D3.2 Core AI Models - Final Version

Lead Partner:	IBM
Version:	1
Dissemination Level:	Public
Work Package:	WP3
Due date:	31/03/2022
Submission Date:	31/03/2022

Abstract:
This document describes the final version of Streams and Incremental learning Library (SAIL). SAIL is an open-source library designed to quickly prototype and experiment with incremental machine learning algorithms (IMLA). This report describes the improvements and features added to SAIL since the first version.

This report documents the core elements of SAIL in the final version and its positioning in the open-source community with the additional features that have been added for the final version. The report covers:

(1) Architecture diagram of SAIL library (2) Base classes and Mixin classes and integration (3) APIs for deep learning models (4) Examples (5) Future work.

Document Revision History

Date	Version	Author/Editor/Contributor	Summary of main changes / Status
20/03/2022	V1	Seshu Tirupathi, Dhaval Salwala, Shivani Tomar IBM	Initial version
28/03/2022	V2	Seshu Tirupathi, Dhaval Salwala, Shivani Tomar IBM	Added architecture diagram, library structure and CI/CD
28/03/2022	V3	George Papastefanatos	Internally reviewed version
29/03/2022	V4	Seshu Tirupathi, Dhaval Salwala, Shivani Tomar IBM	Final version

Disclaimer

Copyright

Table of Contents

1 Introduction

WP3 is responsible for developing IMLA models, ensemble and scalable AutoML pipelines. D3.2 is responsible for the final version of the Streams and Incremental Learning (SAIL) library. D3.2 is a continuation of the effort mentioned in D3.1 which details the initial version the SAIL library. Before detailing the results of WP3.2, the terminology section from D3.1 is reproduced here for consistency and ease of reference for the rest of the document.

1.1 Terminology

Incremental machine learning algorithms are a class of machine learning algorithms where models are updated based on the arrival of new data. Incremental learning is further divided into subclasses that can potentially have conflicting definitions in the literature. Therefore, the subclasses are defined here for consistency in this document:

Incremental machine learning (IML): Algorithms where a machine learning model Mt is updated based on model Mt-1 and the new incoming data without complete retraining.

Continual learning/Lifelong learning (CL): (Deep) Learning a model sequentially without forgetting knowledge obtained. Literature is primarily confined to classification tasks (more particularly in images).

Online learning (OL): Incremental machine learning where model is updated one new sample at a time.

Offline training(OffT): Machine learning models trained on data that is available offline. These models can be batch or incremental trained. Based on the definitions above, continual learning and online learning are subsets of IML.

1.2 Initial version (D3.1)

After extensive literature review and open-source libraries in the space of incremental learning, the initial version of SAIL was released in October 2021 and covered the following features:

- Native models for classification like Incremental Exterme Learning Machine (IELM) (Huang 2007)
- Scikit-learn (Pedregosa 2011) wrapper functionality to online models available through open-source libraries – River (J. M. Montiel 2021) and Scikit-multiflow (J. J. Montiel 2018).
- Ensembling of batch and incremental models.
- Support for AutoML algorithms through model selection.
- Distributed model selection using Ray (Moritz 2018).
- PyTorch (Paszke 2017) models for regression and wrapped through Skorch (Tietz, et al. 2017) library for Scikit-learn consistency.

While the algorithms were developed, the models were only loosely coupled in terms of a common interface. This aspect and additional requirements to make the library more robust, the following aspects were kept in mind to improve the library:

- Mixin classes for incremental machine learning tasks like regression and classification to ensure consistency and reduce chances of errors.
- Richer support for deep learning models and state-of-the-art deep learning models with Scikit-learn interface.
- Consistent Scikit-learn paradigm for incremental models with *partial_fit and predict* functions for online learning and deep learning models to ensure successful implementation of AutoML algorithms.
- Necessity of Scikit-learn API for easy integration with Ray and Ray Tune libraries which provide state-of-the-art distributed HPO functionality.

1.3 Progress from initial version

With the requirements for improving the initial version of SAIL, the following features have been added to SAIL and are explained in detail with examples in the next sections.

- Extended support for standard incremental deep learning networks in Pytorch and wrapped with Skorch library for Scikit-learn interface.
- Added support for incremental TensorFlow deep learning models and wrapped with Scikeras library for Scikit-learn interface.
- Added state of the art deep learning models that handles streaming data and aspects like noise in streaming data better than standard deep learning models.
- More robust setup through abstract base classes and mixin classes for regression and classification incremental learning algorithms.
- Continuous integration and continuous deployment (CI/CD).

1.4 Open-source code

SAIL is an open-source library under MIT license and hosted at https://github.com/IBM/sail and also forked at https://github.com/MORE-EU/sail.

2 Architecture diagram

2.1 Interaction with MORE

SAIL forms the core component of all the three machine learning touchpoints in the MORE platform (Figure 1, as taken from D5.2) i.e. edge analytics engine, IMLA and scalable forecasting component and the AutoML on IMLA. Models developed in SAIL can work on raw as well as decompressed data from the data models of ModelarDB. APIs have been developed for interaction with ModelarDB and to store and retrieve incremental models. Architecture and some of the APIs that have been developed for interacting with the other components of MORE are described in Section 3 of D3.7.

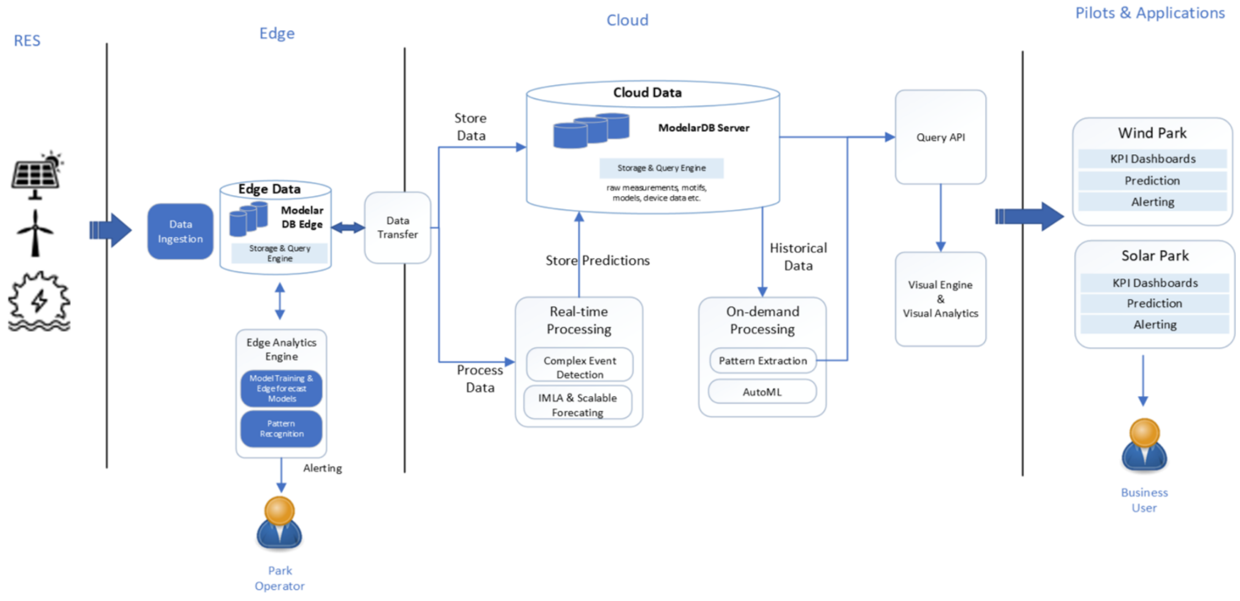


Figure 1 MORE application overview and dataflow

2.2 SAIL components

SAIL library consists of a varied number of incremental machine learning algorithms that are exposed through a common interface to the end user. The various components of SAIL library (Figure 2) are described below:

Original models consist of incremental algorithms that are developed in River, Scikit-Multiflow, Scikit-learn, Keras, TensorFlow, PyTorch or natively using elementary Python libraries like NumPy. An end user who wishes to implement an algorithm in SAIL would start by developing a model using one of the libraries mentioned above.

Model wrappers: Model wrappers are used to expose the original models to NumPy arrays. While Scikit-learn models and Scikit-Multiflow models would not require any wrappers, models in river, PyTorch, Keras and TensorFlow would require model wrappers since the data exposed to these libraries vary across libraries. The open-source wrappers Scikit-learn, Skorch, Scikeras are leveraged for River, TensorFlow and PyTorch models respectively.

SAIL wrapper: Finally SAIL wrapper is exposed to the end-user with the incremental learning and storage functionalities defined through abstract base classes and Mixin classes.

In the future, novel AutoML algorithms will be developed in Ray for scalability.

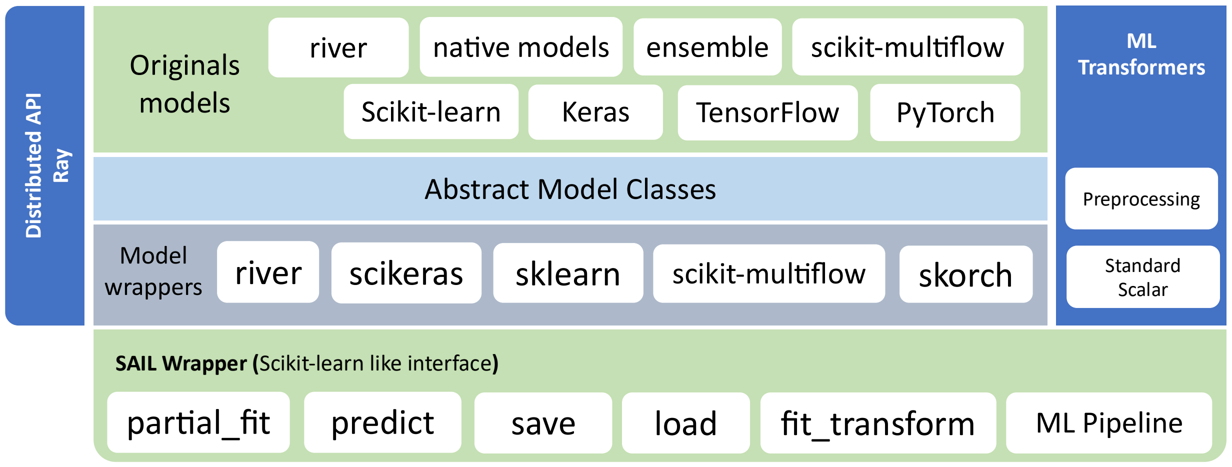
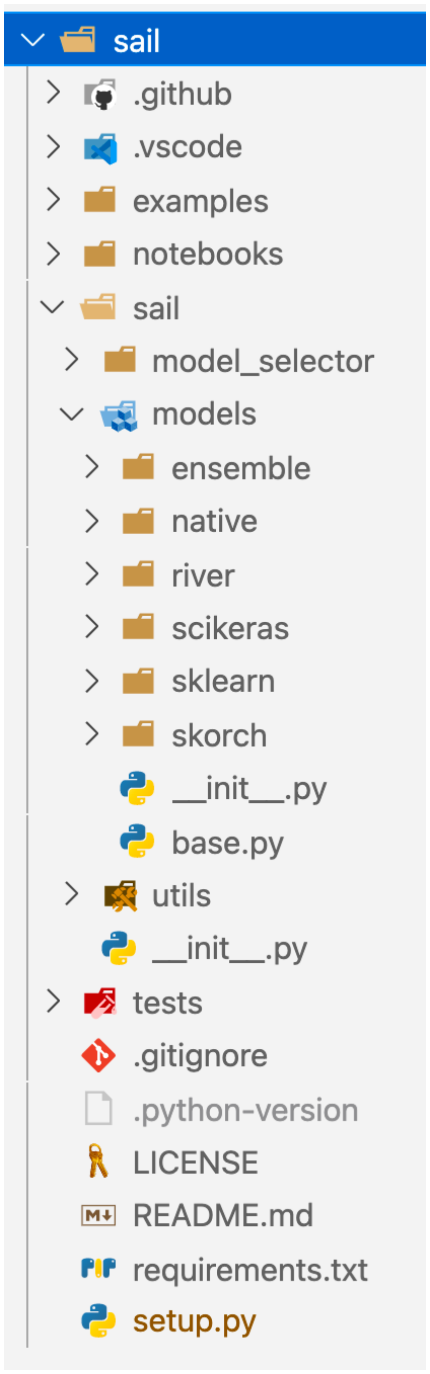


Figure 2 Architecture diagram for SAIL.

2.3 Library Structure

Library structure of SAIL is shown in Figure 3. All models are stored in the *models* directory. The directories *examples* and *notebooks* provide examples on how to use the library. Directory names *tests* is used for unit testing the models in SAIL. Finally, *requirements.txt* and *setup.py* ensures easy installation for development as well as end use.



```
sail
  .github
  .vscode
  examples
  notebooks
  sail
    model_selector
    models
      ensemble
      native
      river
      scikeras
      sklearn
      skorch
      __init__.py
      base.py
    utils
    __init__.py
  tests
  .gitignore
  .python-version
  LICENSE
  README.md
  requirements.txt
  setup.py
```

Figure 3 SAIL directory structure

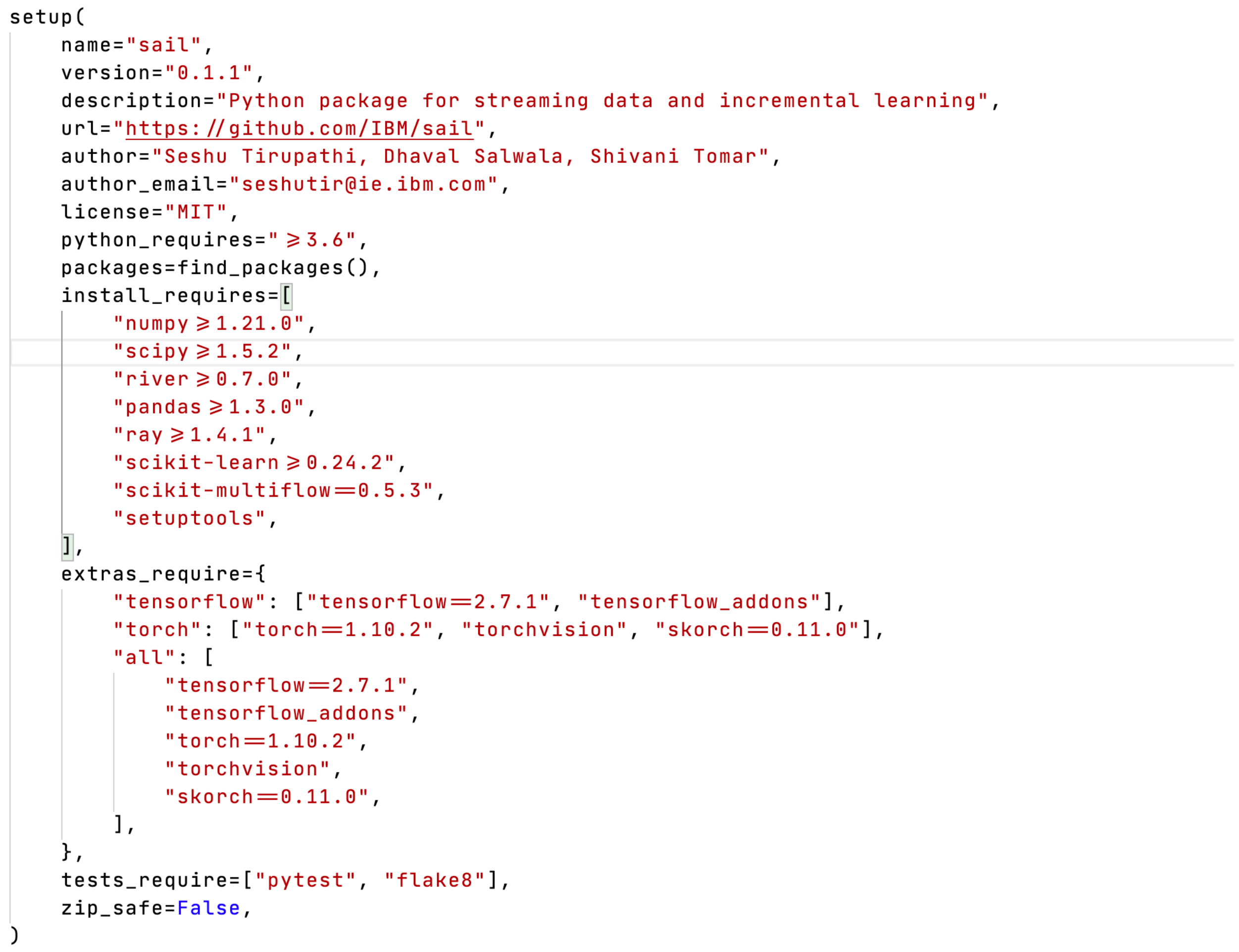
3 User guide

3.1 Installation

The installation instructions remain the same as the initial version i.e., SAIL can be installed through pip. These are the instructions to install SAIL as a package:

1. Clone the repo to the local machine
2. Change directory to the root directory of SAIL
3. Activate conda environment or venv
4. Run the command : *$ pip install .*
5. The above command will install SAIL along with its dependencies.

The setup.py is as below.



```
setup(
    name="sail",
    version="0.1.1",
    description="Python package for streaming data and incremental learning",
    url="https://github.com/IBM/sail",
    author="Seshu Tirupathi, Dhaval Salwala, Shivani Tomar",
    author_email="seshutir@ie.ibm.com",
    license="MIT",
    python_requires=">=3.6",
    packages=find_packages(),
    install_requires=[
        "numpy>=1.21.0",
        "scipy>=1.5.2",
        "river>=0.7.0",
        "pandas>=1.3.0",
        "ray>=1.4.1",
        "scikit-learn>=0.24.2",
        "scikit-multiflow==0.5.3",
        "setuptools",
    ],
    extras_require={
        "tensorflow": ["tensorflow==2.7.1", "tensorflow_addons"],
        "torch": ["torch==1.10.2", "torchvision", "skorch==0.11.0"],
        "all": [
            "tensorflow==2.7.1",
            "tensorflow_addons",
            "torch==1.10.2",
            "torchvision",
            "skorch==0.11.0",
        ],
    },
    tests_require=["pytest", "flake8"],
    zip_safe=False,
)
```

Figure 4 Setup file for SAIL

For development setup, the following command can be run:
$ pip install -r requirements.txt

Optional dependencies can be installed along with SAIL using the following command:
$ pip install .[package_name]

3.2 Testing

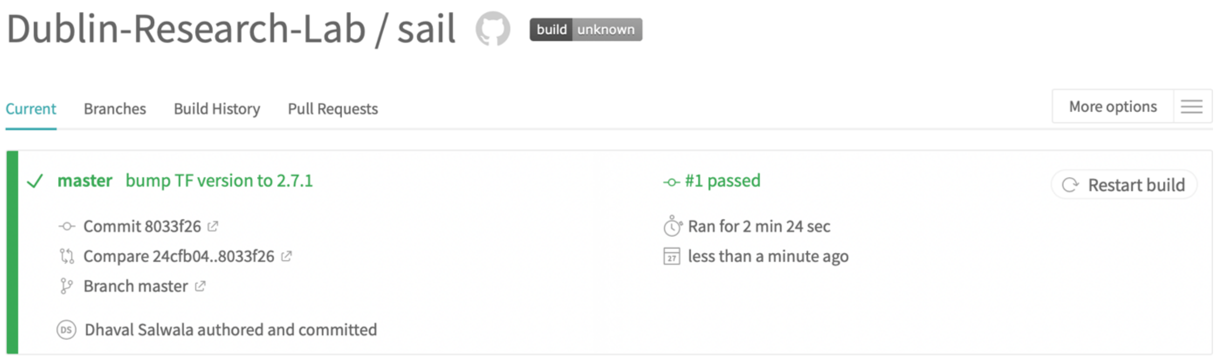
Testing follows the same principles as the initial version of SAIL i.e., unit testing is done through the *unittest* framework in Python and standard open-source datasets, or standard simulated datasets (ex. Seagenerator cite) are used in the examples to benchmark the algorithms.

3.3 Documentation

Documentation in SAIL was generated using *docstring* convention and *Sphinx* with *Google-style* or *reStructuredText* format used for creating directives and provide details about the inputs and outputs of a function. There are no changes to the approach in documentation as compared to the initial version of SAIL.

3.4 CI/CD

CI/CD in SAIL is maintained through Travis build. The latest build of SAIL on the master branch is a success. The Travis build snapshot is below.



✓ master bump TF version to 2.7.1
Commit 8033f26
Compare 24cfb04..8033f26
Branch master
Dhaval Salwala authored and committed
#1 passed
Ran for 2 min 24 sec
less than a minute ago

Figure 5 Travis Build - SAIL

Travis build setup the SAIL environment on the bionic version of Ubuntu. It builds from the master branch and uses Python3.7.

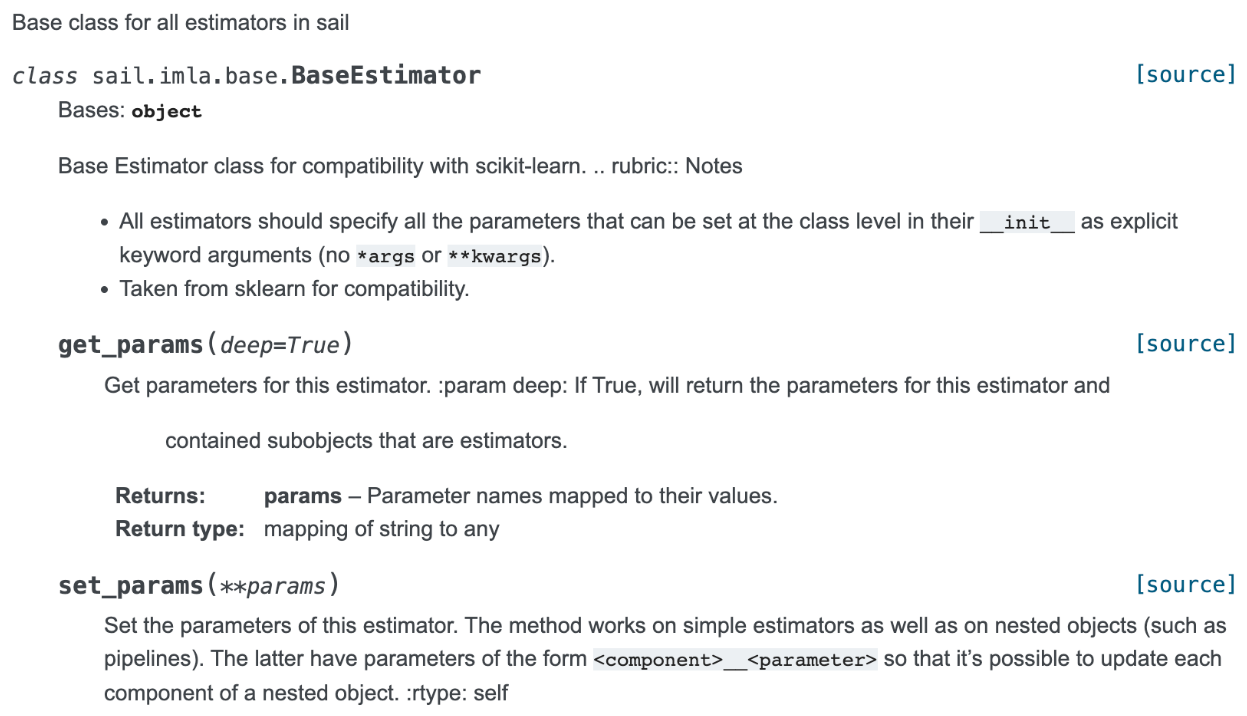
Currently, Travis build take care of the two things,

1. Setup environment as per setup.py file
2. Run unit tests.

4 API description

4.1 Base class for incremental models

Incremental models in SAIL are built on top of base classes consistent with Scikit-learn. In practical terms, the base estimator (Figure 6) follows the exact functions as defined in the Scikit-learn base estimator. (Varoquaux 2021). Further, for incremental learning (*partial_fit)* and prediction (*predict*) functionalities, SAIL follows the mixin classes (Figure 7, Figure 8) completely as defined in Scikit-Multiflow (J. J. Montiel 2018). All incremental models, independent of the library used for the base estimator, inherit from either *BaseEstimator, RegressorMixin or ClassifierMixin* classes. In addition, a similar base class *SAILWrapper* exists for Keras models.



Base class for all estimators in sail

class `sail.imla.base.`**`BaseEstimator`** [source]

Bases: **`object`**

Base Estimator class for compatibility with scikit-learn. .. rubric:: Notes

- All estimators should specify all the parameters that can be set at the class level in their `__init__` as explicit keyword arguments (no `*args` or `**kwargs`).
- Taken from sklearn for compatibility.

`get_params``(`*`deep=True`*`)` [source]

Get parameters for this estimator. :param deep: If True, will return the parameters for this estimator and

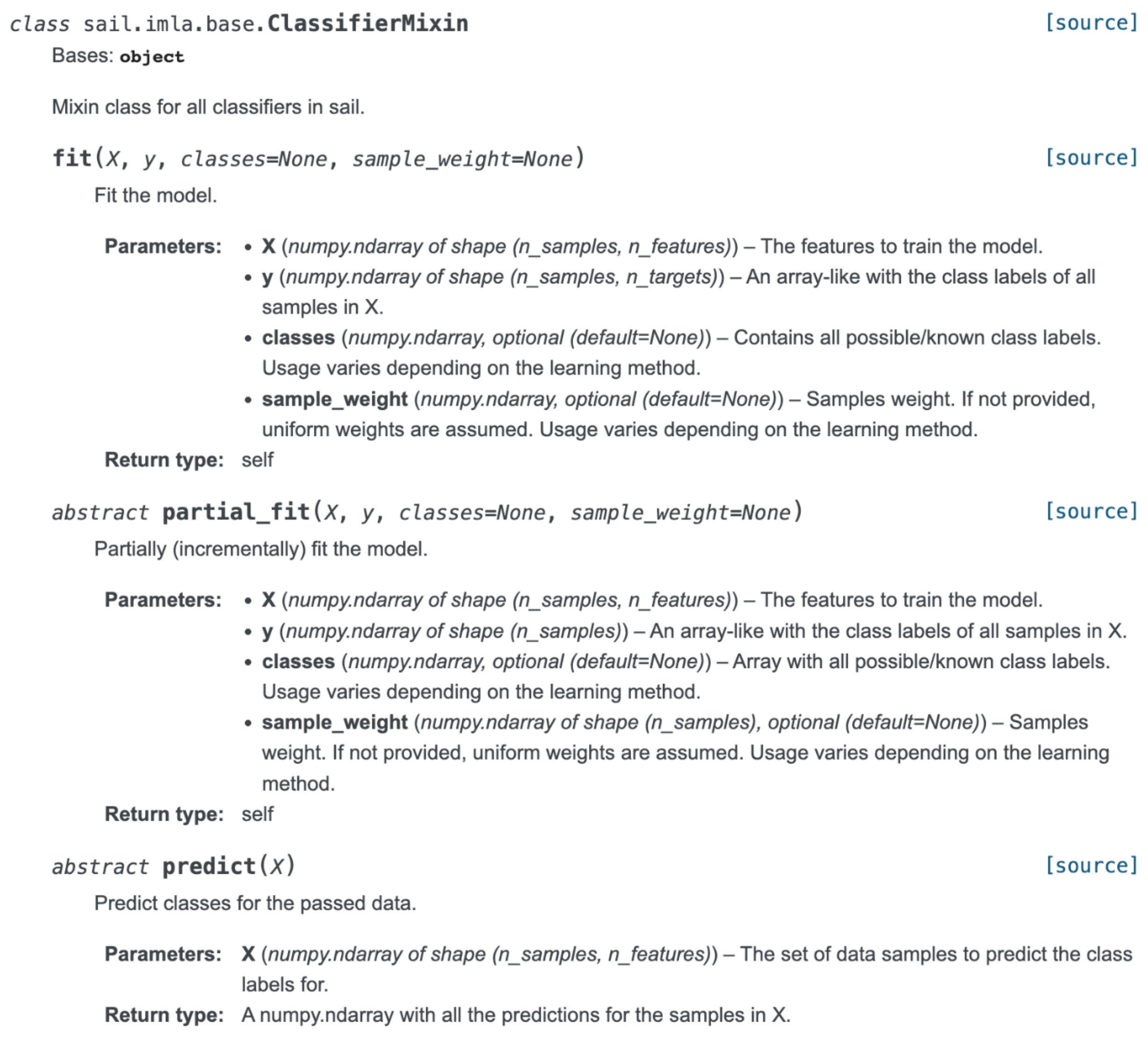
contained subobjects that are estimators.

Returns:	**params** – Parameter names mapped to their values.
Return type:	mapping of string to any

`set_params``(`*`**params`*`)` [source]

Set the parameters of this estimator. The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object. :rtype: self

Figure 6 Base estimator for incremental models in SAIL.



class sail.imla.base.**ClassifierMixin** [source]

Bases: **object**

Mixin class for all classifiers in sail.

fit(*X*, *y*, *classes=None*, *sample_weight=None*) [source]

Fit the model.

Parameters:
- **X** (*numpy.ndarray of shape (n_samples, n_features)*) – The features to train the model.
- **y** (*numpy.ndarray of shape (n_samples, n_targets)*) – An array-like with the class labels of all samples in X.
- **classes** (*numpy.ndarray, optional (default=None)*) – Contains all possible/known class labels. Usage varies depending on the learning method.
- **sample_weight** (*numpy.ndarray, optional (default=None)*) – Samples weight. If not provided, uniform weights are assumed. Usage varies depending on the learning method.

Return type: self

abstract **partial_fit**(*X*, *y*, *classes=None*, *sample_weight=None*) [source]

Partially (incrementally) fit the model.

Parameters:
- **X** (*numpy.ndarray of shape (n_samples, n_features)*) – The features to train the model.
- **y** (*numpy.ndarray of shape (n_samples)*) – An array-like with the class labels of all samples in X.
- **classes** (*numpy.ndarray, optional (default=None)*) – Array with all possible/known class labels. Usage varies depending on the learning method.
- **sample_weight** (*numpy.ndarray of shape (n_samples), optional (default=None)*) – Samples weight. If not provided, uniform weights are assumed. Usage varies depending on the learning method.

Return type: self

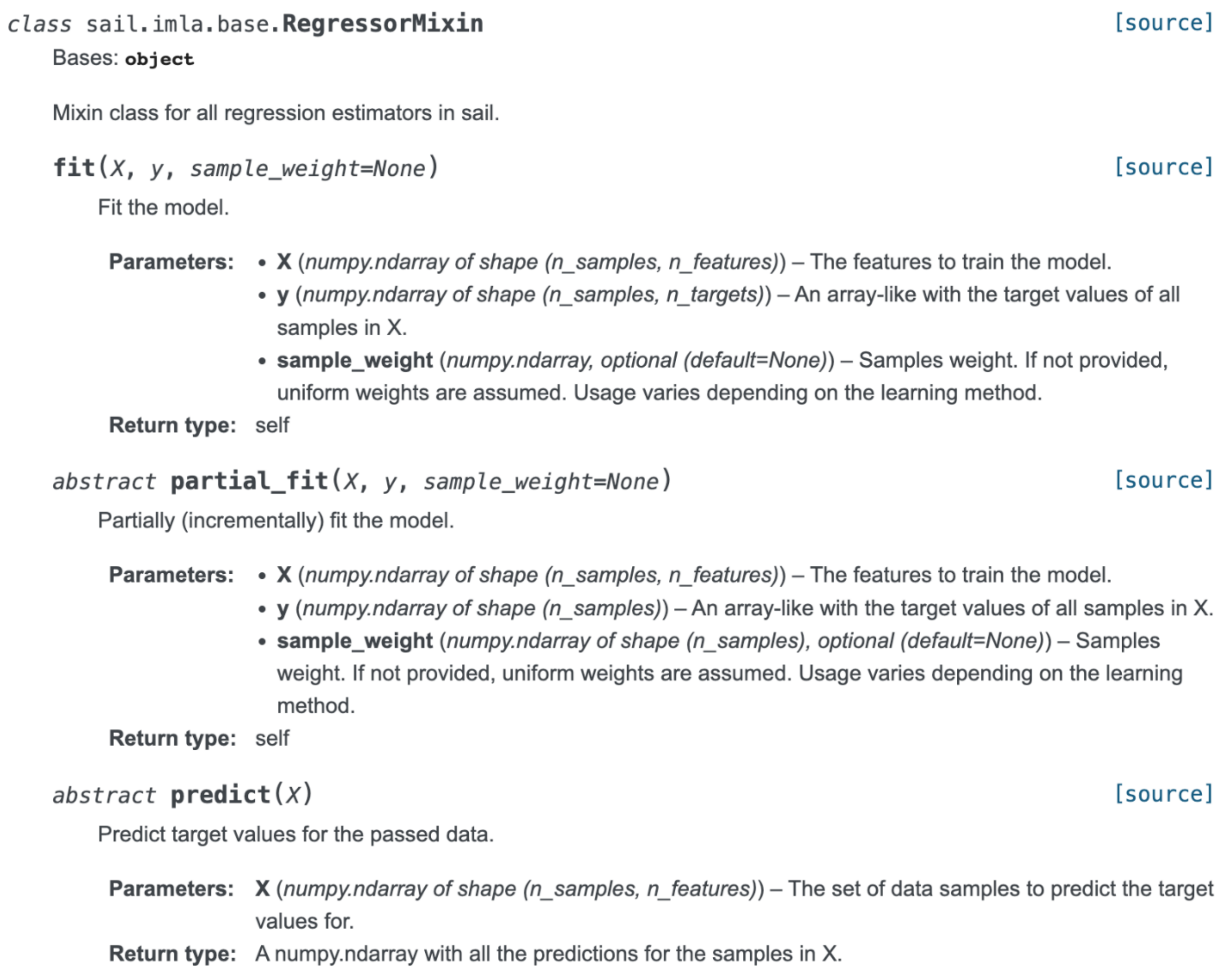
abstract **predict**(*X*) [source]

Predict classes for the passed data.

Parameters: **X** (*numpy.ndarray of shape (n_samples, n_features)*) – The set of data samples to predict the class labels for.

Return type: A numpy.ndarray with all the predictions for the samples in X.

Figure 7 Classifier mixin class for SAIL.



class sail.imla.base.**RegressorMixin** [source]

Bases: **object**

Mixin class for all regression estimators in sail.

fit(*X, y, sample_weight=None*) [source]

Fit the model.

Parameters:
- **X** (*numpy.ndarray of shape (n_samples, n_features)*) – The features to train the model.
- **y** (*numpy.ndarray of shape (n_samples, n_targets)*) – An array-like with the target values of all samples in X.
- **sample_weight** (*numpy.ndarray, optional (default=None)*) – Samples weight. If not provided, uniform weights are assumed. Usage varies depending on the learning method.

Return type: self

abstract **partial_fit**(*X, y, sample_weight=None*) [source]

Partially (incrementally) fit the model.

Parameters:
- **X** (*numpy.ndarray of shape (n_samples, n_features)*) – The features to train the model.
- **y** (*numpy.ndarray of shape (n_samples)*) – An array-like with the target values of all samples in X.
- **sample_weight** (*numpy.ndarray of shape (n_samples), optional (default=None)*) – Samples weight. If not provided, uniform weights are assumed. Usage varies depending on the learning method.

Return type: self

abstract **predict**(*X*) [source]

Predict target values for the passed data.

Parameters: **X** (*numpy.ndarray of shape (n_samples, n_features)*) – The set of data samples to predict the target values for.

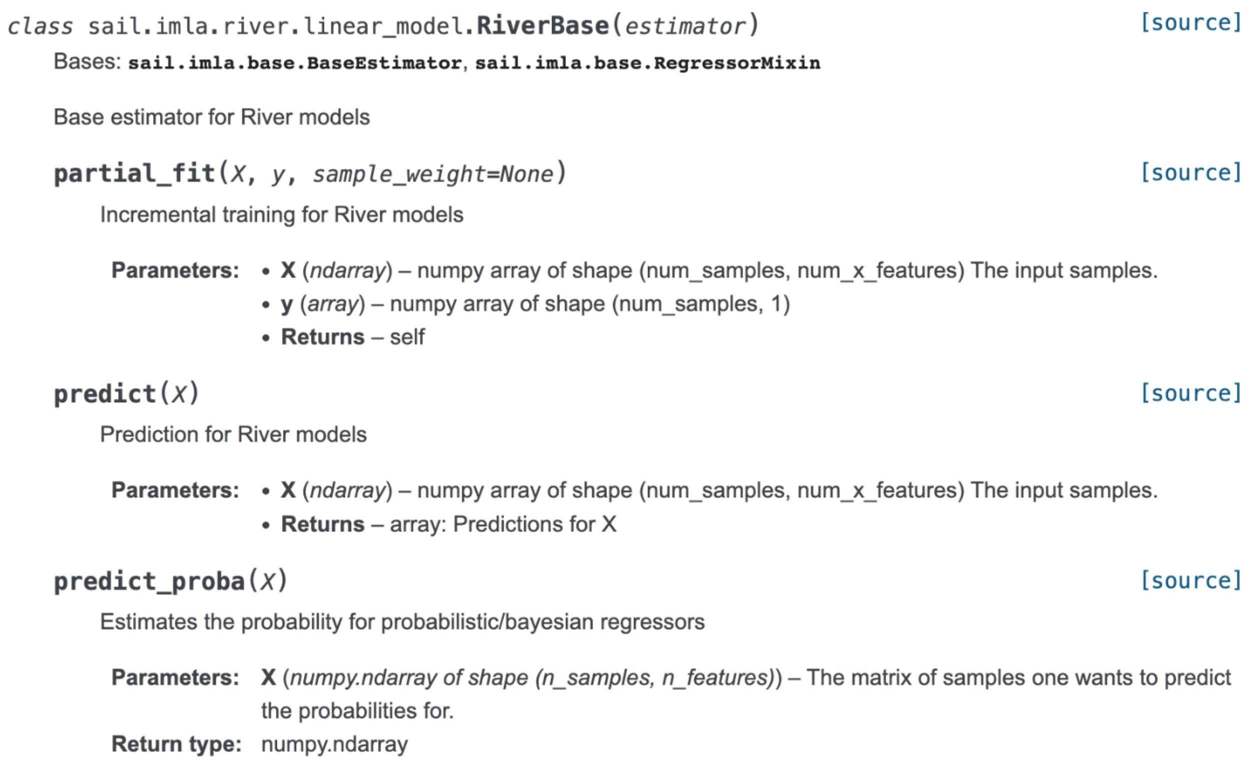
Return type: A numpy.ndarray with all the predictions for the samples in X.

Figure 8 Regressor mixin class for SAIL.

4.2 River, Scikit-learn and corresponding wrappers

Since SAIL expects the input and output to be compatible with Scikit-learn interface, no wrappers or additional code is required to handle incremental models in Scikit-learn.

River library provides called *convert_river_to_sklearn* to convert online models which takes dictionary of features as an input to Scikit-learn input format. This function is leveraged in the *RiverBase* class (Figure 9) developed in MORE. All the online models in River are built on top of the *RiverBase* estimator. An example of the SAIL wrapper for the linear regression model in River is shown in Figure 10.



```
class sail.imla.river.linear_model.RiverBase(estimator)    [source]
```

Bases: **sail.imla.base.BaseEstimator**, **sail.imla.base.RegressorMixin**

Base estimator for River models

```
partial_fit(X, y, sample_weight=None)    [source]
```

Incremental training for River models

Parameters:
- **X** (*ndarray*) – numpy array of shape (num_samples, num_x_features) The input samples.
- **y** (*array*) – numpy array of shape (num_samples, 1)
- **Returns** – self

```
predict(X)    [source]
```

Prediction for River models

Parameters:
- **X** (*ndarray*) – numpy array of shape (num_samples, num_x_features) The input samples.
- **Returns** – array: Predictions for X

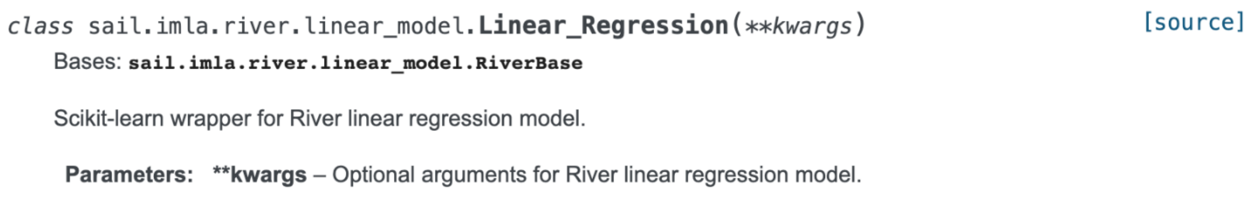
```
predict_proba(X)    [source]
```

Estimates the probability for probabilistic/bayesian regressors

Parameters: **X** (*numpy.ndarray of shape (n_samples, n_features)*) – The matrix of samples one wants to predict the probabilities for.

Return type: numpy.ndarray

Figure 9 Base estimator for River models.



```
class sail.imla.river.linear_model.Linear_Regression(**kwargs)    [source]
```

Bases: **sail.imla.river.linear_model.RiverBase**

Scikit-learn wrapper for River linear regression model.

Parameters: ****kwargs** – Optional arguments for River linear regression model.

Figure 10 Scikit-learn wrapper for linear regression model in River.

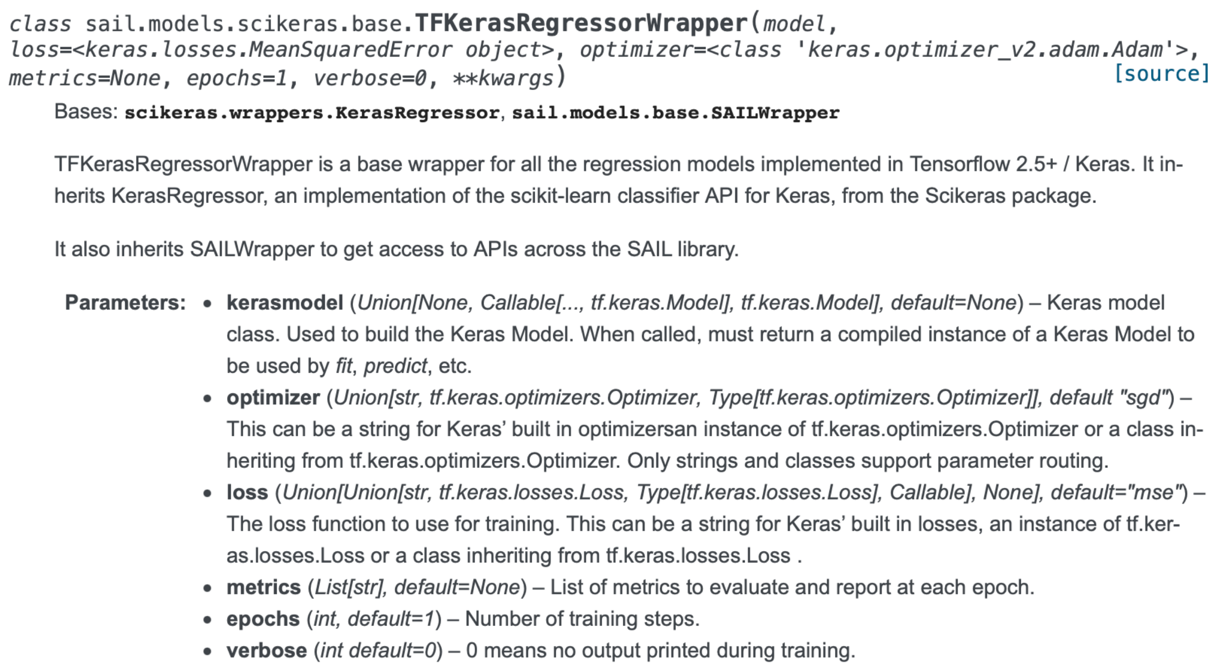
4.3 Tensorflow and Scikeras wrapper

Packages used in the implementation of these wrapper classes:

- scikeras==0.6.1
- TensorFlow >= 2.5.

4.3.1 Regression Wrapper

The internal wrapper (*TFKerasRegressionWrapper*) to use with regression models implemented in TensorFlow 2.5+ / Keras. It inherits all the APIs of KerasRegressor from the Scikeras package. Through the implementation of *SAILWrapper*, it has access to the functions and methods common across the SAIL library.



```
class sail.models.scikeras.base.TFKerasRegressorWrapper(model,
loss=<keras.losses.MeanSquaredError object>, optimizer=<class 'keras.optimizer_v2.adam.Adam'>,
metrics=None, epochs=1, verbose=0, **kwargs)                                      [source]
```

Bases: **scikeras.wrappers.KerasRegressor**, **sail.models.base.SAILWrapper**

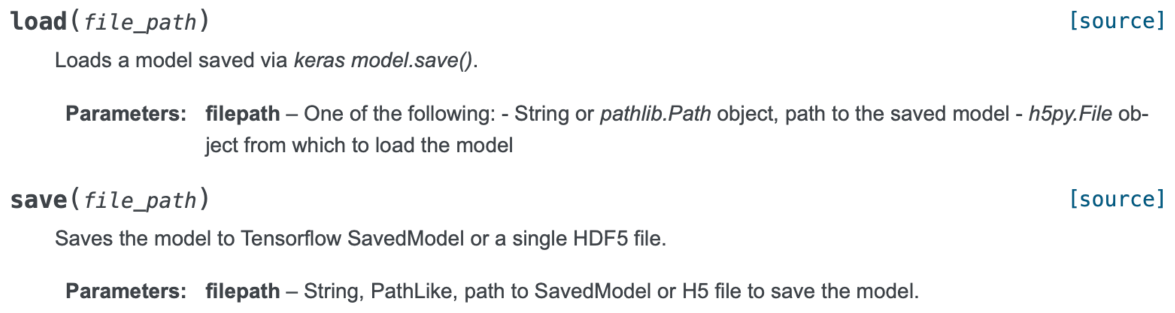
TFKerasRegressorWrapper is a base wrapper for all the regression models implemented in Tensorflow 2.5+ / Keras. It inherits KerasRegressor, an implementation of the scikit-learn classifier API for Keras, from the Scikeras package.

It also inherits SAILWrapper to get access to APIs across the SAIL library.

Parameters:

- **kerasmodel** (*Union[None, Callable[..., tf.keras.Model], tf.keras.Model], default=None*) – Keras model class. Used to build the Keras Model. When called, must return a compiled instance of a Keras Model to be used by *fit*, *predict*, etc.
- **optimizer** (*Union[str, tf.keras.optimizers.Optimizer, Type[tf.keras.optimizers.Optimizer]], default "sgd"*) – This can be a string for Keras' built in optimizersan instance of tf.keras.optimizers.Optimizer or a class inheriting from tf.keras.optimizers.Optimizer. Only strings and classes support parameter routing.
- **loss** (*Union[Union[str, tf.keras.losses.Loss, Type[tf.keras.losses.Loss], Callable], None], default="mse"*) – The loss function to use for training. This can be a string for Keras' built in losses, an instance of tf.keras.losses.Loss or a class inheriting from tf.keras.losses.Loss .
- **metrics** (*List[str], default=None*) – List of metrics to evaluate and report at each epoch.
- **epochs** (*int, default=1*) – Number of training steps.
- **verbose** (*int default=0*) – 0 means no output printed during training.

The *TFKerasRegressionWrapper* takes model class, optimizer, loss function, metrics list, no of epochs, and the verbose flag. The model class is built with arguments before it is passed to the scikeras wrapper.



load(*file_path*) [source]

Loads a model saved via *keras model.save()*.

Parameters: **filepath** – One of the following: - String or *pathlib.Path* object, path to the saved model - *h5py.File* object from which to load the model

save(*file_path*) [source]

Saves the model to Tensorflow SavedModel or a single HDF5 file.

Parameters: **filepath** – String, PathLike, path to SavedModel or H5 file to save the model.

Save() is used to save model configurations

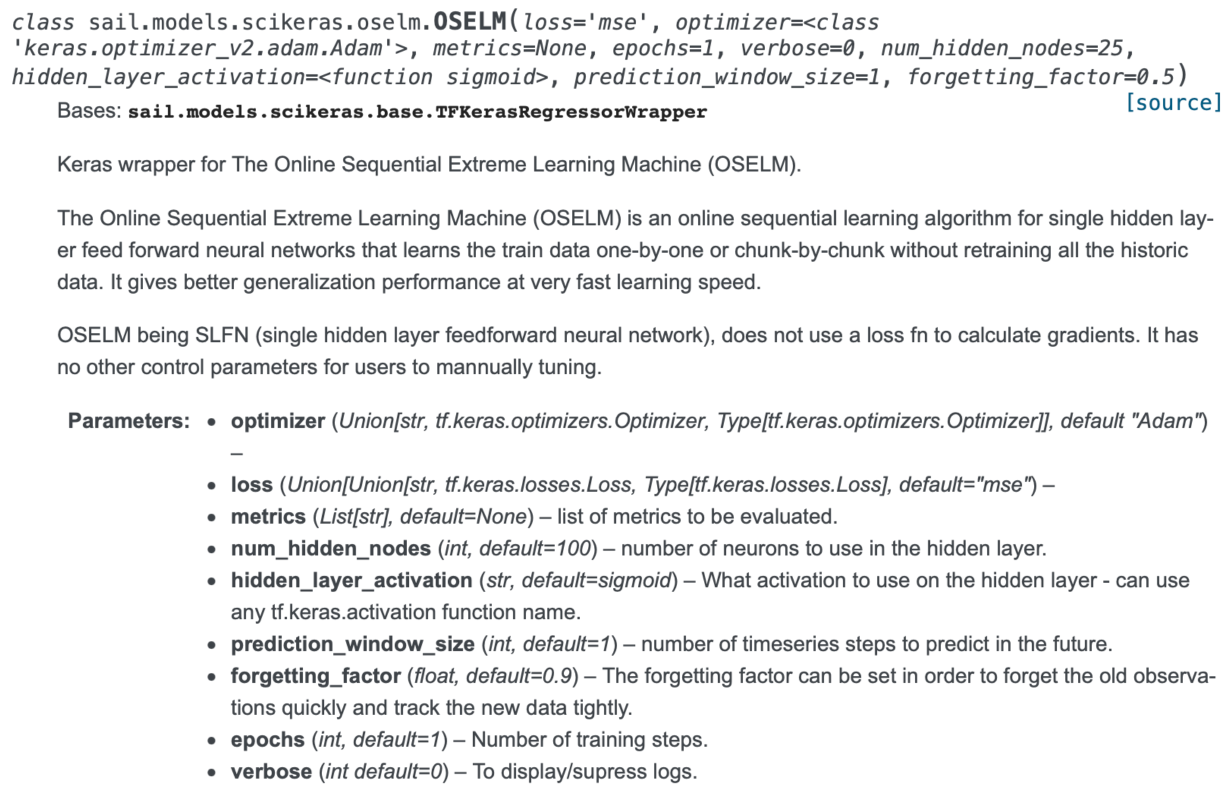
- The architecture, which specifies what layers the model contain, and how they're connected.
- A set of weights values
- An optimizer state.
- A set of losses and metrics.

Load() methods loads a model saved via Keras model.save().

The methods like *partial_fit* and *predict* are defined in the KerasRegressor class.

4.3.1.1 OSELM - The Online Sequential Extreme Learning Machine

The public API is a user-accessible wrapper class for Online Sequential Extreme Learning Machine (OSELM). It inherits an internal class TFKerasRegressorWrapper. It wraps an implementation of the Online Sequential Extreme Learning Machine (OSELM) that subclassed Keras model.



```
class sail.models.scikeras.oselm.OSELM(loss='mse', optimizer=<class
'keras.optimizer_v2.adam.Adam'>, metrics=None, epochs=1, verbose=0, num_hidden_nodes=25,
hidden_layer_activation=<function sigmoid>, prediction_window_size=1, forgetting_factor=0.5)
```
[source]

Bases: `sail.models.scikeras.base.TFKerasRegressorWrapper`

Keras wrapper for The Online Sequential Extreme Learning Machine (OSELM).

The Online Sequential Extreme Learning Machine (OSELM) is an online sequential learning algorithm for single hidden layer feed forward neural networks that learns the train data one-by-one or chunk-by-chunk without retraining all the historic data. It gives better generalization performance at very fast learning speed.

OSELM being SLFN (single hidden layer feedforward neural network), does not use a loss fn to calculate gradients. It has no other control parameters for users to mannually tuning.

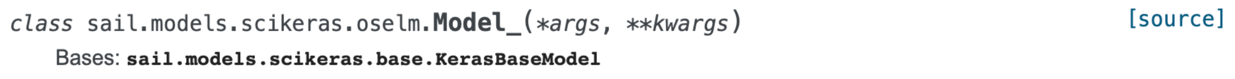
Parameters:

- **optimizer** (*Union[str, tf.keras.optimizers.Optimizer, Type[tf.keras.optimizers.Optimizer]], default "Adam"*) –
- **loss** (*Union[Union[str, tf.keras.losses.Loss, Type[tf.keras.losses.Loss], default="mse"*) –
- **metrics** (*List[str], default=None*) – list of metrics to be evaluated.
- **num_hidden_nodes** (*int, default=100*) – number of neurons to use in the hidden layer.
- **hidden_layer_activation** (*str, default=sigmoid*) – What activation to use on the hidden layer - can use any tf.keras.activation function name.
- **prediction_window_size** (*int, default=1*) – number of timeseries steps to predict in the future.
- **forgetting_factor** (*float, default=0.9*) – The forgetting factor can be set in order to forget the old observations quickly and track the new data tightly.
- **epochs** (*int, default=1*) – Number of training steps.
- **verbose** (*int default=0*) – To display/supress logs.

The OSLEM wrapper takes the number of hidden nodes, hidden layer activation function, prediction window size, and forgetting factor. It takes other parameters (loss, metrics, epochs, verbose) which are passed on to the base TFKerasRegressionWrapper.

OSLEM Model

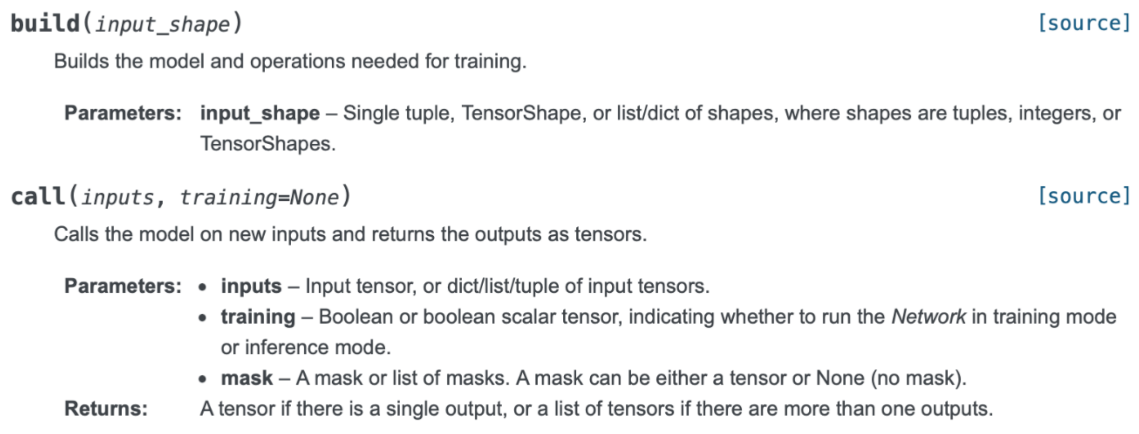
Online Sequential Extreme Learning Machine (OSELM) is an online sequential learning algorithm for single hidden layer feed forward neural networks that learns the train data one-by-one or chunk-by-chunk without retraining all the historic data. It gives better generalization performance at very fast learning speed.



```
class sail.models.scikeras.oselm.Model_(*args, **kwargs)
```
[source]

Bases: `sail.models.scikeras.base.KerasBaseModel`

OSLEM model inherits an internal class KerasBaseModel which inherits Keras model. It provides functions and methods common to SAIL models. It has been implemented by sub-classing the Keras model. It does so to write its training loop from scratch and to take control of the detail. It makes use of the Functional API of Keras.



build(*input_shape*) [source]

Builds the model and operations needed for training.

Parameters: **input_shape** – Single tuple, TensorShape, or list/dict of shapes, where shapes are tuples, integers, or TensorShapes.

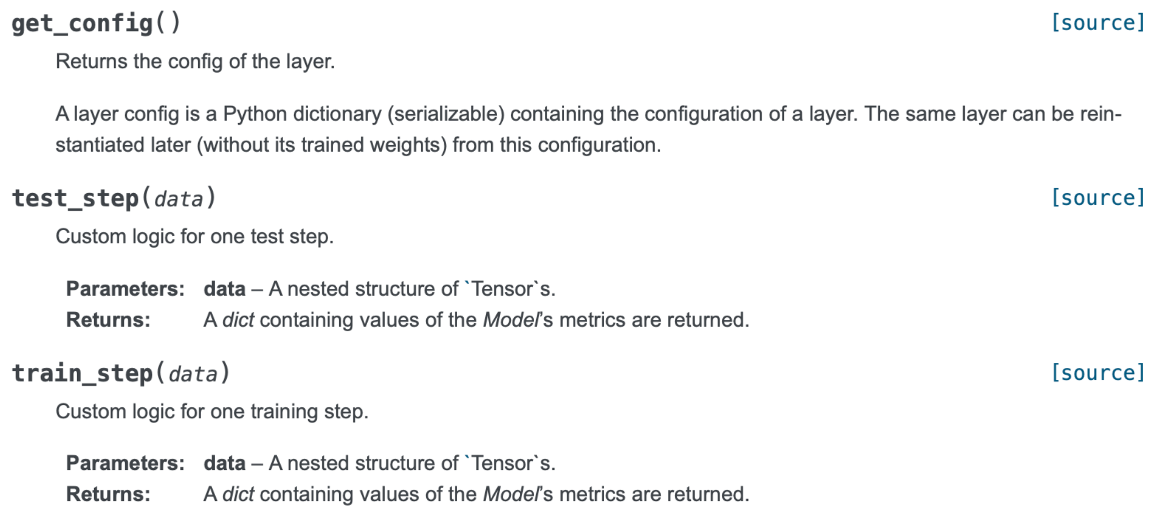
call(*inputs*, *training=None*) [source]

Calls the model on new inputs and returns the outputs as tensors.

Parameters:
- **inputs** – Input tensor, or dict/list/tuple of input tensors.
- **training** – Boolean or boolean scalar tensor, indicating whether to run the *Network* in training mode or inference mode.
- **mask** – A mask or list of masks. A mask can be either a tensor or None (no mask).

Returns: A tensor if there is a single output, or a list of tensors if there are more than one outputs.

- It uses the build() method to create layers and variables needed for the training operations. It builds the model based on input shapes received.
- Call method() to create the model and pass inputs. It acts as a single feed forward pass.



get_config() [source]

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

test_step(*data*) [source]

Custom logic for one test step.

Parameters: **data** – A nested structure of \`Tensor\`s.
Returns: A *dict* containing values of the *Model*'s metrics are returned.

train_step(*data*) [source]

Custom logic for one training step.

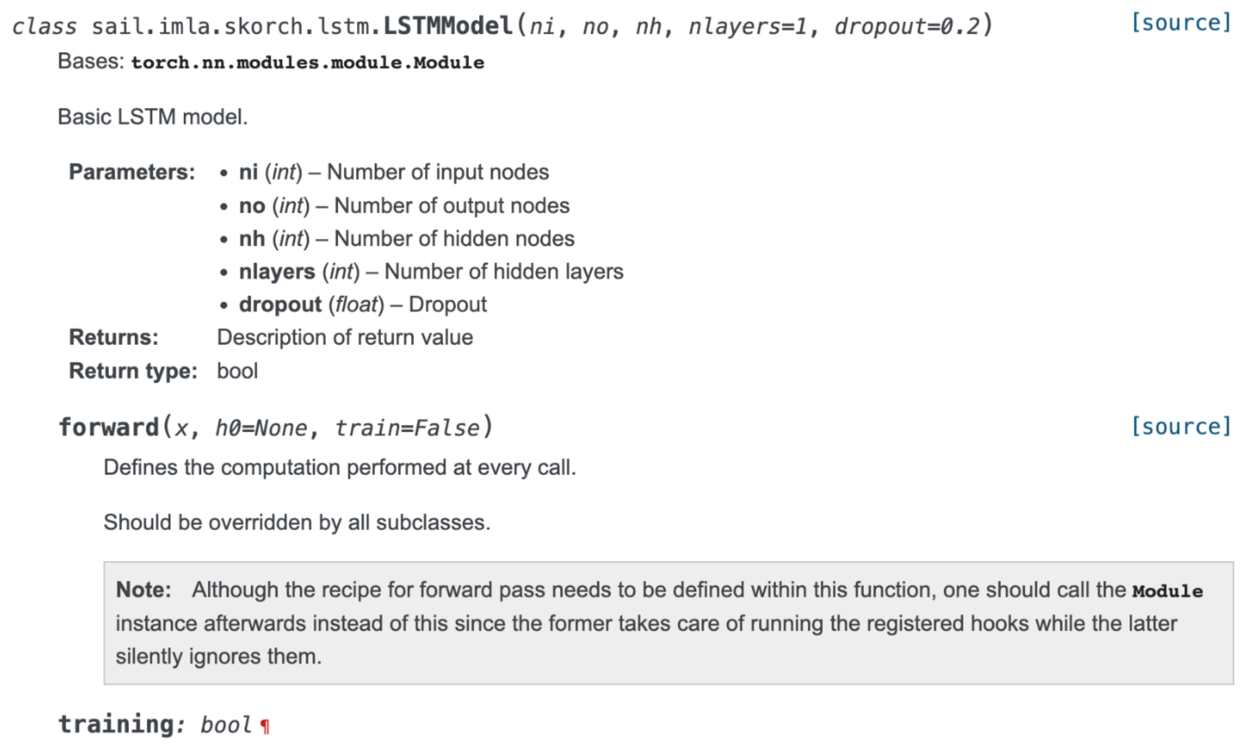
Parameters: **data** – A nested structure of \`Tensor\`s.
Returns: A *dict* containing values of the *Model*'s metrics are returned.

Test_step() and train_step() is overridden to provide a single custom training and evaluation step. Both methods return a dictionary mapping metric name (including the loss) to their current value.

4.4 PyTorch and Skorch wrapper

PyTorch provides an alternative to TensorFlow for deep learning models. In SAIL, standard deep learning models like LSTM, RNN and GRU are implemented. Figure 11 and Figure 12 shows an example set-up for PyTorch models in SAIL. Figure 11 defines the LSTM model and the corresponding forward pass. The *LSTMModel* takes the number of input nodes, output nodes, hidden nodes, number of layers and dropout as parameters. Other optional parameters like learning rate and number of epochs are passed through the Skorch *NeuralNetRegressor* wrapper (Tietz, et al. 2017).

Figure 12 shows the definition of the SAIL model *LSTMRegressor* that is built on top of the PyTorch *LSTMModel* defined in Figure 11. As can be seen, LSTMRegressor inherits from the SAIL *BaseEstimator* and *RegressorMixin*. The abstract methods in *RegressorMixin* like *partial_fit* and *predict* are defined in this class.



`class sail.imla.skorch.lstm.LSTMModel(ni, no, nh, nlayers=1, dropout=0.2)` [source]

Bases: `torch.nn.modules.module.Module`

Basic LSTM model.

Parameters:
- **ni** (*int*) – Number of input nodes
- **no** (*int*) – Number of output nodes
- **nh** (*int*) – Number of hidden nodes
- **nlayers** (*int*) – Number of hidden layers
- **dropout** (*float*) – Dropout

Returns: Description of return value

Return type: bool

`forward(x, h0=None, train=False)` [source]

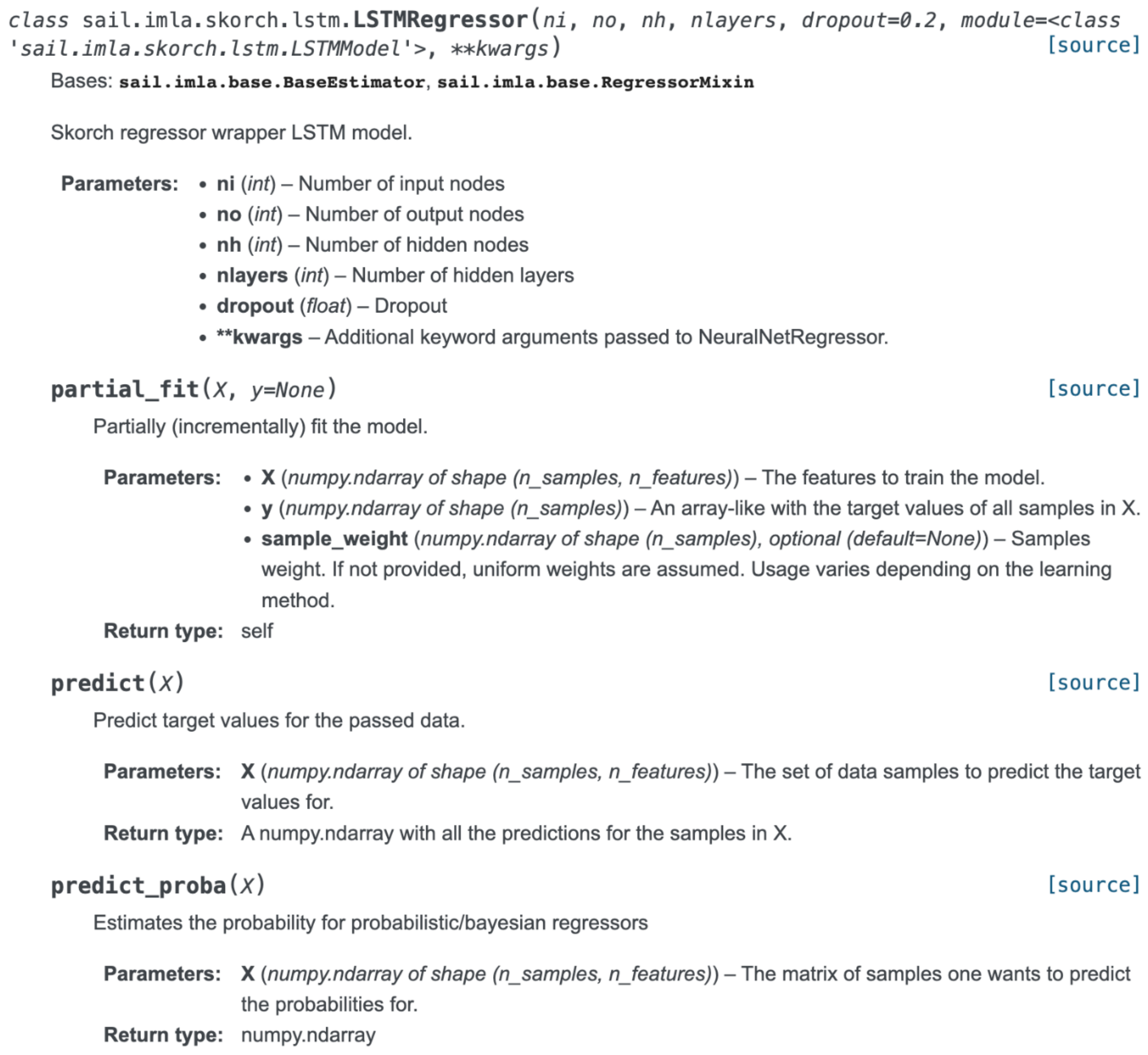
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool` ¶

Figure 11 LSTM model defined in PyTorch.



```
class sail.imla.skorch.lstm.LSTMRegressor(ni, no, nh, nlayers, dropout=0.2, module=<class
'sail.imla.skorch.lstm.LSTMModel'>, **kwargs)                                    [source]
```

Bases: **sail.imla.base.BaseEstimator**, **sail.imla.base.RegressorMixin**

Skorch regressor wrapper LSTM model.

Parameters:
- **ni** (*int*) – Number of input nodes
- **no** (*int*) – Number of output nodes
- **nh** (*int*) – Number of hidden nodes
- **nlayers** (*int*) – Number of hidden layers
- **dropout** (*float*) – Dropout
- ****kwargs** – Additional keyword arguments passed to NeuralNetRegressor.

```
partial_fit(X, y=None)                                                           [source]
```

Partially (incrementally) fit the model.

Parameters:
- **X** (*numpy.ndarray of shape (n_samples, n_features)*) – The features to train the model.
- **y** (*numpy.ndarray of shape (n_samples)*) – An array-like with the target values of all samples in X.
- **sample_weight** (*numpy.ndarray of shape (n_samples), optional (default=None)*) – Samples weight. If not provided, uniform weights are assumed. Usage varies depending on the learning method.

Return type: self

```
predict(X)                                                                       [source]
```

Predict target values for the passed data.

Parameters: **X** (*numpy.ndarray of shape (n_samples, n_features)*) – The set of data samples to predict the target values for.

Return type: A numpy.ndarray with all the predictions for the samples in X.

```
predict_proba(X)                                                                 [source]
```

Estimates the probability for probabilistic/bayesian regressors

Parameters: **X** (*numpy.ndarray of shape (n_samples, n_features)*) – The matrix of samples one wants to predict the probabilities for.

Return type: numpy.ndarray

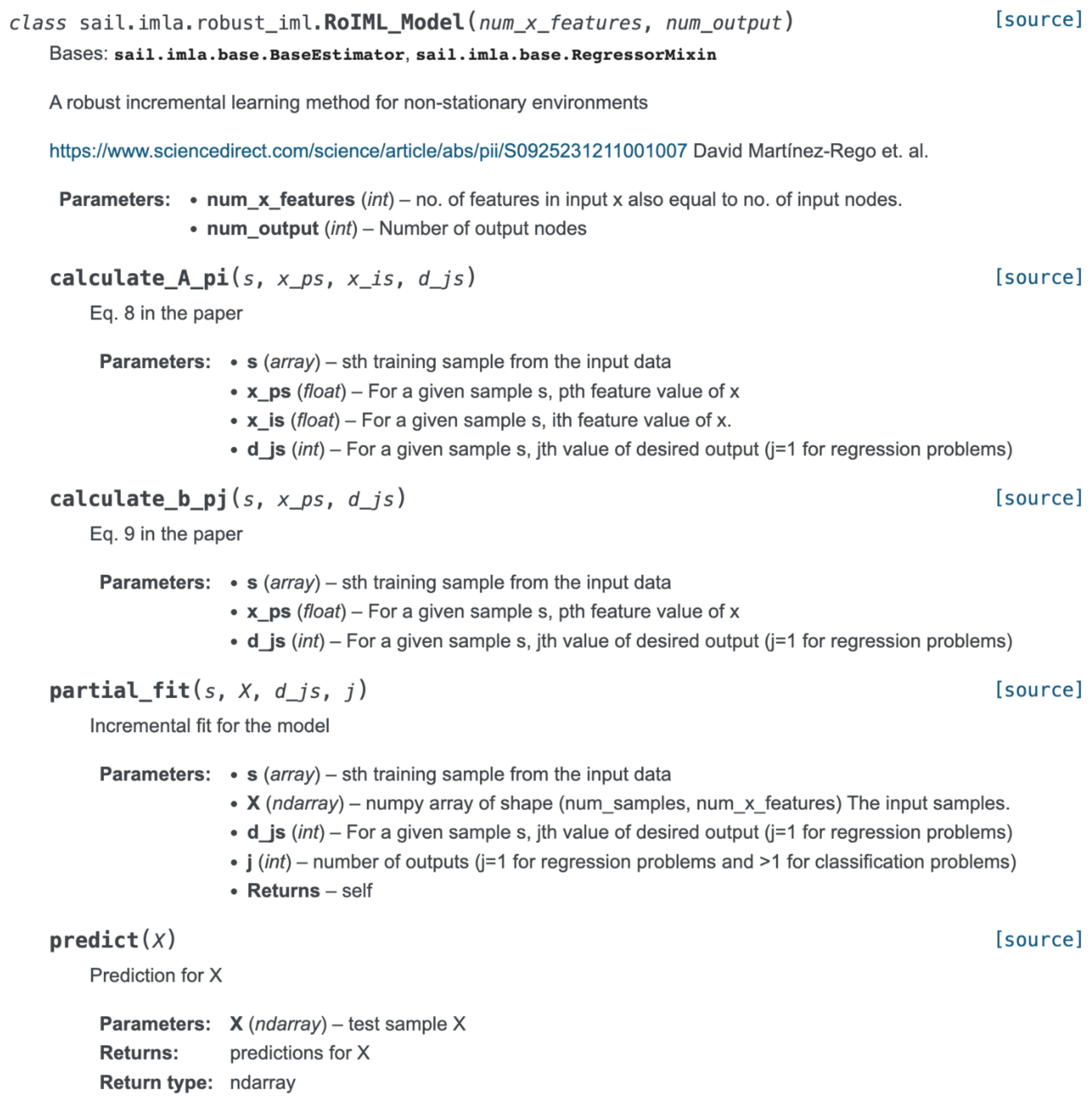
Figure 12 SAIL wrapper for the LSTM model defined in PyTorch.

4.5 State of the art deep-learning models

In addition to standard deep learning models, SAIL also has implementations to state-of-the-art deep learning and neural network models from the literature that follows the same architecture as Figure 2.

4.5.1 Robust incremental learning method for non-stationary environments.

Figure 13 shows the implementation of the paper "A robust incremental learning method for non-stationary environments" by (Martinez-Rego 2011).This is a native implementation in the *Original models* category as shown in Figure 2. For details regarding the features of the algorithm, the reader is referred to (Martinez-Rego 2011).



class sail.imla.robust_iml.**RoIML_Model**(*num_x_features*, *num_output*) [source]

Bases: **sail.imla.base.BaseEstimator**, **sail.imla.base.RegressorMixin**

A robust incremental learning method for non-stationary environments

https://www.sciencedirect.com/science/article/abs/pii/S0925231211001007 David Martínez-Rego et. al.

Parameters:
- **num_x_features** (*int*) – no. of features in input x also equal to no. of input nodes.
- **num_output** (*int*) – Number of output nodes

calculate_A_pi(*s*, *x_ps*, *x_is*, *d_js*) [source]

Eq. 8 in the paper

Parameters:
- **s** (*array*) – sth training sample from the input data
- **x_ps** (*float*) – For a given sample s, pth feature value of x
- **x_is** (*float*) – For a given sample s, ith feature value of x.
- **d_js** (*int*) – For a given sample s, jth value of desired output (j=1 for regression problems)

calculate_b_pj(*s*, *x_ps*, *d_js*) [source]

Eq. 9 in the paper

Parameters:
- **s** (*array*) – sth training sample from the input data
- **x_ps** (*float*) – For a given sample s, pth feature value of x
- **d_js** (*int*) – For a given sample s, jth value of desired output (j=1 for regression problems)

partial_fit(*s*, *X*, *d_js*, *j*) [source]

Incremental fit for the model

Parameters:
- **s** (*array*) – sth training sample from the input data
- **X** (*ndarray*) – numpy array of shape (num_samples, num_x_features) The input samples.
- **d_js** (*int*) – For a given sample s, jth value of desired output (j=1 for regression problems)
- **j** (*int*) – number of outputs (j=1 for regression problems and >1 for classification problems)
- **Returns** – self

predict(*X*) [source]

Prediction for X

Parameters: **X** (*ndarray*) – test sample X

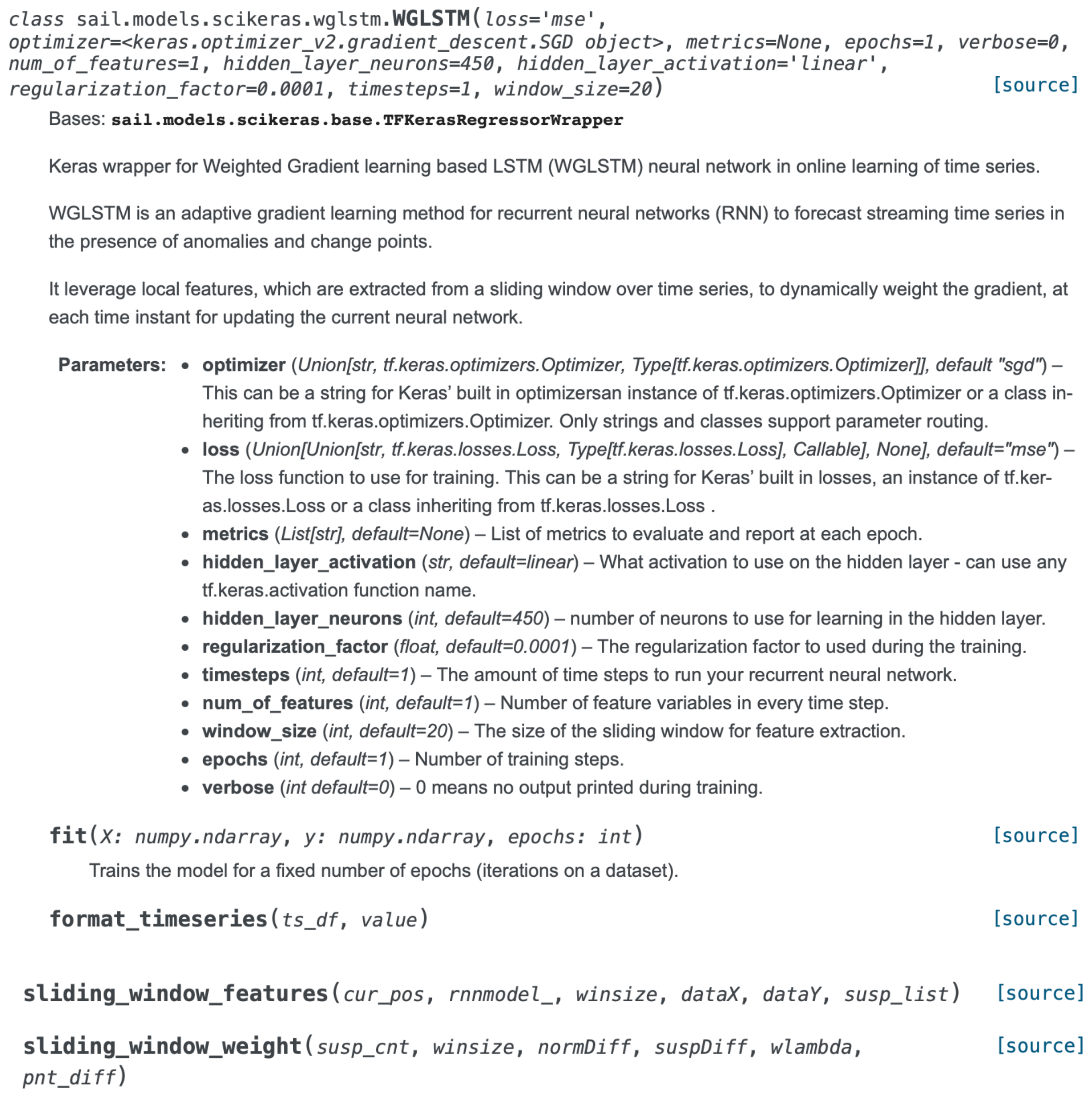
Returns: predictions for X

Return type: ndarray

Figure 13 Robust incremental machine learning algorithm as defined in Martinez-Rego et.al.

4.5.2 Weighted gradient LSTM online learning algorithm (WGLSTM)

The public API is a user-accessible wrapper class for the Weighted Gradient learning-based LSTM (WGLSTM) neural network. It inherits an internal class TFKerasRegressorWrapper (explained in section 4.3.1). It wraps a Keras Sequential model that builds a WGLSTM network.



```
class sail.models.scikeras.wglstm.WGLSTM(loss='mse',
optimizer=<keras.optimizer_v2.gradient_descent.SGD object>, metrics=None, epochs=1, verbose=0,
num_of_features=1, hidden_layer_neurons=450, hidden_layer_activation='linear',
regularization_factor=0.0001, timesteps=1, window_size=20)                      [source]
```

Bases: `sail.models.scikeras.base.TFKerasRegressorWrapper`

Keras wrapper for Weighted Gradient learning based LSTM (WGLSTM) neural network in online learning of time series.

WGLSTM is an adaptive gradient learning method for recurrent neural networks (RNN) to forecast streaming time series in the presence of anomalies and change points.

It leverage local features, which are extracted from a sliding window over time series, to dynamically weight the gradient, at each time instant for updating the current neural network.

Parameters:

- **optimizer** (*Union[str, tf.keras.optimizers.Optimizer, Type[tf.keras.optimizers.Optimizer]], default "sgd"*) – This can be a string for Keras' built in optimizersan instance of tf.keras.optimizers.Optimizer or a class inheriting from tf.keras.optimizers.Optimizer. Only strings and classes support parameter routing.
- **loss** (*Union[Union[str, tf.keras.losses.Loss, Type[tf.keras.losses.Loss], Callable], None], default="mse"*) – The loss function to use for training. This can be a string for Keras' built in losses, an instance of tf.keras.losses.Loss or a class inheriting from tf.keras.losses.Loss .
- **metrics** (*List[str], default=None*) – List of metrics to evaluate and report at each epoch.
- **hidden_layer_activation** (*str, default=linear*) – What activation to use on the hidden layer - can use any tf.keras.activation function name.
- **hidden_layer_neurons** (*int, default=450*) – number of neurons to use for learning in the hidden layer.
- **regularization_factor** (*float, default=0.0001*) – The regularization factor to used during the training.
- **timesteps** (*int, default=1*) – The amount of time steps to run your recurrent neural network.
- **num_of_features** (*int, default=1*) – Number of feature variables in every time step.
- **window_size** (*int, default=20*) – The size of the sliding window for feature extraction.
- **epochs** (*int, default=1*) – Number of training steps.
- **verbose** (*int default=0*) – 0 means no output printed during training.

```
fit(X: numpy.ndarray, y: numpy.ndarray, epochs: int)                            [source]
```

Trains the model for a fixed number of epochs (iterations on a dataset).

```
format_timeseries(ts_df, value)                                                 [source]
```

```
sliding_window_features(cur_pos, rnnmodel_, winsize, dataX, dataY, susp_list)   [source]
```

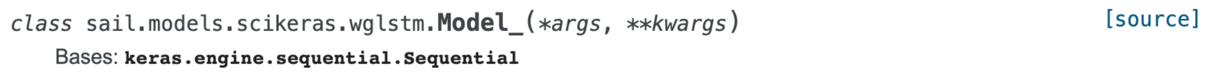
```
sliding_window_weight(susp_cnt, winsize, normDiff, suspDiff, wlambda,           [source]
pnt_diff)
```

Weighted Gradient learning based LSTM (WGLSTM) neural network is an adaptive gradient learning method for recurrent neural networks (RNN) to forecast streaming time series in the presence of anomalies and change points. WGLSTM incrementally learn the streaming time series and provide robust predictions adapting to the changing patterns as well as resisting to outliers. It introduces the weighted gradient to the online SGD for the RNN models.

It leverages local features, which are extracted from a sliding window over time series, to dynamically weight the gradient, at each time instant for updating the current neural network.

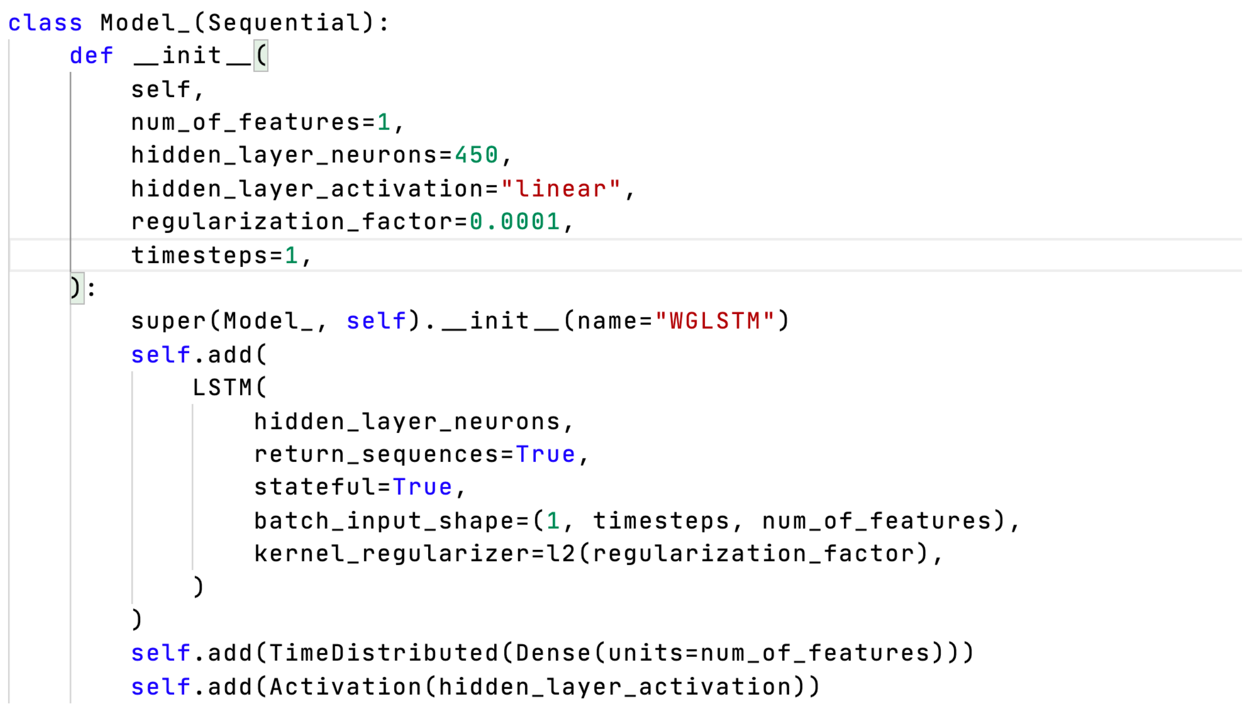
- In the above APIs, WGLSTM wrapper takes num_of_features, hidden_layer_neurons, hidden_layer_activation function, regularization_factor, and timesteps.
- The fit() function is overridden to accommodate custom training steps.
- format_timeseries() is used to create windows of features and targets from a time series.
- The key idea here is to leverage local features, which are extracted from a sliding_window_features() over time series, to dynamically weight the gradient via sliding_window_weight().

Sequential Model



class sail.models.scikeras.wglstm.**Model_**(**args*, ***kwargs*) [source]

Bases: **keras.engine.sequential.Sequential**



```
class Model_(Sequential):
    def __init__(
        self,
        num_of_features=1,
        hidden_layer_neurons=450,
        hidden_layer_activation="linear",
        regularization_factor=0.0001,
        timesteps=1,
    ):
        super(Model_, self).__init__(name="WGLSTM")
        self.add(
            LSTM(
                hidden_layer_neurons,
                return_sequences=True,
                stateful=True,
                batch_input_shape=(1, timesteps, num_of_features),
                kernel_regularizer=l2(regularization_factor),
            )
        )
        self.add(TimeDistributed(Dense(units=num_of_features)))
        self.add(Activation(hidden_layer_activation))
```

A novel LSTM neural network to be capable of making robust prediction over time series in the presence of both outliers and change points.

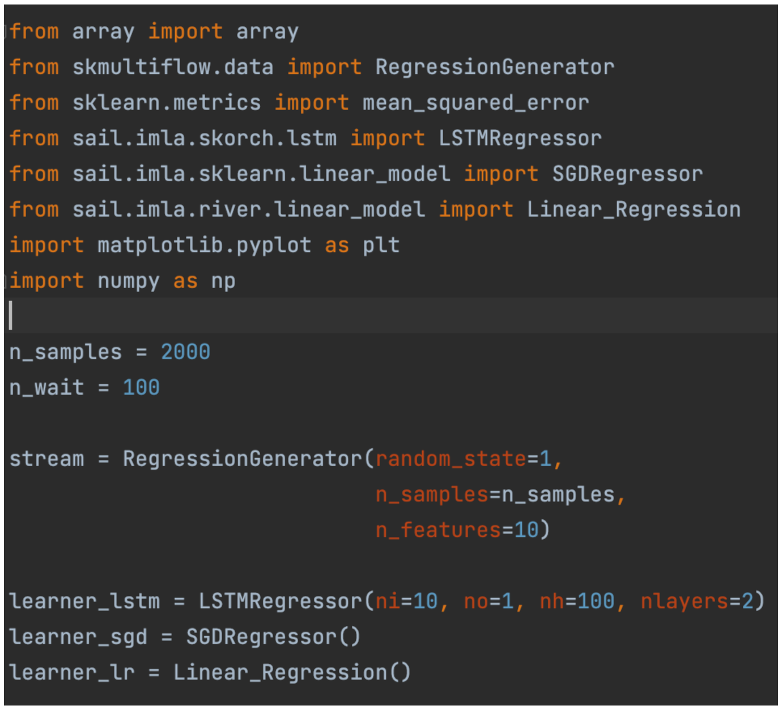
5 Examples

This section illustrates the two main advantages of SAIL, namely:

- Working with online models (River), simple incremental learning models (Scikit-learn) and incremental deep learning models (PyTorch) using a common interface.
- Ensemble of batch and incremental models. Both batch and incremental models can be online models, simple incremental models or incremental deep learning models.

5.1 Online learning, simple models and deep learning models

For the first example, we consider a regression problem with the streaming data generated as defined by the algorithm in (Street 2001). The implementation of this algorithms as shown in Scikit-multiflow is used for this example. As can be seen in Figure 14 and Figure 15, SAIL ensures that the interface for the end user is consistent for online, simple incremental or incremental deep learning models. All the models take numpy arrays as input and produce a numpy array for predictions. Finally, Figure 16, shows the results for the predictions for the three models as compared to the ground truth as a sanity check.



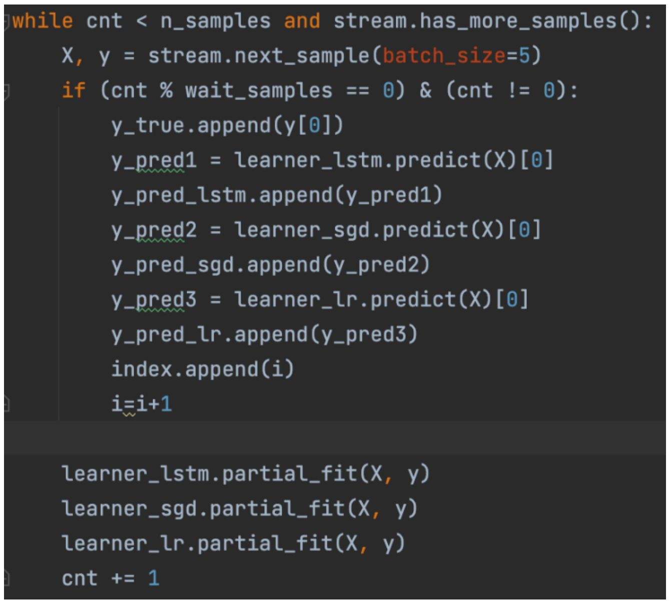
```
from array import array
from skmultiflow.data import RegressionGenerator
from sklearn.metrics import mean_squared_error
from sail.imla.skorch.lstm import LSTMRegressor
from sail.imla.sklearn.linear_model import SGDRegressor
from sail.imla.river.linear_model import Linear_Regression
import matplotlib.pyplot as plt
import numpy as np

n_samples = 2000
n_wait = 100

stream = RegressionGenerator(random_state=1,
                             n_samples=n_samples,
                             n_features=10)

learner_lstm = LSTMRegressor(ni=10, no=1, nh=100, nlayers=2)
learner_sgd = SGDRegressor()
learner_lr = Linear_Regression()
```

Figure 14 Consistent instantiations and imports of various types of incremental models.



```
while cnt < n_samples and stream.has_more_samples():
    X, y = stream.next_sample(batch_size=5)
    if (cnt % wait_samples == 0) & (cnt != 0):
        y_true.append(y[0])
        y_pred1 = learner_lstm.predict(X)[0]
        y_pred_lstm.append(y_pred1)
        y_pred2 = learner_sgd.predict(X)[0]
        y_pred_sgd.append(y_pred2)
        y_pred3 = learner_lr.predict(X)[0]
        y_pred_lr.append(y_pred3)
        index.append(i)
        i=i+1

    learner_lstm.partial_fit(X, y)
    learner_sgd.partial_fit(X, y)
    learner_lr.partial_fit(X, y)
    cnt += 1
```

Figure 15 Partial_fit and predict APIs for base models in PyTorch, Scikit-learn and River.

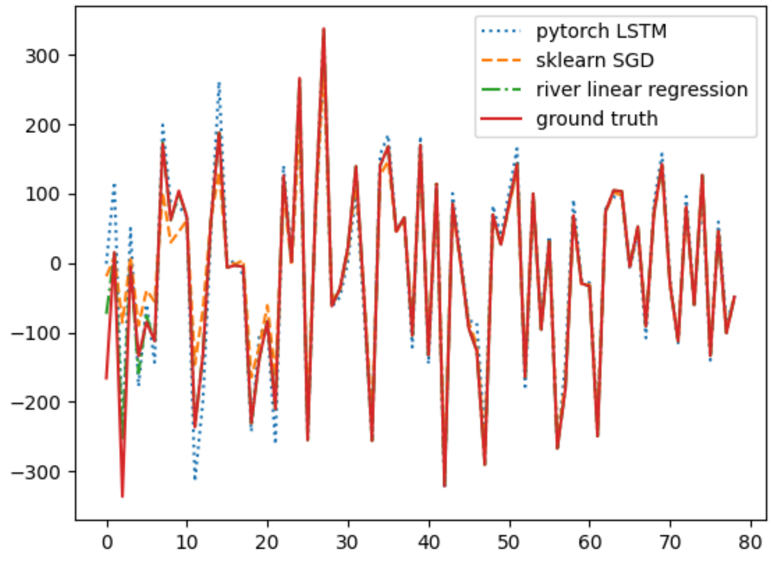
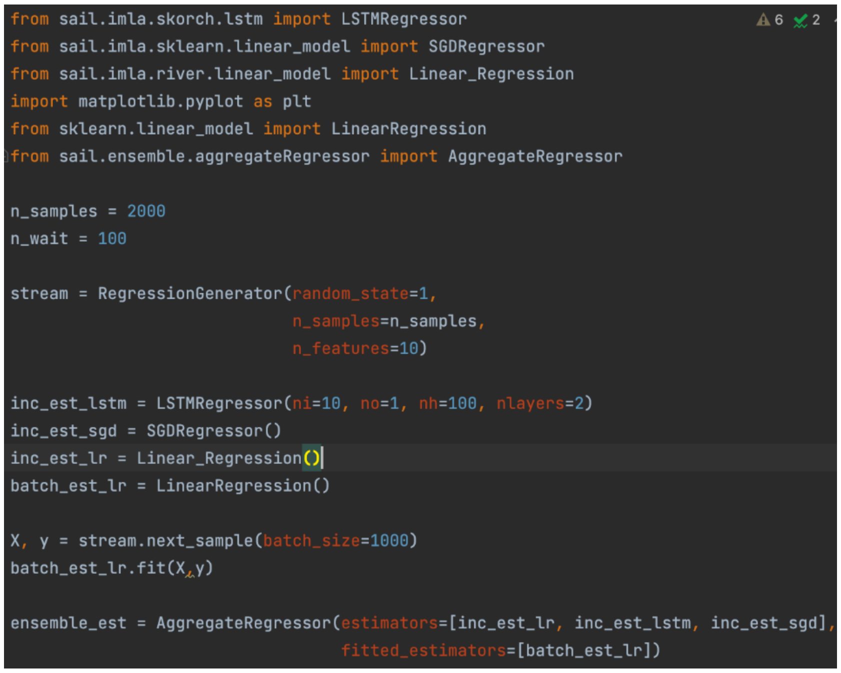


Figure 16 Prediction vs ground truth for all the base estimators defined in Section 5.1.

5.2 Ensembling of batch and incremental models

It has been recently shown in (Von Krannichfeldt 2020), ensembling of batch and incremental models results in improved load forecasting. Ensembling in SAIL allows for both batch and incremental models. As an example, we consider the same data stream model and same incremental models as defined in Section 5.1. A linear regression model from Scikit-learn is defined in addition as a batch model (Figure 17.) . The *AggregateRegressor* ensemble class takes *estimators* for incremental training as well as *fitted_estimators* as batch learning models that are not trained incrementally. Ensemble models follow the same *partial_fit* and *predict* APIs for incremental training and prediction as the base estimators. The ensemble *partial_fit* in this case only trains the incremental models when new data arrives while the batch models as well as incremental models are used for ensembling. Figure 18 shows the results of the ensemble model as compared to the ground truth for the SEA Generator dataset.



```
from sail.imla.skorch.lstm import LSTMRegressor
from sail.imla.sklearn.linear_model import SGDRegressor
from sail.imla.river.linear_model import Linear_Regression
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sail.ensemble.aggregateRegressor import AggregateRegressor

n_samples = 2000
n_wait = 100

stream = RegressionGenerator(random_state=1,
                             n_samples=n_samples,
                             n_features=10)

inc_est_lstm = LSTMRegressor(ni=10, no=1, nh=100, nlayers=2)
inc_est_sgd = SGDRegressor()
inc_est_lr = Linear_Regression()
batch_est_lr = LinearRegression()

X, y = stream.next_sample(batch_size=1000)
batch_est_lr.fit(X,y)

ensemble_est = AggregateRegressor(estimators=[inc_est_lr, inc_est_lstm, inc_est_sgd],
                                  fitted_estimators=[batch_est_lr])
```

Figure 17 Ensemble of models developed in PyTorch, Scikit-learn and River.

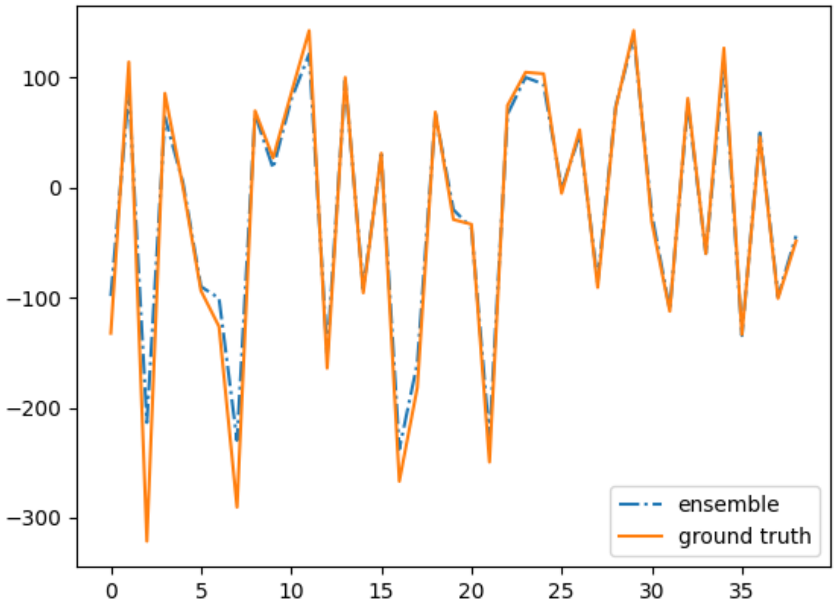


Figure 18 Prediction vs ground truth for all the base estimators defined in Section 5.2.

6 Conclusion and Future work

A robust framework has been setup to build and work on incremental machine learning algorithms in SAIL. The library supports simple online learning algorithms from libraries like River, Scikit-Multiflow and also native incremental deep learning models developed using PyTorch and TensorFlow. Common APIs have been developed to incrementally train and score these models. Ensembling of these models has also been developed and tested.

As stated earlier, SAIL forms the core component for incremental models and AutoML pipelines. So future work will concentrate on leveraging the models in this library for the remaining tasks in MORE. These include:

- Develop AutoML pipelines for incremental models.
- Develop distributed AutoML pipelines for incremental models.
- Test these algorithms for RES use-cases like soiling in solar panels, yaw misalignment in wind turbines etc.
- Implement incremental models with compressed data models from SAIL.

7 Bibliography

n.d. *https://github.com/scikit-learn/scikit-learn/blob/1fe00b58949a6b9bce45e9e15eb8b9c138bd6a2e/sklearn/base.py.*

Huang, Guang-Bin, and Lei Chen. 2007. ""Convex incremental extreme learning machine."." *Neurocomputing 70, no. 16-18 3056-3062.*

Martinez-Rego, David, Beatriz Perez-Sanchez, Oscar Fontenla-Romero, and Amparo Alonso-Betanzos. 2011. ""A robust incremental learning method for non-stationary environments." ." *Neurocomputing 74, no. 11.*

Montiel, Jacob, Jesse Read, Albert Bifet, and Talel Abdessalem. 2018. ""Scikit-multiflow: A multi-output streaming framework." ." *The Journal of Machine Learning Research 19, no. 1.*

Montiel, Jacob, Max Halford, Saulo Martiello Mastelini, Geoffrey Bolmier, Raphael Sourty, Robin Vaysse, Adil Zouitine et al. 2021. ""River: machine learning for streaming data in Python." ."

Moritz, Philipp, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol et al. 2018. ""Ray: A distributed framework for emerging {AI} applications." ." *13th USENIX Symposium on Operating Systems Design and Implementation.*

Paszke, Adam, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. ""Automatic differentiation in pytorch." ."

Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel et al. 2011. ""Scikit-learn: Machine learning in Python." ." *Journal of machine Learning research .*

Street, W. Nick, and YongSeog Kim. 2001. ""A streaming ensemble algorithm (SEA) for large-scale classification."." *In Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining.*

Tietz, Marian, Thomas Fan, Daniel Nouri, and Benjamin Bossan. 2017. "skorch: A scikit-learn compatible neural network library that wraps PyTorch."

Varoquaux, Gael. 2021. *https://scikit-learn.org/stable/modules/generated/sklearn.base.BaseEstimator.html.*

Von Krannichfeldt, Leandro, Yi Wang, and Gabriela Hug. 2020. ""Online ensemble learning for load forecasting."." *IEEE Transactions on Power Systems 36, no. 1.*