dataset*)

transform(*dataset*)

class simdeep.survival_utils.**MadScaler**
Bases: object

fit_transform(*X*)

class simdeep.survival_utils.**RankCorrNorm**(*dataset*)
Bases: object

class simdeep.survival_utils.**RankNorm**
Bases: object

fit_transform(*X*)

class simdeep.survival_utils.**SampleReducer**(*perc_sample_to_keep=0.9*)
Bases: object

sample_to_keep(*datasets*, *index=None*)

class simdeep.survival_utils.**VarianceReducer**(*nb_features=200*)
Bases: object

fit(*dataset*)

fit_transform(*dataset*)

transform(*dataset*)

simdeep.survival_utils.**convert_metadata_frame_to_matrix**(*frame*)

simdeep.survival_utils.**load_data_from_tsv**(*use_transpose=False*, ***kwargs*)

simdeep.survival_utils.**load_entrezID_to_ensg**()

simdeep.survival_utils.**load_survival_file**(*f_name*, *path_data='/home/docs/checkouts/readthedocs.org/user_builds/deepprog-garmires-lab/checkouts/stable/simdeep/../examples/data/'*, *sep='\t'*, *survival_flag={'event': 'recurrence', 'patient_id': 'barcode', 'survival': 'days'}*)

simdeep.survival_utils.**return_intersection_indexes**(*ids_1*, *ids_2*)

simdeep.survival_utils.**save_matrix**(*matrix*, *feature_array*, *sample_array*, *path_folder*, *project_name*, *key=''*, *sep='\t'*)

simdeep.survival_utils.**select_best_classif_params**(*clf*)
select best classifier parameters based uniquely on test errors

simdeep.survival_utils.**translate_index**(*original_ids*, *new_ids*)

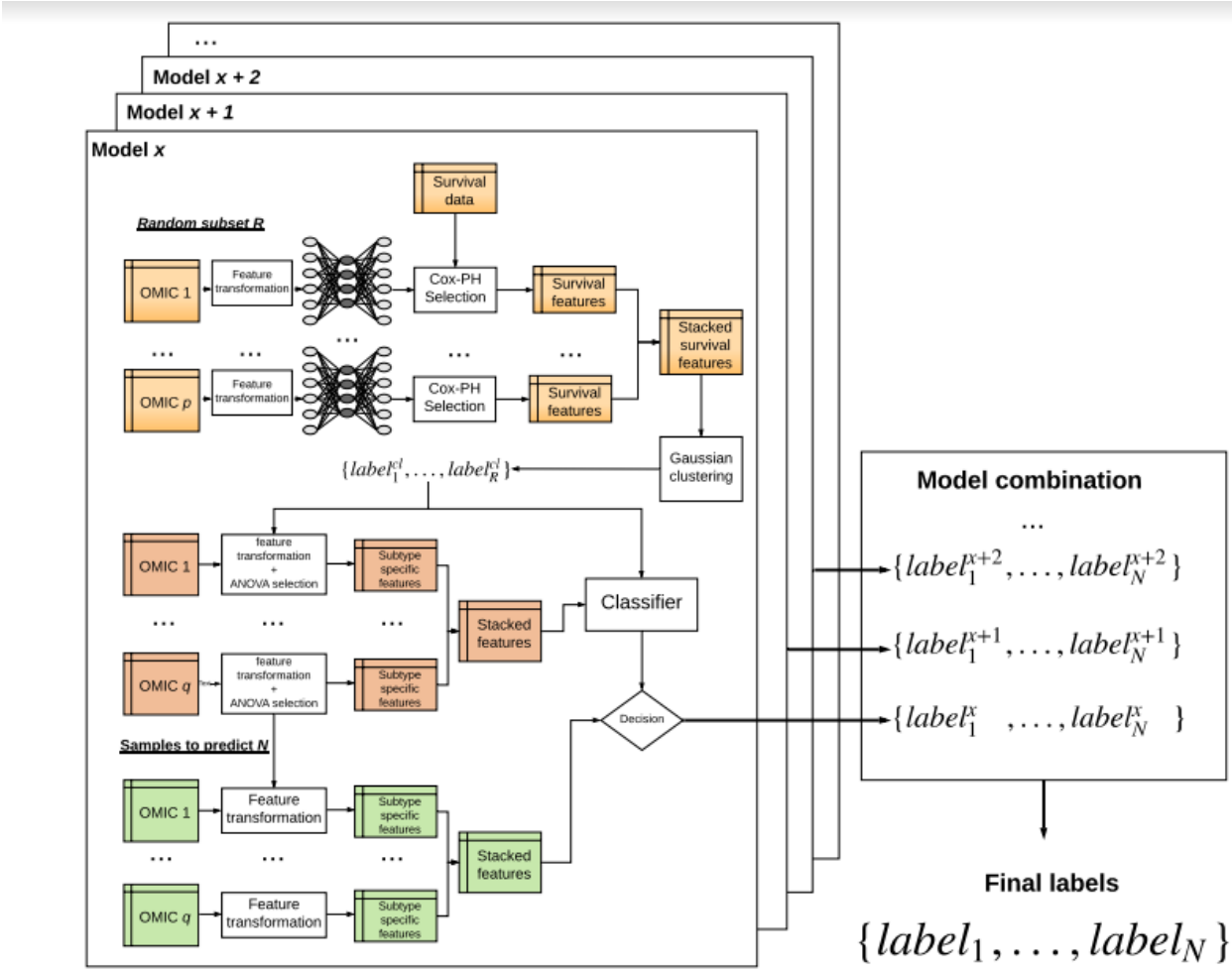
9.13 Module contents

CHAPTER

TEN

INTRODUCTION

This package allows to combine multi-omics data for individual samples together with survival. Using autoencoders (default) or any alternative embedding methods, the pipeline creates new set of features and identifies those linked with survival. In a second time, the samples are clustered with different possible strategies to obtain robust subtypes linked to survival. The robustness of the obtained subtypes can then be tested externally on one or multiple validation datasets and/or the *out-of-bags* samples. The omic data used in the original study are RNA-Seq, MiR and Methylation. However, this approach can be extended to any combination of omic data. The current package contains the omic data used in the study and a copy of the model computed. However, it is easy to recreate a new model from scratch using any combination of omic data. The omic data and the survival files should be in tsv (Tabular Separated Values) format and examples are provided. The deep-learning framework to produce the autoencoders uses Keras with either Theano / tensorflow/ CNTK as background.


...
Model x + 2
Model x + 1
Model x
Survival data
Random subset R
OMIC 1
Feature transformation
Cox-PH Selection
Survival features
OMIC p
Feature transformation
Cox-PH Selection
Survival features
Stacked survival features
Gaussian clustering
$\{label_1^{cl}, \ldots, label_R^{cl}\}$
OMIC 1
feature transformation + ANOVA selection
Subtype specific features
OMIC q
feature transformation + ANOVA selection
Subtype specific features
Stacked features
Classifier
Decision
Samples to predict N
OMIC 1
Feature transformation
Subtype specific features
OMIC q
Feature transformation
Subtype specific features
Stacked features
Model combination
...
$\{label_1^{x+2}, \ldots, label_N^{x+2}\}$
$\{label_1^{x+1}, \ldots, label_N^{x+1}\}$
$\{label_1^{x} \quad, \ldots, label_N^{x} \quad\}$
Final labels
$\{label_1, \ldots, label_N\}$

CHAPTER

ELEVEN

ACCESS

The package is accessible at this link: https://github.com/lanagarmire/DeepProg.

CHAPTER

TWELVE

CONTRIBUTE

- Issue Tracker: github.com/lanagarmire/DeepProg/issues
- Source Code: github.com/lanagarmire/DeepProg

CHAPTER

THIRTEEN

SUPPORT

If you are having issues, please let us know. You can reach us using the following email address:

Olivier Poirion, Ph.D. o.poirion@gmail.com

CHAPTER

FOURTEEN

DATA AVAILABILITY

The matrices and the survival data used to compute the models are available here: http://ns102669.ip-147-135-37.us/DeepProg/matrices/

CHAPTER

FIFTEEN

CITATION

This package refers to our preprint paper: Multi-omics-based pan-cancer prognosis prediction using an ensemble of deep-learning and machine-learning models, https://www.medrxiv.org/content/10.1101/19010082v1

CHAPTER

SIXTEEN

LICENSE

The project is licensed under the MIT license.

PYTHON MODULE INDEX

S

simdeep, 51
simdeep.config, 47
simdeep.coxph_from_r, 47
simdeep.extract_data, 48
simdeep.plot_utils, 49
simdeep.simdeep_utils, 49
simdeep.survival_utils, 50

INDEX

C

c_index() (*in module simdeep.coxph_from_r*), 47
c_index_from_python() (*in module simdeep.coxph_from_r*), 47
c_index_from_r() (*in module simdeep.coxph_from_r*), 47
c_index_multiple() (*in module simdeep.coxph_from_r*), 47
c_index_multiple_from_python() (*in module simdeep.coxph_from_r*), 47
c_index_multiple_from_r() (*in module simdeep.coxph_from_r*), 47
convert_metadata_frame_to_matrix() (*in module simdeep.survival_utils*), 50
convert_to_rmatrix() (*in module simdeep.coxph_from_r*), 47
CorrelationReducer (*class in simdeep.survival_utils*), 50
coxph() (*in module simdeep.coxph_from_r*), 47
coxph_from_python() (*in module simdeep.coxph_from_r*), 47
coxph_from_r() (*in module simdeep.coxph_from_r*), 47
create_a_cv_split() (*simdeep.extract_data.LoadData method*), 48

F

feature_selection_usage_type() (*in module simdeep.simdeep_utils*), 49
fit() (*simdeep.survival_utils.CorrelationReducer method*), 50
fit() (*simdeep.survival_utils.VarianceReducer method*), 50
fit_transform() (*simdeep.survival_utils.CorrelationReducer method*), 50
fit_transform() (*simdeep.survival_utils.MadScaler method*), 50
fit_transform() (*simdeep.survival_utils.RankNorm method*), 50
fit_transform() (*simdeep.survival_utils.VarianceReducer method*), 50

L

load_array() (*simdeep.extract_data.LoadData method*), 48
load_data_from_tsv() (*in module simdeep.survival_utils*), 50
load_entrezID_to_ensg() (*in module simdeep.survival_utils*), 50
load_labels_file() (*in module simdeep.simdeep_utils*), 49
load_matrix_full() (*simdeep.extract_data.LoadData method*), 48
load_matrix_test() (*simdeep.extract_data.LoadData method*), 48
load_matrix_test_fold() (*simdeep.extract_data.LoadData method*), 48
load_meta_data() (*simdeep.extract_data.LoadData method*), 48
load_meta_data_test() (*simdeep.extract_data.LoadData method*), 49
load_model() (*in module simdeep.simdeep_utils*), 49
load_new_test_dataset() (*simdeep.extract_data.LoadData method*), 49
load_survival() (*simdeep.extract_data.LoadData method*), 49
load_survival_file() (*in module simdeep.survival_utils*), 50
load_survival_test() (*simdeep.extract_data.LoadData method*), 49
LoadData (*class in simdeep.extract_data*), 48

M

MadScaler (*class in simdeep.survival_utils*), 50
main() (*in module simdeep.coxph_from_r*), 48
make_color_dict() (*in module simdeep.plot_utils*), 49
make_color_dict_from_r() (*in module simdeep.plot_utils*), 49
make_color_list() (*in module simdeep.plot_utils*), 49

metadata_usage_type() (*in module simdeep.simdeep_utils*), 49
module
    simdeep, 51
    simdeep.config, 47
    simdeep.coxph_from_r, 47
    simdeep.extract_data, 48
    simdeep.plot_utils, 49
    simdeep.simdeep_utils, 49
    simdeep.survival_utils, 50

N

normalize_training_array() (*simdeep.extract_data.LoadData method*), 49

P

plot_kernel_plots() (*in module simdeep.plot_utils*), 49
predict_with_coxph_glmnet() (*in module simdeep.coxph_from_r*), 48

R

RankCorrNorm (*class in simdeep.survival_utils*), 50
RankNorm (*class in simdeep.survival_utils*), 50
reorder_matrix_array() (*simdeep.extract_data.LoadData method*), 49
return_intersection_indexes() (*in module simdeep.survival_utils*), 50

S

sample_to_keep() (*simdeep.survival_utils.SampleReducer method*), 50
SampleHTML (*class in simdeep.plot_utils*), 49
SampleReducer (*class in simdeep.survival_utils*), 50
save_matrix() (*in module simdeep.survival_utils*), 50
save_model() (*in module simdeep.simdeep_utils*), 49
save_ref_matrix() (*simdeep.extract_data.LoadData method*), 49
select_best_classif_params() (*in module simdeep.survival_utils*), 50
simdeep
    module, 51
simdeep.config
    module, 47
simdeep.coxph_from_r
    module, 47
simdeep.extract_data
    module, 48
simdeep.plot_utils
    module, 49
simdeep.simdeep_utils
    module, 49
simdeep.survival_utils
    module, 50
subset_training_sets() (*simdeep.extract_data.LoadData method*), 49
surv_mean() (*in module simdeep.coxph_from_r*), 48
surv_mean_from_python() (*in module simdeep.coxph_from_r*), 48
surv_mean_from_r() (*in module simdeep.coxph_from_r*), 48
surv_median() (*in module simdeep.coxph_from_r*), 48
surv_median_from_python() (*in module simdeep.coxph_from_r*), 48
surv_median_from_r() (*in module simdeep.coxph_from_r*), 48

T

transform() (*simdeep.survival_utils.CorrelationReducer method*), 50
transform() (*simdeep.survival_utils.VarianceReducer method*), 50
transform_matrices() (*simdeep.extract_data.LoadData method*), 49
translate_index() (*in module simdeep.survival_utils*), 50

V

VarianceReducer (*class in simdeep.survival_utils*), 50