D3.1 Core AI Models - Initial Version

Lead Partner:	IBM
Version:	1
Dissemination Level:	Public
Work Package:	WP3
Due date:	30/09/2021
Submission Date:	30/09/2021

Abstract:
This document describes the initial 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). Initial version of SAIL explores incremental machine learning algorithms, deep learning models, distributed ensemble models and distributed model selection algorithms for IMLA.

This report documents the core elements of SAIL and its positioning in the open source community. In particular, the report covers:
(1) Existing solutions and current challenges in incremental machine learning algorithms, (2) missing gap addressed by SAIL, (3) user guide to run SAIL, (4) Examples of SAIL, (5) future work.

ICT-H2020-51-2020 - Big Data technologies and extreme-scale analytics

Document Revision History

Date	Version	Author/Editor/Contributor	Summary of main changes / Status
28/09/2021	V1	Seshu Tirupathi, IBM	First version
25/9/2021		Reviewed by Manolis Terrovitis and Giorgos Giannopoulos	
30/09/2021	V2	Seshu Tirupathi, IBM	Updated based on review comments

Disclaimer

Copyright

Table of Contents

1 Introduction 4
1.1 WP Description and Deliverable 3.1 4
2 Literature review 6
2.1 Incremental machine learning 6
2.2 Out-of-core scalable computing 8
3 SAIL features 11
3.1 Core AI models 11
3.2 Ensembling – Ensembling with batch and incremental models 11
3.3 AutoML and model selector 12
3.4 Distributed computing through Ray 12
4 User guide 13
4.1 Installation 13
4.2 Documentation 13
5 Testing procedure and experiments 19
5.1 Classification example 20
5.2 Parallelization example 21
6 Conclusion and Future work 23
7 References 24

1 Introduction

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 M_t is updated based on model M_{t-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.

1.1 WP Description and Deliverable 3.1

WP3 is responsible for developing IMLA and AutoML pipelines on IMLA. These models are aimed to be used for high frequency and high volume forecasting and prediction tasks. Ensemble models are also planned to be developed as these methods are important tools to handle concept drift in streaming data [1].
SAIL was developed as part of Tasks 3.1 and 3.2 where IMLA models, AutoML pipelines, and ensemble models on benchmark problems need to be explored. In particular, links have been created to existing libraries to gain access to a host of incremental models. The models mentioned in Task 3.1 are covered by existing libraries or have been implemented in SAIL. All the algorithms that cover the breadth of incremental models [1] as defined in Task 3.1, especially for classification, are covered as part of SAIL through native algorithms or wrappers to existing libraries. These algorithms include:

1. Incremental Support Vector Machine (ISVM), 2. LASVM, 3. On-line Random Forest, 4. Incremental Learning Vector Quantization, 5. Learn++ (LPP CART), 6. Incremental Extreme Learning Machine, 7. Naive Bayes, 8. Stochastic Gradient Descent

The library iteslf will be packaged as a micro-service and consistent with the APIs to interact with ModelarDB defined in WP2 for summarized data and WP5 for the use cases and system requirements as defined in D5.1 and D5.2 respectively. .

In addition, SAIL also includes direct support for neural networks and deep learning models through Pytorch and Keras. The neural network models coupled with native IMLA models form the foundation of AutoML pipelines, ensemble learning. Further, these algorithms will eventually be used for the use cases in solar and wind farms.

In summary, Deliverable 3.1 documents the features of SAIL and a user guide to use or contribute to the library. SAIL is an open source library and hosted at https://github.com/IBM/sail and also forked at https://github.com/MORE-EU/sail.

2 Literature review

Incremental machine learning for big data streaming applications can be addressed by handling data efficiently, improving IMLA, and developing scalable architecture to handle the volume and velocity of big data. At a high level, the research directions for these three approaches that can be explored in MORE are shown in Table. 1. Hybrid methods can further be developed by considering a combination of the columnar topics mentioned in Table. 1. For example, MORE would consider a hybrid of using summarized data with IMLA for renewable energy sector.

Data	**Machine learning**	**Systems**
Data summarization techniques.	Incremental machine learning algorithms and AutoML on IMLA.	Architectural aspects/choices for handling high frequency data (apache arrow, Spark streaming etc).
Model training and updates depending on the incoming data.	Deep learning continual AI models extended for time series data (handling catastrophic forgetting).	Methods to reduce latency, communication etc.
Decisions on persistence of high volume and/or high velocity incoming data.	Federated learning for streaming data.	Systems to ensure resiliency and reliability of exteme scale computations.
Abstracting ideas from custom models developed for applications in renewable energy sector.	Abstracting ideas from custom models developed for applications in renewable energy sector.	Abstracting ideas from custom models developed for applications in renewable energy sector.

Table 1: Research directions in WP3 and interlinks possible with other WPs in MORE.

2.1 Incremental machine learning

Incremental machine learning algorithms had peripheral support in established libraries like Scikit-learn and other popular batch learning algorithms. However, as the popularity and applicability of these models is gaining prominence, dedicated libraries have been developed that cater exclusively to incremental models and especially online models. In particular, Creme and Scikit-Multiflow were the popular choices in the early part of this year. These two libraries have since been merged to form a unified library called River. In addition to River which concentrates on providing online models without active support for deep learning models. Pytorch and Keras are the two most popular options for deep learning models. There are other options for incremental models and the advantages and disadvantages of these libraries are mentioned in Table. 2.

Open source library	Features	Leveraged (Yes/No/Later)
River [2]	**Pros:** De-facto library for online models. Has a wide range of models, metrics, statistics, streams, built-in datasets. **Cons:** Limited support for neural networks (mainly through a compatible class).	Yes
Scikit-learn [3]	**Pros:** De-facto library for batch models. Has a wide range of models, metrics, statistics, built-in datasets for batch learning models. **Cons:** Limited support for neural networks (mainly through a compatible class). Limited support for incremental models.	Yes
Pytorch [4]	**Pros:** Popular library for deep learning models. **Cons:** Support for only naive incremental deep learning models by default.	Yes
Keras [5]	**Pros:** Popular library for deep learning models. **Cons:** Support for only naive incremental deep learning models by default.	Yes
Sktime [6]	**Pros:** Unified framework for machine learning with time series. **Cons:** Only support for naive incremental models by default. No support for deep learning models.	Yes
Sktime-dl [7]	**Pros:** Companion framework for deep learning with time series using Keras **Cons:** Only support for naive incremental deep learning models by default.	Optional

Scikit-multiflow [8]	**Pros:** Follows Scikit-learn framework for online learning. Has a wide range of online learning algorithms implemented in the Scikit-learn pipeline. **Cons:** Deprecated as this library has been merged with crème to a unifying library called River.	Yes*
crème [9]	**Pros:** Library for online models. Has a wide range of models, metrics, statistics, streams, built-in datasets. **Cons:** Deprecated as this library has been merged with Scikit-multiflow to a unifying library called River.	No
Spark Mllib [10]	**Pros:** Scalable machine learning library that runs on Apache Spark. Has limited set of batch learning algorithms, metrics, statistics, streams, built-in datasets. Very good for feature engieering and transformations. **Cons:** Connections have to be developed to use other state of the art machine learning libraries. Constrained to use Apache Spark.	No

Table 2: Open source libraries for IMLA

While multiple libraries exist for incremental models, there are no open source libraries for state of the art online learning models based on neural networks and deep learning algorithms.

2.2 Out-of-core scalable computing

For an end-to-end machine learning pipeline in MORE, frameworks/libraries are also required to handle extract-transform-load (ETL) aspects of big data and the ability to perform out-of-core computations in addition to incremental models. Extensive literature and open source libraries exist to handle different aspects of big data transformations and scalable computing for machine learning algorithms. Some of the libraries that were considered to integrate with SAIL are mentioned in Table. 3.

Open source library	Features	Leveraged (Yes/No/Optional)
Dask [11]	**Pros:** Provides dask arrays, dask dataframes to scale machine learning models on big data. Dask-ML scales machine learning APIs like scikit-learn for big data. Easy scalability APIs from local machine to cluster mode. Uses familiar Pandas-like APIs for arrays and dataframes. **Cons:** Deprecated as this library has been merged with Scikit-multiflow to a unifying library called River.	Optional (One of Ray backends)
Modin [12]	**Pros:** Provides parallel light weight dataframes for working with big data. Uses familiar Pandas-like APIs for arrays and dataframes. **Cons:** Relatively less popular compared to Pandas and user acceptance might be an issue.	Optional
Vaex [13]	**Pros:** Data wrangling library that is useful for big data applications to be tested even on a local machine. Uses lazy evaluations and parallelization for big data applications. **Cons:** Limited similarity to Pandas-like API as compared to Modin.	Optional
Rapids [14]	**Pros:** GPU deployment of machine learning pipelines. **Cons:** Difficult to use with limited similarity between cuDF and Pandas dataframes.	No

Spark [15]	**Pros:** Very popular for distributed computing. User friendly APIs for big data. **Cons:** Limited support for direct hyperparameter optimization. User defined funtions (UDF) have to be created to use popular machine learning algorithms and libraries. Constrained to run in one integrated environment.	No
Flink [16]	**Pros:** Very popular for distributed computing on streaming data. Has ready to use functions to link with Pandas dataframe. **Cons:** Limited support for direct hyperparameter optimization. User defined funtions (UDF) have to be created to use popular machine learning algorithms and libraries. Constrained to run in one integrated environment.	No
Ray [17]	**Pros:** Very popular for distributed computing or parallel execution. Minimal changes to existing code to scale from single CPU to cluster computing. Provides wrappers for many hyperparameter tuning libraries. **Cons:** Uptime to get the cluster running for parallel execution might make it unusable for small scale problems.	Yes

Table 3: Scalable computing libraries for big data and machine learning

Considering the flexibility that Ray provides and the wrappers it covers to other distributed computing frameworks (Dask, Modin etc) and hyperparameter tuning libraries (Ray-tune, Optuna etc.), it was chosen as the backend for distributed computing in SAIL. In addition libraries like Vaex can be considered optional since they can be used mainly during ETL operations.

3 SAIL features

As discussed in the last section, SAIL leverages the existing machine learning libraries with incremental learning support like River, sklearn etc and creates a common set of APIs to run these models in the backend. Also, as noted in particular, while River provides minimal utilities for deep learning models, it does not focus on deep learning models. SAIL provides wrappers to Pytorch and Keras models through Skorch and Scikeras to ensure common APIs to train and score the models. In addition, the common APIs for the models in SAIL ensures streamlined APIs for AutoML pipelines and hyperparameter tuning using Ray. One other drawback in the current libraries which is addressed in SAIL is that batch models and incremental models are disjoint. Practical applications would require batch learning and incremental models to be used in AutoML pipelines as well as ensemble models.

To summarize, the main advantages that SAIL provides as compared to existing libraries are:

- Bridge between incremental models and batch models for ensembling and AutoML components like model selection.
- Faster computational times for ensemble models (distributed through Ray).
- Faster computational times for ensemble of forecasts (distributed through Ray).
- Incremental models for deep learning models.
- Creates a clean Scikit-learn interface for developing AutoML algorithms for incremental models with traditional incremental models as well as deep learning models as the base estimators.

3.1 Core AI models

SAIL model provides a common Scikit-learn compatible interface for online models available in River (using the *compat* class), deep learning models in Pytorch using skorch [18]. Deep learning algorithms in Keras are planned to be integrated through Scikeras. In addition, initial experiments on the contiual AI workflow that has so far been used primarily for image data in libraries Avalanche [19] and CL-Gym [20] have been carried out and a stable integration is planned. With Continual AI, vanilla deep learning models and traditional incremental and online machine learning algorithms, SAIL will provide a unified interface for all the popular subclasses of incremental algorithms.

3.2 Ensembling – Ensembling with batch and incremental models

Ensembling models are important for reducing the variance in the predictions of machine learning models. Ensembling is also important when features are missing in streaming data or new classes are generated in the target variable. Ensembling libraries that are available in open source libraries either cater to combining exclusively online models (ex. River) or exclusively batch models (ex. Sklearn, combo). In practice, batch learning models or incremental models are optimized on offline models and can be used as an ensemble member. The ensembling algorithms are extended in this library to combine forecasts from both offline models (batch or incremental) with forecasts from incremental models. In addition, ensembling models on big data can be highly time consuming as the number of models in the ensemble increases. This issue is more pertinent when the ensemble models are trained on offline data. Ray has been used to parallelize the ensemble models for model based parallelization.

3.3 AutoML and model selector

One of the fundamental requirements of AutoML pipelines is to select the best model for a given prediction task. The model selection can be done using prequential data (score and then train) or holdout dataset (as usually done in batch learning models). Model selection algorithms for incremental models for both prequential and holdout mechanisms are available in SAIL. Like ensembling, model selection can be time intensive, especially with hyperparameter tuning. Hence, Ray has been used to parallelize model selection algorithms.

3.4 Distributed computing through Ray

Distributed computing for machine learning models is an active area of research and development and many open source libraries exist with respective inherent advantages for distributed computing. For example, joblib is a common choice for parallelization of sklearn pipelines. Dask and Ray are popular choices for parallelization of sklearn pipelines. Ray was the preferred option as the workflow is flexible in Ray with wrappers available for Dask or Joblib. This ensures that the incremental models in SAIL can be distributed directly in Ray or any other custom library which can be easily integrated with Ray. In addition, since the streaming algorithms were chosen to be Scikit-learn compatible, hyperparameter optimization algorithms available in Ray can be directly utilized.

4 User guide

4.1 Installation

SAIL can be installed through pip. The github repo can be cloned to the local machine and in the root directory where *setup.py* is located, the following command needs to be run

pip install .

This command will install SAIL along with its dependencies. It is strongly recommended that a virtual environment (like venv or conda) is created to run SAIL to avoid dependency conflicts with already installed packages. As seen in the setup file (*Illustration-1*), there are optional dependencies in SAIL (like skorch) which can be installed using the following command:

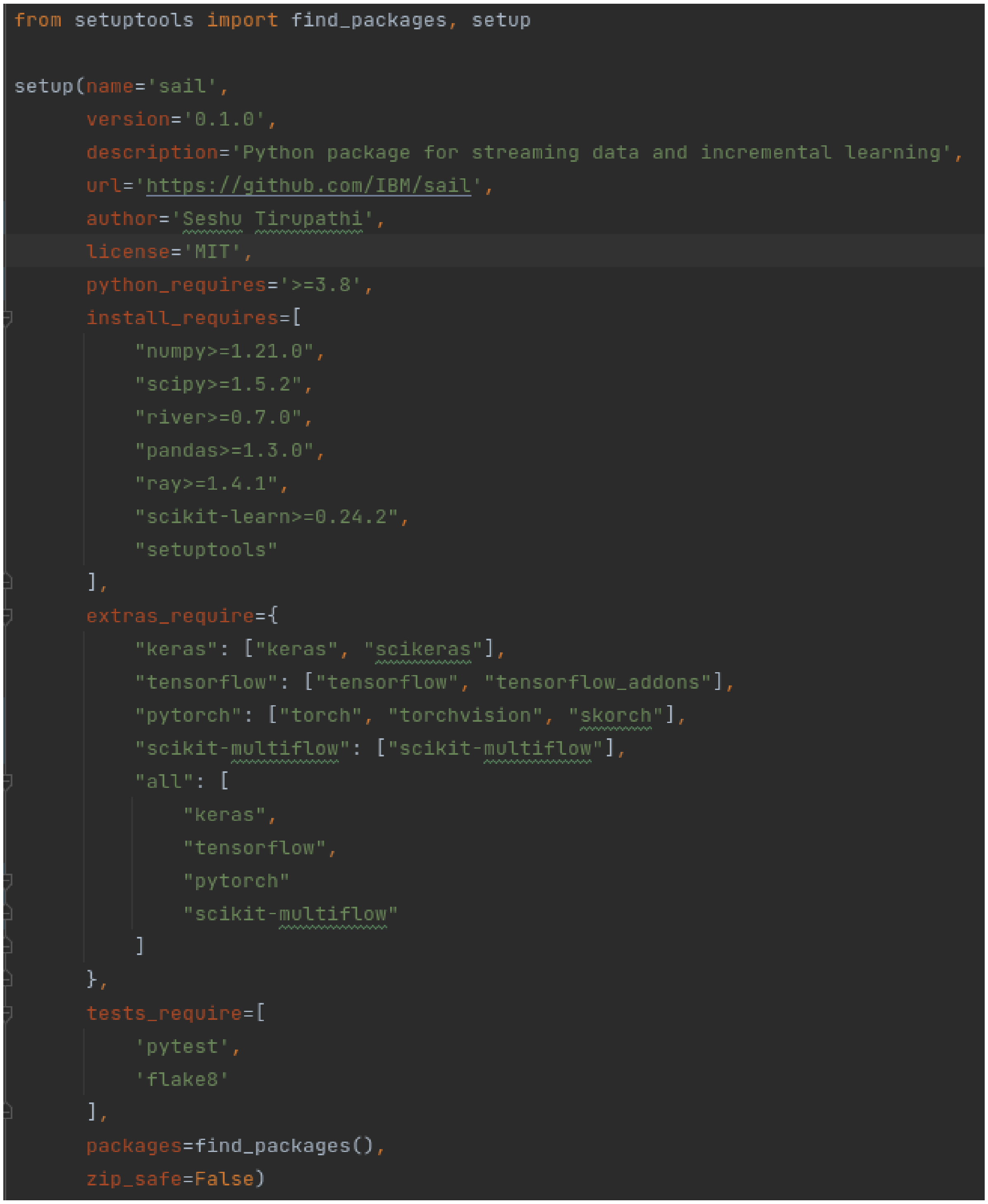
pip install -e .[package_name]

4.2 Documentation

Documentation in SAIL was generated using *docstring* convention and *Sphinx*. *Illustration-2* shows the summary page for the documentation generated through *Sphinx*. The documentation in the code clearly describes the inputs and outputs that a given function expects. Brief description on some of the models developed in SAIL and corresponding documentation generated by Sphinx on how to use the functions are given below:

Ensemble models – *Illustration-3* gives an overview of the base class for ensembling models. As stated in the introduction, it takes both *base_estimators (incremental)* and *fitted_estimators (trained)* as members of the class. Distributed versions for classification (*Illustration-4*) and regression (*Illustration-5*) are inherited from the base ensemble class. Weighted ensemble (*Illustration-6*) for incremental models also inherits from this base class without a *fitted_estimators* parameter defined.

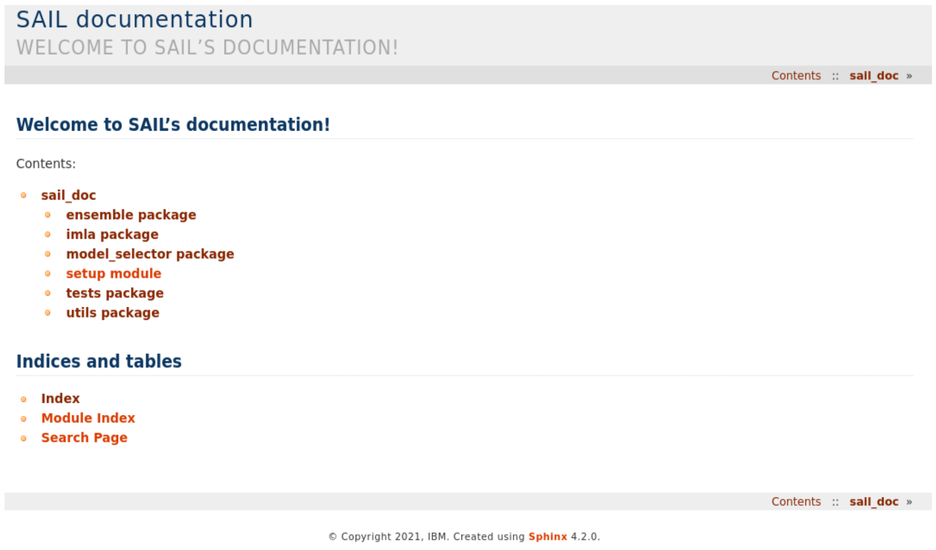
Model selector – *Illustration-7* shows the APIs that are publicly accessible for the base class for model selection. It provides functions to retrieve the best model, the index of the best model, list of the models considered etc. This class is inherited by two model selector classes where the criterion for selection changes, namely holdout or prequential (*Illustration-8*).



```
from setuptools import find_packages, setup

setup(name='sail',
      version='0.1.0',
      description='Python package for streaming data and incremental learning',
      url='https://github.com/IBM/sail',
      author='Seshu Tirupathi',
      license='MIT',
      python_requires='>=3.8',
      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",
          "setuptools"
      ],
      extras_require={
          "keras": ["keras", "scikeras"],
          "tensorflow": ["tensorflow", "tensorflow_addons"],
          "pytorch": ["torch", "torchvision", "skorch"],
          "scikit-multiflow": ["scikit-multiflow"],
          "all": [
              "keras",
              "tensorflow",
              "pytorch"
              "scikit-multiflow"
          ]
      },
      tests_require=[
          'pytest',
          'flake8'
      ],
      packages=find_packages(),
      zip_safe=False)
```

Illustration 1: setup.py script for installing SAIL.



SAIL documentation
WELCOME TO SAIL'S DOCUMENTATION!

Contents :: sail_doc »

Welcome to SAIL's documentation!

Contents:

- sail_doc
  - ensemble package
  - imla package
  - model_selector package
  - setup module
  - tests package
  - utils package

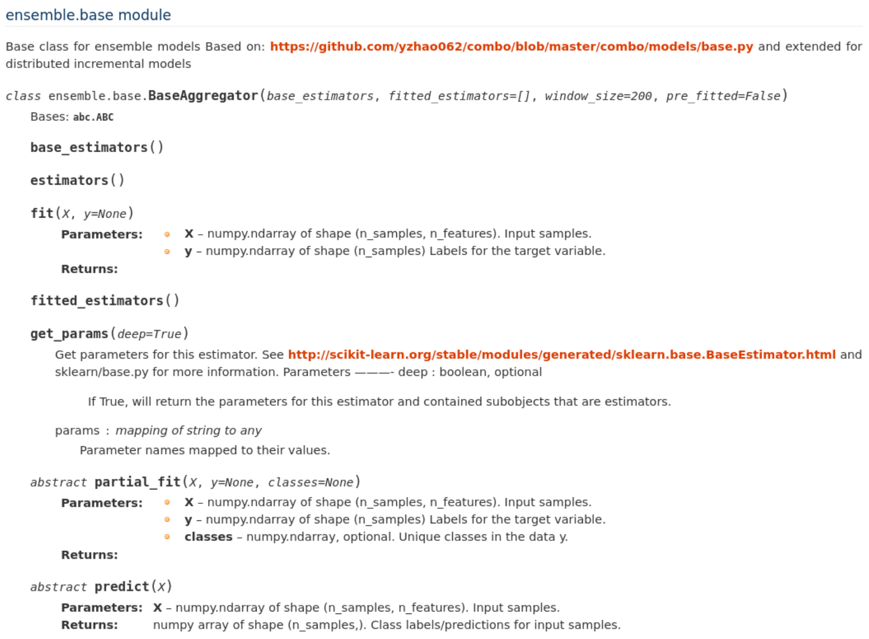
Indices and tables

- Index
- Module Index
- Search Page

Contents :: sail_doc »

© Copyright 2021, IBM. Created using Sphinx 4.2.0.

Illustration 2: Overview of SAIL documentation generated through Sphinx.



ensemble.base module

Base class for ensemble models Based on: https://github.com/yzhao062/combo/blob/master/combo/models/base.py and extended for distributed incremental models

class ensemble.base.**BaseAggregator**(*base_estimators, fitted_estimators=[], window_size=200, pre_fitted=False*)

Bases: abc.ABC

base_estimators()

estimators()

fit(*X, y=None*)

Parameters:
- **X** – numpy.ndarray of shape (n_samples, n_features). Input samples.
- **y** – numpy.ndarray of shape (n_samples) Labels for the target variable.

Returns:

fitted_estimators()

get_params(*deep=True*)

Get parameters for this estimator. See http://scikit-learn.org/stable/modules/generated/sklearn.base.BaseEstimator.html and sklearn/base.py for more information. Parameters ———- deep : boolean, optional

If True, will return the parameters for this estimator and contained subobjects that are estimators.

params : *mapping of string to any*

Parameter names mapped to their values.

abstract **partial_fit**(*X, y=None, classes=None*)

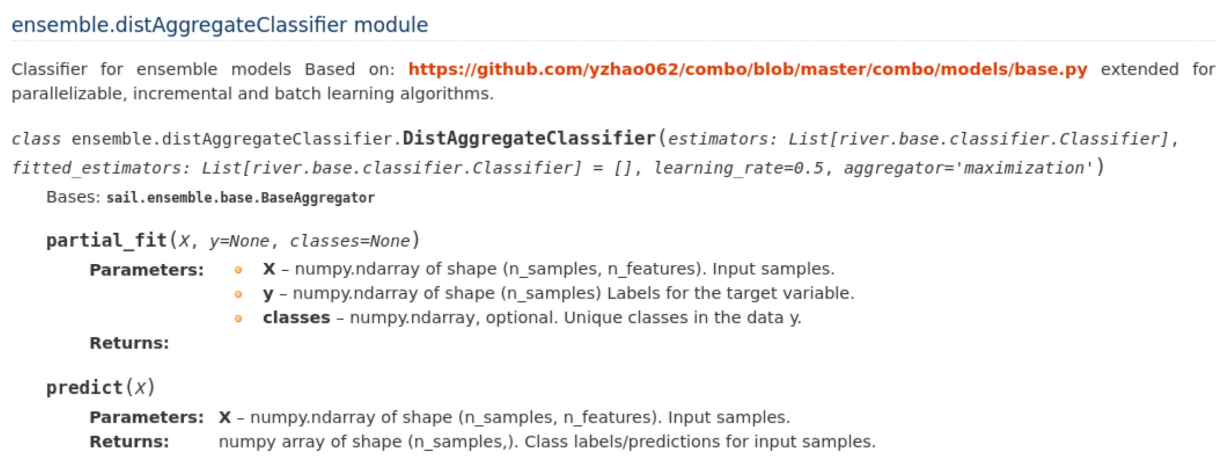
Parameters:
- **X** – numpy.ndarray of shape (n_samples, n_features). Input samples.
- **y** – numpy.ndarray of shape (n_samples) Labels for the target variable.
- **classes** – numpy.ndarray, optional. Unique classes in the data y.

Returns:

abstract **predict**(*X*)

Parameters: **X** – numpy.ndarray of shape (n_samples, n_features). Input samples.
Returns: numpy array of shape (n_samples,). Class labels/predictions for input samples.

Illustration 3: APIs for base class in ensemble models.



ensemble.distAggregateClassifier module

Classifier for ensemble models Based on: https://github.com/yzhao062/combo/blob/master/combo/models/base.py extended for parallelizable, incremental and batch learning algorithms.

```
class ensemble.distAggregateClassifier.DistAggregateClassifier(estimators: List[river.base.classifier.Classifier], fitted_estimators: List[river.base.classifier.Classifier] = [], learning_rate=0.5, aggregator='maximization')
```

Bases: `sail.ensemble.base.BaseAggregator`

`partial_fit(X, y=None, classes=None)`

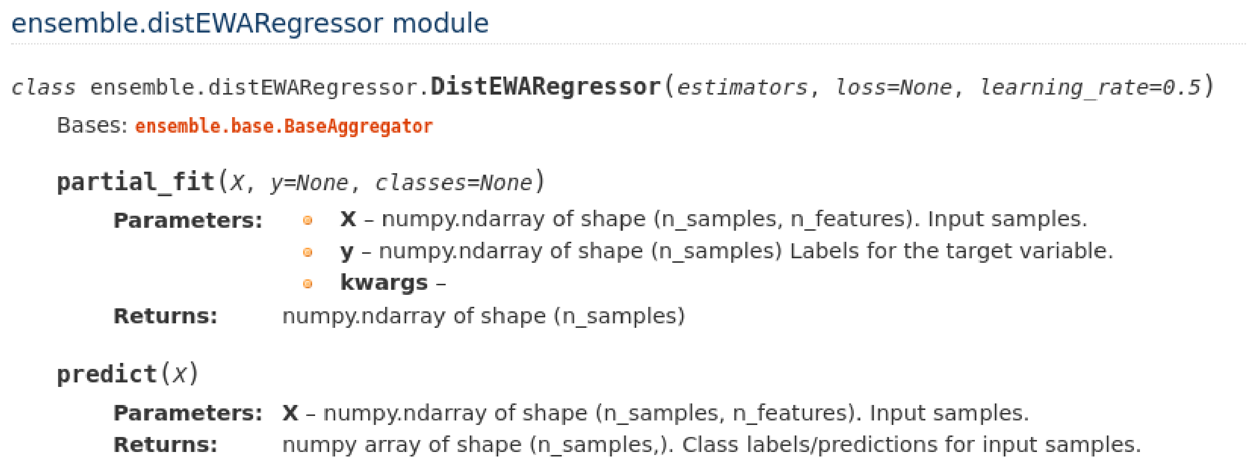
Parameters:
- **X** – numpy.ndarray of shape (n_samples, n_features). Input samples.
- **y** – numpy.ndarray of shape (n_samples) Labels for the target variable.
- **classes** – numpy.ndarray, optional. Unique classes in the data y.

Returns:

`predict(X)`

Parameters: **X** – numpy.ndarray of shape (n_samples, n_features). Input samples.
Returns: numpy array of shape (n_samples,). Class labels/predictions for input samples.

Illustration 4: APIs for distribued ensemble models for classification.



ensemble.distEWARegressor module

```
class ensemble.distEWARegressor.DistEWARegressor(estimators, loss=None, learning_rate=0.5)
```

Bases: `ensemble.base.BaseAggregator`

`partial_fit(X, y=None, classes=None)`

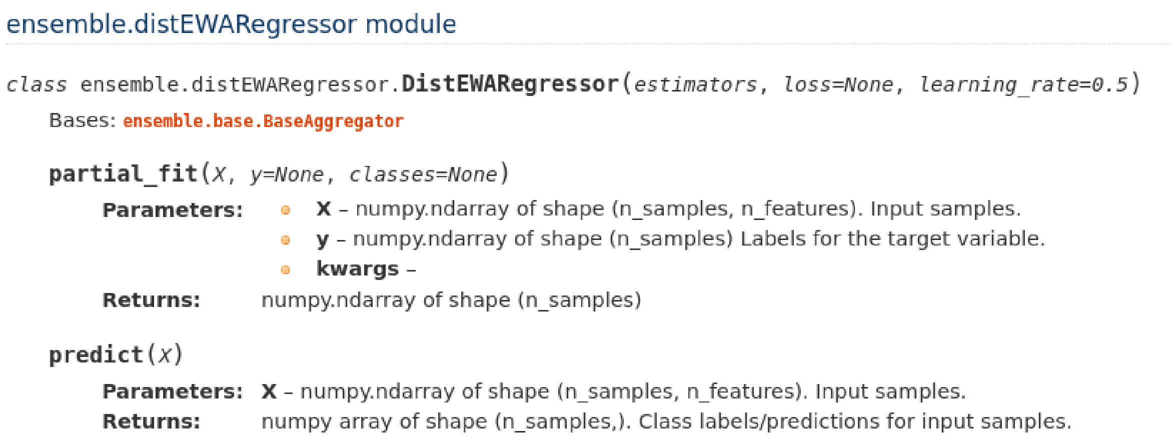
Parameters:
- **X** – numpy.ndarray of shape (n_samples, n_features). Input samples.
- **y** – numpy.ndarray of shape (n_samples) Labels for the target variable.
- **kwargs** –

Returns: numpy.ndarray of shape (n_samples)

`predict(X)`

Parameters: **X** – numpy.ndarray of shape (n_samples, n_features). Input samples.
Returns: numpy array of shape (n_samples,). Class labels/predictions for input samples.

Illustration 5: APIs for distribued ensemble models for regression.



ensemble.distEWARegressor module

class ensemble.distEWARegressor.**DistEWARegressor**(*estimators*, *loss=None*, *learning_rate=0.5*)

Bases: `ensemble.base.BaseAggregator`

partial_fit(*X*, *y=None*, *classes=None*)

Parameters:
- **X** – numpy.ndarray of shape (n_samples, n_features). Input samples.
- **y** – numpy.ndarray of shape (n_samples) Labels for the target variable.
- **kwargs** –

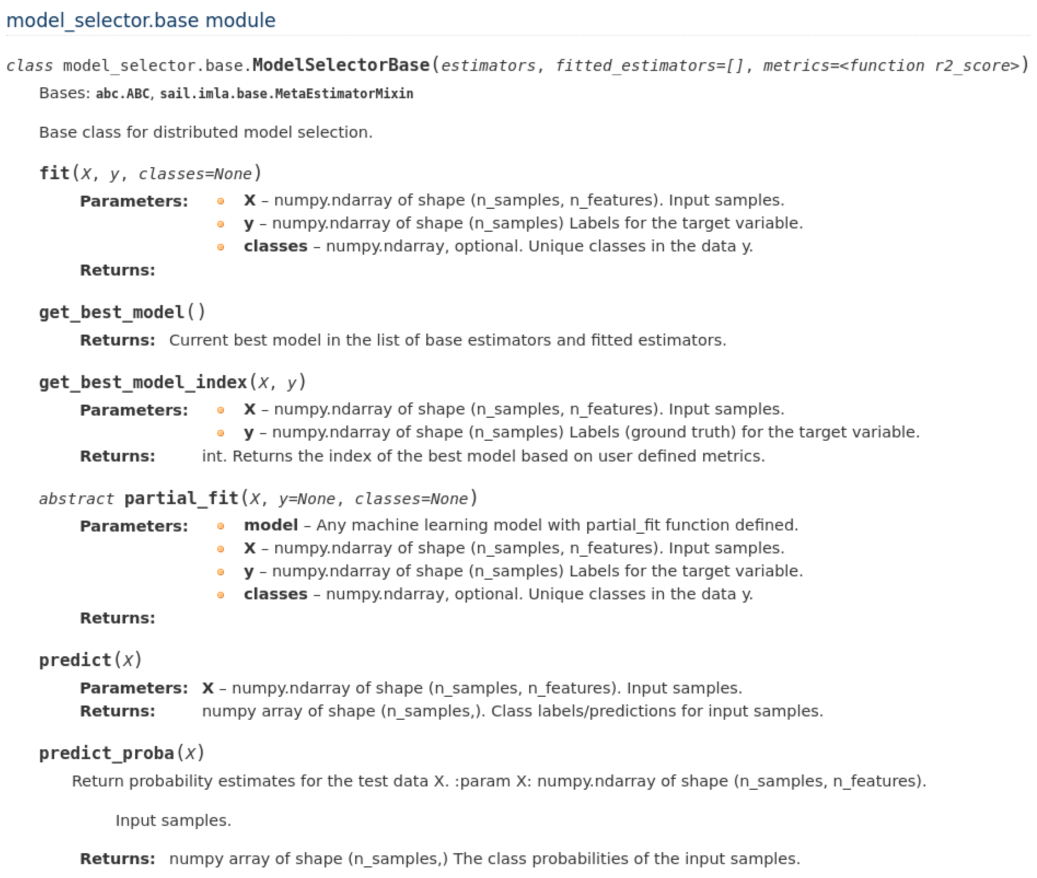
Returns: numpy.ndarray of shape (n_samples)

predict(*X*)

Parameters: **X** – numpy.ndarray of shape (n_samples, n_features). Input samples.

Returns: numpy array of shape (n_samples,). Class labels/predictions for input samples.

Illustration 6: APIs for distribued ensemble models for exponentially weighted average regressor.



model_selector.base module

class model_selector.base.**ModelSelectorBase**(*estimators*, *fitted_estimators=[]*, *metrics=<function r2_score>*)

Bases: `abc.ABC`, `sail.imla.base.MetaEstimatorMixin`

Base class for distributed model selection.

fit(*X*, *y*, *classes=None*)

Parameters:
- **X** – numpy.ndarray of shape (n_samples, n_features). Input samples.
- **y** – numpy.ndarray of shape (n_samples) Labels for the target variable.
- **classes** – numpy.ndarray, optional. Unique classes in the data y.

Returns:

get_best_model()

Returns: Current best model in the list of base estimators and fitted estimators.

get_best_model_index(*X*, *y*)

Parameters:
- **X** – numpy.ndarray of shape (n_samples, n_features). Input samples.
- **y** – numpy.ndarray of shape (n_samples) Labels (ground truth) for the target variable.

Returns: int. Returns the index of the best model based on user defined metrics.

abstract **partial_fit**(*X*, *y=None*, *classes=None*)

Parameters:
- **model** – Any machine learning model with partial_fit function defined.
- **X** – numpy.ndarray of shape (n_samples, n_features). Input samples.
- **y** – numpy.ndarray of shape (n_samples) Labels for the target variable.
- **classes** – numpy.ndarray, optional. Unique classes in the data y.

Returns:

predict(*X*)

Parameters: **X** – numpy.ndarray of shape (n_samples, n_features). Input samples.

Returns: numpy array of shape (n_samples,). Class labels/predictions for input samples.

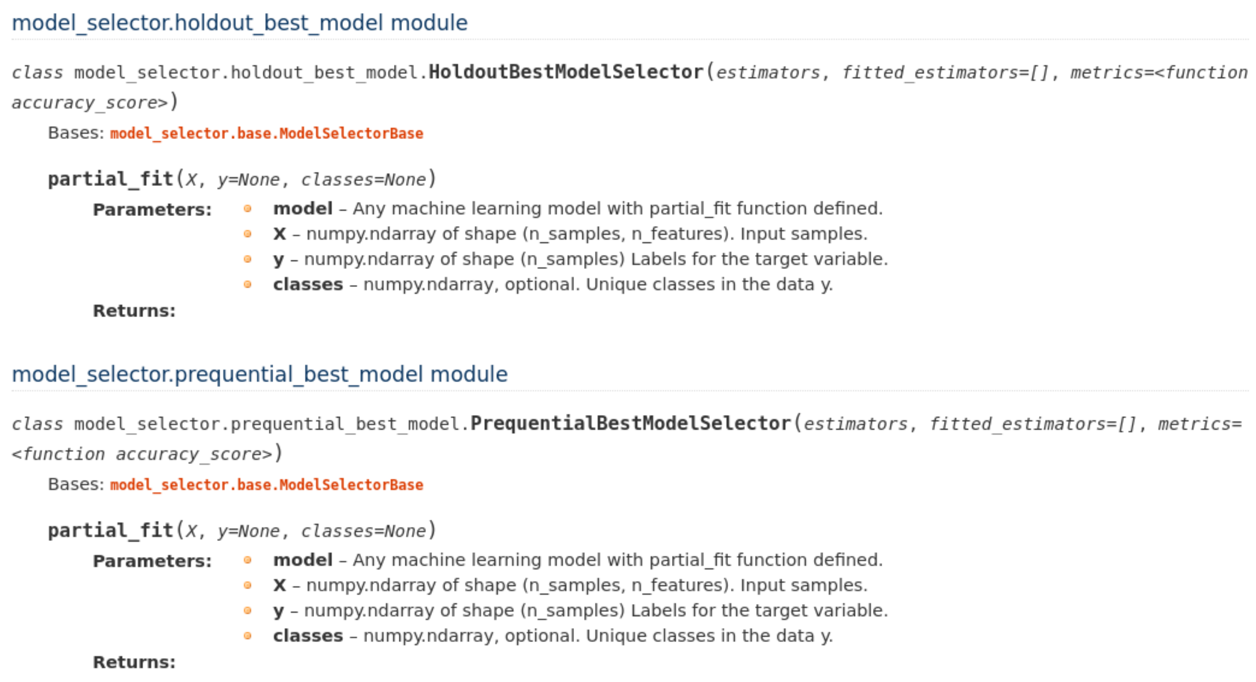
predict_proba(*X*)

Return probability estimates for the test data X. :param X: numpy.ndarray of shape (n_samples, n_features).

Input samples.

Returns: numpy array of shape (n_samples,) The class probabilities of the input samples.

Illustration 7: Base class for distributed model selection.



model_selector.holdout_best_model module

class model_selector.holdout_best_model.**HoldoutBestModelSelector**(*estimators*, *fitted_estimators=[]*, *metrics=<function accuracy_score>*)

Bases: `model_selector.base.ModelSelectorBase`

partial_fit(*X*, *y=None*, *classes=None*)

Parameters:
- **model** – Any machine learning model with partial_fit function defined.
- **X** – numpy.ndarray of shape (n_samples, n_features). Input samples.
- **y** – numpy.ndarray of shape (n_samples) Labels for the target variable.
- **classes** – numpy.ndarray, optional. Unique classes in the data y.

Returns:

model_selector.prequential_best_model module

class model_selector.prequential_best_model.**PrequentialBestModelSelector**(*estimators*, *fitted_estimators=[]*, *metrics=<function accuracy_score>*)

Bases: `model_selector.base.ModelSelectorBase`

partial_fit(*X*, *y=None*, *classes=None*)

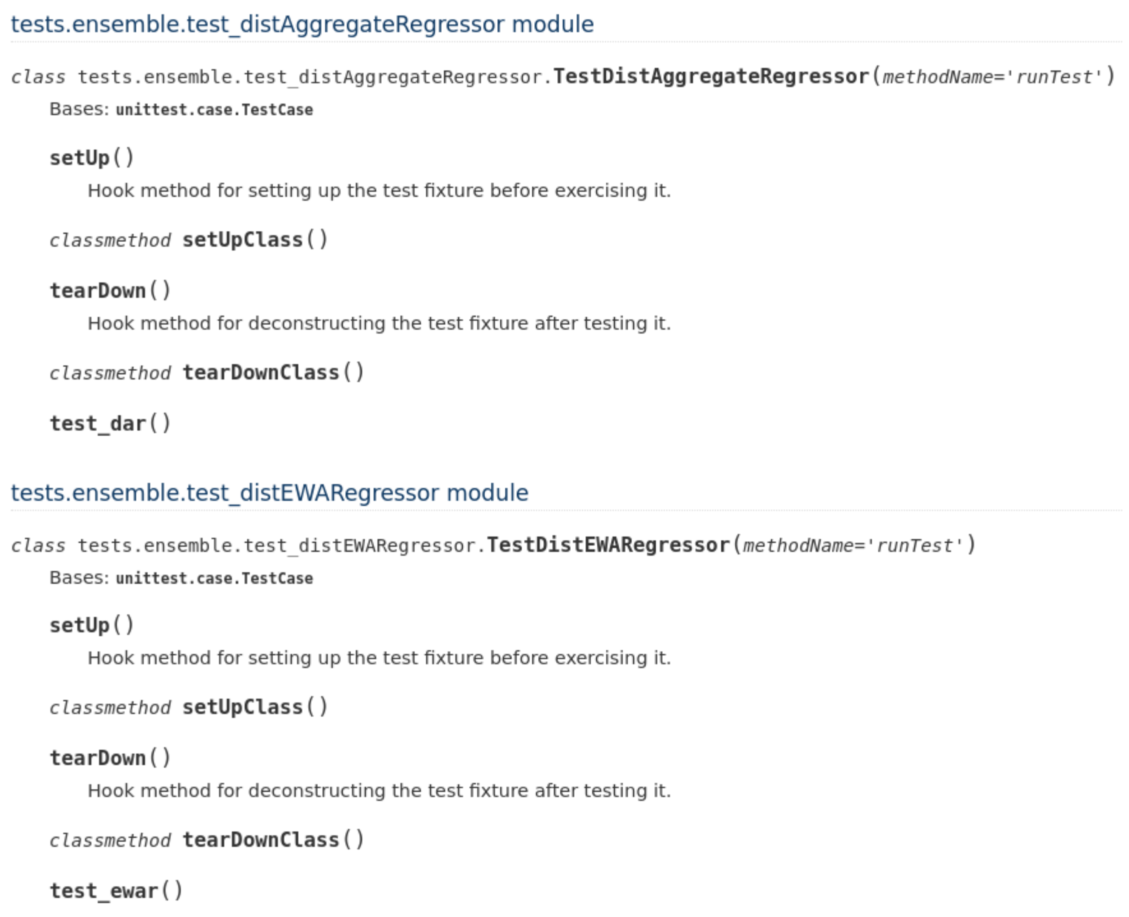
Parameters:
- **model** – Any machine learning model with partial_fit function defined.
- **X** – numpy.ndarray of shape (n_samples, n_features). Input samples.
- **y** – numpy.ndarray of shape (n_samples) Labels for the target variable.
- **classes** – numpy.ndarray, optional. Unique classes in the data y.

Returns:

Illustration 8: APIs for holdout and prequential based model selection.

5 Testing procedure and experiments

SAIL was tested using two methods. The first method involves standard software engineering practice to develop unit tests for the modules developed in SAIL. In particular, *unittest* framework was used for unit testing for the algorithms in SAIL (*Illlustration-9*). This ensures that the CI/CD principles can be safely followed as more algorithms are added to the library. Further, standard benchmark datasets have been used to better understand the performance of the models developed through SAIL. Two examples are described in the following subsections.


tests.ensemble.test_distAggregateRegressor module

```
class tests.ensemble.test_distAggregateRegressor.TestDistAggregateRegressor(methodName='runTest')
    Bases: unittest.case.TestCase

    setUp()
        Hook method for setting up the test fixture before exercising it.

    classmethod setUpClass()

    tearDown()
        Hook method for deconstructing the test fixture after testing it.

    classmethod tearDownClass()

    test_dar()
```

tests.ensemble.test_distEWARegressor module

```
class tests.ensemble.test_distEWARegressor.TestDistEWARegressor(methodName='runTest')
    Bases: unittest.case.TestCase

    setUp()
        Hook method for setting up the test fixture before exercising it.

    classmethod setUpClass()

    tearDown()
        Hook method for deconstructing the test fixture after testing it.

    classmethod tearDownClass()

    test_ewar()
```

Illustration 9: Unit tests for ensemble models.

5.1 Classification example

A synthetic example of streaming dataset was generated to test the performance of IMLA as compared to batch learning algorithms. The dataset is based on Agrawal [21] with 16,000 samples. There is abrupt drift obesrved in the data between 12,000 and 13,000 samples. Random Forest classifier (RFC) was taken as a benchmark for the batch learning algorithms. RFC was trained for the first 10,000 samples and scored on the remaining samples. Adaptive Random Forest (ARF) and Logistic Regression based on stochastic gradient descent were considered for the incremental models. The incremental models are scored and then trained for the whole 16,000 samples. As seen in *Illustration-10*, the performance of RFC degrades when concept drift occurs in the dataset. Incremental models, on the other hand, experience only a little drop in accuracy. This shows the importance of incremental algorithms for streaming data. It is important to note that all the three algorithms were used with the default options without hyperparameter tuning as the study was only to understand the robustness of the models to concept drift.

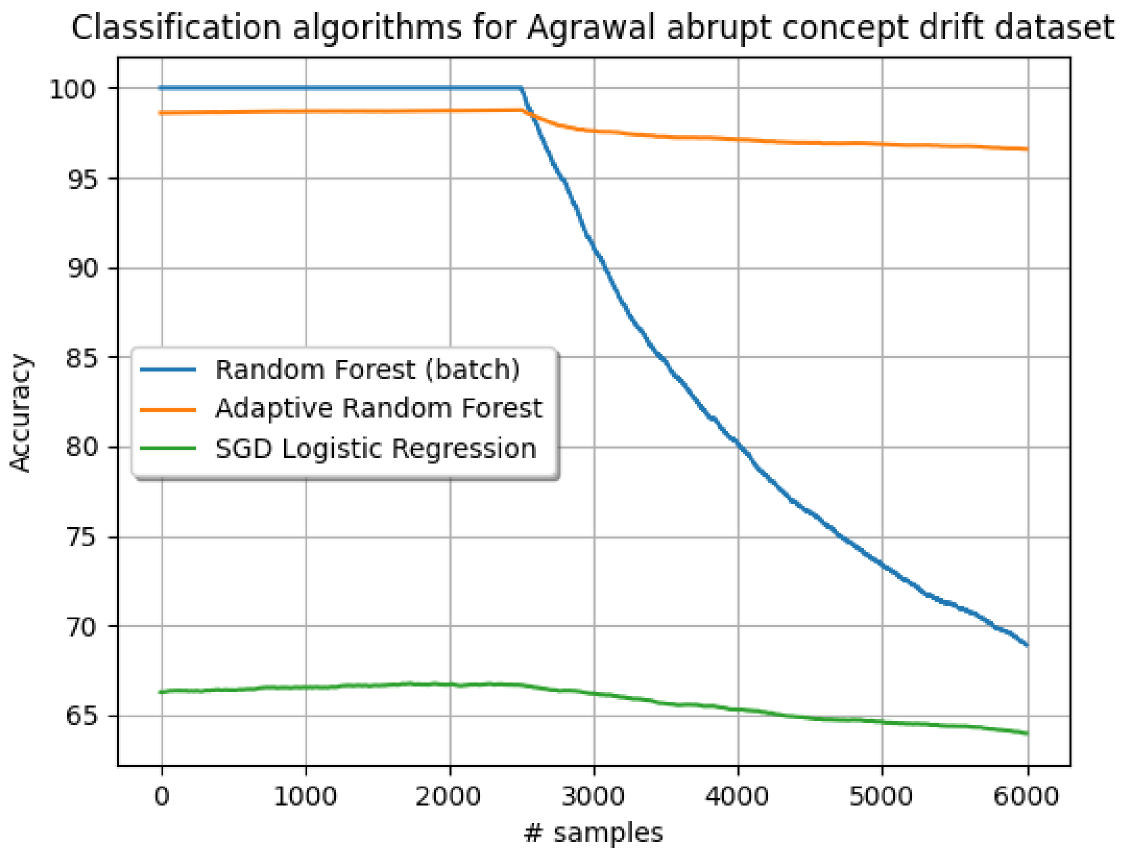
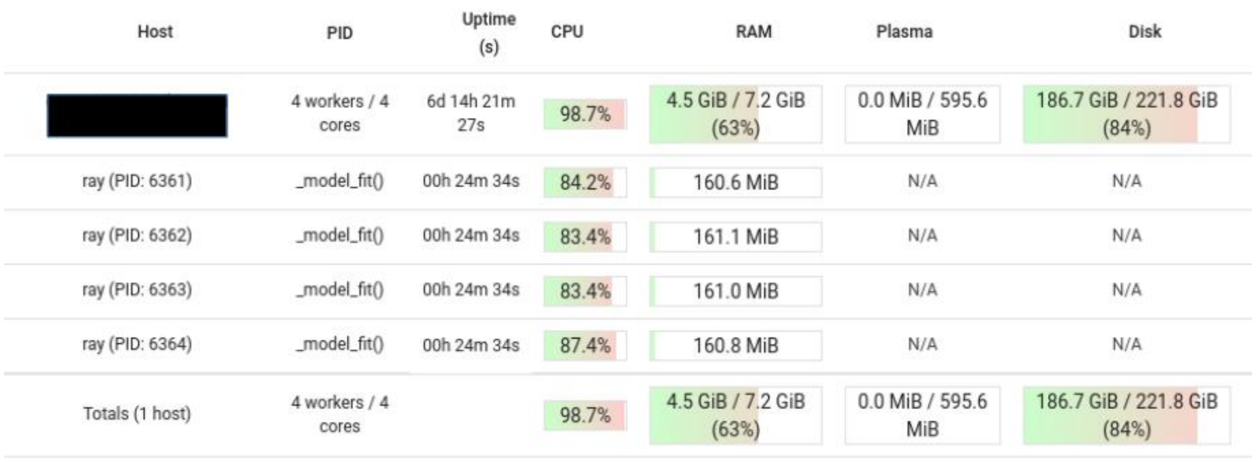


Illustration 10: Accuracy metric for batch and incremental models on Agrawal dataset.

5.2 Parallelization example

In this example, we consider the importance of Ray for parallel computations in model selection. We consider a simple linear regression dataset by considering a linear equation and randomly perturbing the target variable. Three sample sizes were considered in this example, namely, [1000, 2000, 4000]. Incremental models were run on this dataset for 10 partial-fit iterations and 20 partial-fit iterations using Ray in local mode and distributed mode respectively. Local mode does not have any communication costs and is equivalent to serial computation. Model selection was done with 40 linear regression models with different learning rates. The system was parallelized in Ray using 4 workers/cores (*Illustration-11*).

As can be seen in *Illustration-12,* as the sample size increases or the number of iterations increase, the computational time improves with parallelization through Ray. A realistic example, at least for offline training with incremental algorithms will have terabytes of data and the computational gain will be much more. Also, as the number of cores increases, there would be further gains in computational time.



Host	PID	Uptime (s)	CPU	RAM	Plasma	Disk
[redacted]	4 workers / 4 cores	6d 14h 21m 27s	98.7%	4.5 GiB / 7.2 GiB (63%)	0.0 MiB / 595.6 MiB	186.7 GiB / 221.8 GiB (84%)
ray (PID: 6361)	_model_fit()	00h 24m 34s	84.2%	160.6 MiB	N/A	N/A
ray (PID: 6362)	_model_fit()	00h 24m 34s	83.4%	161.1 MiB	N/A	N/A
ray (PID: 6363)	_model_fit()	00h 24m 34s	83.4%	161.0 MiB	N/A	N/A
ray (PID: 6364)	_model_fit()	00h 24m 34s	87.4%	160.8 MiB	N/A	N/A
Totals (1 host)	4 workers / 4 cores		98.7%	4.5 GiB / 7.2 GiB (63%)	0.0 MiB / 595.6 MiB	186.7 GiB / 221.8 GiB (84%)

Illustration 11: Distibution of incremental training tasks through Ray.

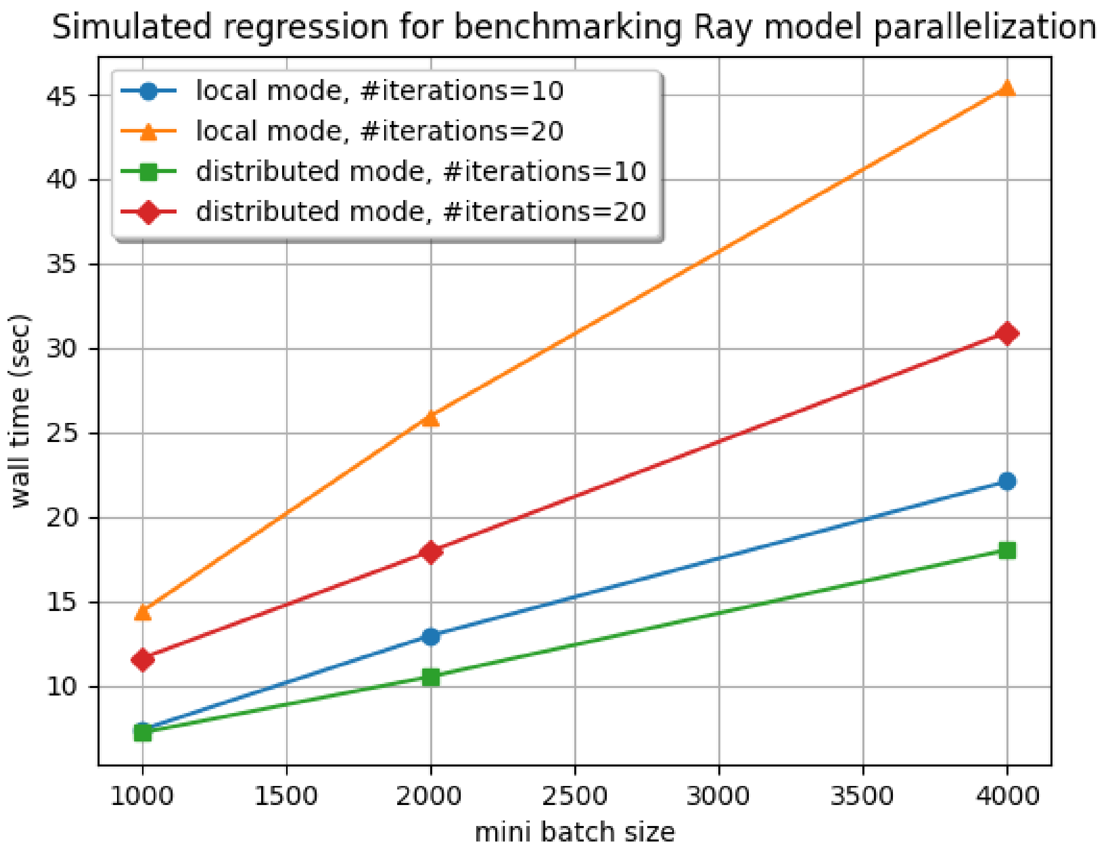


Illustration 12: Scalability analysis for model selection through Ray.

6 Conclusion and Future work

State of the art incremental machine learning algorithms with a common set of APIs were required to ensure easy testing, ensembling and creating AutoML pipelines. After reviewing, the existing libraries and features they provide, SAIL was developed with some local algorithms, wrappers for traditional online ML algorithms in River, and deep learning models in Pytorch and Keras to ensure a common set of APIs. Further, Ray was selected for distributed processing to work with a flexible distributed platform and hyperparameter engine. Preliminary results suggest that model based distributed computing would be a valuable tool for IMLA on streaming data.

Future work will focus on:

- Develop a stable release for continual-AI pipelines for time series data.
- Develop pipeline classes so that cleaner and less error-prone programmes can be written by the end user using SAIL.
- Work on Task 3.4 which starts at M12 to consider the applications of these machine learning models for the wind and solar data available through MORE.
- Look at working on applying IMLA models on summarized data available through ModelarDB (WP2).
- Research on using the motif based models of WP4 for continual-AI algorithms.
- Hyperparameter tuning through Ray for incremental models.
- APIs and prototypes to showcase the use-cases for MORE.

7 References

[1] Losing, Viktor, Barbara Hammer, and Heiko Wersing. "Incremental on-line learning: A review and comparison of state of the art algorithms." Neurocomputing 275 (2018): 1261-1274.

[2] https://github.com/online-ml/river

[3] https://scikit-learn.org/stable/

[4] https://pytorch.org/

[5] https://keras.io/

[6] https://github.com/alan-turing-institute/sktime

[7] https://github.com/sktime/sktime-dl

[8] https://scikit-multiflow.github.io/

[9] https://pypi.org/project/creme/

[10] https://spark.apache.org/mllib/

[11] https://dask.org/

[12] https://modin.readthedocs.io/en/latest/

[13] https://vaex.io/docs/index.html

[14] https://rapids.ai/

[15] https://spark.apache.org/

[16] https://flink.apache.org/

[17] https://www.ray.io/

[18] https://github.com/skorch-dev/skorch

[19] https://github.com/ContinualAI/avalanche

[20] https://github.com/imirzadeh/CL-Gym

[21] Agrawal, Rakesh, Tomasz Imielinski, and Arun Swami. "Database mining: A performance perspective. "*IEEE transactions on knowledge and data engineering* 5.6 (1993): 914-925.