Adaptive Extreme Gradient Boost Regressor for Evolving Data Streams - Demo

Hélder Vieira[1], Samuel Aduroja[2], and Vânia Guimarães[3]

[1] Faculty of Science, University of Porto, Portugal
up201503395@edu.fc.up.pt
[2] Faculty of Science, University of Porto, Portugal
up202009191@edu.fc.up.pt
[3] Faculty of Science, University of Porto, Portugal
up200505287@edu.fc.up.pt

1 Introduction

Ensemble methods are very popular in several domains. Within this category, one of the main machine learning competitions winners is eXtreme Gradient Boosting (XGB). It has proven to be a good choice in terms of performance and speed. In context of a data stream, resources, such as running time and space memory, are limited, however, they are fundamental due to the large volume of data generated and the speed of its production. The power of XGB lies on its accuracy with introduction of a regularization term, its efficient memory usage and its scalability. These features have aroused the interest of researchers who seek to adapt the method to deal with stream data. Adaptive XGBoost for Evolving Data Streams was proposed by Jacob Montiel et al. [1]. The model described is focused on binary classification. Although, its structure may adapt to regression problems. Our contribution consists of implementing this proposed method for regression data in python language using the xgboost library and the scikit-multiflow data stream software suite. This work is organized as follows: The next section after we present a brief theoretical background, we will review the drift detector Page-Hinkley test and Extreme Gradient Boosting algorithm. Then, we will explain the Adaptive Extreme Gradient Boosting suitable for data streams. In Section 4, a description of our algorithm is presented. A demo and its results are shown in Section 5. Finally we conclude the paper in Section 6.

2 Theoretical framework

In a data stream, the concept can change over time. If the probability distribution of data is stable, the model should be learn from past data to make accuracy predictions. However, in case of concept drift, the model should adapt to new concept as soon as possible. There are several ways to detect and handle with concept drift. Incremental learning is one of these strategies. As new data arrives, the model is retrained and updated. As a result, the model adapts to the most recent structure of data. However, the process can be too slow in case

of fast drifts. Another approach is use a detector algorithm. When a new concept is detected, an updated strategy is activated in order to quickly learn new distribution. Predictive method has possibility to use Page-Hinkley test.

2.1 Drift Detector

Page-Hinkley (PH) test [4] was used to monitor the evolution of loss function. At each time stamp, we compare the error with the mean absolute error. If the difference between two metrics is significantly high, a change alert is triggered. The basic idea of PH is compute the cumulative sum of differences of the observed values and their mean until the present moment and signals the presence of a drift whenever $m_T - m_{min}$ goes above a predefined threshold (λ). The method also allows a variance around the mean which is represented by the parameter α:

$$m_t = \sum_{t=1}^{T} (x_t - \overline{x}_T + \alpha) \tag{1}$$

where $\overline{x}_T = 1/T \sum_{t=1}^{T} x_t$ and $t = 1, ..., T$

$$m_{min} = min(m_t) \tag{2}$$

The user has to defined two parameters, α which represents the magnitude of accepted change, and λ that expresses tolerable false alarm rate. With a low threshold, almost all change is considered as concept drift, but the algorithm may incur in false alarms. Whereas, increasing the threshold would lead PH test to miss or delay some changes.

2.2 XGBoost

Extreme Gradient Boosting (XGBoost) is a popular implementation of a decision tree ensemble based on gradient boosting. The algorithm was designed to outperform several methods in a fast and scalable manner. Given a training dataset $D = \{x_i, y_i\}_1^N$, the goal of a machine learning problem is to find a model $F(x)$ to make predictions $\hat{y}_i$ more accurate as possible, considering a set of features $\overrightarrow{x}_i$. Similarly to Gradient Boosting, XGBoost combines the contributions of many weak learners, during the gradient boosting process, to produce a strong learner. The basic idea is, at each iteration k train a decision tree f_k based on the residuals of the previous model and add it to the ensemble. Each new tree improves the model most by minimizing the following objective function:

$$L(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k^K \Omega(f_k) \tag{3}$$

where $l(\hat{y}_i, y_i)$ is the differentiable loss function for each individual prediction, that means the difference between predicted value, $\hat{y}_i$ and the target value, y_i, which controls the predictive power; $\Omega(f) = \gamma T + \frac{1}{2}\lambda||w||^2$ is the regularization

term which controls the complexity of trees preventing the chance of overfitting. T represents the total number of leaves in the tree, therefore a higher value of γ produces a simpler tree. Influencing score of each leaf w is the shrinkage parameter λ, which reduces the step size in the additive expansion.

After K iterations, the ensemble model uses a weighted sum of all trees to make the final prediction:

$$\hat{y}_i = \sum_{k=1}^{K} \eta . f_k(x_i) \tag{4}$$

where η is the contribution of the new base learner on the prediction. It also has a role in prevention overfitting. The benefits of tree complexity reduction obtained by mentioned shrinkage methods and others, such as maximum depth of trees, are beyond avoiding overfit. It contributes to train the model faster and use less storage space.

3 Adaptation XGB for Stream Learning

3.1 Adaptive Extreme Gradient Boosting

XGB was designed to train a model over all collected data. The statistical properties of the data remain the same in training and test set. Inversely, in stream scenario, data is continuously arriving to the system and its distribution usually change and evolve over time. To handle with these new challenges, Adaptive Extreme Gradient Boosting was proposed. Regardless of how data arrives, W number of instances are stored in a buffer $w = (\overrightarrow{x}_i, y_i)$, where $i \in \{1, 2, ..., W\}$. In the first iteration, a single weak learner f_1 is trained over this chunk of examples. In the next iterations, the model will sequentially add new single learners to the ensemble. When $k > 1$, the w_k of data samples will pass through the ensemble model to compute the residuals and built a new tree trained over these same errors. As theoretically, we not stop receive stream data, the model could grow indefinitely. For that reason, and essentially to have an updated model based on more recent data, the maximum number of base learners is predefined. There are two strategies to replace old base functions, when the ensemble model is full:

- A push strategy ($AXGB_p$) reminds a queue formed by multiple learners, as shown in figure 1. When the maximum number of single models is reached, the oldest base function is pushed out of the ensemble in order to append the new member.
- By replace strategy ($AXGB_r$), each time the ensemble model is full, a completely new model is created from scratch. The new weak learner is trained using only the samples stored in the current buffer and it goes to the position of the oldest one. In the next iteration, the new booster is trained based on the prediction errors generated by the previous model. The process is continually repeated. Figure 2 depicts all the sequence (green colored circles represent the model used to train the new booster).

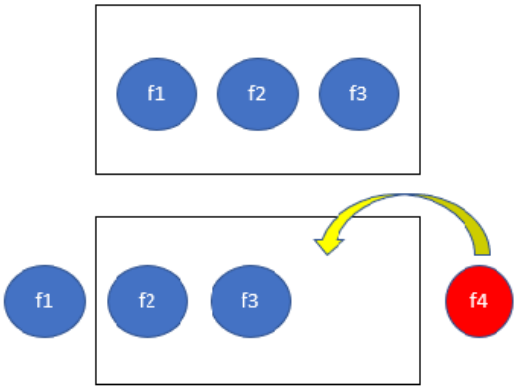

Fig. 1. Push strategy ($AXGB_p$)

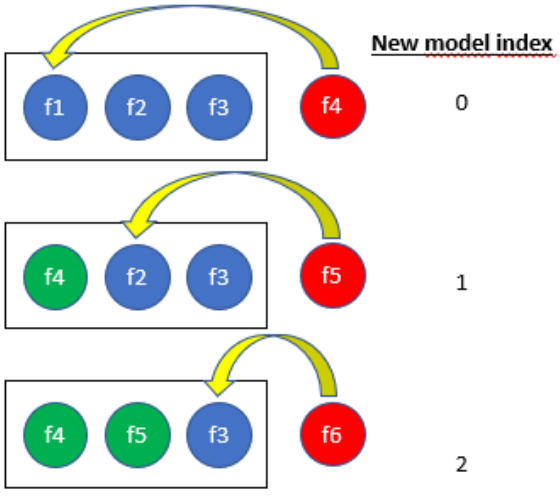

Fig. 2. Replace strategy ($AXGB_r$)

The full ensemble model is only achieved after K iterations. As a result, the predictions made by uncompleted model will be less accurate. Although, stream models should be ready to predict at any point in time. In order to accelerate the process of creating the full ensemble with $K.W$ samples, AXGB has the option of doubling in each iteration the number of examples contained in the buffer of a minimum size W_{min} until reaching the maximum buffer size W_{max}, that is initially established. The window size, W_i, for the ith iteration is defined as follow:

$$W_i = min(W_{min} \times 2^i, W_{max}) \tag{5}$$

By the dynamic window mechanism, we do not have to wait for W_{max} samples to build a base function f_i, a smaller number of examples, $W_{min} \times 2^i$, are enough. After having all the estimators, with both ensemble update strategies, older learners trained on fewer examples, are replaced by newest one, trained on more samples.

3.2 Track Concept Drift

As mentioned before, the strategy of update ensemble allows model to self-adapt to changing data. This is a passive approach, since the method does not have an immediate reaction to concept drifts. In most cases, it may take long time to adapt to a new concept. Page-Hinkley test can be used to overcome this problem. When drift detector warns us that there is a change in concept, the buffer size is back to W_{min} samples. In the next iterations, window size will follow the process of dynamic window. The learning and update phase will depend on the ensemble strategy:

- By push strategy, new single model will be trained on small window and added to the ensemble, while outdated members, the oldest one, is removed. At the beginning, the window size grows exponentially, therefore a completely new full ensemble model is created quickly.
- By replacement strategy, the method follows the same process as the first time. A new base function is trained on the most recent data and the errors of prediction of oldest learners will not be used on the next estimators. Only the residuals of the model built with new concept are considered.

4 Implementation and user guide

This project is an implementation of XGBoost regressor for evolving datastreams. It was inspired in the classifier implementation by J. Montiel et al. (2020) [1]. The implementation is on python 3 and built on top of scikit-multiflow. The open source code is available on github: https://github.com/hmcvieira/Data-Stream-Mining/tree/main/HW1/AXGBregression. The following files are available:

– *axgb_regression.py* contains the script that implements the regression algorithm
– *axgb_regression_test.py* contains an example execution script
– A README file that briefly describes the file structure

Adaptive Extreme Gradient Boosting algorithm can be used in a variety of regression problems. The implementation of AXGB was made using *scikit* − *multiflow* [3] and *XGBoost* [2] packages, data streams softwares written in Python. For this purpose, we create the module *axgb_regressor* with a class *AdaptiveXGBoostRegressor*. The available hyper-parameters are:

– **n_estimators**: int (default=5)
  The number of estimators in the ensemble.
– **learning_rate**: float (default=0.3)
  Learning rate, a.k.a eta.
– **max_depth**: int (default = 6)
  Max tree depth.
– **max_window_size**: int (default=1000)
  Max window size.
– **min_window_size**: int (default=None)
  Minimum window size. If this parameters is not set, then a fixed size window of size "max_window_size" will be used.
– **update_strategy**: str (default="replace")
  'push' - the ensemble resembles a queue
  'replace' - oldest ensemble members are replaced by newer ones
– **detect_drift**: bool (default=False)
  If set will use a drift detector (Page-Hinkley).
– **threshold**: int (default=200)
  The change is detected if the difference is greater than a threshold (lambda)
– **min_instances**: int (default=200)
  Minimum number of instances before detecting change.
– **delta**: float (default=0.5)
  Magnitude of changes that are allowed

Despite all hyper-parameters have default values, the user should adapt them to the specific problem. *AdaptiveXGBoostRegressor* class has *partial_fit*, *_partial_fit*, *_train_on_mini_batch*, *_train_booster* and *_update_model_idx* functions which transform data in numpy array. Then, the elements are stored in buffers of window size, one with features and another with target. When a buffer is full a new estimator will be trained over this data or residuals of ensemble model depending on the situation. The structure of the ensemble model results from the chosen strategy. If a Page-Hinkley test warns that there is a concept change, the size of buffer will be minimum and for that we use *_reset_window_size* function. Then, the buffer size will follow the dynamic window size process by *_adjust_window_size* function. All code can be found in Appendix A.

5 Demonstration and Results

The implementation of our algorithm was tested in an artificial dataset. We used *RegressionGenerator* method available on *skmultiflow.data* module to generate three random datasets with 15 features. The first one has 2,000 samples and 8 informative features, the second one has 3,000 examples and 10 informative features, and the last one has 5,000 samples and 12 informative features. All the elements together create our dataset with a concept drift in 2,000 sample and another in the position 5,000.

The parameters of our model are the same for all situations, whether we use push or replacement strategy, or even with or without drift detector for a fair comparison.

```
n_estimators = 20          # Number of members in the ensemble
learning_rate = 0.3        # Learning rate or eta
max_depth = 6              # Max depth for each tree
max_window_size = 100      # Max window size
min_window_size = 10       # set to activate dynamic window
detect_drift = True        # Enable/disable drift detection
threshold = 5100           # Page-Hinkley threshold
```

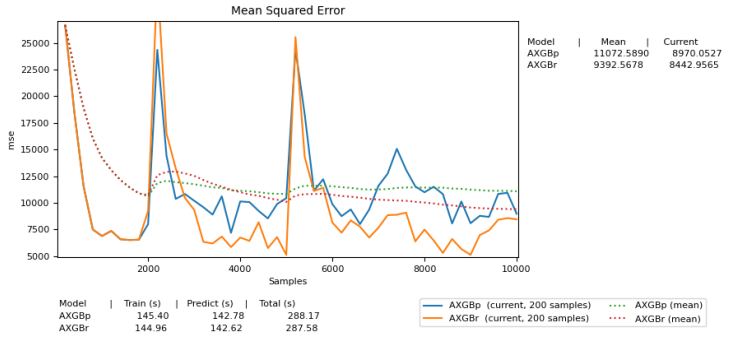

Fig. 3. MSE with Page-Hinkley test

Figure 3 depicts the evolution of mean squared error for both push and replace strategies, with drift detector. As we can see, initially they have same results since they add new trees in a similar way until reach full ensemble, at position 1800. Then, the loss of both models increase, but $AXGB_p$ has better predictions because of contribution of previous learners. Unlike the predictions of $AXGB_r$ that are based only on the most recent data. PageHinkley detector triggers the warning after seeing 2010 samples. Both models have a poor performance due to concept drift. However, $AXGB_p$ predicts a little better than $AXGB_r$ because diversity of ensembles can improve the generalization of models, while $AXGB_r$ has to make a first "guess" with only a small amount of data.

Around 3,850 samples, ensemble model is full and it is updated, increasing mean squared error as a result. In that case, the strategy that benefited from this data distribution was replacement with lower residuals. Again, a new concept occurs at position 5,000. The drift detector in the model with replace strategy detected the change exactly when it happened. Page-Hinkley test of $AXGB_p$, with same hyper-parameters, gave the warning 100 samples later. Both models had a bad and similar performance when a new concept occurred. Despite $AXGB_p$ model has members trained on old concept, dynamic window size allows the model in few iterations to remove older members and create new ones base on new concept. Then, without concept drift, we see some variations of MSE metric, due, essentially, to adaptation strategy, having a better performance on $AXGB_r$. In the long run, they seem to converge. Considering the running time, $AXGB_p$ is slightly slower in training because it uses more ensemble members to train new learners.

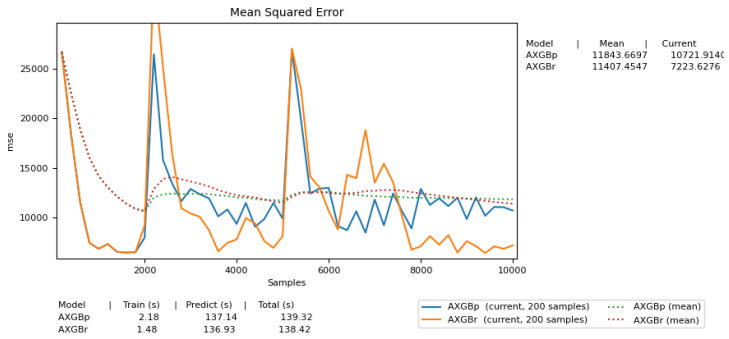

Fig. 4. MSE without drift detector

In the following demonstration, our model will learn on the same dataset, but without drift detector. Figure 4 shows the mean square error for each 200 examples. We can see the evolution of this metric for push and replace strategies. At the beginning, both models have the same evolution as in previous situation until suffer the effect of concept drift. At this point the behaviour is very similar. The residuals of both models increased significantly but $AXGB_p$ performs slightly better than $AXGB_r$. In this case, base learners created over old concept, more specifically based on data that arrived before position 1,800, were important to reduce the error. However, after 2,800 samples, $AXGB_r$ outperforms $AXGB_p$. The ensemble model of $AXGB_r$ is full again after seeing sample 3,600. Initially, the model does not benefit from the update strategy, since its accuracy decreases expressively, however it recovers quickly. At position 5,000 a new concept occurs. Both models had similar predictions. This is explained by the fact that $AXGB_r$ model is trained on almost same samples than $AXGB_p$. Around 5,400 samples $AXGB_r$ model is full again, but it benefited from not having base functions

trained on old concept and the accuracy improved. Although, with more samples, we see that few members of the ensemble model were not able to extract the structure of data and the errors increased quickly. On the other hand, with push strategy, the model the accuracy decreased but with a lower rate. Then, when $AXGB_r$ model applied the update strategy, at around position 7,200 and 9,000 the MSE increased, however, in few steps, the model was able to understand the data and get good predictions. $AXGB_p$ is more stable since the update strategy does not change once the model is full. After 10,000 samples, in this particular example, the mean of MSE of both models is very close, even so the model with the replace strategy shows better results. When we analyse the velocity, once again $AXGB_r$ is slightly faster because, in general, it creates a new model, from time to time, using less weak learners to make predictions and to train the new one.

At the end of this experiment, mean of MSE was very similar for the 3 models (AXGBr, AXGBp and ARF) in each experimental setup (with and without drift detector). Although in some cases a model slightly outperforms other, this difference is potentially negligible in the context of the problem. However, accuracy is not the only important factor in stream learning. Run time may compromise the viability of the stream application. In this particular experiment, when we use methods with drift detector, the total time spent in train and predict processes more than doubled, compared to models that only use incremental learning to adapt to dynamic data. The choice of method will depend on the balance we can find between accuracy and resources, such as time and memory. Even with enough resources to allow us to take advantage of the Page-Hinkley test, tuning its hyper-parameters is not an easy task. The consequence of the adaptation of model to false alarms is reduction of accuracy. In general, choosing appropriate hyper-parameters plays a crucial role in the success of model. Hyper-parameters setting like number of estimators, window size and learning rate, have a significant impact on the performance of the model is being trained. Hence, we can't get conclusions of the best approach with this example. All strategies are interesting for some applications with adequate hyper-parameters.

5.1 Benchmark comparison

The implementation of new algorithm in a dataset does not guarantee that all methods are working as intended. In order to minimize this question we will evaluate the performance of AXGB against other ensemble model that uses decision trees as weak learners, the Adaptive Random Forest algorithm. From $skmultiflow.meta$ module we import $AdaptiveRandomForestRegressor$ class. Figure 5 depicts the comparison of the different versions of AXGB without drift detector against ARF with also 20 estimators. The green line, which represents the evolution of mean square error of Adaptive Random Forest is very close the others metrics, showing similarity in behaviour of all three methods. Another proof of similarity is mean MSE. Adaptive Random Forest was the model with lower error (10,889), although the value is close to mean MSE of $AXGBr$ and $AXGBp$ (11,407 and 11,844 respectively). If we compared with current residuals,

ARF had the highest error, 10,802, against 7,224 from $AXGBr$ which was the best result. In terms of running time, Adaptive Random Forest took almost twice as long to train the model and to predict in order to get almost the same accuracy than AXGB models. This experience shows that our implemented algorithms can outperform ARF, having good accuracy in a short period of time.

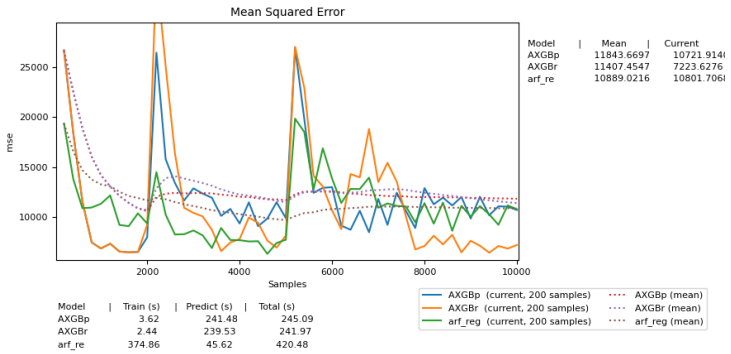

Fig. 5. AXGB vs Random Forest without drift detector

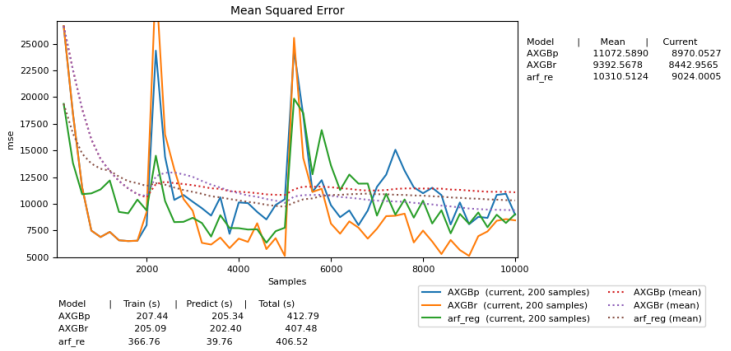

Fig. 6. AXGB vs Random Forest with drift detector

The comparison between models with drift detector is represented in figure 6. Visually, ARF does not appear to have major discrepancies from the other models. In terms of mean MSE, the model with higher accuracy is AXGBr, the second one is ARF, and finally we have AXGBp. Analyzing the MSE at position 10,000, the best model was AXGBr and the worst was ARF. However, all three values were around 9,000. Taking into account the speed to train and to predict, the differences are almost negligible, but ARF did the execution in less time. As

we expected, the results of this analysis support the quality and feasibility of our algorithm implementation.

6 Conclusion

Our project consisted in the implementation of a version of Extreme Gradient Boosting for data streams, the Adaptive Extreme Gradient Boosting for regression problems. Setting hyper-parameter *update_strategy*, the user can select push or replacement strategy. *detect_drift* parameter allows the user to opt for a model with or without drift detector. The required packages are scikit-multiflow and xgboost. For the demo, we used an artificial dataset generated by *RegressionGenerator*, a method available on scikit-multiflow. The performance of a model in each phase give us the perception of the impact of each approach in different situations. However, general conclusions can not be drawn about what model has the best overall performance.

Appendixes

A Code

```
import numpy as np
import xgboost as xgb
from skmultiflow.drift_detection import PageHinkley
from skmultiflow.utils import get_dimensions

class AdaptiveXGBoostRegressor():
    _PUSH_STRATEGY = 'push'
    _REPLACE_STRATEGY = 'replace'
    _UPDATE_STRATEGIES = [_PUSH_STRATEGY, _REPLACE_STRATEGY]

    def __init__(self,
                 n_estimators=30,
                 learning_rate=1,
                 max_depth=6,
                 max_window_size=1000,
                 min_window_size=None,
                 detect_drift=False,
                 update_strategy='replace',
                 threshold = 100):

        super().__init__()
        self.learning_rate = learning_rate
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.max_window_size = max_window_size
        self.min_window_size = min_window_size
```

```
        self._first_run = True
        self._ensemble = None
        self.detect_drift = detect_drift
        self._drift_detector = None
        self._X_buffer = np.array([])
        self._y_buffer = np.array([])
        self.threshold = threshold
        self._model_idx = 0
        if update_strategy not in self._UPDATE_STRATEGIES:
            raise AttributeError("Invalid update_strategy:
    {}\n"
                                 "Valid options: {}".format(
    update_strategy,

    self._UPDATE_STRATEGIES))
        self.update_strategy = update_strategy
        self._configure()

    def _configure(self):
        if self.update_strategy == self._PUSH_STRATEGY:
            self._ensemble = []
        elif self.update_strategy == self._REPLACE_STRATEGY:
            self._ensemble = [None] * self.n_estimators
        self._reset_window_size()
        self._init_margin = 0.0
        self._boosting_params = {"verbosity": 0,
                                 "objective": "reg:
    squarederror",
                                 "eta": self.learning_rate,
                                 "max_depth": self.max_depth}
        if self.detect_drift:
            self._drift_detector = PageHinkley( threshold=
    self.threshold,min_instances = 200, alpha = 0.5)


    def partial_fit(self, X, y, classes=None, sample_weight=
    None):
        """
        Partially (incrementally) fit the model.
        Parameters
        ----------
        X: numpy.ndarray
            An array of shape (n_samples, n_features) with
    the data upon which
            the algorithm will create its model.
        y: Array-like
            An array of shape (, n_samples) containing the
    classification
            targets for all samples in X. Only binary data is
     supported.
```

```
        classes: Not used.
        sample_weight: Not used.
        Returns
        -------
        AdaptiveXGBoostClassifier
            self
        """
        for i in range(X.shape[0]):
            self._partial_fit(np.array([X[i, :]]), np.array([
    y[i]]))
        return self

    def _partial_fit(self, X, y):
        if self._first_run:
            self._X_buffer = np.array([]).reshape(0,
    get_dimensions(X)[1])
            self._y_buffer = np.array([])
            self._first_run = False
        self._X_buffer = np.concatenate((self._X_buffer, X))
        self._y_buffer = np.concatenate((self._y_buffer, y))
        while self._X_buffer.shape[0] >= self.window_size:
            self._train_on_mini_batch(X=self._X_buffer[0:self
    .window_size, :],
                                      y=self._y_buffer[0:self
    .window_size])
            delete_idx = [i for i in range(self.window_size)]
            self._X_buffer = np.delete(self._X_buffer,
    delete_idx, axis=0)
            self._y_buffer = np.delete(self._y_buffer,
    delete_idx, axis=0)

            # Check window size and adjust it if necessary
            self._adjust_window_size()

        # Support for concept drift
        if self.detect_drift:
            error = abs (self.predict(X)-y)
            # Check for warning
            self._drift_detector.add_element(error)
            # Check if there was a change
            if self._drift_detector.detected_change():
                # Reset window size
                self._reset_window_size()
                if self.update_strategy == self.
    _REPLACE_STRATEGY:
                    self._model_idx = 0

    def _adjust_window_size(self):
        if self.window_size * 2 < self.max_window_size:
            self.window_size *= 2
```

```
        else:
            self.window_size = self.max_window_size

    def _reset_window_size(self):
        if self.min_window_size:
            self.window_size = self.min_window_size
        else:
            self.window_size = self.max_window_size

    def _train_on_mini_batch(self, X, y):
        if self.update_strategy == self._REPLACE_STRATEGY:
            booster = self._train_booster(X, y, self._model_idx)
            # Update ensemble
            self._ensemble[self._model_idx] = booster
            self._update_model_idx()
        else:
            booster = self._train_booster(X, y, len(self._ensemble))
            # Update ensemble
            if len(self._ensemble) == self.n_estimators:
                self._ensemble.pop(0)
            self._ensemble.append(booster)

    def _train_booster(self, X: np.ndarray, y: np.ndarray, last_model_idx: int):
        d_mini_batch_train = xgb.DMatrix(X, y)
        # Get margins from trees in the ensemble
        margins = np.asarray([self._init_margin] * d_mini_batch_train.num_row())
        for j in range(last_model_idx):
            margins = np.add(margins,
                             self._ensemble[j].predict(d_mini_batch_train, output_margin=True))
        d_mini_batch_train.set_base_margin(margin=margins)
        booster = xgb.train(params=self._boosting_params,
                            dtrain=d_mini_batch_train,
                            num_boost_round=1,
                            verbose_eval=False)
        return booster

    def _update_model_idx(self):
        self._model_idx += 1
        if self._model_idx == self.n_estimators:
            self._model_idx = 0

    def predict(self, X):
        """
        Predict the class label for sample X
        Parameters
```

```
        ----------
        X: numpy.ndarray
            An array of shape (n_samples, n_features) with the samples to
            predict the class label for.
        Returns
        -------
        numpy.ndarray
            A 1D array of shape (, n_samples), containing the
            predicted class labels for all instances in X.
        """
        if self._ensemble:
            if self.update_strategy == self._REPLACE_STRATEGY:
                trees_in_ensemble = sum(i is not None for i in self._ensemble)
            else:   # self.update_strategy == self._PUSH_STRATEGY
                trees_in_ensemble = len(self._ensemble)
            if trees_in_ensemble > 0:
                d_test = xgb.DMatrix(X)
                for i in range(trees_in_ensemble - 1):
                    margins = self._ensemble[i].predict(d_test, output_margin=True)

                    d_test.set_base_margin(margin=margins)
                predicted = self._ensemble[trees_in_ensemble - 1].predict(d_test)

                return np.array(predicted).astype(float)
        # Ensemble is empty, return default values (0)
        return np.zeros(get_dimensions(X)[0])
```

References

1. Montiel, J., Mitchell, R., Frank, E., Pfahringer, B., Abdessalem, T., Bifet, A. (2020, July). Adaptive XGBoost for evolving data streams. In 2020 International Joint Conference on Neural Networks (IJCNN) (pp. 1-8). IEEE.
2. Chen, T., Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 785–794). New York, NY, USA: ACM. https://doi.org/10.1145/2939672.2939785
3. Montiel, J., Read, J., Bifet, A., Abdessalem, T. (2018). Scikit-multiflow: A multi-output streaming framework. The Journal of Machine Learning Research, 19(72):15.
4. E. S. Page. 1954. Continuous inspection schemes. Biometrika 41, 1/2 (1954), 100–115