This is a portion of the eBook at doi:10.7551/mitpress/10654.001.0001

2 Big Data Stream Mining

In this chapter we give a gentle introduction to some basic methods for learning from data streams. In the next chapter, we show a practical example of how to use MOA with some of the methods briefly presented in this chapter. These and other methods are presented in more detail in part II of this book.

2.1 Algorithms

The main algorithms in data stream mining are classification, regression, clustering, and frequent pattern mining.

Suppose we have a stream of items, also called instances or examples, that are continuously arriving. We are in a classification setting when we need to assign a label from a set of nominal labels to each item, as a function of the other features of the item. A classifier can be trained as long as the correct label for (many of) the examples is available at a later time. An example of classification is to label incoming email messages as spam or not spam. Regression is a prediction task similar to classification, with the difference that the label to predict is a numeric value instead of a nominal one. An example of regression is predicting the value of a stock in the stock market tomorrow.

Classification and regression need a set of properly labeled examples to learn a model, so that we can use this model to predict the labels of unseen examples. They are the main examples of *supervised* learning tasks. When examples are not labeled, one interesting task is to group them in homogeneous clusters. Clustering can be used, for example, to obtain user profiles in a website. It is an example of an *unsupervised* learning task.

Frequent pattern mining looks for the most relevant patterns within the examples. For instance, in a sales supermarket dataset, it is possible to know what items are bought together and obtain association rules, as for example: *Most times customers buy cheese, they also buy wine*.

The most significant requirements for a stream mining algorithm are the same for predictors, clusterers, and frequent pattern miners:

Requirement 1: Process an instance at a time, and inspect it (at most) once.

Requirement 2: Use a limited amount of time to process each instance.

Requirement 3: Use a limited amount of memory.

Requirement 4: Be ready to give an answer (prediction, clustering, patterns) at any time.

Requirement 5: Adapt to temporal changes.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2263139/9780262346047_cah.pdf by guest on 21 June 2024

This is a portion of the eBook at doi:10.7551/mitpress/10654.001.0001

2.2 Classification

In batch or offline classification, a classifier-building algorithm is given a set of labeled examples. The algorithm creates a model, a classifier in this case. The model is then deployed, that is, used to predict the label for unlabeled instances that the classifier builder never saw. If we go into more detail, we know that it is good methodology in the first phase to split the dataset available into two parts, the training and the testing dataset, or to resort to cross-validation, to make sure that the classifier is reasonably accurate. But in any case, there is a first training phase, clearly separated in time from the prediction phase.

In the online setting, and in particular in streaming, this separation between training, evaluating, and testing is far less clear-cut, and is interleaved. We need to start making predictions before we have all the data, because the data may never end. We need to use the data whose label we predict to keep training the model, if possible. And probably we need to continuously evaluate the model in some way to decide if the model needs more or less aggressive retraining.

Generally speaking, a stream mining classifier is ready to do either one of the following at any moment:

1. Receive an unlabeled example and make a prediction for it on the basis of its current model.
2. Receive the label for an example seen in the past, and use it for adjusting the model, that is, for training.

For example, an online shop may want to predict, for each arriving customer, whether the customer will or will not buy a particular product (prediction). When the customer session ends, say, minutes later, the system gets the "label" indicating whether indeed the customer bought the product or not, and this feedback can be used to tune the predictor. In other cases, the label may never be known; for example, if the task is to determine whether the customer is a robot or a human, the true label may be available for a few customers only. If trying to detect fraudulent transactions in order to block them, transactions predicted to be fraudulent are not executed, so their true labels are never known.

This simple description glosses over a number of important practical issues. First, how many of the unlabeled instances eventually receive their correct label? Clearly, the fewer labels received, the harder the prediction task. How long should we wait for an instance label to arrive, before we drop the

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2263139/9780262346047_cah.pdf by guest on 21 June 2024

This is a portion of the eBook at doi:10.7551/mitpress/10654.001.0001

instance? Efficiently managing the buffer of instances waiting for their labels is a very delicate implementation problem when dealing with massive, high-speed streams. Finally, should we use all labeled instances for training? If in fact many labels are available, perhaps there is a diminishing return in accuracy for the increased computational cost of training on all instances.

It is difficult to deal with these issues in generic ways, because often the good solution depends too much on the details of a specific practical scenario. For this reason, a large part of the research in stream classification deals with a simplified cycle of training/prediction: we assume that we get the true label of *every* unlabeled instance, and that furthermore we get it *immediately* after making the prediction and before the next instance arrives. In other words, the algorithm executes the following loop:

- Get an unlabeled instance x.
- Make a prediction $\hat{y} = f(x)$ for x's label, where f is the current model.
- Get the true label y for x.
- Use the pair (x, y) to update (train) f, and the pair $(\hat{y}, y)$ to update statistics about classifier performance.
- Proceed to the next instance.

This model is rightly criticized by practitioners as too simple, because it ignores the very real problem of *delayed* and *missing* label feedback. It is however quite useful for comparing learning algorithms in a clean way, provided we have access to, or can simulate, a stream for which we have all labels.

Two variations of these cycles are worth mentioning. In *semi-supervised* learning, we use unlabeled examples for training as well, because at least they provide information on the distribution of the examples. Unfortunately, there is little work on semi-supervised stream learning, even though the abundance of data in high-speed streams makes it promising and a good approach to the delayed/missing label feedback problem. In *active learning*, the algorithm does not expect the labels of all instances but selectively chooses which ones to request. This is a good approach when every label is theoretically available but obtaining it has a cost. In this book we cover active learning in streams but not semi-supervised learning.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2263139/9780262346047_cah.pdf by guest on 21 June 2024

This is a portion of the eBook at doi:10.7551/mitpress/10654.001.0001

2.2.1 Classifier Evaluation in Data Streams

Given this cycle, it is reasonable to ask: How do we evaluate the performance of a classification algorithm? In traditional batch learning, evaluation is typically performed by randomly splitting the data into training and testing sets (holdout); if data is limited, cross-validation (creating several models and averaging results across several random partitions in training and test data) is preferred.

In the stream setting, (effectively) unlimited data tends to make cross-validation too expensive computationally, and less necessary anyway. But it poses new challenges. The main one is to build an accurate picture of accuracy over time. One solution involves taking snapshots at different times during the induction of a model to see how the model accuracy varies. Two main approaches arise:

- **Holdout:** This is measuring performance on a single holdout partition. It is most useful when the division between train and test sets has been predefined, so that results from different studies can be directly compared. However, holdout only gives an accurate estimation of the current accuracy of a classifier if the holdout set is similar to the current data, which may be hard to guarantee in practice.
- **Interleaved test-then-train or prequential:** Each individual example is used to test the model before it is used for training, and from this the accuracy can be incrementally updated. When the evaluation is intentionally performed in this order, the model is always being tested on instances 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 also ensures a smooth plot of accuracy over time, as each individual example will become less and less significant to the overall average. In test-then-train evaluation, all examples seen so far are taken into account to compute accuracy, while in prequential, only those in a sliding window of the most recent ones are.

As data stream classification is a relatively new field, such evaluation practices are not nearly as well researched and established as they are in the traditional batch setting.

Next we name and describe a few of the best-known stream classifiers. The first two are so simple that they are usually only considered baselines for the evaluation of other classifiers.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2263139/9780262346047_cah.pdf by guest on 21 June 2024

This is a portion of the eBook at doi:10.7551/mitpress/10654.001.0001

2.2.2 Majority Class Classifier

The majority class classifier is one of the simplest classifiers. It stores a count for each of the class labels, and predicts as the class of a new instance the most frequent class label.

2.2.3 No-Change Classifier

The no-change classifier is another one of the simplest: predict the last class in the data stream. It exploits autocorrelation in the label assignments, which is very common.

2.2.4 Lazy Classifier

The lazy classifier is based on a very simple idea: 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. In particular, the k-nearest neighbor or k-NN method outputs the majority class label of the k instances closest to the one to predict.

For this classifier, a predefined notion of closeness or distance is required, and the performance depends on the meaningfulness of this distance with respect to labels.

Example 2.1 Consider the following dataset of tweets, on which we want to build a model to predict the sentiment ($+$ or $-$) of new incoming tweets.

ID	Text	Sentiment
T1	glad happy glad	$+$
T2	glad glad joyful	$+$
T3	glad pleasant	$+$
T4	miserable sad glad	$-$

Assume we want to classify the following new instance:

ID	Text	Sentiment
T5	glad sad miserable pleasant glad	?

The simple classifiers will perform the following predictions:

- *Majority classifier:* The majority class label is $+$, so the prediction is $+$.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2263139/9780262346047_cah.pdf by guest on 21 June 2024

This is a portion of the eBook at doi:10.7551/mitpress/10654.001.0001

- *No-change classifier:* The class label of the last instance (T4) is −, so the prediction is −.
- *Lazy classifier:* If we measure similarity by the number of common words, the tweet closest to T5 is T4, with label −. Therefore, the 1-NN classifier predicts − for T5.

2.2.5 Naive Bayes

The Naive Bayes classifier is a simple classifier based on the use of the Bayes' theorem. The basic idea is to compute a probability for each one of the classes based on the attribute values, and select the class with the highest probability. Under the naive assumption that the attributes are all independent, the class probabilities can be computed by multiplying over all attributes the probability of having that particular class label conditioned on the attribute having a particular value. The independence assumption is almost always false, but it can be shown that weaker assumptions suffice, and Naive Bayes does surprisingly well for its simplicity on a variety of tasks.

2.2.6 Decision Trees

Decision tree learners build a tree structure from training examples to predict class labels of unseen examples. The main advantage of decision trees is that it is easy to understand their predictions. That is why they are among the most used classifiers in settings where black-box classifiers are not desirable, for example, in health-related applications.

In stream mining, the state-of-the art decision tree classifier is the *Hoeffding tree*, due to Domingos and Hulten [88], and its variations. Traditional decision trees scan the entire dataset to discover the best attribute to form the initial split of the data. Once this is found, the data is split by the value of the chosen attribute, and the algorithm is applied recursively to the resulting datasets, to build subtrees. Recursion is applied until some stopping criterion is met. This approach cannot be adopted directly in the stream setting, as we cannot afford the resource cost (time and memory) of storing instances and repeatedly scanning them.

The Hoeffding tree is based on the idea that, instead of looking at previous (stored) instances to decide what splits to do in the trees, we can wait to receive enough instances and make split decisions when they can be made confidently. The main advantage of this approach is that it is not necessary to store

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2263139/9780262346047_cah.pdf by guest on 21 June 2024

This is a portion of the eBook at doi:10.7551/mitpress/10654.001.0001

instances. Instead, sufficient statistics are kept in order to make splitting decisions. The sufficient statistics make it easy to incorporate Naive Bayes models into the leaves of the tree.

The *Hoeffding adaptive tree* [33] is an extension of the Hoeffding tree that is able to create and replace new branches when the data stream is evolving and the class label distribution or instance distribution is changing.

2.2.7 Ensembles

Ensembles are sets of classifiers that, when combined, can predict better than any of them individually. If we use the same algorithm to build all the classifiers in the ensemble, we will need to feed the algorithm with different subsets of the data to make them different. *Bagging* is an ensemble method that (1) uses as input for each run of the classifier builder a subset obtained by sampling with repetition of the original input data stream, and (2) uses majority voting of the classifiers as a prediction strategy.

The *ADWIN bagging* method [38], implemented as OzaBagAdwin in MOA, is an extension of bagging that it is able to create and replace new classifiers when the data stream is evolving and the class label distribution is changing.

2.3 Regression

As in classification, the goal in a regression task is to learn a model that predicts the value of a label attribute for instances where the label is not (yet) known. However, here the label is a real value, and not one of a discrete set of values. Predicting the label exactly is irrealistic, so the goal is to be close to the correct values under some measure, such as average squared distance.

Several classification algorithms have natural counterparts for regression, including lazy learning and decision trees.

2.4 Clustering

Clustering is useful when we have unlabeled instances and we want to find homogeneous groups or clusters in them, according to their similarities or affinities. The main difference from classification is that the groups are unknown before the learning process, and we do not know whether they are

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2263139/9780262346047_cah.pdf by guest on 21 June 2024

This is a portion of the eBook at doi:10.7551/mitpress/10654.001.0001

the “correct” ones after it. This is why it is a case of so-called *unsupervised* learning. Uses of clustering include segmentation of customers in marketing and finding communities in social networks.

We can see clustering as an optimization problem where we want to minimize a cost function that evaluates the “badness” of a partition of examples into clusters. Some clustering methods need to be told the desired number of clusters to find in the data, and others will find some number of clusters by themselves.

The k-means clustering method is one of the most used methods in clustering, due to its simplicity. It starts by selecting k centroids in a random way. After that, two steps are repeated: first, assign each instance to the nearest centroid, and second, recompute the cluster centroids by taking the center of mass of the instances assigned to it. This is repeated until some stopping criterion is met, such as no changes in the assignments. It is not a streaming method, as it requires several passes over the same data.

Streaming methods for clustering typically have two levels, an online one and an offline one. At the online level, a set of microclusters is computed and updated from the stream efficiently; in the offline phase, a classical batch clustering method such as k-means is performed on the microclusters. The online level only performs one pass over the data; the offline phase performs several passes, but not over all the data, only over the set of microclusters, which is usually a pretty small set of controllable size. The offline level can be invoked once, when (if) the stream ends, or periodically as the stream flows to update the set of clusters.

2.5 Frequent Pattern Mining

Frequent pattern mining is an important task in unsupervised learning. It can be used to simply describe the structure of the data, to find association rules, or to find discriminative features that can be used for classification or clustering tasks. Examples of pattern classes are itemsets, sequences, trees, and graphs [255].

The problem is as follows: given a source of data (a batch dataset or a stream) that contains patterns, and a threshold σ, find all the patterns that appear as a subpattern in a fraction σ of the patterns in the dataset. For example, if our source of data is a stream of supermarket purchases, and $\sigma = 10\%$, we would call $\{cheese, wine\}$ frequent if at least 10% of the purchases contain at

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2263139/9780262346047_cah.pdf by guest on 21 June 2024

This is a portion of the eBook at doi:10.7551/mitpress/10654.001.0001

least cheese *and* wine, and perhaps other products. For graphs, a triangle could be a graph pattern, and if we have a database of graphs, this pattern would be frequent if at least a fraction σ of the graphs contain at least one triangle.

In the batch setting, Apriori, Eclat, and FP-growth are three of the best-known algorithms for finding frequent itemsets in databases, where each item is a set. Similar algorithms exist for data structures such as sequences and graphs. However, it is difficult to translate them directly to the stream setting, either because they perform several passes over the data or because they store too much information.

Algorithms for frequent pattern mining on streams typically use a batch miner as a base, but need to implement other ideas; furthermore, they are often approximate rather than exact. Moment and IncMine are algorithms for frequent itemset mining in data streams.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2263139/9780262346047_cah.pdf by guest on 21 June 2024

This is a portion of the eBook at doi:10.7551/mitpress/10654.001.0001

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2263139/9780262346047_cah.pdf by guest on 21 June 2024

This is a section of doi:10.7551/mitpress/10654.001.0001

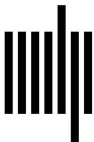
Machine Learning for Data Streams

with Practical Examples in MOA

By: Albert Bifet, Ricard Gavaldà, Geoffrey Holmes, Bernhard Pfahringer

Citation:

Machine Learning for Data Streams: with Practical Examples in MOA
By: Albert Bifet, Ricard Gavaldà, Geoffrey Holmes, Bernhard Pfahringer
DOI: 10.7551/mitpress/10654.001.0001
ISBN (electronic): 9780262346047
Publisher: The MIT Press
Published: 2023



The MIT Press

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2263139/9780262346047_cah.pdf by guest on 21 June 2024

MIT Press Direct

© 2017 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form or by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

This book was set in Times Roman and Mathtime Pro 2 by the authors.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data is available

ISBN: 978-0-262-03779-2

10 9 8 7 6 5 4 3 2 1

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2263139/9780262346047_cah.pdf by guest on 21 June 2024