xiRT

Release 1.0.33

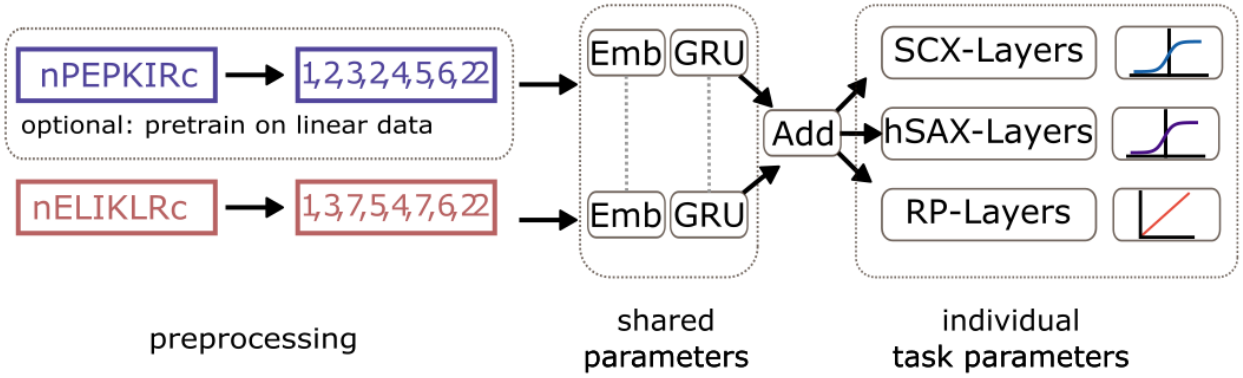
Jan 30, 2022

Contents:

xiRT is a versatile python package for multi-dimensional retention time prediction for linear and crosslinked peptides.

xiRT requires identified peptide sequences with an assigned confidence (FDR) to learn the retention behavior from multiple dimensions. The high confidence identifications are necessary to reduce the noise in the data which allows more accurate retention time prediction. However, typically we want to supply higher FDR (>1%) data to also predict the retention times for peptide spectrum matches where the search score was not sufficient for passing the FDR cutoff. Post-search validation algorithms such as percolator can then be used to rescore the given set of PSMs with the predicted retention times.

Approach.



xiRT uses a deep neural network architecture to realize the simultaneous learning for multiple retention times. In brief, xiRT builds a multi-layer network that can be divided into a Siamese part and individual task subnetworks. The Siamese part takes the peptide sequences as input and applies an embedding and recurrent function to the input. For linear peptides the output of the recurrent layer is directly forwarded to the task subnetworks. For crosslinked peptides, each peptide has its own input and after the recurrent layer the two outputs are first combined and then passed towards the individual task networks. In contrast, to typical regression models the input data (peptide) sequences are not transformed into features but rather the entire peptide sequence including modifications is used as input.

Supported Prediction Tasks

xiRT is versatile in the input and experimental design. An arbitrary number of prefractionation methods are supported as well as a standard reversed phase RT prediction. In addition, similar tasks such has collision-cross section prediction can be learned.

CHAPTER 1

xiRT - Introduction

xiRT is a deep learning tool to predict the retention times(s) of linear and crosslinked peptides from multiple fractionation dimensions including RP (typically coupled to the mass spectrometer). xiRT was developed with a combination of SCX / hSAX / RP chromatography. However, xiRT supports all available chromatography methods.

1.1 Description

xiRT is meant to be used to generate additional information about CSMs for machine learning-based rescoring frameworks but the usage can be extended to spectral libraries, targeted acquisitions etc.

xiRT offers several training / prediction modes that need to be configured depending on the use case. At the moment training, prediction, crossvalidation are the supported modes. - *training*: trains xiRT on the input CSMs (using 10% for validation) and stores a trained model - *prediction*: use a pretrained model and predict RTs for the input CSMs - *crossvalidation*: load/train a model and predict RTs for all data points without using them in the training process. Requires the training of several models during CV.

Note: all modes can be supplemented by using a pre-trained model ("transfer learning").

Installation To install xiRT simply run the command below. We recommend to use an isolated python environment, for example by using pipenv **or** conda. Installation should finish within minutes.

Using pipenv: >pipenv shell > >pip install xirt

To enable CUDA support, using a [conda environment](https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html#creating-an-environment-with-commands) is the easiest solution. Conda will take care of the CUDA libraries and other dependencies. Note, xiRT runs either on CPUs or GPUs. To use a GPU specify CuDNNGRU/CuDNNLSTM as type in the LSTM settings, to use a CPU set the type to GRU/LSTM.

> conda create –name xirt_env python=3.8 > >conda activate xirt_env > > pip install xirt

Hint: The plotting functionality for the network is not enabled per default because pydot and graphviz sometimes make trouble when they are installed via pip. If on linux, simply use *sudo apt-get install graphviz*, on windows download latest graphviz package from [here](https://www2.graphviz.org/Packages/stable/windows/), unzip the content of the file and the *bin* directory path to the windows PATH variable. These two packages allow the visualization of the neural network architecture. xiRT will function also without this functionality.

Older versions of TensorFlow will require the separate installation of tensorflow-gpu. We recommend to install tensorflow in conda, especially if GPU usage is desired.

CHAPTER 2

General Usage

The command line interface (CLI) requires three inputs:

1) input spectra matches file (CSMs or PSMs)
2) a YAML file to configure the neural network architecture
3) another YAML file to configure the general training / prediction behaviour, called setup-config

Configs are available via github). Alternatively, up-to-date configs can be generated using the xiRT package itself:

```
> xirt -p learning_params.yaml
>
> xirt -s xirt_params.yaml
```

To use xiRT these options are combined as shown below:

```
>xirt -i CSMs.csv -o out_dir -x xirt_params.yaml -l learning_params.yaml
```

To adapt xiRT's parameters, a yaml config file needs to be prepared. The configuration file determines network parameters (number of neurons, layers, regularization) but also defines the prediction task (classification / regression / ordinal regression). In general, the default values do not need to be changed for standard use cases. Depending on the decoding of the target variable, the output layers may need to be adapted. For a usual RP prediction, regression is essentially the only viable option. For SCX / hSAX (e.g. general classification from fractionation experiments) the prediction task can be formulated as classification, regression or ordinal regression. For the usage of regression for fractionation, we recommend to use fraction-specific eluent concentrations as prediction target variable (raw fraction numbers are also possible). Please see some examples below to learn more about the different parameterizations.

2.1 Quick start

The GitHub repository contains a few example files. Download the following files from HERE:

- DSS_xisearch_fdr_CSM50percent_minimal.csv
- xirt_params_rp.yaml

- learning_params_training_cv.yaml

This set of files can now be used to perform a RP (only) prediction on crosslink data. To run xiRT on the data call the main function as follows after successfull installation:

```
> xirt -i DSS_xisearch_fdr_CSM50percent_minimal.csv -o xirt_results/ -x xirt_params_
↪rp.yaml -l learning_params_training_cv.yaml
```

CHAPTER 3

Examples

This section covers a few use case examples. Please check the *Parameters* section to gain a better understanding for each of the variables.

3.1 Reversed-phase Prediction

While xiRT was developed for multi-dimensional RT prediction, it can also be used for a single dimension. For this, the xiRT YAML parameter file needs to be adapted as follows:

```
output:
  rp-activation: linear
  rp-column: rp
  rp-dimension: 1
  rp-loss: mse
  rp-metrics: mse
  rp-weight: 1

predictions:
  continues:
    - rp
  # simply write fractions: [] if no fraction prediction is desired
  fractions: []
```

This configuration assumes that the target column in the input data is named "rp" and that the scale is continuous (*rp-activation: linear*). If that is the case, the other parameters should not be changed (dimension, loss, metric, weight).

3.2 2D RT Prediction - Ordinal Task

Many studies apply a pre-fractionation method (e.g. SEC, SCX) and then measure the obtained fractions. For this given experimental setup, the xiRT config could look like this:

```
output:
  rp-activation: linear
  rp-column: rp
  rp-dimension: 1
  rp-loss: mse
  rp-metrics: mse
  rp-weight: 1

  scx-activation: sigmoid
  scx-column: scx_ordinal
  scx-dimension: 15
  scx-loss: binary_crossentropy
  scx-metrics: mse
  scx-weight: 50

predictions:
  continues:
    - rp
  # simply write fractions: [] if no fraction prediction is desired
  fractions: [scx]
```

In this config, 15 fractions (or pools) were measured. While RP prediction is modeled as regression problem, the SCX prediction is handled as ordinal regression. This type of regression performs classification while accounting for the magnitude of the classification errors. E.g. in a regular classification it does not matter whether an observed PSM from fraction 5, got predicted to elute in fraction 10 or in fraction 4. The error would only count as *false classification*. However, in ordinal regression the margin of error is incorporated to the loss function and thus (theoretically) ordinal regression should perform better than classification. The weight here defines how the losses from the two prediction tasks are added to derive the final loss. This parameter needs to be adapted for differences in scale and type of the output.

3.3 2D RT Prediction - Classification Task

Despite the theoretical advantage of ordinal regression, classification also delivered good results during the development of xiRT. Therefore, we kept this as an option.

For this experimental setup, the xiRT config could look like this:

```
output:
  rp-activation: linear
  rp-column: rp
  rp-dimension: 1
  rp-loss: mse
  rp-metrics: mse
  rp-weight: 1

  scx-activation: softmax
  scx-column: scx_1hot
  scx-dimension: 15
  scx-loss: categorical_crossentropy
  scx-metrics: accuracy
  scx-weight: 50

predictions:
  continues:
    - rp
```

(continues on next page)

(continued from previous page)

```
  # simply write fractions: [] if no fraction prediction is desired
  fractions: [scx]
```

Here we have the same experimental setup as above but the scx prediction task is modeled as classification. For classification, the activation function, column name and loss function must be defined as in the example.

3.4 Transfer Learning

xiRT supports multiple types of transfer-learning. For instance, training the exact same architecture (dimensions, sequence lengths) on a data set (e.g. BS3 crosslinked proteome) and then fine tune the learned weights on the actual data set (e.g. DSS crosslinked protein complex) is possible. This requires a simple change in the learning (-l parameter) config. The *pretrained_model* parameter needs to be adapted for the location of the weights file from the BS3 model.

Additionally, the the underlying model can be changed even more. This might become necessary when the training was done with e.g. 10 fractions but only 5 got acquired eventually. In this scenario, the weights cannot be used from the last layers. Therefore, the *pretrained_weights* and the *pretrained_model* parameter need to be defined in the learning (-l) config.

The files in the repository ("sample_data" and "DSS_transfer_learning_example" folder) provide examples to achieve the transfer learning. Two calls to xiRT are necessary:

1) Train the reference model without crossvalidation:

```
>xirt -i sample_data\DSS_xisearch_fdr_CSM50percent_minimal.csv \
-x sample_data\xirt_params_3RT_best_ordinal.yaml \
-l sample_data\learning_params_training_nocv.yaml \
-o models\3DRT_full_nocv
```

2) Use the model for transfer-learning:

```
>xirt -i sample_data\DSS_xisearch_fdr_CSM50percent_transfer_scx17to23_hsax2to9_
↪minimal.csv \
-x models/3DRT_full_nocv/callbacks/xirt_params_3RT_best_ordinal_scx17to23_hsax2to9.
↪yaml \
-l models/3DRT_full_nocv/callbacks/learning_params_training_nocv_scx17to23_hsax2to9.
↪yaml \
-o models\3DRT_transfer_dimensions
```

3.5 Further extensions

To further expand the tasks, two steps need to be done. First, the *predictions* section needs to be adapted such that a list of values, for example, [scx, hsax] is supplied. Further, each entry in the *predictions* section needs to have a matching set of entries in the *output* section. Carefully adjust the combination of activation, loss and column parameters as shown above. xiRT allows to have 3x regression tasks, 1x regression task + 1x classification task, etc.

In principle, the learning and prediction is agnostic to the type of input data. That means that not only RT can be learned but also other experimentally observed properties. Simply follow the notation and decoding of the training parameters to add other (non-liquid-chromatography) columns.

3.6 Note

It is important to follow the conventions above. Otherwise learning results might vary a lot.

For classification always use the following setup:

```
output:
    scx-activation: softmax
    scx-column: scx_1hot
    scx-dimension: 15
    scx-loss: categorical_crossentropy
    scx-metrics: accuracy
```

For **ordinal regression** always use the following setup:

```
output:
    scx-activation: sigmoid
    scx-column: scx_ordinal
    scx-dimension: 15
    scx-loss: binary_crossentropy
    scx-metrics: mse
```

For **regression** always use the following setup:

```
output:
    rp-activation: linear
    rp-column: rp
    rp-dimension: 1
    rp-loss: mse
    rp-metrics: mse
```

CHAPTER 4

Results

This section covers the results that are generated from a successful xiRT run. In the command line interface, the output folder needs to be specified. Typically, csv/xls files are the outputs of interest for most applications. The created folder will contain the following results:

1) log file
2) callbacks
3) quality control visualizations
4) tables (CSV/XLS)

For more details, please see the following paragraphs.

4.1 Log File

The log file contains useful information, including the xiRT version and parameters. Moreover the various steps performed during the analysis with xiRT are documented (e.g. number of duplicated entries, amino acid alphabet, maximum sequence length etc.). The logs also contain short numeric summaries from the CV training of xiRT.

```
2021-01-04 17:21:31,708 - xirt - INFO - Init logging file.
2021-01-04 17:21:31,708 - xirt - INFO - Starting Time: 17:21:31
2021-01-04 17:21:31,708 - xirt - INFO - Starting xiRT.
2021-01-04 17:21:31,708 - xirt - INFO - Using xiRT version: 1.0.63
2021-01-04 17:21:31,781 - xirt.__main__ - INFO - xi params: sample_data/xirt_params_
↪3RT.yaml
2021-01-04 17:21:31,781 - xirt.__main__ - INFO - learning_params: sample_data/
↪learning_params_training_cv.yaml
2021-01-04 17:21:31,781 - xirt.__main__ - INFO - peptides: sample_data/DSS_xisearch_
↪fdr_CSM50percent.csv
2021-01-04 17:21:31,781 - xirt.predictor - INFO - Preprocessing peptides.
2021-01-04 17:21:31,781 - xirt.predictor - INFO - Input peptides: 17886
2021-01-04 17:21:31,781 - xirt.predictor - INFO - Reordering peptide sequences.␣
↪(mode: crosslink)
```

(continues on next page)

(continued from previous page)

```
2021-01-04 17:21:43,726 - xirt.processing - INFO - Preparing peptide sequences for␣
↪columns: Peptide1,Peptide2
2021-01-04 17:21:44,296 - xirt.predictor - INFO - Duplicatad entries (by sequence␣
↪only): 5426/17886
2021-01-04 17:21:44,312 - xirt.predictor - INFO - Encode crosslinked residues.
2021-01-04 17:21:46,910 - xirt.predictor - INFO - Applying length filter: 17886␣
↪peptides left
2021-01-04 17:21:46,920 - xirt.processing - INFO - Setting max_length to: 59
2021-01-04 17:21:47,012 - xirt.processing - INFO - alphabet: ['-OH' 'A' 'D' 'E' 'F' 'G
↪' 'H' 'H-' 'I' 'K' 'L' 'M' 'N' 'O' 'P' 'Q' 'R'
 'S' 'T' 'V' 'W' 'Y' 'clA' 'clD' 'clE' 'clF' 'clG' 'clI' 'clK' 'clL' 'clM'
 'clN' 'clP' 'clQ' 'clR' 'clS' 'clT' 'clV' 'clY' 'clcmC' 'cloxM' 'cmC'
 ...
 ...
2021-01-04 17:28:38,903 - xirt.qc - INFO - Metrics: r2: 0.30 f1: 0.16 acc: 0.25 racc:␣
↪0.61
2021-01-04 17:28:39,207 - xirt.qc - INFO - QC: rp
2021-01-04 17:28:39,215 - xirt.qc - INFO - Metrics: r2: 0.69
2021-01-04 17:28:43,643 - xirt.__main__ - INFO - Writing output tables.
2021-01-04 17:29:01,207 - xirt.__main__ - INFO - Completed xiRT run.
2021-01-04 17:29:01,207 - xirt.__main__ - INFO - End Time: 17:29:01
2021-01-04 17:29:01,208 - xirt.__main__ - INFO - xiRT CV-training took: 7.20 minutes
2021-01-04 17:29:01,209 - xirt.__main__ - INFO - xiRT took: 7.49 minutes
```

4.2 Callbacks

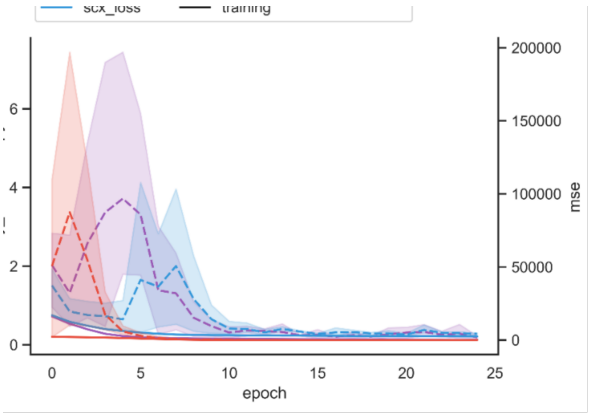
Callbacks are used throughout xiRT to select the best performing model which is not necessarily the last (epoch) model trained. To reuse already trained models for transfer-learning and predictions on other data sets, the neural network model ("xirt_model_XX.h5"), as well as the parameters/weights ("xirt_weights_XX.h5") are stored. In addition, training results per epoch are stored ("xirt_epochlog_XX.log"). "XX" refers to the cross-validation fold, e.g. 01, 02 and 03 for k=3; -1 refers the predictions for the 'unvalidated' fold (e.g. all PSMs/CSMS with FDR > e.g. 1%. The epoch log contains losses and metrics for the training and validation data. For some applications the used encoder (mapping of amino acids to integers) needs to be transferred. Therefore, the callbacks also include a trained label encoder from sklearn as pickled object ("encoder.p"). The last file additionally contains the formatted input data as pickled data. It can be used programmatically for debugging, exploration and manual retention time prediction using an already existing model. The data can be parsed in python via:

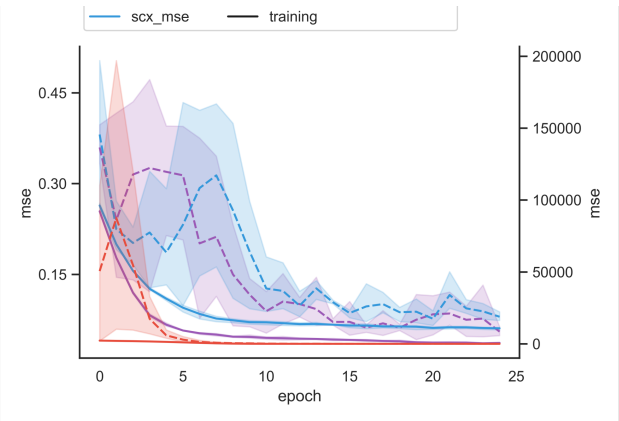
```
import pickle
X, y = pickle.load(open("Xy_data.p", "rb"))
alpha_peptides, beta_peptides = X[0], X[1]
# assuming 3 RT dimensions
RT1, RT2, RT3 = y
```

4.3 Visualizations

xiRT will create a rich set of QC plots that should always be investigated. The plots are stored in svg/pdf format.

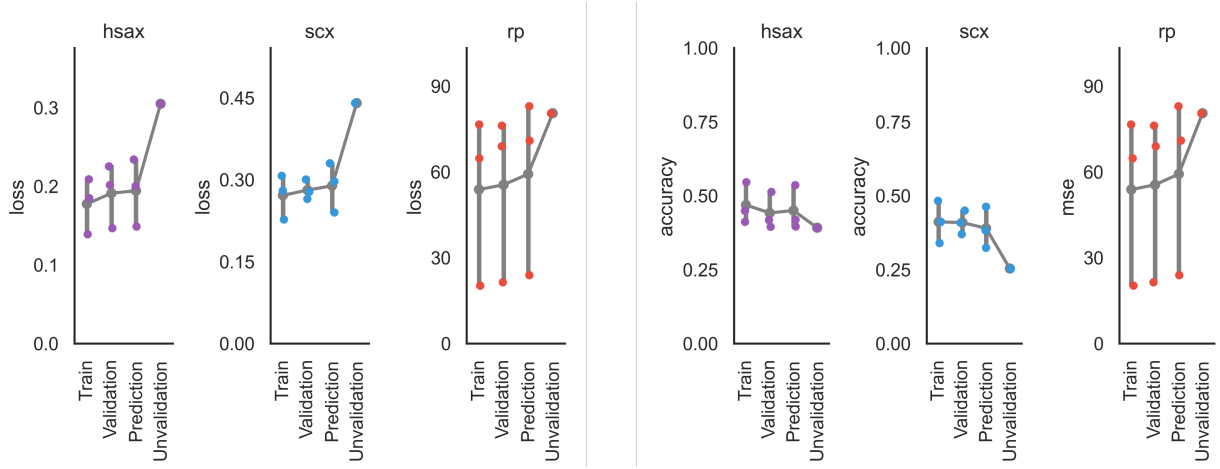
4.3.1 Epoch Loss / Metrics





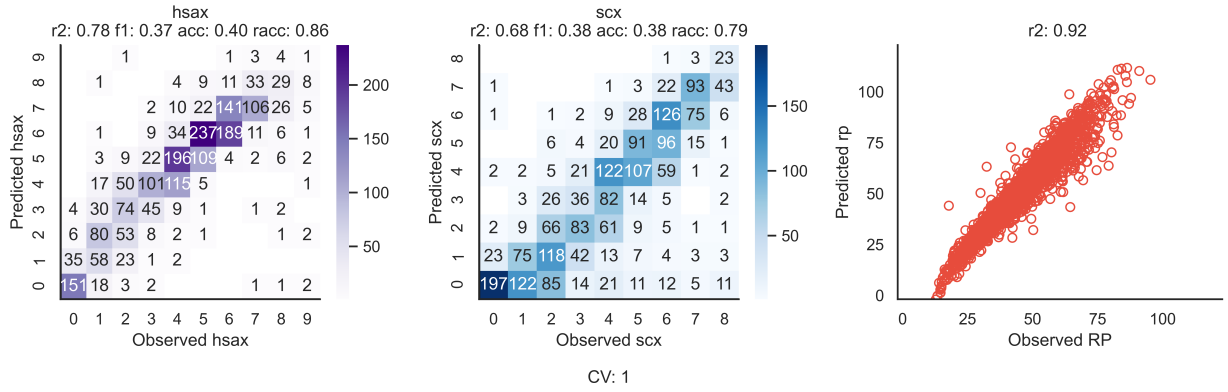
The epoch loss / metrics plot shows the training behavior over the epochs and is a good diagnostic tool to assess robustness across CV-folds, learning rate adjustment, overfit-detection and general learning behavior across tasks. In the example above, we see quick convergence and robust learning behavior after 10 epochs. In non-regression tasks, loss and metrics are not necessarily the same.

4.3.2 CV Summary



The CV summary shows the point estimates of the loss/metric for the training, validation and prediction folds for all training tasks. Unvalidation refers to the data not passing the training confidence (here: FDR) cutoff.

4.3.3 CV Observations



This plot shows the prediction performance for each CV-fold on all tasks. It also reports some key metrics that are not reported in the epoch log (r2, f1, accuracy, relaxed accuracy).

4.4 Tables

The tables contain a lot of extra information (some of which is used for the QC plots above). Please find an example of each file on (GitHub)[https://github.com/Rappsilber-Laboratory/xiRT/tree/master/sample_data/DSS_results_example].

4.4.1 Processed PSMs

This table ("processed_psms.csv") contains the input data together with internally performed processing steps. The additional columns, as exemplified on a test dataset, are:

- swapped (indicator if peptide order was swapped)
- Seq_Peptide1/Seq_Peptide2 (peptide sequences in modX format)
- Seqar_Peptide1/Seqar_Peptide2 (peptide sequences in array format
- Duplicate (indicator if combination of sequences and charge is unique within the xiRT definition)
- scx0_based (0-based fraction number)
- scx_1hot (1-hot encoded fraction variable)
- scx_ordinal (ordinal encoded fraction variable)
- fdr_mask (indicator if PSM passed the FDR for training)

4.4.2 Epoch History

This table ("epoch_history.csv") has similar data as the callbacks version but the CV results are concatenated and learning rate decay is documented.

4.4.3 Error Features

This table ("error_features.csv") contains the input PSMID, crossvalidation split annotation and the predicted retention times (including their basic error terms).

4.4.4 Error Features Interactions

This table ("error_features_interactions.csv") contains the input PSMID, and the some engineered error terms from the previous table.

4.4.5 Model Summary

This table ("model_summary.csv") contains important metrics that summarize the performance of the learned models across CV-splits and their corresponding train/validation/prediction splits.

CHAPTER 5

Parameters

xiRT needs two sets of parameters that are supplied via two YAML files. The *xiRT parameters* contain the settings that define the network architecture and learning tasks. With different / new types of chromatography or other separation settings, the learning behavior is influenced and hence needs adjustement. The *learning parameters* are used to define the learning data (e.g. filtered to a desired confidence limit) and some higher-level learning behaviour. For instance, settings for loading pretrained models and cross-validation are controlled.

5.1 xiRT-Parameters

The xiRT-Parameters can be divided into several categories that either reflect the individual layers of the network or some higher level parameters. Since the input file structure is very dynamic, the xiRT configuration needs to be handled with care. For example, the RT information in the input data is encoded in the *predictions* section. Here, the column names of the RT data needs to be defined. Accordingly, the learning options in the *output* section must be adapted. Each prediction task needs the parameters x-activation, x-column, x-dimension, x-loss, x-metrics and x-weight, where "x" represents the seperation method of interest.

Please see here for an example YAML file including comments (form xiRT v. 1.0.32):

```
LSTM:
  activation: tanh      # activiation function
  activity_regularization: l2       # regularization to use
  activityregularizer_value: 0.001  # lambda value
  bidirectional: true               # if RNN-cell should work bidirectional
  kernel_regularization: l2         # kernel regularization method
  kernelregularizer_value: 0.001    # lambda value
  lstm_bn: true                     # use batch normalization
  nlayers: 1                        # number of layers
  type: GRU                         # RNN type of layer to use: GRU, LSTM and␣
↪CuDNNGRU, CuDNNGRU
  units: 50                         # number of units in the RNN cell
dense:          # parameters for the dense layers
  activation:   # type of activiations to use for the layers (for each layer)
  - relu    # activiation function
```

(continues on next page)

(continued from previous page)

```
  - relu
  - relu
  dense_bn: # use batch normalization
  - true
  - true
  - true
  dropout:  # dropout usage rate
  - 0.1
  - 0.1
  - 0.1
  kernel_regularizer:   # regularizer for the kernel
  - l2
  - l2
  - l2
  neurons:  # number of neurons per layer
  - 300
  - 150
  - 75
  nlayers: 3    # number of layers, this number must be matched by the parameters
  regularization:   # use regularization
  - true
  - true
  - true
  regularizer_value:    # lambda values
  - 0.001
  - 0.001
  - 0.001
embedding:      # parameters for the embedding layer
  length: 50    # embedding vector dimension
learning:       # learning phase parameters
  batch_size: 128   # observations to use per batch
  epochs: 75        # maximal epochs to train
  learningrate: 0.001   # initial learning rate
  verbose: 1        # verbose training information
output:     # important learning parameters
  callback-path: data/results/callbacks/       # network architectures and weights␣
↪will be stored here
  # the following parameters need to be defined for each chromatography variable
  hsax-activation: sigmoid  # activiation function, use linear for regression
  hsax-column: hsax_ordinal # output column name
  hsax-dimension: 10    # equals number of fractions
  hsax-loss: binary_crossentropy    # loss function, must be adapted for regression /␣
↪classification
  hsax-metrics: mse     # report the following metric
  hsax-weight: 50       # weight to be used in the loss function
  rp-activation: linear
  rp-column: rp
  rp-dimension: 1
  rp-loss: mse
  rp-metrics: mse
  rp-weight: 1
  scx-activation: sigmoid
  scx-column: scx_ordinal
  scx-dimension: 9
  scx-loss: binary_crossentropy
  scx-metrics: mse
  scx-weight: 50
```

(continues on next page)

(continued from previous page)

```
siamese:        # parameters for the siamese part
  use: True          # use siamese
  merge_type: add    # how to combine individual network params after the Siamese␣
↪network
  single_predictions: True  # use also single peptide predictions
callbacks:                  # callbacks to use
  check_point: True
  log_csv: True
  early_stopping: True
  early_stopping_patience: 15
  tensor_board: False
  progressbar: True
  reduce_lr: True
  reduce_lr_factor: 0.5
  reduce_lr_patience: 15
predictions:
    # parameters that define how the input variables are treated
    # "continues" means that linear (regression) activation functions are used for␣
↪the learning.
    # if this should be done, the above parameters must also be adapted (weight, loss,
↪ metric, etc)
  continues:
    - rp
  fractions: # simply write fractions: [] if no fraction prediction is desired
    # if (discrete) fraction numbers should be used for the learning, this needs to be
    # indicated here
    # For fractions, either ordinal regression or classification can be used in the
    # fractions setting (regression is possible too).
    - scx
    - hsax
```

Apart from the very important neural network architecture definitions, the target variable encoding is also defined in the YAML.

5.2 Learning-Parameters

Parameters that govern the separation of training and testing data for the learning.

Here is an example YAML file with comments (form xiRT v. 1.0.32):

```
# preprocessing options:
# le: str, label encoder location. Only needed for transfer learning, or usage of␣
↪pretrained
# max_length: float, max length of sequences
# cl_residue: bool, if True crosslinked residues are decoded as Kcl or in modX format␣
↪clK
preprocessing:
    le: None
    max_length: -1 # -1
    cl_residue: True


# fdr: float, a FDR cutoff for peptide matches to be included in the training process
# ncv: int, number of CV folds to perform to avoid training/prediction on the same␣
↪data
```

(continues on next page)

(continued from previous page)

```
# mode: str, must be one of: train, crossvalidation, predict
# train and transfer share the same options that are necessary to run xiML, here is a␣
↪brief rundown:
# augment: bool, if data augmentation should be performed
# sequence_type: str, must be linear, crosslink, pseudolinear. crosslink uses the␣
↪siamese network
# pretrained_weights: "None", str location of neural network weights. Only embedding/
↪RNN weights
#   are loaded. pretrained weights can be used with all modes, essentially resembling␣
↪a transfer
#   learning set-up
# sample_frac: float, (0, 1) used for downsampling the input data (e.g. for learning␣
↪curves).
#   Usually, left to 1 if all data should be used for training
# sample_state: int, random state to be used for shuffling the data. Important for␣
↪recreating
#   results.
# refit: bool, if True the classifier is refit on all the data below the FDR cutoff␣
↪to predict
# the RT times for all peptide matches above the FDR cutoff. If false, the already␣
↪trained CV
# classifier with the lowest validation loss is chosen
train:
  fdr: 0.01
  ncv: 3
  mode: "crossvalidation" # other modes are: train / crossvalidation / predict
  augment: False
  sequence_type: "crosslink"
  pretrained_weights: "None"
  test_frac: 0.10
  sample_frac: 1
  sample_state: 21
  refit: False
```

Generally, it is better to supply more high-quality data than more data. Sometimes considerable drops in performance can be observed when 5% instead of 1% input data is used. However, there is no general rule of thumb and this needs to be optimized per run / experiment.

5.3 Hyperparameter-Optimization

Neural Networks are very sensitive to their hyperparameters. To automate the daunting task of finding the right hyperparameters two utils are shipped with xiRT. 1) a convenience function that generates YAML files from a *grid YAML* file. 2) a snakemake workflow that can be used to run xiRT with each parameter combination.

The grid will be generated based on all entries where not a single value is passed but a list of values. This can lead to an enormous search space, so step-wise optimization is sometimes the only viable option.

CHAPTER 6

Frequently Asked Questions

6.1 1. What is xiRT?

xiRT is a python package for multi-dimensional RT prediction for linear and cross-linked peptides.

6.2 2. How does xiRT work?

xiRT is a deep learning application and uses a Siamese network to encode crosslinked peptides. xiRT can predict continuous and discrete retention times (e.g. from reversed phase or fractionation experiments).

6.3 3. What are the requirements for xiRT?

xiRT requires a running python installation, please follow the installation guide to get xiRT running. To visualize the neural network pydot and graphviz are also needed.

6.4 4. Do I need a GPU?

A GPU is not necessary to use xiRT. It speeds things up but xiRT can run on any desktop computer. Make sure to specify the correct layer in the xirt_params file (e.g. GRU instead of CudNNGRU).

6.5 5. What's the run time of xiRT?

Depends heavily on the settings (e.g. cross-validation folds, epochs, number input PSMs). For the example data (3-fold crossvalidation, 17k PSMs, 25 epochs) the analysis finishes within 10 minutes on a desktop pc.

6.6 6. Where can I get help using xiRT?

Please create an GitHub issue if we can assist you with your analysis or if anything is unclear.

6.7 7. Which chromatography types are supported?

xiRT is agnostic to the type of chromatography and supports to learn 1, 2,3 . . . , n chromatography dimensions at the same time. Continuous (e.g. reversed phase) and discrete (fractionation) retention time measurements are supported.

6.8 8. xirt_params.yaml - File not found error

When using xirt over the command line, make sure to allways use the relative or absolute path to the input files.

6.9 9. AttributeError: module 'sip' has no attribute 'setapi'

The current matplotlib version (3.3) seems to have a bug. Please install matplitlib 3.2 (pip install matplotlib==3.2).

CHAPTER 7

xiRT-Modules

7.1 xirt package

7.1.1 Submodules

7.1.2 xirt.features module

7.1.3 xirt.predictor module

7.1.4 xirt.processing module

7.1.5 xirt.sequences module

7.1.6 xirt.xirtnet module

7.1.7 Module contents

CHAPTER 8

xiRT - development

This section covers some guidelines for the development of xiRT. Especially, how to generate a new release.

8.1 Preparing a new release

The preparation include to run all tests locally and make sure that they pass. In addition, the test coverage badge needs to be generated by calling:

```
` >coverage-badge -o documentation/imgs/coverage.svg -f `
```

If everything passes, the documentation needs to generated. Make sure to include new pages in the index page and build the documentation by calling:

```
` >sphinx-build -b html documentation/source documentation/build `
```

8.2 Publishing a new release

To release a new version two steps are necessary: 1) create the PyPi release 2) create the github release

For 1) execute the following command and have the PyPi and github credentials and hand: ` >python setup.py upload `

Make sure to bump the version number with the last commit before executing the setup file. This will automatically generate a tag in github along the PyPi package.

2) Can be done via the github project page. Simply navigate to the "create release" page and upload the binaries / wheels that have been generated with the setup.py file. Make sure tom summarize the changes to the last version.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search