COMPSCI361: Introduction to Machine Learning

Data Stream Mining II

CS762 S1 2023

Instructor: Thomas Lacombe
Adapted from Yun Sing Koh

Overview

- Classifier Evaluation
- Classifier for Data Streams
- Ensembles for Data Streams

Classifier Evaluation

An evaluation framework should be composed of the following parts:

- error estimation,
- evaluation performance measures,
- statistical significance validation,
- a cost measure of the process.

Can we use traditional classifier evaluation techniques?

Would the following technique work?

- 10-fold cross-validation

Can we use traditional classifier evaluation techniques?

Static techniques do not work?

- Traditional evaluation that assumes a fixed training-test set is not adequate.
- The evaluation should also take into account the evolving nature of the data.

Error Estimation

- Holdout: When data is so abundant that it is possible to have test sets periodically, then we can measure the performance on these holdout sets.
- Interleaved test-then-train: Each individual example can be used to test the model before it is used for training, and from this, the accuracy can be incrementally updated.
- Prequential: Like interleaved test-then-train but the idea that more recent examples are more important, using a sliding window or a decaying factor.
- Interleaved chunks: Also like interleaved test-then-train, but with chunks of data in sequence.

Holdout

- Source of holdout examples is new examples from the stream that have not yet been used to train the learning algorithm.
- When no concept drift is assumed, a single static held out set should be sufficient, which avoids the problem of varying estimates between potential test sets.
- Assuming that the test set is independent and sufficiently large relative to the complexity of the target concept, it will provide an accurate measurement of generalization accuracy

Prequential and Interleaved test-then-train

- Test set is dynamic!
- Assumption: It assumes the direct availability of the labels of the arriving instances for testing.
- Each individual example can be used to test the model before it is used for training, and from this the accuracy can be incrementally updated. When intentionally performed in this order, the model is always being tested on examples it has not seen. This scheme has the advantage that no holdout set is needed for testing, making maximum use of the available data.
- It is considered as a **pessimistic estimator**.

Performance Evaluation Measures

- Prequential accuracy measure is only appropriate when all classes are balanced. The Kappa statistic is a more sensitive measure for quantifying the predictive performance of streaming classifiers.
- The Kappa statistic κ defined as follows:

$$\frac{p_0 - p_c}{1 - p_c}$$

- The quantity p_0 is the classifier's prequential accuracy, and p_c is the probability that a chance classifier—one that randomly assigns to each class the same number of examples as the classifier under consideration—makes a correct prediction.
- If the classifier is always correct, then $\kappa = 1$.
- If its predictions coincide with the correct ones as often as those of a chance classifier, then $\kappa = 0$.

Performance Evaluation Measures

- The Kappa M statistic is a measure that compares against a majority class classifier instead of a chance classifier:

$$\frac{p_0 - p_m}{1 - p_m}$$

- In cases where the distribution of predicted classes is substantially different from the distribution of the actual classes, a majority class classifier can perform better than a given classifier while the classifier has a positive κ statistic

Baseline Classifiers

- The Majority Class algorithm is one of the simplest classifiers: it predicts the class of a new instance to be the most frequent class.
- No-change classifier, which predicts the label for a new instance to be the true label of the previous instance. It exploits autocorrelation in the label assignments, which is very common.
- Lazy Classifier, the classifier consists of keeping some of the instances seen, and predicting using the class label of the closest instances to the instance whose class label we want to predict.

Decision Trees from Data Stream

- In the case of data streams data arrive continuously to the considered node.
- Decision tree for static data: (1) Which attribute to choose?
- Decision tree for stream data: (1) Which attribute to choose? (2) When to make a split?

Decision Trees from Data Stream

- The aim of a decision tree learning streaming algorithm is to provided output nearly identical to that of a conventional/static learner.

Intuition

- Internal node: test on example's attribute value
- Leaf node: class labels
- Key idea: 1) pick an attribute to test at root 2) divide the training data into subsets D_i for each value the attribute can take on 3) build the tree for each D_i and splice it in under the appropriate branch at the root.

Hoeffding Tree Algorithm

Main ideas applied in the Hoeffding tree algorithm:

- Sufficient statistics
- Splitting criterion

Hoeffding Tree (Domingos and Hulten)

- In the data stream setting, where we cannot store all the data, the main problem of building a decision tree is the need to reuse instances to compute the best splitting attributes.
- Hoeffding Tree, a very fast decision tree algorithm for streaming data, where instead of reusing instances, we wait for new instances to arrive.

Hoeffding tree algorithm: sufficient statistics

- In each node information collected in a form of "sufficient statistics": N_{ijk} number of data elements from the k-th class which take the j-th value of the i-th attribute

Hoeffding tree algorithm: splitting criterion

- Compute the split measure function $f_i(S)$ each attribute $i = 1, \ldots, d$ based on the currently collected instances $S = X_1, \ldots, X_N$.
- Find two attributes with the highest values of f of $f_{a_{max1}}(S)$ and $f_{a_{max2}}(S)$.
- If $f_{a_{max1}}(S) - f_{a_{max2}}(S) > \epsilon = \epsilon(N, \delta)$ with a probability of $1 - \delta$

$$\mathrm{E}(f_{a_{max1}}(S)) > \mathrm{E}(f_{a_{max2}}(S)) \quad (1)$$

and choose a_{max1} as a splitting attribute.

Hoeffding tree algorithm: splitting criterion

Splitting criterion

$$f_{a_{max1}}(S) - f_{a_{max2}}(S) > \epsilon = \epsilon(N, \delta)$$

with a probability of $1 - \delta$

Hoeffding's inequality

The main mathematical tool used in this algorithm was the Hoeffding's inequality Theorem: If $X_1, X_2, \ldots, X_N$ independent random variables and $a_i \leq X_i \leq b_i \, (i = 1, 2, \ldots N)$ then for $\epsilon > 0$.

$$P(X - \mathrm{E}[X] \geq \epsilon) \leq \exp^{-2N^2\epsilon^2 / \sum_{i=1}^{N} (b_i - a_i)^2} \tag{2}$$

Challenge: How to find formula for $\epsilon(N, \delta)$?

Hoeffding's inequality

$$\epsilon = \epsilon(N, \delta) = \frac{\sqrt{R^2 \ln 1/\delta}}{2N} \quad (3)$$

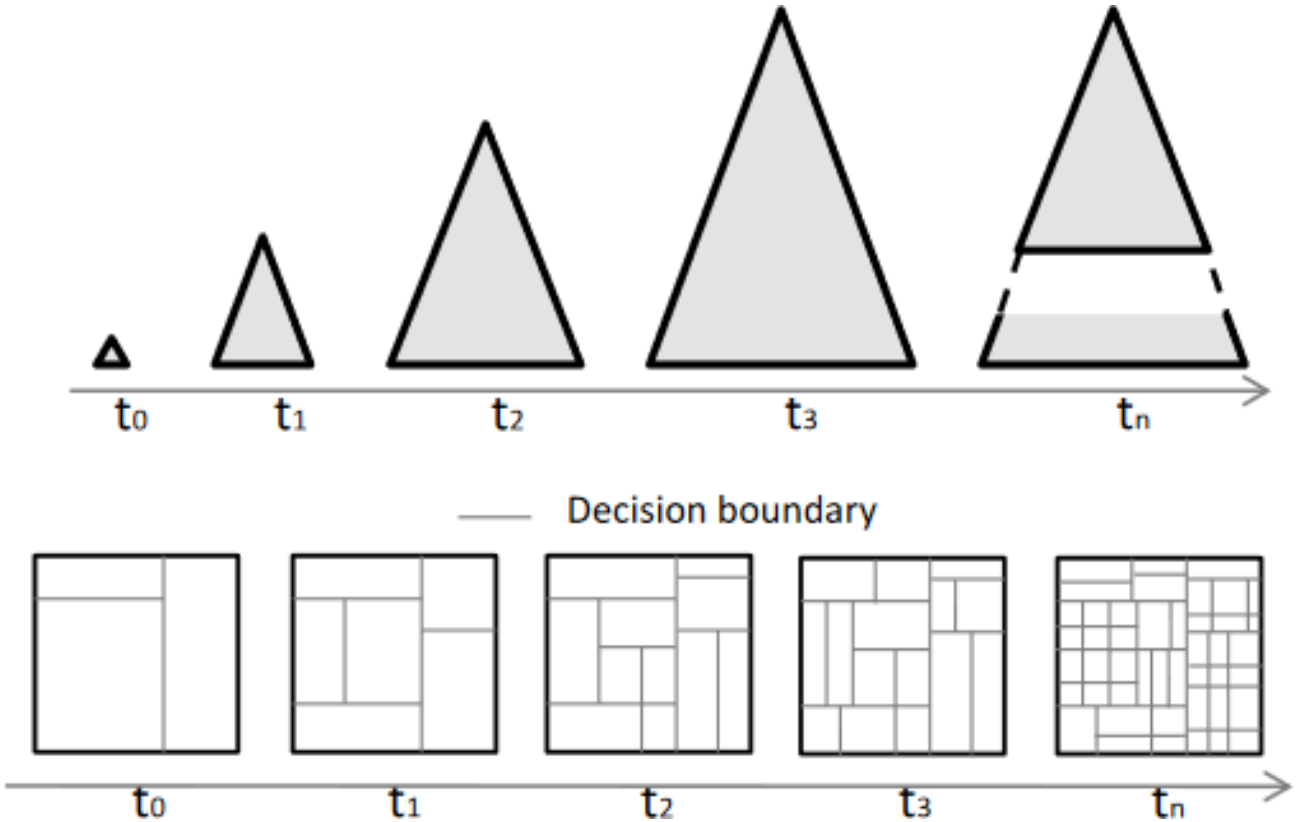
where R is a range of values of the applied split measure.

Hoeffding Tree

Result: Hoeffding Tree
Let HT be a tree with a single leaf(root);
Init counts n_{ijk} at root;
for *each example (x, y) in Stream* **do**
  HTGROW((x, y), HT);
end

Algorithm 1: Hoeffding Tree

Hoeffding Tree



Hoeffding Tree or VFDT

Result: Hoeffding Tree
Let HT be a tree with a single leaf(root);
Init counts n_{ijk} at root;
for *each example (x, y) in Stream* **do**
  HTGROW((x, y), HT);
end
def *HTGROW((x, y), HT)*:
  Sort (x, y) to leaf l using HT ;
  Update counts n_{ijk} at leaf l ;
  if *examples seen so far at l are not all of the same class* **then**
    Compute G for each attribute
  end
  if *G(Best Attr.)- G(2nd best)* $> \sqrt{\frac{R^2 \ln 1/\delta}{2n}}$ **then**
    Split leaf on best attribute
  end
  for *each branch* **do**
    Start new leaf and initiliatize counts
  end

Algorithm 2: Hoeffding Tree

VFDT

The VFDT (Very Fast Decision Tree) algorithm makes several modifications to the Hoeffding tree algorithm to improve both speed and memory utilization. The modifications include:

- breaking near-ties during attribute selection more aggressively,
- computing function f_a after a number of training examples,
- deactivating the least promising leaves whenever memory running low,
- dropping poor splitting attributes,
- improving the initialization method.

VFDT

Ties: when two attributes have similar G, split if.

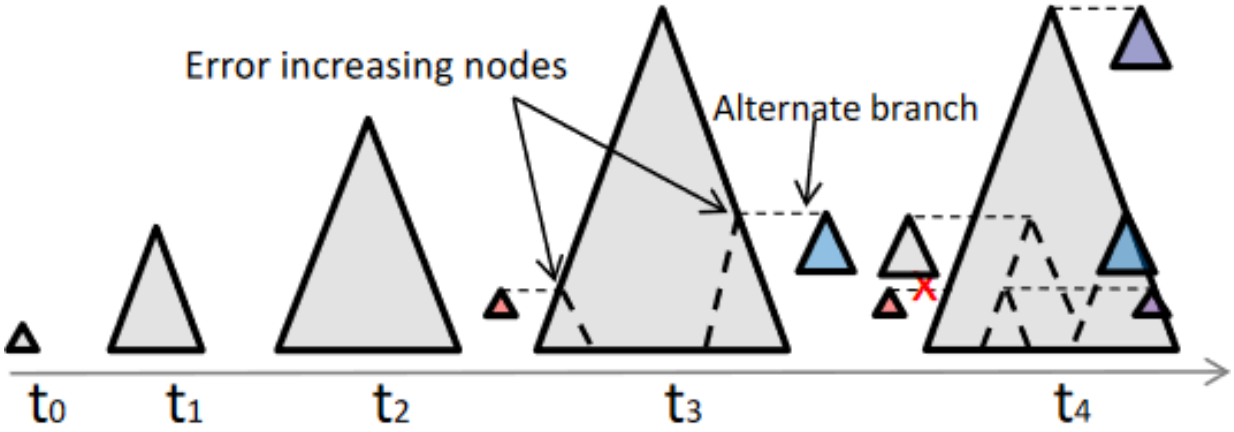
$$G(BestAttr.) - G(2ndbest) < \frac{\sqrt{R^2 \ln 1/\delta}}{2N} < \tau$$

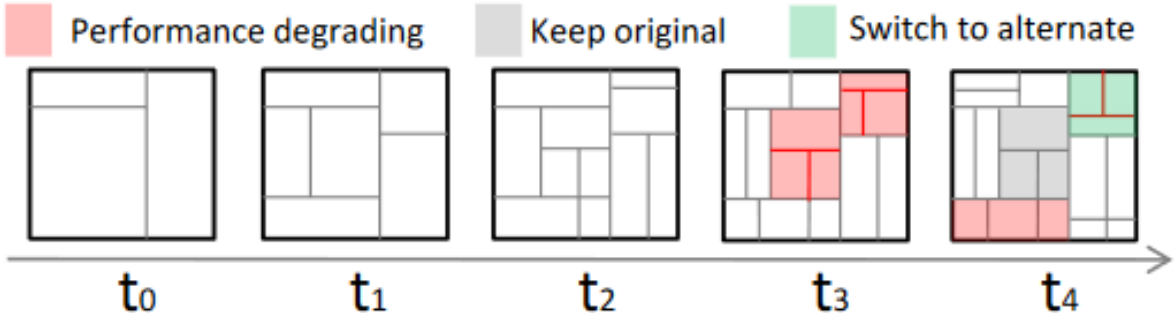
To speed up the process, instead of computing the best attributes to split every time a new instance arrives, the VFDT computes them every time a number n_{min} of instances has arrived.

CVFDT (Hulten et al.)

- Concept-adapting very fast decision tree (CVFDT) algorithm as an extension of VFDT to deal with concept drift, maintaining a model that is consistent with the instances stored in a sliding window.
- It keeps its model consistent with a sliding window of examples
- Construct "alternative branches" as preparation for changes. It does not construct a new model from scratch each time.
- It updates statistics at the modes by incrementing the counts associated with new examples and decrementing the counts associated with old ones,
- If the alternative branch becomes more accurate, switch of tree branches occurs
- Disadvantage, such trees do not have theoretical guarantees like Hoeffding trees.

CVFDT





CVFDT (Hulten et al.)

Result: Hoeffding Tree
Let HT be a tree with a single leaf(root);
Init counts n_{ijk} at root;
for *each example (x, y) in Stream* **do**
    add remove and forget Examples;
    CVFDTGROW((x, y), HT);
    CheckSplitValidity(HT, n, δ);
end
def *CVFDTGROW((x, y), HT)*:
    Sort (x, y) to leaf l using HT ;
    Update counts n_{ijk} at leaf l ;
    if *examples seen so far at l are not all of the same class* **then**
        Compute G for each attribute
    end
    if *G(Best Attr.)- G(2nd best)* > $\sqrt{\frac{R^2 \ln 1/\delta}{2n}}$ **then**
        Split leaf on best attribute
    end
    for *each branch* **do**
        Start new leaf and initiliatize counts
    end
    Create alternate Subtree

Algorithm 3: CVFDT

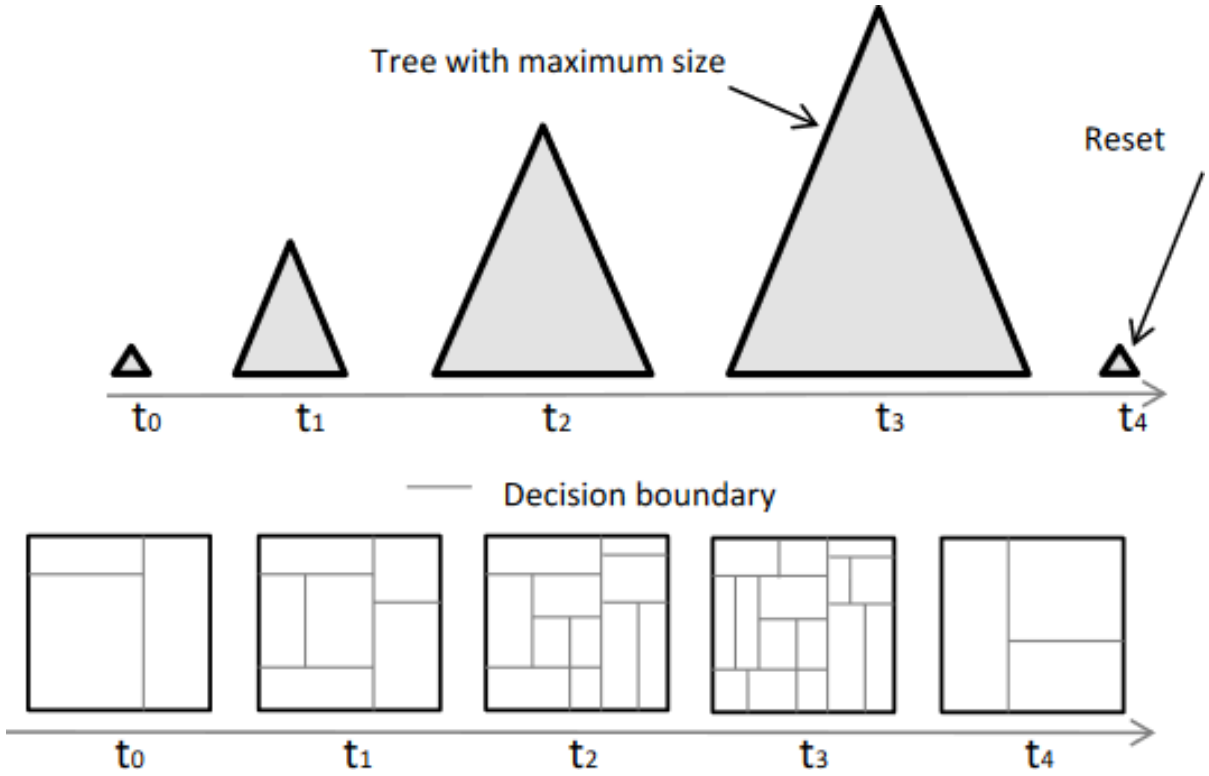
CVFDT (Hulten et al.)

- The main method maintains a sliding window with the latest instances, so it has to add, remove, and forget instances.
- The main method calls procedure CVFDTGROW to process an example, but also method CheckSplitValidity to check whether the chosen splits are still valid.
- CVFDTGrow also updates counts of the nodes traversed in the sort.
- CheckSplitValidity creates an alternate subtree if the attributes chosen to split are now different from the ones that were chosen when the split was done.
- Periodically, the algorithm checks whether the alternate branch is performing better than the original branch tree, and if so it replaces the original branch, and if not, it removes the alternate branch.

Hoeffding Adaptive Tree (HAT) (Bifet et al.)

- Replace frequency counters by estimators
- No need for window of instances
- Sufficient statistics kept by estimators separately
- Parameter-free change detector + estimator with theoretical guarantees for subtree swap (ADWIN)
- Keeps sliding window consistent with "no-change hypothesis"

Hoeffding Adaptive Tree



Other splitting criterion

Drawback of Hoeffding Inequality.

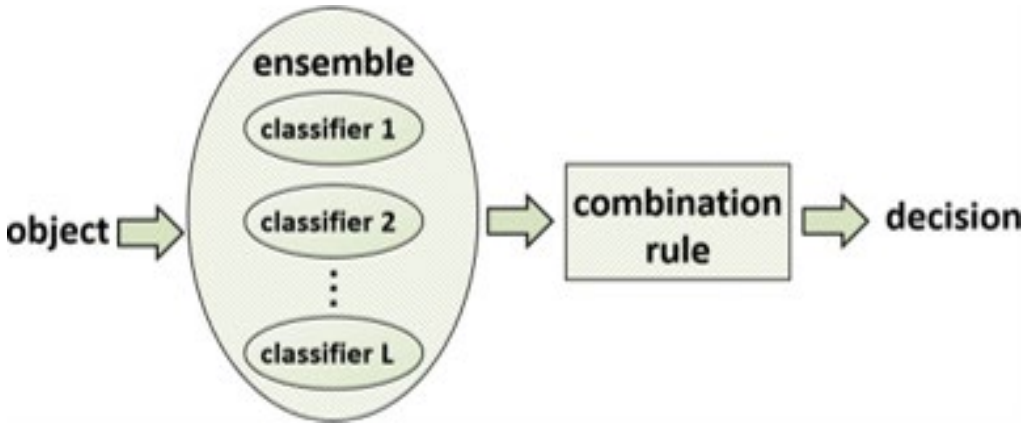
- it only operates on numerical variables
- it demands an input that can be expressed as a sum of the independent variables

This is not the case for Information Gain and Gini Index.

Ensemble

Ensemble learning is a well-established area in static machine learning due to the following reasons:

- Classifiers combination can improve the performance of the best individual ones and it can exploit unique classifier strengths
- Avoiding the selection of the worst classifier. Usually they other more flexible decision boundary and at the same time they do not suffer from overfitting
- Can be simply and efficiently applied to distributed environments?



Ensemble

Ensemble learning can be seen as a natural choice for mining non-stationary data streams:

- It can use the changing concept as a way to maintain diversity It has flexibility to incorporate new data:
  - Adding new components
  - Updating existing components
- It offers natural forgetting mechanism via ensemble pruning
- It reduces the variance of base classifiers, thus increasing the stability It allows to model changes in data as weighted aggregation of base classifiers.

Ensembles for stream mining

- Ensembles according to processing modes:
  - Block ensembles
  - Online ensembles
- Ensembles according to their method for adapting to drifting streams:
  - Dynamic combiners: base classifiers learned in advance, combination rule adapts to changes
  - Ensemble updating: all / some base classifiers updated with incoming examples
  - Dynamic ensemble line-up: new classifiers added for incoming data, weakest ones removed from the committee

Ensembles for stream mining

- Ensemble based approaches provide a natural fit to the problem of learning in a nonstationary setting
  - Ensembles tend to be more accurate than single classifier-based systems due to reduction in the variance of the error
  - (Stability) They have the flexibility to easily incorporate new data into a classification model when new data are presented, simply by adding new members to the ensemble
  - (Plasticity) They provide a natural mechanism to forget irrelevant knowledge, simply by removing the corresponding old classifier(s) from the ensemble

Dynamic combiners

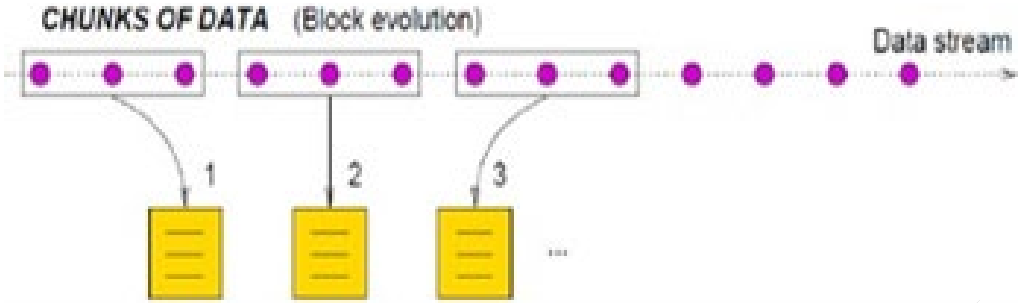
- Based on assumption that concept drift can be modeled as varying classifier combination scheme, e.g., with weights assigned to each classifier.
- In order to work we require an efficient pool of initial classifiers with high diversity to capture different properties of the analyzed stream.
- Classifier combination block is subject to identical limitations as standard classifiers in regard to time and memory consumption.
- Untrained combiners - less accurate, low computational complexity, fast adaptation.
- Trained combiners - more accurate, increased complexity, require additional data for training (big limitation for streams).

Ensemble updating

- This approach assumes that our ensemble consist of classifiers that can be updated in batch or online modes.
- At the beginning we train a set of classifiers that will be continually adapted to the current state of the data stream.
- This requires a diversity assurance method, usually realized as initial training on different examples (online Bagging) or different features (online Random Subspaces or online Random Forest). Additional diversity may be assured by using incoming examples to update only some of the classifiers in a random or guided manner.

Dynamic ensemble line-up

- This approach assumes that we have a flexible ensemble set-up and add new classifiers for each incoming chunk of data.
- Generic scheme:
  - Train single initial classifier or K initial classifiers (subject to training data availability)
  - For each incoming chunk of data:
    - Train a new component classifier
    - Test other classifiers against the recent chunk Assign weight to each classifier
    - Select top L classifiers (remove the weaker classifiers)



Ensembles

- OzaBag
- ADWIN Bagging for M models
- Leveraging Bagging
- Oza and Russell's Online Boosting
- Adaptive Random Forest

OzaBag (Oza and Russell)

- It is difficult at first to draw a sample with replacement from the stream.
- Bootstrap replica, the number of copies K of each of the n examples in the training set follows a binomial distribution For large values of n, this binomial distribution tends to a Poisson(1) distribution.

OzaBag (Oza and Russell)

Algorithm 1 OzaBag

Result: OzaBag

Initialize base models h_m for all $m \in 1, 2, \ldots, M$

for *all training examples* **do**

  for $m = 1, 2, \ldots, M$ **do**

    Set $w = Poisson(1)$

    Update h_m with the current example with weight w

  end

end

anytime output

return hypothesis: $h_{fin}(x) = \arg\max \sum_{t=1}^{T} I(h_t(x) = y)$

Leveraging Bagging ensemble classifier

- Leveraging Bagging is an improvement over the Oza Bagging algorithm. The bagging performance is leveraged by increasing the re-sampling. It uses a Poisson distribution to simulate the re-sampling process. To increase re-sampling it uses a higher w value of the Poisson distribution (average number of events), 6 by default, increasing the input space diversity, by attributing a different range of weights to the data samples.
- To deal with concept drift, Leveraging Bagging uses the ADWIN algorithm to monitor the performance of each member of the ensemble.
- If concept drift is detected, the worst member of the ensemble (based on the error estimation by ADWIN) is replaced by a new (empty) classifier.

Adaptive Random Forest (Gomes et al., 2017)

- It adds diversity through resampling ("bagging")
- It adds diversity through randomly selecting subsets of features for node splits
- It has one drift and warning detector per tree, which cause selective resets in response to drifts. It also allows training background trees, which start training if a warning is detected and replace the active tree if the warning escalates to a drift.

Coping with concept drift – Background trees

- To cope with evolving data streams a drift detection algorithm is usually coupled with the ensemble algorithm The default approach is to reset learners immediately after a drift is signaled. This may decrease the ensemble classification performance, since this learner has not been trained on any instance, thus making it unable to positively impact the overall ensemble predictions.
- Instead of resetting trees as soon as drifts are detected, in ARF use a more permissive threshold to detect warnings and create "background" trees that are trained along the ensemble without influence the ensemble predictions. If a drift is detected for the tree that originated the warning signal it is then replaced by its respective background tree.

ARF

```
1: function AdaptiveRandomForests(m, n, δ_w, δ_d)
2:     T ← CreateTrees(n)
3:     W ← InitWeights(n)
4:     B ← ∅
5:     while HasNext(S) do
6:         (x, y) ← next(S)
7:         for all t ∈ T do
8:             ŷ ← predict(t, x)
9:             W(t) ← P(W(t), ŷ, y)
10:            RFTreeTrain(m, t, x, y)          ▷ Train t on the current instance (x, y)
11:            if C(δ_w, t, x, y) then          ▷ Warning detected?
12:                b ← CreateTree()             ▷ Init background tree
13:                B(t) ← b
14:            end if
15:            if C(δ_d, t, x, y) then          ▷ Drift detected?
16:                t ← B(t)                     ▷ Replace t by its background tree
17:            end if
18:        end for
19:        for all b ∈ B do                     ▷ Train each background tree
20:            RFTreeTrain(m, b, x, y)
21:        end for
22:    end while
23: end function
```

Recurrent concept frameworks

- Recurring or recurrent concept frameworks have been proposed for these scenarios.
- The main idea is to keep a library of classifiers that have been useful in the past but have been removed from the ensemble, for example because at some point they underperformed.
- Their accuracy is rechecked periodically in case they seem to be performing well again, and if so, they are added back to the ensemble.

Resources

Packages

- MOA Java
- Scikit-Multiflow Python
- Scikit-Ika Python

Theory

- Machine Learning for Data Streams with Practical Examples in MOA (https://moa.cms.waikato.ac.nz/book-html/) – Chapter 6 and 7
- Part of the slides are from Machine Learning for Data Streams with Practical Examples
- Part of the slides are https://www.dbs.ifi.lmu.de/Lehre/BigData-Management&Analytics/WS16-17/skript/07-Stream_Clustering_Classification.pdf