Online Learning in Open Feature Spaces

Exploring Decision Tree-based Methods for Learning from Data Streams with Varying Feature Spaces

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Christian Schreckenberger
aus Mannheim

Mannheim, 2025

Dekan:	Prof. Dr. Claus Hertling
Referent:	Prof. Dr. Heiner Stuckenschmidt, *Universität Mannheim*
Korreferent:	Prof. Dr. Yi He, *College of William & Mary*

Tag der mündlichen Prüfung: 6. Februar 2026

Abstract

In an era of rapid technological advancements, the ability to efficiently understand and extract knowledge from dynamic and ever-evolving data environments is crucial. This thesis presents machine learning methods to deal with such intricacies by leveraging the foundations of decision trees and random forests. Specifically, the proposed approaches address the issues of monotonically increasing feature spaces and varying feature spaces in online learning environments. Beyond that, we argue that our proposed methods naturally convey a higher level of interpretability over their machine learning competitors tailored for the same evolving environments.

The proposed methods are built on the foundation of decision tree-based models, which are inherently known for their interpretability. On the other hand, decision tree-based ensemble models are also particularly known for their performance. Hence, we envisioned adjusting and developing models in this group to provide performant and interpretable models that are capable of taking on the challenges that come with dynamic and ever-evolving environments.

The first method, Dynamic Fast Decision Tree, is targeted at monotonically increasing feature spaces, i.e., feature spaces where new features emerge over time. The proposed approach extends the established Hoeffding tree method, with additional restructuring and pruning capabilities. The second approach, Dynamic Forest, is targeted at varying feature spaces, i.e., feature spaces where new features may emerge, while others may vanish. It builds on the well-established principles of random forest and enhances them by dynamically managing the ensemble to deal with the implications of varying feature spaces. The final method presented in this thesis is called ORF^{3}V, which also focuses on varying feature spaces. The method dynamically manages so-called feature forests and forms a random forest-like ensemble, based

on efficient approximation of feature statistics.

Empirical evaluations were conducted for all three methods, demonstrating that the proposed methods achieve competitive results while having the advantage of being interpretable. Finally, we provide a roadmap for future directions and open challenges, which were identified in our previous work.

Zusammenfassung

In Zeiten rasanter technologischer Fortschritte ist die Fähigkeit, effizient Wissen aus dynamischen und sich stetig wandelnden Datenumgebungen zu verstehen und zu extrahieren von zentraler Bedeutung. In dieser Arbeit werden drei Maschinelle Lernansätze vorgestellt, die genau das ermöglichen. Sie bauen auf den Grundlagen von Entscheidungsbäumen und Random Forests auf. Im Einzelnen gehen die Ansätze auf die Herausforderungen von monoton wachsenden Merkmalsräumen und variierenden Merkmalsräumen in online Lernumgebungen ein. Dabei wird gezeigt, dass die präsentierten Ansätze, im Gegensatz zu anderen Maschinellen Lernmethoden, die für denselben Zweck entwickelt wurden, einen höheren Grad an Interpretierbarkeit zulassen. Das ist möglich, da die vorgestellten Ansätze auf auf Entscheidungsbäumen basierende Modelle beruhen, die bekannt für ihre inhärente Interpretierbarkeit sind. Des Weiteren zeichnen sich auf Entscheidungsbäumen basierende Ensemble Modelle durch ihre hohe Leistung aus. Folglich wurde sich dafür entschieden, Modelle aus dieser Gruppe anzupassen und weiterzuentwickeln, um leistungsstarke und interpretierbare Modelle bereitfstellen zu können, die dazu fähig sind, mit den Herausforderungen von dynamischen und sich stetig wandelnden Datenumgebungen umgehen zu können.

Die erste Methode, Dynamic Fast Decision Tree, befasst sich mit monoton wachsenden Merkmalsräumen, d. h. mit Merkmalsräumen, in denen mit der Zeit neue Merkmale erscheinen. Die methodische Umsetzung erweitert die etablierte Hoeffding-Tree-Methode um zusätzliche Umstrukturierungs- und Pruningverfahren. Der zweite Ansatz, Dynamic Forest, behandelt variierende Merkmalsräume, bei denen neue Merkmale erscheinen können, während andere verschwinden. Der vorgestellte Ansatz basiert auf den bewährten Prinzipien des Random Forest-Verfahrens und verbessert diesen durch ein

dynamisches Management des Ensembles, um so den Auswirkungen variierender Merkmalsräume entgegenzuwirken. Die letzte Methode, ORF^{3}V, fokussiert ebenfalls variierende Merkmalsräume. Sie verwaltet dynamisch sogenannte Feature-Forests und bildet ein Random Forest-ähnliches Ensemble, das auf einer effizienten Approximation von Merkmalsstatistiken basiert.

Alle drei Methoden wurden empirisch evaluiert. Dabei wurde gezeigt, dass die vorgeschlagenen Ansätze wettbewerbsfähige Ergebnisse erzielen und dabei den zusätzlichen Vorteil der Interpretierbarkeit bieten. Abschließend wird ein Plan für zukünftige Weiterentwicklungen sowie offene Herausforderungen vorgestellt, die in vorausgehenden Arbeiten identifiziert wurden.

Acknowledgements

I would like to specifically acknowledge three individuals who have had a significant impact on me, my academic journey, and especially this dissertation: Heiner Stuckenschmidt, Christian Bartelt, and Yi He.

Heiner, I am incredibly grateful for the environment you have provided during the (too long) journey of this dissertation. Your approach to mentoring left room for me to develop my own ideas and grow independently, yet you always provided the necessary support and guidance when needed and found the right words in times of setbacks.

Special thanks also go to Christian. Your support in the initial stages of this journey has taught me a lot, extending beyond academia. The team you assembled was a significant factor in why I always enjoyed coming to the office.

Yi, getting to know you had a very significant impact on this dissertation. Your academic support and input have been unmatched. I am deeply thankful to have collaborated with your brilliance on several projects for this dissertation.

Special mentions go out to Christian Meilicke, who sparked the idea of pursuing a PhD, and to Tim Glockner, whom I had the pleasure of supervising for his Bachelor's thesis.

I would also like to thank all my colleagues who have been part of this endeavor and made it enjoyable: Fabian Burzlaff, Jakob Kappenberger, Lea Cohausz, Nils Wieber, Nils Wilken, Sascha Marton, Christian Schindler, Kristian Kolthoff, Michael Oesterle, Lars Hoffmann, and many others.

My final words of gratitude go out to my family and closest friends. I want to thank my partner Anna, my parents Ursula and Manfred, my brother Tobias, and my friends Lukas and Philipp for their unconditional emotional support.

Contents

List of Figures

List of Tables

List of Algorithms

Nomenclature

The following list describes the general and algorithm specific notation.

General Notation

x	An instance of dataset X
y	The class label of instance x
c	The class value
C	The number of classes
X	All observations of x
Y	All observations of y
$\mathcal{D}$	The tuple (X,Y)
$\mathcal{X}$	The domain of X
$\mathcal{Y}$	The domain of Y
F_d	The d-th feature
D_t	The dimensionality of the feature space at time t
$\mathcal{F}$	The universal feature space
$\mathcal{F}(x)$	The available features of observation x
$f(x)$	The goal function we want to approximate
$\hat{f}(x)$	The learned function/model approximated from the data
$\hat{y}$	The predicted class given $\hat{f}(x)$
$\mathbb{I}(\hat{y}_t = y_t)$	Identity function, one if $\hat{y}_t = y_t$, zero otherwise

DynamicFastDecisionTree

$\mathcal{T}$	A decision tree or subtree $\mathcal{T}$
S	A split node S
S_F	A split node that splits on feature F
$\mathbf{S}$	An array of split nodes

L	A leaf node of decision tree
$\mathbf{L}$	An array of leaf nodes
n_{ijk}	The matrix to store sufficient statistics in Hoeffding trees
$L_{n_{ijk}}$	The n_{ijk} matrix of a leaf node L
M	Decision matrix extracted from tree $\mathcal{T}$
$\mathcal{T}_{left}$	The left sided subtree of $\mathcal{T}$
$\mathcal{T}_{right}$	The right sided subtree of $\mathcal{T}$

DynamicForest

$\mathcal{L}$	Ensemble of all decision stumps
$\mathcal{L}_b$	The b-th decision stump in the ensemble
$\mathbf{w}$	All weights of the ensemble
w_b	Weight associated with the b-th decision stump
$\mathcal{L}_b^{F_d}$	The b-th decision stump that splits on feature F_d
$\mathcal{F}_b$	The accepted feature space of the b-th weak learner
$\mathcal{W}_t$	The sliding window at time t

ORF3V

$\mathcal{L}$	Ensemble of all feature forests
$\mathcal{L}_d$	Feature forests corresponding to the d-th feature in the ensemble
$\mathbf{w}$	All weights of the ensemble
w_d	Weight associated with the d-th feature forest
Φ_d	The feature statistics associated with the d-th feature forest

Part I

Introduction

Chapter 1

Motivation

Computing technologies, in terms of hardware and software, constantly evolve. Devices get smaller and cheaper, computer programs get more capable and efficient. This not only leads to more productivity but also to an ever-increasing ubiquitous ability to capture data in its various forms [Friedewald and Raabe, 2011]. We are able to capture the world in data with a resolution that becomes finer and finer, converging toward reality. Machine learning is the field of research with the goal of extracting patterns from data to turn them into actionable insights and knowledge [Zhou et al., 2017]. The more generalizable this knowledge is, the more valuable it can be, as it allows us to apply it to unseen cases with higher confidence [James et al., 2023]. As of now, given the characteristics of the data (e.g., sequential, structured, high-dimensional) and the constraints of the task at hand (e.g., low-memory, real-time), various learning strategies have been proposed [Hoi et al., 2021b, James et al., 2023].

Traditionally, machine learning focuses on *offline* learning. Hereby, a batch of data is fed to the learning algorithm, which can make multiple passes over the data to derive a model. All data is available to the learning algorithm when building the model. However, this was only feasible as data was not as abundantly available as it is today. As data became more available, problems started to arise because the computing power in terms of main memory is now a limiting factor, and not all data can be inspected at once. Another factor is that as more data emerges, models must be retrained entirely from scratch, which is expensive and results in poor scalability for real-world applications [Hoi et al., 2021b].

Not only the ever-growing size of data but also the velocity at which data is produced, thanks to advances in and the ubiquity of sensing techniques [Gama and Gaber, 2007, Nittel, 2015, Shi and Abdel-Aty, 2015, Pardo et al., 2015], has led to what is called *online* learning. Hereby, the model is built incrementally in consecutive rounds. At each round, more data becomes available to the learning algorithm. This led to many powerful algorithms for streaming data analytics [Aggarwal, 2007, Shalev-Shwartz et al., 2011, Yu et al., 2017, Leite et al., 2013], which can be employed in a diverse set of use cases, such as stock price prediction [Tantisripreecha and Soonthomphisaj, 2018], dynamic pricing [Levina et al., 2009], or fraud detection[Paladini et al., 2023].

However, despite this advancement and the general applicability of online learning algorithms to a variety of use cases, there is one shortcoming: online learning algorithms assume that the feature space remains static. However, this assumption does not hold in many applications, some of which are presented as follows:

- In sensor networks, sensors can be added over time, hence the feature space can grow. On the other side, sensors will eventually fail. This can lead to two scenarios: the sensor is replaced, resulting in a short-term gap in the data, meaning we have to deal with missing data, or the sensor is not replaced, causing a vanished feature; thus, the feature space can shrink.

- Considering natural language processing tasks, such as sentiment classification, there will be new words or hashtags that emerge, which leads to a growing feature space in a bag-of-words representation. On the other hand, language is constantly evolving, and the use of words vanishes, leading to a decrease in feature space size.

- In a recommender system built from an e-commerce application, new products will be added over time, while others will be discontinued, leading to new features as users add these new products to their baskets, while other features vanish as users can't add the discontinued products to their baskets anymore.

Considering these applications, the need for online learning algorithms that can deal with *open feature spaces*, as opposed to closed feature spaces, can be

justified. Initial research on an incrementally growing feature space [Zhang et al., 2015, 2016] showed the feasibility of such online learning algorithms and sparked several research endeavors in the open feature space direction [Hou et al., 2017, Beyazit et al., 2019, He et al., 2021a]. The common denominator of these approaches is the use of linear and non-interpretable models.

1.1 Problem Statement

As outlined, in many real-world scenarios, the characteristics of incoming data can evolve over time. This situation arises when data streams vanish or emerge. Consequently, a model trained on a set of features may encounter future instances containing an altered feature space. On top of this, many real-world problems demand solutions that are interpretable [Kim, 2020] so that when certain features emerge or vanish, we can understand how and why the model's behavior shifts. Interpretable machine learning models offer the advantage of providing insights into their decision-making, thereby fostering trust in users [Ribeiro et al., 2016]. A highly dynamic environment adds additional complexity for a user. Alleviating this complexity by providing insights as to what or why something is important can increase such trust [Virgolin et al., 2022]. Considering the sensor network example, knowing which sensors are essential for accurate predictions and the ability to identify them can provide hints on when sensors should be replaced or where it could be beneficial to deploy more.

1.2 Requirements

Given the motivation, the outlined problem in Section 1.1, and the data stream classification cycle in Figure 1.1, we can postulate six requirements that need to be satisfied. Requirement 1, Requirement 2, Requirement 3, and Requirement 5 have been proposed by Kirkby [2007] in their work concerned with online learning. As they still hold, we follow their description closely. However, to learn from open feature spaces, two additional requirements are needed. We need to adapt to changes in the feature space (Requirement 4) and provide interpretability (Requirement 6) to foster trust in the model in a dynamic environment.

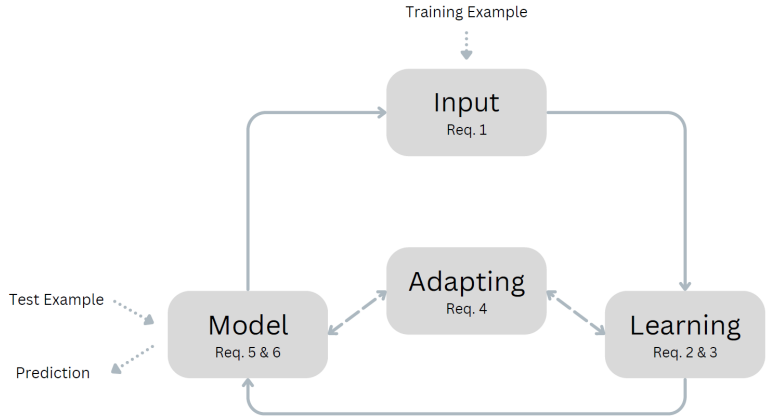


Figure 1.1: The data stream classification cycle, initially introduced by Kirkby [2007], now represents a new diagram when adapted to our setting of online learning in open feature spaces with interpretable methods.

1.2.1 Requirement 1: Online processing.

A defining feature of learning from data streams lies in the instantaneous processing of new examples. Each instance is available upon arrival, without the option to retrieve previous examples at a later time. Hence, there is no option for random access to the data.

This paradigm dictates the input of the algorithm; however, there is no prohibition to store examples internally temporarily. For example, the algorithm might gather a subset of examples in a sliding window to apply offline machine learning algorithms. Eventually, it is required to purge all examples.

The inspect-once policy can be softened to allow resending entire data streams, which is equivalent to scanning the database multiple times. Hereby, the algorithm is allowed to refine the learned model in a successive iteration. Nonetheless, any online learning algorithm requiring more than a single pass is limited in its general applicability to data streams.

1.2.2 Requirement 2: Finite memory.

The primary reason for using the stream model is its ability to process data that is significantly larger than the available memory. Due to the nature of data streams, there is a risk of running out of memory if no limits are

set. Hence, the algorithms should minimize their memory footprint where possible.

Generally, the memory used within an algorithm can be categorized into two types: memory for storing ongoing statistics, i.e., aggregated descriptions of the data, and memory for storing the current model. In the most memory-efficient scenarios, these categories merge, meaning the ongoing statistics serve as the model utilized for prediction.

The limitation on memory can be alleviated by utilizing external storage, such as temporary files. However, any such workaround must be approached with careful consideration of Requirement 3, i.e., runtime constraints. Note that the algorithms in this thesis do not require such workarounds.

1.2.3 Requirement 3: Finite processing time.

To accommodate any volume of examples, the runtime complexity of the algorithm should scale linearly with the number of examples. This scalability is attainable within the data stream framework by imposing a constant, preferably minimal, upper limit on the processing load per example.

Due to the nature of data streams and the rapid arrival of new observations, the learning algorithm must process the examples as quickly as possible. Processing the observations slower than new observations arriving could lead to data loss, as the examples can not be persisted.

While absolute timing may not be paramount in less demanding scenarios, such as when the algorithm is applied to classify a large but persistent data source, slower performance diminishes its utility for users, who expect fast results within a shorter timeframe.

1.2.4 Requirement 4: Feature Space Adaptation.

The ideal algorithm can adapt to changes in the feature space. Regardless of the number of features it has observed, it can adjust to a new feature space and prioritize features with a higher importance. In reality, there may be phases where the feature space remains static and no adaptation is necessary, but it is of utmost importance that the algorithm possesses this capability.

Adhering to Requirement 2, the algorithm should possess the ability to keep the model small. This means pruning parts of it where the data streams

ceased to exist. Generally, smaller models also provide better interpretability, hence benefiting Requirement 6.

1.2.5 Requirement 5: Anytime Predictions.

An ideal algorithm can learn an optimal model based on the data it has observed up to this point in time. However, in practice, there may be intervals during which the model remains static, such as when a batch-based algorithm accumulates the next batch of data.

Learning the model is required to be as efficient as possible. It should be avoided to relearn the model. Instead, updates to the model should be done by manipulating the model itself.

Additionally, the algorithm should be able to predict instances that only carry a subset of the previously observed features, meaning it is capable of handling missing inputs for certain features.

1.2.6 Requirement 6: Interpretability.

The learned model from the algorithm should not only make correct predictions but also be interpretable. This requirement is essential for fostering trust in the model among users and for ultimately gaining an understanding of the captured world in a highly dynamic environment.

Furthermore, with respect to Requirements 1 and 2, it is especially challenging to inspect the data produced, thereby voiding any opportunity to analyze the data retrospectively and gain insights. Hence, interpretable models should be preferred to provide insights into the data.

1.2.7 Data Stream Classification Cycle Extension

The data stream classification cycle with adaptations for learning from open feature spaces is illustrated in Figure 1.1. Following the depiction of Kirkby [2007] closely, the extended cycle contains one additional step, where the learning and the model may invoke an adaptation process when it is required. The three steps of receiving a new instance that needs to be processed instantaneously (Req. 1), the learning with space and time bound restrictions (Req. 2 and Req. 3), and finally having a model that is ready to predict at any time (Req. 5), essentially remain the same. The adaptation

step is required whenever the feature space changes, requiring adaptation of the learning process as well as the model to the new feature space (Req. 4). This dynamic then, in turn, requires the model to have an additional requirement, namely that it is interpretable (Req. 6).

1.3 Contribution & Published Work

By providing solutions to the outlined problem statement and the requirements of the previous sections, we contribute in many ways to the state of learning from data streams with open feature spaces. The research conducted for this thesis has led to three conference publications and one workshop publication:

- Schreckenberger, C., Glockner, T., Stuckenschmidt, H., & Bartelt, C. (2020, November). Restructuring of Hoeffding trees for trapezoidal data streams. In 2020 International Conference on Data Mining Workshops (ICDMW) (pp. 416-423). IEEE.
- Schreckenberger, C., Bartelt, C., & Stuckenschmidt, H. (2022, September). Dynamic forest for learning from data streams with varying feature spaces. In International Conference on Cooperative Information Systems (pp. 95-111). Cham: Springer International Publishing.
- Schreckenberger, C., He, Y., Lüdtke, S., Bartelt, C., & Stuckenschmidt, H. (2023, June). Online random feature forests for learning in varying feature spaces. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 37, No. 4, pp. 4587-4595).
- He, Y., Schreckenberger, C., Stuckenschmidt, H., & Wu, X. (2023, August). Towards utilitarian online learning–a review of online algorithms in open feature space. In Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence (pp. 6647-6655).

1.4 Outline

In the remainder of Part I, we elaborate on the background of the presented work in Chapter 2. Hereby, we lay the foundation by reviewing key concepts

in machine learning, with emphasis on online learning and open feature spaces (Section 2.1). Based on this brief introduction, we then proceed to introduce the key methods on which our presented approaches are grounded. Hereby, we first introduce decision trees (Section 2.2), which includes an introduction to their induction, prediction mechanisms, and interpretability. Then, we proceed similarly with random forests in Section 2.3. Finally, we present how open feature spaces are evaluated in Section 2.4. This includes a short introduction to the experimental design as well as the evaluation metrics commonly used.

In Part II are our core contributions of this thesis. We begin by examining the problem of learning from monotonically increasing feature spaces in Chapter 3. Here, we introduce the Dynamic Fast Decision Tree approach (Section 3.1), which adapts decision trees to dynamically adjust to new features by restructuring and pruning the given model. This is followed by situating the approach in the context of related work in Section 3.2, which is then complemented by a parameter and performance evaluation in Section 3.3. The chapter is concluded with a discussion on how the proposed approach fulfills the presented requirements Section 3.4. Following this, Chapter 4 addresses the complexities of learning from varying feature spaces. First, we introduce the problem of varying feature spaces and justify its significance with the crowdsense dataset (Section 4.1), which was collected within the scope of this thesis. We then proceed to introduce our two random forest-based approaches Dynamic Forest (Section 4.2) and ORF^{3}V (Section4.3). Both methods are designed to handle varying feature spaces by dynamically managing ensemble membership. The introduction of the methods is followed by relating them to the overall field of online learning in varying feature spaces, as well as online learning with trees and tree-ensemble methods in Section 4.4. After providing a thorough evaluation of the established benchmark evaluation case, we also provide insights into their interpretability (Section 4.5). As with the previous chapter, we conclude this chapter with a discussion by situating the proposed approaches with regard to the requirements (Section 4.6).

Looking forward, Part III is comprised of two chapters. In Chapter 5, open challenges and future directions that can emerge from our research are described. Here, we discuss issues where we see research potential, such as robustness against label imbalance and noise, adaptation to concept drift,

the open-world crisis, as well as security, privacy, and algorithmic fairness. Finally, in Chapter 6, we wrap up the thesis by summarizing the contributions and providing final thoughts.

Chapter 2

Background

This chapter provides the framework in which our research resides and gives context for the challenges inherent in open feature spaces. We begin by outlining the evolution of machine learning from offline to online learning, highlighting the shift from static to dynamic feature spaces. We then explore the fundamental models for our research, i.e., decision trees and random forests. Ultimately, we offer an introduction to the evaluation procedures in open feature spaces.

2.1 Machine Learning

At the highest level, machine learning can be divided into two sub-areas: unsupervised learning and supervised learning [Bishop and Nasrabadi, 2006]. In the unsupervised learning subarea, data is given with the goal of detecting patterns, e.g., clusters or outliers, from it. In contrast, supervised learning additionally has a target value $y_i \in Y$ associated with every data example $x_i \in X$, where $\mathcal{D} = (X, Y)$ refers to the dataset, i.e., all observations, and N is the total number of observations. Depending on the target value, we can further distinguish between two types of problems, i.e., regression and classification [Bishop and Nasrabadi, 2006]. In a regression problem, the target value is a real-valued number, whereas in a classification task, there is a finite number of classes. This thesis focuses on classification tasks. The domain of X and Y is referred to as $\mathcal{X}$ and $\mathcal{Y}$, respectively. A single feature (or data stream), which characterizes an observation x_i, is referred to as F_d,

where d is the d-th feature of that example. The union of all features is known as the feature space $\mathcal{F}$.

Definition 1 *In a **supervised classification task**, real valued feature vectors $X \in \mathbb{R}^{N \times D}$ with $D \in \mathbb{N}^+$ are associated with a class vector $Y \in \{1, ..., C\}^N$, where $C \in \mathbb{N}, C \geq 2$. The goal is to learn a function f with $f : \mathcal{X} \to \mathcal{Y}$. The function f takes an observation x as input and predicts the class $f(x) = \hat{y}$.*

The special case $C = 2$ is referred to as *binary classification*, whereas for any $C > 2$, the task is called *multi-class classification* [Murphy, 2012]. Note that in this thesis, we only focus on single membership classes, whereas in a multi-class multi-label classification task, it is also possible for examples to be associated with multiple classes [Chakraborty and Dey, 2024].

To determine f, a certain amount $N \in \mathbb{N}$ of observations $\{(x_i, y_i)\}_{i=1}^N$ are required. In *offline* learning, i.e., classical machine learning, this amount is *finite* and *known a priori* [James et al., 2023]. Furthermore, it is assumed that all observations are *independent and identically distributed* (i.i.d.), meaning that all examples are statistically independent of each other and drawn from the same probability distribution [Raschka, 2018]. The examples used to determine f are referred to as training examples. Usually, an exact determination of f is not possible. Therefore, we try to find an optimal estimation of f, which we call $\hat{f}$, with $\hat{f} : \mathcal{X} \to \mathcal{Y}$. Formally, $\hat{f}$ is a mapping that depends on the given training examples $\hat{f} = \hat{f}(x_1, y_1, ..., x_N, y_N)$. It provides us instructions on how to obtain a mapping $\mathcal{X} \to \mathcal{Y}$ for each observation (x_i, y_i) for $i = 1, .., N$. The function $\hat{f}$ is also referred to as a classifier. The predicted value $\hat{y}$ of the classifier $\hat{f}$ cam ne obtained with $\hat{f}(x) = \hat{y}$ [James et al., 2023].

Definition 2 *Given that $\mathbb{E}(Z)$ is the expected value of a random variable Z. A **loss function** is a measurable mapping $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_{\geq 0}$. Applying $\hat{f}$ to an observation x, with $\hat{f}(x) = \hat{y}$, leads to the loss $L(\hat{y}, y)$. The **risk** of $\hat{f}$ is $R(\hat{f}) := \mathbb{E}(L(y, \hat{f}(x)))$ [Vapnik, 1991].*

The loss function only serves as a means to determine the distance between the actual value of y and the predicted value of $\hat{y}$. In contrast, the risk $R(\hat{f})$ expresses the average expected error when applying $\hat{f}$. Therefore,

given all possible configurations of a classifier, we try to deduce a classifier that minimizes the risk. This principle is known as empirical risk minimization [Vapnik, 1991], which we follow in all the proposed approaches.

Generally, the correctness of a prediction made by a classifier is easily quantifiable. However, a classifier can also be evaluated from a qualitative perspective. Hence, the quality of a classifier $\hat{f}$ can be evaluated with respect to two main criteria [Bishop and Nasrabadi, 2006]:

- Predictive power (quantitative evaluation): Does $\hat{f}(x)$ predict the right y for a new observation x? The quality of the mapping $f : \mathcal{X} \rightarrow \mathcal{Y}$ can be measured with a loss function L, which determines the distance between $f(x)$ and y.
- Interpretability (qualitative evaluation): Can the classifier $\hat{f}$ be used to gain an understanding of how the features influence a decision and which features matter the most?

The ultimate goal of machine learning is to generalize beyond the examples seen in the training set [Domingos, 2012]. To gauge the generalization of a classifier, it is necessary to evaluate the predictive power of this classifier. The predictive power is usually assessed on the *test examples*. This set of examples contains, for the classifier $\hat{f}$, unseen observations to estimate the performance. Standard evaluation techniques, such as hold-out or cross-validation, divide the entire dataset into a training and test set [Raschka, 2018], i.e., just presenting the training examples to the learning algorithm and using the test examples for the evaluation.

With the surge of black box models, such as artificial neural networks, in recent times, the need for approaches to interpret these models has also risen [Zhang et al., 2021]. While these methods require distinct approaches to make them interpretable, other methods, such as decision trees and random forests, are inherently interpretable [Louppe, 2014]. In decision trees, for example, feature importance can be calculated directly from the given tree, or we can interpret the structure in a way that more important features are positioned closer to the root of the tree.

In this dissertation, we use a running example based on a sensor network deployed for weather monitoring. Each sensor station records temperature, humidity, wind speed, and precipitation at regular intervals. Machine learning methods can be applied to these data streams to classify weather condi-

tions (e.g., hot, mild). The weather sensor network can be expressed more formally using the standard terminology of supervised learning, as outlined below:

1. An **example** (or data instance) represents a single observation, e.g., the weather on the 3rd of April at 1 pm.

2. A **feature** (or attribute or data stream) characterizes the example, e.g., the measured temperature from a single weather station.

3. The **feature space** is made up of the union of all features, e.g, all sensors for this scenario.

4. The **class** indicates the membership of an example x_i to an exclusive group, e.g., when all weather stations indicate temperatures above 30°C, the observation can be associated with the class *hot*.

5. A **classifier** (or model) is a learned function $\hat{f}$ that maps an example x onto the class y, e.g., a decision tree model.

2.1.1 Online Learning

The assumption of offline learning, where the data is static, i.e., all the observations are known beforehand, does not hold in online learning [Domingos and Hulten, 2000]. Due to the volume of data generated, which is enabled by technological advances, algorithms can no longer assume that they have all observations available at training time [Domingos and Hulten, 2000]. Notably, the "three Vs" (volume, velocity, variety) of big data [Favaretto et al., 2020] underscore the need for machine learning algorithms that are tailored to this paradigm. Opposed to the static data given in traditional offline machine learning, we have streaming data in online learning, which is continuously generated and can be considered a subset of big data with the following characteristics [Eva Bartz, 2024]:

- Volume: Streaming data is generated in large quantities.
- Velocity: Streaming data is generated at a high rate.
- Variety: Streaming data is available in different formats.

- Variability: Streaming data can be structureless and vary over time, e.g., concept drifts.
- Volatility: Streaming data is available once.

Definition 3 *In an **online supervised classification task**, we have a continuously growing number of observations $x_t, t = 1, .., T$, where T denotes the end of the classification task. Each observation $x_t \in \mathbb{R}^D$ adhered to the pre-defined feature space of size $D \in \mathbb{N}^+$ and is associated with a class $y \in \{1, ..., C\}$, where $C \in \mathbb{N}, C \geq 2$. Each observation is available only once at round t and cannot be retrieved arbitrarily again at a later point in time. The goal is to incrementally learn a function f with $f : \mathcal{X} \to \mathcal{Y}$.*

In online supervised classification tasks, T is usually considered to be infinite, hence classical offline learning is not feasible on this amount of data as the memory requirements and, in some cases, the computing requirements would explode (cf. Req. 2 and Req. 3) [Shalev-Shwartz and Ben-David, 2014]. Generally, models learned in a streaming setting are learned iteratively [Hazan, 2023]. This means that at the start of the learning process, we may not have the optimal model, but we will converge over time to the optimal model based on the adjustments made by the learning algorithm [Cesa-Bianchi and Lugosi, 2006]. However, opposed to the number of observed instances, the feature space from which the model is learned remains fixed. Meaning that there are no changes in the feature space over time; hence, it is known a priori.

Running Example. In the weather sensor network, data arrive continuously in 5-minute intervals. An online learning algorithm can update its model incrementally with each new reading, rather than retraining from scratch. For instance, after processing the 12:00 measurements, the model is immediately refined using the 12:05 data, enabling timely adaptation.

2.1.2 Open Feature Spaces

The online learning setting can be extended to a variation with no restrictions on the stability of the feature space [Beyazit et al., 2019]. Instead of

having a fixed feature space, the feature space can then evolve by changing its dimensionality. The feature space observed until the point in time t, for $t = 1, \ldots, T$, is called the observed feature space $\mathcal{F}_t$. It is the union of all observed features from all observed instances x_t until the point in time t, with $\mathcal{F}_t = \mathcal{F}(x_1) \cup \ldots \cup \mathcal{F}(x_t)$. The function $\mathcal{F}(x_t)$ returns the feature space of an observation x_t.

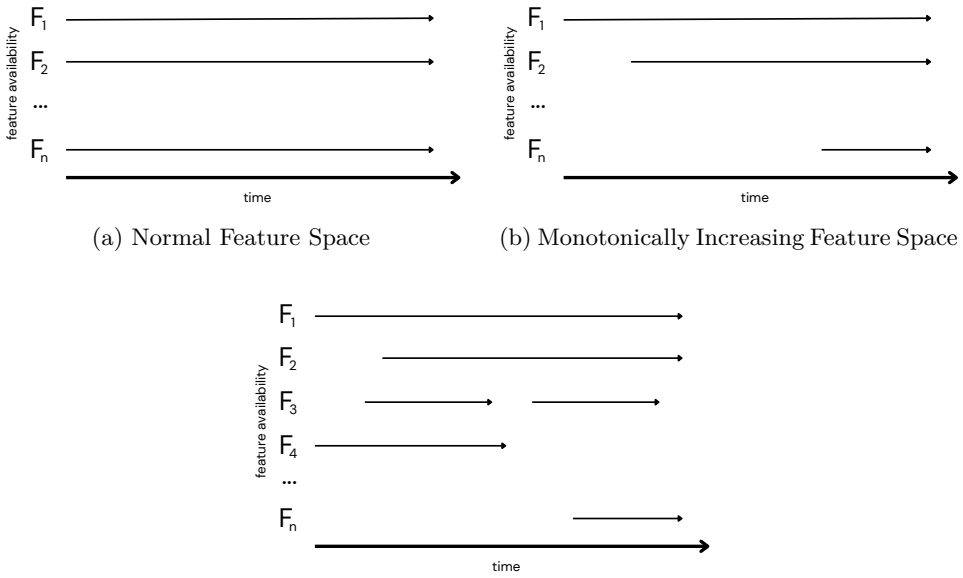
Definition 4 *The online supervised classification task can be extended to a setting with an **open feature space**. Hereby for each observation $x_t, t = 1, .., T$ the feature space of an observation $x_t \in \mathcal{X}_t \subseteq \mathbb{R}^{D_t}$ is dynamic and the dimension $D_t \in \mathbb{N}^+$ and can change arbitrarily in every round t.*

Generally, there are three cases for D_t in each round, i.e., it remains the same ($D_{t+1} = D_t$), it increases ($D_{t+1} > D_t$), or it decreases ($D_{t+1} > D_t$). However, in all three cases, this does not imply that we have the same features available in $t+1$ as we had in t. For example, despite an increase in the total feature space, another feature may have vanished or is temporarily missing. Given the literature [He et al., 2023], there are a few sub-cases with additional assumptions for the evolution of D_t, e.g., feature evolvable streams [Hou et al., 2017].

In this thesis, the primary focus is on the case with no constraints on the feature evolution, i.e., varying feature spaces. In this case, as described above, the features can vary arbitrarily between every observed instance. Furthermore, we also consider the case of monotonically increasing feature spaces, also commonly referred to as trapezoidal data streams [Zhang et al., 2015, 2016], where $D_{t+1} \geq D_t$, as this case highlights the component of an increasing feature space with no regard to features vanishing or missing for a certain amount of time.

Running Example. In the weather sensor network, the feature space is not fixed, but changes as the system evolves. Two patterns can be observed:

- **Varying Feature Spaces.** A sensor may fail temporarily or transmit irregularly, causing features to appear and disappear unpredictably. For instance, a wind-speed sensor might provide readings at 12:00, go offline at 12:05, and return again at 12:15. In some cases, a sensor



(a) Normal Feature Space

(b) Monotonically Increasing Feature Space

(c) Varying Feature Space

Figure 2.1: All three figures show the availability of features over time (feature is available, where arrow is present) in a setting where data is streamed: (a) shows the assumption of the feature space in a traditional online learning setting, i.e. all features are always available, (b) shows the assumption of a monotonically increasing feature space, i.e. features are added over time, and (c) shows the characteristics of a varying feature space, i.e. no assumption on how the feature space behaves.

may break completely and be decommissioned, requiring the learning algorithm to permanently remove this feature from the model while still ensuring reliable predictions from the remaining sensors.

- **Monotonically Increasing Feature Spaces.** When new sensors are deployed, the feature space expands permanently. For example, at 12:10, a new barometric pressure sensor is added, and from that point onward, its values appear every 5 minutes. Algorithms must integrate this new dimension while still leveraging previously learned information from the original sensors.

2.1.2.1 Challenges

As published in our work [He et al., 2023], an open feature space leads to two distinct challenges. i.e., how to solve the addition of emerged features

and how to deal with vanished or missing features. The descriptions, based on our previous work, are given below.

Emerging features: When introducing a new feature, it is important to ensure that current decisions remain unbiased by avoiding the risk of either overvaluing or undervaluing the feature. A common strategy for assigning weight to a feature is proportional apportionment [Penrose, 1946], which can be mapped to a learning problem by gauging the amount of information conveyed by the feature for prediction using mutual information between the feature and the label [Kraskov et al., 2004]. However, in an online process, a new feature may only be described by a few instances, making a precise information measurement next to impossible.

Vanishing features: When a majority of features are absent from the decision-making process, the voting opinions of those remain unobserved. This can lead to less informed decisions, particularly in extreme cases where the vast majority of features are missing. In such situations, decision-making may be dominated by a few remaining features that are less informative, leading to educated guesses. Furthermore, if a feature vanished for an extended period or indefinitely, it is unclear how to redistribute their voting weights in the model so that the subsequent decisions made by the remaining features still guarantee maximization of utility gain.

2.2 Decision Trees

A decision tree is a predictive model that represents decisions and their possible consequences in a tree-like structure [Quinlan, 1986]. As explained by Han et al. [2011], nodes in a decision tree represent the decisions based on a feature and a value or threshold. Branches then indicate the outcome of the decision and connect to the subsequent node or leaf. In the leaves of a decision tree, the final class or the class distributions are stored. A decision tree is induced based on the given training examples. To make a prediction with a decision tree, we have to follow the decision tree structure, starting from the root node and applying the decision at the split nodes, until we reach a leaf node that indicates the outcome for the example at hand.

Generally, decision trees are known to have several advantages [James et al., 2023], two of which are especially relevant with regard to this thesis:

- Decision trees are straightforward to explain due to their intuitive sequential decision structure.
- Decision trees can be displayed graphically, which enables the interpretability of the models even by non-experts.

On the other hand, decision trees are said to have less predictive power than other machine learning models. However, this can be alleviated by aggregating many decision trees [James et al., 2023] into so-called random forests, which we will focus on in the subsequent Section 2.3. In this section, we generally follow the explanations and definitions of Han et al. [2011] to introduce offline learning decision trees.

2.2.1 Induction of Decision Trees

The process of inducing a decision tree involves constructing a tree from a given set of training data, i.e., decision trees are applied in offline learning. This methodical construction has proven to create models that can generalize well to unseen data [Hastie et al., 2009]. In the induction process, the training data is recursively partitioned into subsets, resulting in a tree structure that represents the relationships within the data [Han et al., 2011]. This section will dive into the formalization of decision tree induction, detailing the foundations and algorithmic procedures involved.

According to Murphy [2012], the process can be formalized to the following, where we adjusted for our notation: Let $\mathcal{D} = (X, Y)$ be the training dataset consisting of N instances, where each instance x_i is described by a vector of features $\mathcal{F} = \{F_1, F_2, ..., F_D\}$ and a class label y_i for classification problems. The goal is to construct a decision tree $\mathcal{T} : \mathcal{X} \to \mathcal{Y}$ that maps an attribute vector to its corresponding output. Given our previous definition of a learned model $\hat{f}$, $\mathcal{T}$ is the equivalent, but signifies that it is a learned model based on a decision tree.

The induction of a decision tree is a recursive process that involves the following steps at each node:

1. **Select the Best Split Attribute:** Choose the attribute $F^* \in \mathcal{F}$ and a split value v that best partitions the dataset (X, Y) into subsets. The

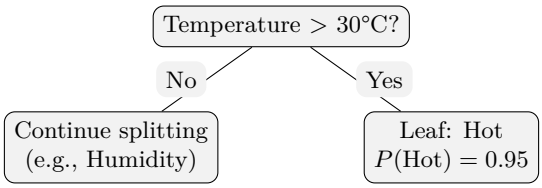


Figure 2.2: Decision tree induction for weather classification.

choice of F^* is based on a splitting criterion, such as information gain or Gini impurity.

2. **Split the Dataset:** Partition the dataset $\mathcal{D} = (X, Y)$ into subsets $\mathcal{D}_1 = (X_1, Y_1), \mathcal{D}_2 = (X_2, Y_2), \ldots, \mathcal{D}_k = (X_k, Y_k)$ based on the values v of the selected feature F^*.

3. **Create Split Nodes and Leaf Nodes:** If the partitioned subsets $(X_1, Y_1), (X_2, Y_2), \ldots, (X_k, Y_k)$ are homogeneous (i.e., all instances in (X_i, Y_i) belong to the same class) or some other stopping criterion is met, create a leaf node L_i representing the class label or the class label distribution. If the subsets are not homogeneous, or some other stopping criterion is met, create a split node S_{F_d} for F^* and recursively apply the partitioning process to each subset (X_i, Y_i).

Running Example. Suppose we want to classify 5-minute weather observations from the sensor network into two classes: *Hot* and *Mild*. Each observation consists of four features:

- Temperature (°C)
- Humidity (%)
- Wind speed (km/h)
- Precipitation (mm)

The training dataset contains labeled examples collected over the past week.

1. **Select the Best Split Attribute:** The induction algorithm evaluates candidate features using a splitting criterion such as information gain. It finds that splitting on **temperature** with threshold **30 °C** yields the highest information gain. Thus, F^* = temperature, with split value $v = 30$ °C.

2. **Split the Dataset:** The dataset is partitioned into two subsets:

   - (X_1, Y_1): All observations where temperature ≤ 30 °C
   - (X_2, Y_2): All observations where temperature > 30 °C

   Example:

   - At 12:00 $\rightarrow$ Temp = 28 °C, Humidity = 60% $\rightarrow$ assigned to (X_1, Y_1)
   - At 12:05 $\rightarrow$ Temp = 33 °C, Humidity = 35% $\rightarrow$ assigned to (X_2, Y_2)

3. **Create Split Nodes and Leaf Nodes:** If (X_1, Y_1) is not homogeneous, the algorithm continues splitting, perhaps next on humidity. If (X_2, Y_2) is nearly homogeneous (e.g., 95% "Hot"), the algorithm creates a leaf node L_2 predicting "Hot" with $P(\text{Hot}) = 0.95$. The root node becomes a split node $S_{\text{Temperature}}$, branching into two children. The resulting intermediate tree is shown in Figure 2.2.

2.2.1.1 Splitting Criterion

Different algorithms, such as ID3 [Quinlan, 1986], C4.5 [Quinlan, 1993], or CART [Breiman, 2017], use different *splitting criteria* to determine the best attribute. Two common measures in the domain of classification are information gain (IG) and Gini impurity (GI). The two criteria are defined by Han et al. [2011] as follows.

Information gain can be generally defined as,

$$IG(\mathcal{D}, F_d) = H(\mathcal{D}) - \sum_{v \in V(F_d)} \frac{|\mathcal{D}_v|}{|\mathcal{D}|} H(\mathcal{D}_v) \tag{2.2.1}$$

where $H(\mathcal{D})$ is the entropy of $\mathcal{D}$, and $V(F_d)$ are the observed values in feature F_d, and $\mathcal{D}_v$ refers to the subset that carry this value. In cases with continuous values in a feature, a splitting point $v \leq \theta$ is used, where values for a selected feature are below or equal to the splitting point θ are considered for the first subset, and the values greater than the splitting point θ are considered for the second subset.

Gini impurity is defined as,

$$GI(\mathcal{D}, F_d) = \sum_{v \in V(F_j)} \frac{|\mathcal{D}_v|}{|\mathcal{D}|}(1 - \sum_{c=1}^{C} p_{c|v}^2) \tag{2.2.2}$$

where $p_{c|v}$ is the proportion of instances in $\mathcal{D}$ that belong to class c with value v in the given feature F_d and C is the number of classes. Gini impurity ranges from 0 to 0.5, where lower is better. To calculate the optimal split, we would calculate, similarly to information gain, the reduction of impurity in the child nodes that results from splitting a feature or a feature-value combination.

As both splitting criteria, information gain and Gini impurity, operate similarly in terms of determining the best possible split, we will refer to them as $G(\mathcal{D}, F_j)$, where they can be used interchangeably.

Running Example. Returning to the weather sensor network, assume we have collected labeled observations in 5-minute intervals, where each instance is classified into either *Hot* or *Mild*. The candidate features are temperature, humidity, wind speed, and precipitation.

To induce a decision tree, the algorithm evaluates each feature using a splitting criterion $G(\mathcal{D}, F_j)$. Suppose the entropy $H(\mathcal{D})$ of the dataset is high, as both classes are equally represented. When calculating the expected reduction in entropy for different attributes, the temperature feature yields the largest gain at the threshold $\theta = 30°\text{C}$. Concretely:

- Instances with temperature $\leq 30°\text{C}$ are predominantly labeled *Mild*.
- Instances with temperature $> 30°\text{C}$ are predominantly labeled *Hot*.

As a result, the split

$$\text{Temperature} \leq 30°\text{C} \quad \text{vs.} \quad \text{Temperature} > 30°\text{C}$$

maximizes $G(\mathcal{D}, F_j)$ compared to alternative splits on humidity, wind speed, or precipitation. Thus, the root node of the induced tree becomes a split node on temperature.

Subsequent splits can then refine the decision boundaries. For example, in the subset (X_1, Y_1) where Temperature $\leq 30°\text{C}$, humidity may provide

the next best split, separating mild but humid intervals from cooler, drier conditions. This illustrates how $G(\mathcal{D}, F_j)$ guides the recursive induction process towards a structure that progressively reduces impurity or entropy across the partitions.

2.2.1.2 Pseudo Algorithm

In this thesis, our focus is on binary decision trees. Binary decision trees are restricted to two child nodes. Instead of sorting the subsets based on their values of, e.g., a nominal feature, a splitting point θ is required.

A description of how to build a binary decision tree recursively is given in Algorithm 2.1. We closely follow the outlined pseudo code of Han et al. [2011]. First, the entire dataset is given to the algorithm, and it is checked whether the defined stopping criterion is met, e.g., the remaining sample size is below a specified threshold. If this is the case, a leaf node representing the class distribution is created and returned. Otherwise, the algorithm proceeds by selecting the feature that optimizes the splitting criterion, e.g., information gain or Gini impurity. Subsequently, a split node is created for this feature, and for each value in this feature, we recursively call the DecisionTree function with the subset of the data that matches the value in the selected feature. The *child_node* created in the call with the subset of the dataset is attached to the split node. Finally, the split node is returned.

Therefore, a decision tree model $\mathcal{T}$ is learned by calling the recursive decision tree function with $\mathcal{T} \leftarrow \text{DecisionTree}(\mathcal{D}, \mathcal{F})$.

2.2.1.3 Pruning

Pruning is a technique in decision tree algorithms designed to reduce the complexity of the learned tree model and prevent overfitting [Bramer, 2007]. Overfitting occurs when a decision tree is excessively complex. This typically occurs when the model captures noise from the training data, rather than learning the underlying patterns [Esposito et al., 1995]. Pruning improves the model's generalization by simplifying it, thereby enhancing its performance on unseen data [Esposito et al., 1995]. In the following, we describe the two primary categories of pruning, which were first introduced by Quinlan [1993]: *pre-pruning* (early stopping) and *post-pruning*.

Algorithm 2.1 Binary Decision Tree Induction

DecisionTree($\mathcal{D}$, $\mathcal{F}$)

if stopping criterion is met for $\mathcal{D}$ **then**
  return a leaf node L with the class distribution
else
  $F^*, \theta \leftarrow$ select the attribute and splitting point that optimizes the splitting criterion
  Create split node $S_{F^*,\theta}$ for F^* with θ
  $\mathcal{D}_{v \leq \theta} \leftarrow$ subset of $\mathcal{D}$ where $F^* \leq v$
  Attach DecisionTree($\mathcal{D}_{v \leq \theta}$, $\mathcal{F} \setminus \{F^*\}$) to $S_{F^*,\theta}$
  $\mathcal{D}_{v > \theta} \leftarrow$ subset of $\mathcal{D}$ where $F^* > v$
  Attach DecisionTree($\mathcal{D}_{v > \theta}$, $\mathcal{F} \setminus \{F^*\}$) to $S_{F^*,\theta}$
  return the split node S
end if

Pre-pruning, also known as early stopping, involves halting the tree construction process before it becomes fully developed. Several criteria have been introduced to limit the growth of a decision tree, such as maximum depth, minimum samples required in a node, or a minimum gain based on the splitting criterion.

Post-pruning, on the other hand, allows the decision tree to grow to its full depth and complexity initially and then prunes it back by removing branches that contribute little to the tree's predictive power.

Both pre-pruning and post-pruning methods are essential tools for managing the trade-off between the decision tree's accuracy and its ability to generalize [Zhou, 2021]. So, by effectively pruning a decision tree, it can be ensured that it generalizes well to new, unseen data [Han et al., 2011]. Furthermore, it can be stated that a smaller decision tree is also more interpretable, as the paths from the root to the leaf nodes become less complex in terms of the evaluated decisions.

Running Example. In the weather sensor network, pruning helps prevent overfitting to noise, e.g., from faulty sensor readings. A temporary spike in humidity, for instance, might otherwise create a meaningless split. With pre-pruning, the tree stops growing if the gain in $G(\mathcal{D}, F_j)$ is too small. With post-pruning, subtrees such as those based on precipitation can be removed if validation shows they do not improve predictive performance. This yields a simpler and more robust tree.

2.2.2 Prediction with Decision Trees

The prediction process using a decision tree is a straightforward traversal through the tree structure, guided by the values of the input features [Quinlan, 1993]. Given a trained decision tree model $\mathcal{T}$, the goal is to predict the output $\hat{y}$ for a new, unseen instance x by following a path from the root node to a leaf node, where the prediction is made by $\hat{y} = \mathcal{T}(x)$.

Following Han et al. [2011]'s explanation, when predicting with a decision tree, each internal node (including the root node) represents a split decision based on a specific feature and value from the input instance. The prediction process starts at the root node of the decision tree. Hereby, the first decision is evaluated. Depending on the feature-value combination of the input instance, we traverse the tree to the succeeding child node. If the next node is a split node, the procedure is repeated until a leaf node is reached. Once the leaf node of the decision tree is reached, the output is determined. This leaf either assigns the most frequent class among training samples or provides a probability distribution over possible classes [Murphy, 2012].

Formally, probabilities from a decision tree $\mathcal{T}$ are obtained by considering the distribution of class labels within the leaf node L_i reached by an instance x:

$$P_{\mathcal{T}}(y = c \mid x, L_i) = \frac{|L_{i,c}|}{|L_i|} \tag{2.2.3}$$

where $|L_{i,c}|$ denotes the number of instances of class c in leaf node L_i, and $|L_i|$ denotes the total number of instances in that node.

Running Example. Consider a decision tree trained on the weather sensor network to classify 5-minute intervals as *Hot* or *Mild*. A new observation arrives at 12:20 with the following values: temperature = 32°C, humidity = 35%, wind speed = 5 km/h.

The prediction process starts at the root node:

- Root split: *Temperature* $> 30°C$*?* $\rightarrow$ Yes, move to the right child.
- Next node: *Humidity* $\leq 40\%$*?* $\rightarrow$ Yes, move to the left child.
- Reached a leaf node L_i containing mostly *Hot* examples.

At this leaf, suppose 18 training instances ended up here during induction, of which 15 are labeled *Hot* and three are labeled *Mild*. The prediction probabilities are therefore

$$P_{\mathcal{T}}(y = \text{Hot} \mid x, L_i) = \tfrac{15}{18} = 0.83, \quad P_{\mathcal{T}}(y = \text{Mild} \mid x, L_i) = \tfrac{3}{18} = 0.17.$$

Thus, the model outputs *Hot* with high confidence for the 12:20 interval.

2.2.3 Hoeffding Trees

Hoeffding trees represent an extension to the previously introduced decision tree algorithm designed to handle streaming data. They are particularly suited for scenarios where data is continuously incoming and decisions must be made in real time [Domingos and Hulten, 2000]. The core advantage of Hoeffding trees lies in their ability to construct a model incrementally and update it as more data becomes available, all while guaranteeing that the model will be asymptotically nearly as accurate as a conventional decision tree built with all data available at once [Domingos and Hulten, 2000]. In the following, we adhere to the introduction and explanation of Hoeffding trees, as given in Domingos and Hulten [2000].

The theoretical foundation of Hoeffding trees is based on the Hoeffding bound, which quantifies the amount of data needed to make a decision with a certain level of confidence that the decision is close to the best possible decision that could be made given infinite data. The Hoeffding inequality [Hoeffding, 1994] is given by:

$$P(|\bar{x} - \mu| > \epsilon) \leq 2\exp(-2n\epsilon^2) \tag{2.2.4}$$

where μ is the true mean of a random variable, $\bar{x}$ is the observed mean, ϵ is a small threshold, and n is the sample size. In the context of decision trees, this bound helps determine whether enough data has been seen to make a reliable split decision at any node in the tree.

In the Hoeffding tree, the Hoeffding bound is used to decide if the difference in benefit between the best and the second-best feature is significant. If the difference is greater than a threshold determined by the bound, the

split is made. The threshold ϵ, is computed as

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}} \tag{2.2.5}$$

where R is the range of the attribute's benefit. In the case of information gain, this is 1 for binary classification and $log_2(|C|)$ for multiclass classification; when using Gini Impurity, it is 0.5 for binary classification and $1 - \frac{1}{|C|}$ for multiclass classification. δ is the probability of not choosing the best attribute, and n is the number of instances seen at the node.

As Hoeffding trees are required to process streaming data, i.e., one instance at a time, the algorithm must store sufficient statistics of the observed data instances at each leaf node. This is done via the n_{ijk} matrix, where i corresponds to a specific feature, j corresponds to a specific value of the feature F_i, and k corresponds to the class label. For each entry at i, j, k, the n_{ijk} indicates how many instances have been observed with this feature-values-class combination. This matrix enables the computation of class distribution for each attribute value, which can then be used to calculate Gini impurity or information gain.

Running Example. In the weather sensor network, suppose we want to split a node using the feature *humidity*, discretized into two values:

- v_1: Humidity $\leq 40\%$
- v_2: Humidity $> 40\%$

The prediction task is to classify each 5-minute interval as either *Hot* (c_1) or *Mild* (c_2).

During incremental training, the Hoeffding Tree maintains sufficient statistics in the form of the n_{ijk} matrix, where n_{ijk} counts the number of instances with feature F_i taking value v_j and belonging to class c_k. Assume the following counts have been observed at a specific node:

Humidity (F_i)	Class Hot (c_1)	Class Mild (c_2)
$\leq 40\%$ (v_1)	$n_{i,1,1} = 20$	$n_{i,1,2} = 10$
$> 40\%$ (v_2)	$n_{i,2,1} = 5$	$n_{i,2,2} = 15$

This means that for humidity $\leq 40\%$, the node has seen 20 instances

Algorithm 2.2 Hoeffding Tree Algorithm

Setup:
  Initialize tree $\mathcal{T}$ with a single leaf node L_0.
  Initialize n_{ijk} for all features F_i, values v_j, and class labels c_k at the root node L_0.
Process Instance:
  for $t = 1, 2, ...T$ **do**
    Pass instance x_t down the tree $\mathcal{T}$ until it reaches a leaf node L.
    Update n_{ijk} at leaf node L for the feature values in x_t and its class label y_t.
    $F^* \leftarrow$ feature with the highest splitting criterion.
    $F^{**} \leftarrow$ feature with the second-highest splitting criterion.
    Compute the Hoeffding bound:

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}$$

    if $G(n_{ijk}, F^*) - G(n_{ijk}, F^{**}) > \epsilon$ **then**
      Split the leaf node L into split node S_{F^*} using attribute F^*.
      Create child nodes $L_1, L_2, \ldots, L_m$ based on the values of F^*.
      Initialize sufficient statistics n_{ijk} for each child node.
    end if
  end for

labeled *Hot* and 10 labeled *Mild*, while for humidity $> 40\%$, the counts are 5 *Hot* and 15 *Mild*.

The Hoeffding Tree uses these accumulated counts to estimate splitting criteria (e.g., Information Gain or Gini Impurity) without storing the raw data stream. Once the difference between the best and second-best attributes exceeds the Hoeffding bound, the algorithm commits to a split using the sufficient statistics in the n_{ijk} matrix.

2.2.3.1 Pseudo Algorithm

We follow the description of the Hoeffding tree algorithm as given by Domingos and Hulten [2000], shown in Algorithm 2.2. The Hoeffding tree algorithm requires two initial tasks to be executed prior to learning from streaming data. A tree with a single leaf node L_0 is created, and within this leaf node, the n_{ijk} matrix is initialized for all features F_i, values v_j, and labels c_k. From this point on, single data instances x are passed into the tree and processed. Given that a new data instance arrives, it is passed

down the tree $\mathcal{T}$ until it reaches a leaf node L. Given the label and the values of the data instance, the n_{ijk} matrix is updated for the respective features. In a subsequent step, a splitting criterion, either Gini impurity or information gain is used to determine the best and the second best split, with e.g. $G(n_{ijk}, F^*)$, where the aggregate statistics in n_{ijk} can be used instead of X in traditional offline decision trees. Based on the seen instances n in a leaf node, the Hoeffding bound ϵ is computed. Suppose the benefit of the best computed splitting criterion minus the second best splitting criterion is greater than the Hoeffding bound ϵ. In that case, a split is made for the best feature-value combination. The leaf node at hand is then transformed to a split node S with the determined feature-value combination, and child leaf nodes with new n_{ijk} matrices are initialized based on the values of F^*.

2.2.3.2 Implementations of Hoeffding Trees

The Very Fast Decision Tree (VFDT) [Domingos and Hulten, 2000] is a specialized and most prominent version of the generalizable Hoeffding tree algorithm. The VFDT algorithm provides a set of operations that are beneficial over the standard Hoeffding tree algorithm, e.g., a tie-breaking mechanism, better memory management, and efficiency. Tie-breaking is, for example, required when features have a similar benefit. This might encompass two similar highly informative features, which would lead to indecision and stagnation in growth for the Hoeffding tree, as it would not be able to make a split, or rather, very late. VFDT also employs growth-limiting methods to minimize memory consumption and pruning of sufficient statistics for features that are deemed irrelevant. A smaller extension to the VFDT algorithm is the VFDTc algorithm [Gama et al., 2003], which enables the VFDT algorithm to handle continuous features. Another extension CVFDT to VFDT is aimed at handling concept drift [Hulten et al., 2001].

2.2.4 Decision Tree Interpretability

As previously noted and introduced by Han et al. [2011], decision trees provide clear and visual paths from the root to the leaf nodes, indicating how individual features contribute to the decision-making process. Each internal node represents a decision based on a feature value, while branches indicate the outcome of the decision, leading to the next step in the path

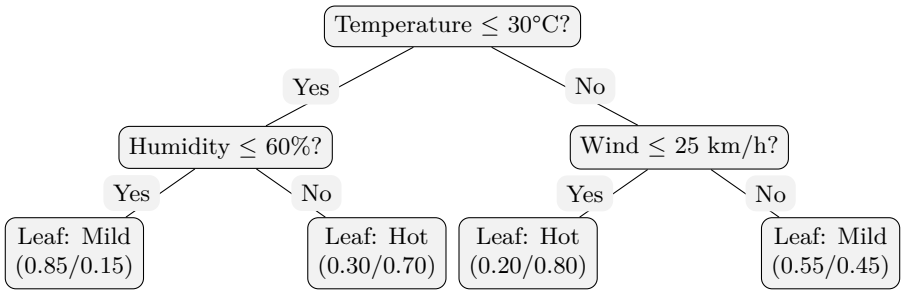


Figure 2.3: Example of an interpretable decision tree for the weather sensor network.

until a final decision (leaf node) is reached. This straightforward structure makes it easy to trace how a model arrived at a particular decision. Key elements for interpretation include:

- **Feature Importance**: Decision trees allow easy extraction of feature importance scores, showing which features play a significant role in predicting the outcome [Lundberg and Lee, 2017].
- **Path Analysis**: One can follow the path from the root to a leaf node to understand the combination of feature values that lead to a specific decision [Breiman, 2017].
- **Rule Extraction**: Each path can be converted into a set of human-readable rules, facilitating the understanding of the decision-making process [Rokach and Maimon, 2008].

Running Example. Figure 2.3 shows a small decision tree learned from 5-minute intervals in the weather sensor network. Each internal node stores the feature and threshold, i.e., *Temperature* $\leq$ *30°C?*, then *Humidity* $\leq$ *60%?* or *Wind* $\leq$ *25 km/h?*). Each leaf node contains the class distribution observed during training (shown as *(Mild/Hot)*), from which either a majority label or a probability can be derived.

Feature importance. Even without formal feature importance metrics, the root split (`Temperature` $\leq 30°$C`?`) is clearly the most influential since it partitions all observations first. Subsequent splits on humidity and wind refine the classification, but their effects are limited to specific subsets of the data.

Path analysis. To explain why a particular 5-minute interval is predicted as *Hot* in the tree, one can trace the path:

$$\texttt{Temperature} > 30^{\circ}\text{C} \;\rightarrow\; \texttt{Wind} \leq 25 \text{ km/h} \;\Rightarrow\; \texttt{Hot}.$$

Rule extraction. Transforming the tree into *IF–THEN* rules provides interpretable explanations for people with the respective domain knowledge. The following rules can be extracted from the example decision tree:

R1: $\texttt{Temp} \leq 30^{\circ}\text{C}$ **and** $\texttt{Humidity} \leq 60\% \Rightarrow \texttt{Mild}$ (0.85),

R2: $\texttt{Temp} \leq 30^{\circ}\text{C}$ **and** $\texttt{Humidity} > 60\% \Rightarrow \texttt{Hot}$ (0.70),

R3: $\texttt{Temp} > 30^{\circ}\text{C}$ **and** $\texttt{Wind} \leq 25$ km/h $\Rightarrow \texttt{Hot}$ (0.80),

R4: $\texttt{Temp} > 30^{\circ}\text{C}$ **and** $\texttt{Wind} > 25$ km/h $\Rightarrow \texttt{Mild}$ (0.55).

In general, these methods can be applied to gain an understanding of the decision tree and the underlying data from which it was derived. As stated in Zhang et al. [2021], the core of interpretability lies in providing explanations that offer insights into why a decision was made.

2.3 Random Forests

Random forests are an ensemble learning method extensively used for classification and regression tasks in machine learning. The concept builds upon the *random subspace method for constructing decision forests* introduced by Ho [1998], which was further refined by Breiman [2001]. Random forests extend the concept of decision trees by constructing multiple trees during training and aggregating their results. This ensemble approach enhances predictive accuracy and controls overfitting by reducing variance through averaging without significantly increasing bias [Hastie et al., 2009]. We base our description of the random forest algorithm on Hastie et al. [2009] and Breiman [2001].

2.3.1 Fundamental Principle

The core idea behind random forests involves two key concepts: bootstrap aggregating (bagging) and random feature selection [Breiman, 2001]. Ac-

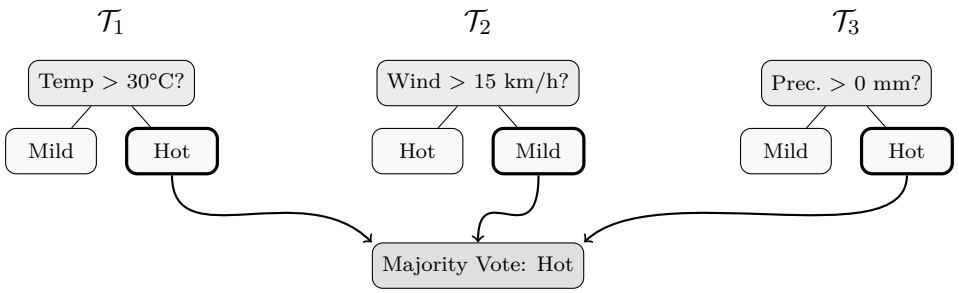


Figure 2.4: Random forest for weather sensor network.

cording to Hastie et al. [2009], bagging is a technique that creates multiple subsets of data from the original dataset by sampling with replacement. Each decision tree in the forest is trained on a different bootstrap sample, ensuring diversity among the trees. This randomness reduces the variance of the model, as averaging multiple trees tends to smooth out anomalies and mitigate the effects of overfitting inherent in single decision trees.

Random feature selection at every split node further enhances diversity by altering the feature space for each tree [Breiman, 2001]. At each node within a tree, a random subset of features is selected, and the best split is found only within this subset [Breiman, 2001]. This process decorrelates the trees, preventing any single strong predictor from dominating the model across all trees and allowing weaker predictors to contribute, hence capturing a broader range of patterns [Hastie et al., 2009].

Given a training dataset (X, Y) the random forest algorithm constructs B decision trees $\{\mathcal{T}_b\}_{b=1}^{B}$. Each tree $\mathcal{T}_b$ is trained on a bootstrap sample (X_b, Y_b) drawn from (X, Y) with replacement. The final prediction $\hat{y}$ for a new instance x is determined by majority voting:

$$\hat{y} = \text{mode}\,\{\mathcal{T}_b(\mathbf{x})\}_{b=1}^{B} \tag{2.3.1}$$

At each node within a tree, random feature selection involves selecting a subset of m features from the total D features ($m \ll D$). The best split is then found by considering only these m features.

Running Example. To illustrate the principle of random forests, assume we want to build an ensemble of three trees ($B = 3$) for the weather classification task. Each tree is trained on a different bootstrap sample of the

data, ensuring that no two trees see the exact same training set. The final random forest is depicted in Figure 2.4.

Bagging. Suppose we build a random forest with $B = 3$ trees. From the original training set of 100 intervals, we create three bootstrap samples by sampling with replacement:

- $\mathcal{T}_1$ is trained on a bootstrap sample where some intervals appear multiple times and others are omitted.
- $\mathcal{T}_2$ is trained on a different bootstrap sample.
- $\mathcal{T}_3$ is trained on yet another bootstrap sample.

This ensures that each tree sees slightly different training data, increasing diversity.

Random Feature Selection. At each node, instead of considering all four features, a random subset is chosen (e.g., $m = 2$ features):

- In $\mathcal{T}_1$, only {Temperature, Humidity} are considered. The best split is on Temperature.
- In $\mathcal{T}_2$, only {Wind speed, Precipitation} are considered. The best split is on Wind speed.
- In $\mathcal{T}_3$, only {Temperature, Precipitation} are considered. The best split is on Precipitation.

This randomness prevents the same strong feature (e.g., Temperature) from dominating every tree, encouraging weaker predictors like Wind speed or Precipitation to also contribute.

Prediction. Now, consider a new observation at 12:20 with values: Temperature = 32°C, Humidity = 35%, Wind speed = 12 km/h, Precipitation = 0 mm.

- $\mathcal{T}_1$ predicts: *Hot*
- $\mathcal{T}_2$ predicts: *Mild*
- $\mathcal{T}_3$ predicts: *Hot*

The final random forest prediction is determined by majority vote:

$$\hat{y} = \text{mode}\{\textit{Hot}, \textit{Mild}, \textit{Hot}\} = \textit{Hot}.$$

Algorithm 2.3 Random Forest Algorithm

```
1: for b = 1 to B do
2:    Generate a bootstrap sample (X_b, Y_b) by sampling n instances from
      D with replacement.
3:    Initialize the root node with (X_b, Y_b).
4:    while stopping criterion not met do
5:       for each internal node do
6:          Randomly select m features from the total D features without
            replacement.
7:          Determine the best split among the m features based on the cho-
            sen splitting criterion.
8:          Split the node into two child nodes based on the best split.
9:       end for
10:   end while
11: end for
```

2.3.2 Pseudo Algorithm

The pseudo-code algorithm presented in Algorithm 2.3 is based on Hastie et al. [2009]. To generate a random forest, we repeat the following procedure B times, where B is the parameter that determines how many bootstrap samples (X_b, Y_b) and hence trees in the ensemble are created. The size of the bootstrap sample, which is sampled with replacement from the dataset $\mathcal{D}$, is determined by the n parameter. Given the drawn bootstrap sample, the tree is then built similarly to the introduced decision tree (cf. Algorithm 2.1), with the exception that at every split node, only a subset of the features is regarded. The size of the subset of the features is specified with the m parameter.

2.3.3 Random Forest Extensions

Random forests have been extensively studied, and several extensions and adaptations to specific scenarios have been proposed [Cutler et al., 2012, Kulkarni and Sinha, 2012]. In this section, we introduce two extension principles that are relevant to our proposed approaches.

2.3.3.1 Random Forest of Stumps

In a random forest, the tree depth of the decision trees used is generally not restricted. A specialized version of the random forest uses decision stumps,

i.e., decision trees with a single decision node [Iba and Langley, 1992], to build a forest of stumps [Alharthi et al., 2018]. Learning a random forest with decision stumps has the benefit that it is less complex and, hence, easier to compute and more interpretable. On the downside, decision stumps are less predictive than deeper trees, resulting in lower individual accuracy.

2.3.3.2 Weighted Random Forests

Weighted Random Forests (WRFs) are an extension of the traditional random forest algorithm, where each decision tree in the forest is assigned a specific weight [Winham et al., 2013]. These weights are used to adjust the contribution of each tree's prediction to the model's final output. WRFs provide additional flexibility and interpretability compared to standard random forests, especially in scenarios where the predictive reliability or importance of individual trees varies across subsets of the data.

When integrated with random forests, WRFs provide a framework for understanding how each tree contributes to the overall prediction. Generally, WRFs are very similar to random forests with a weight component w_b for each tree in the ensemble. In this context, the random forest can be seen as an ensemble where each decision tree has a weighted contribution that can be expressed as:

$$\hat{y} = \sum_{b=1}^{B} w_b * (\mathcal{T}_b(x)) \tag{2.3.2}$$

Various voting mechanisms exist [Rokach, 2010] that vary depending on if the given task is a classification or regression task. In this thesis, we use distribution summation, introduced by Clark and Boswell [1991]:

$$\hat{y} = \underset{c \in \mathcal{Y}}{\operatorname{argmax}} \sum_{b=1}^{B} P_{\mathcal{T}_b}(y = c|x). \tag{2.3.3}$$

Hereby, the idea is to sum over the conditional probability vector, which can be obtained from each classifier, indicating the probability of each class given the input vector x. The selected class is then chosen according to the highest value in the total vector [Rokach, 2010]. This requires the decision trees used in the ensemble to be able to predict class probabilities, which can be considered an inherent feature of decision trees.

2.3.4 Random Forest Interpretability

While random forests are more complex to interpret compared to decision trees, there is still active research going on exploring opportunities to interpret random forest models [Haddouchi and Berrado, 2024]. Generally, the methods provided can be distinguished into four categories, as of Haddouchi and Berrado [2024]:

- **Size reduction:** Reduce the number of RF trees to produce a simpler model, which facilitates interpretation.
- **Rule extraction:** Extract a representative set of rules for approximating the RF model. Interpretation based on the set of extracted rules.
- **Feature oriented:** Explain how changes in the values of the descriptive features affect the predictions.
- **Sample similarity-based:** Investigate the similarity among instances to explain predictions.

Moreover, feature importance can be easily inferred in random forests, providing another angle from which to interpret the model and the impact of the features on the final decision [Breiman, 2001]. Feature importance can also be easily calculated for WRFs, following the same scheme as for random forests, but attributing more or less importance according to the weights [Winham et al., 2013]. Using decision stumps as weak learners in the random forest also allows for more interpretability, as the decisions for each weak learner are less complex and, hence, easier to comprehend [Hastie et al., 2009].

2.4 Evaluation

Evaluating online machine learning models poses distinct challenges compared to traditional batch learning models due to the dynamic nature of data streams and the need for models to adapt continuously [Gama et al., 2013]. This chapter introduces strategies and metrics designed to evaluate the performance of online machine learning algorithms with respect to changes in the feature space. Evaluations must be not only accurate and

reflective of real-time data conditions, but also robust enough to handle changes in the feature space.

2.4.1 Experimental Design

In the following, we describe the general experiment setup of online machine learning evaluation. It provides the framework for evaluating open feature spaces. Subsequently, we describe how monotonically increasing feature spaces and varying feature spaces are simulated within this framework.

2.4.1.1 Online Learning Environments

In the work Gama et al. [2013], the setup is formalized as follows. Consider a potentially infinite data stream $\{(\mathbf{x}_t, y_t)\}_{t=1}^{\infty}$, where $\mathbf{x}_t \in \mathcal{X}$ is the feature vector observed at time t, and $y_t \in \mathcal{Y}$ is the corresponding ground-truth label. Let $\mathcal{X}$ be the input space and $\mathcal{Y}$ be the output space.

An online learning algorithm maintains a model $\hat{f}_t$ at each time step t. The key characteristic of online learning is that the model is updated sequentially as new instances arrive, without reprocessing the entire historical dataset [Cesa-Bianchi and Lugosi, 2006].

In this thesis, we follow the prequential (predict-then-update) error mechanism, which continuously assesses performance as new data arrives, without needing separate training and test sets [Gama et al., 2013]. Generally, the data used in the evaluation of online learning algorithms are pre-existing datasets with fixed instances, as it would be essentially impossible to evaluate a truly infinite data stream. Hence, in the evaluation, many times it is resorted to simulate the online learning environment by giving the machine learning algorithm one instance at a time, use the result in the evaluation, and finally update the model, following the aforementioned predict-then-update mechanism.

At time t, the model $\hat{f}_{t-1}$ receives an instance (x_t, y_t). Before updating, it produces a prediction:

$$\hat{y}_t = \hat{f}_{t-1}(x_t). \tag{2.4.1}$$

Once the prediction is made, the ground-truth label y_t is revealed, and the model updates its parameters:

$$\hat{f}_t = \text{Update}(\hat{f}_{t-1}, (x_t, y_t)). \tag{2.4.2}$$

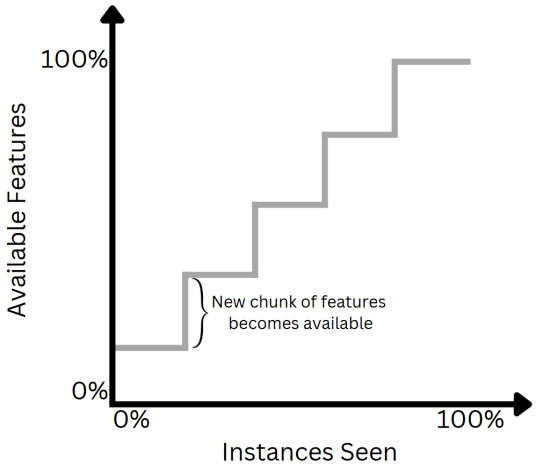


Figure 2.5: Experiment simulation for monotonically increasing feature spaces, where new chunks of the feature space become available as time progresses.

Running Example. In the weather sensor network, new measurements arrive every 5 minutes in the form of temperature, humidity, wind speed, and precipitation values. The task is to classify each interval as *Hot* or *Mild*.

At time step t, the online model $\hat{f}_{t-1}$ receives a new observation, for example at 12:20:

$$x_t = (32^{\circ}\mathrm{C}, 35\%, 12\ \mathrm{km/h}, 0\ \mathrm{mm}).$$

Before updating, the model predicts a class:

$$\hat{y}_t = \hat{f}_{t-1}(x_t).$$

Suppose the model predicts *Hot*. Afterwards, the ground-truth label y_t from the sensor annotation is revealed, e.g., $y_t = \mathit{Hot}$, and the provided learning algorithm can now use this information to update its parameters. Over time, this predict-then-update cycle repeats for each 5-minute interval. The accumulated sequence of prediction errors provides an online evaluation of model performance.

2.4.1.2 Simulation of Monotonically Increasing Feature Spaces

Monotonically increasing feature spaces are characterized by additional features that emerge later on in the learning process. Hence, not every feature is available at the start of the process. As we utilize the online learning evaluation framework to evaluate monotonically increasing feature spaces, which employ fixed feature spaces, we are also required to simulate the emergence of features. In the literature, it has been established to add chunks of features over time [Zhang et al., 2015, 2016]. An example of such a simulation setup is given in Figure 2.5.

Consider the same setting as in an online learning environment with T as the size of the dataset that is simulated, $\{(\mathbf{x}_t, y_t)\}_{t=1}^{T}$, where $\mathbf{x}_t \in \mathcal{X}_t$ is the feature vector observed at time t, and $y_t \in \mathcal{Y}$ is the corresponding ground-truth label. Let $\mathcal{X}_t \subseteq \mathbb{R}^{D_t}$ be the input space at time t with dimensionality d_t and $\mathcal{Y}$ be the output space. After a number of observed instances m, the input space is extended so that

$$D_{t+m} \geq D_t \tag{2.4.3}$$

holds.

Running Example. In the weather sensor network, not all features are necessarily available at the beginning of the stream. For example, at the start of deployment, only temperature and humidity sensors may be installed at all stations. Thus, the input space $\mathcal{X}_t$ at time t initially has dimensionality $d_t = 2$.

Following the online learning framework, every 5-minute interval provides a new instance (x_t, y_t), such as

$$x_t = (28°\mathrm{C}, 60\%), \quad y_t = \textit{Mild}.$$

The model $\hat{f}_{t-1}$ makes a prediction before updating, and performance is evaluated prequentially.

After m time steps, new sensors are added to the network, e.g., wind speed sensors. The dimensionality of the feature space increases to $d_{t+m} = 3$.

Each subsequent observation now includes the additional feature:

$$x_{t+m} = (30°\text{C}, 55\%, 14 \text{ km/h}), \quad y_{t+m} = \textit{Mild}.$$

Later in the process, precipitation sensors may be deployed, further increasing the dimensionality to $d_{t+m+k} = 4$.

As not every feature is available from the beginning, this scenario illustrates a *monotonically increasing feature space*. In practice, the online evaluation must simulate this setting by revealing new chunks of features at specific points in time, as shown in Figure 2.5. The learner must then adapt its model incrementally to exploit the richer input space while still making prequential predictions at every time step.

2.4.1.3 Simulation of Varying Feature Spaces

Varying feature spaces are characterized by features that emerge later in the learning process, as well as by features that are vanishing or missing at random. We again use an online learning environment, where we have a pre-determined length of input observations. In the literature, it has been established that a certain fraction of features is removed at every time step [Beyazit et al., 2019, He et al., 2021b], thereby simulating the emergence of previously unseen features and the vanishing of previously observed features. Commonly, removal ratios of 0.25, 0.5, and 0.75 are used Beyazit et al., 2019.

We again consider the same base setting as in an online learning environment with T as the size of the dataset that is simulated, $\{(\mathbf{x}_t, y_t)\}_{t=1}^{T}$, where $\mathbf{x}_t \in \mathcal{X}_t$ is the feature vector observed at time t, and $y_t \in \mathcal{Y}$ is the corresponding ground-truth label. Let $\mathcal{X}_t \subseteq \mathbb{R}^{D_t}$ be the input space at time t with dimensionality d_t and $\mathcal{Y}$ be the output space. At every step t, a predefined removal chance r checks if a feature is omitted in x_t. Given that $|\mathcal{F}|$ is the total amount of features available in the simulated dataset, then

$$D_t \approx r * |\mathcal{F}| \tag{2.4.4}$$

at every step of t, where the three cases, i.e. $D_{t+1} = D_t$, $D_{t+1} > D_t$, and $D_{t+1} > D_t$, for varying feature spaces hold.

The example given in Figure 2.6 describes the scenario of a simulated varying feature space with eight features and a predetermined dataset length

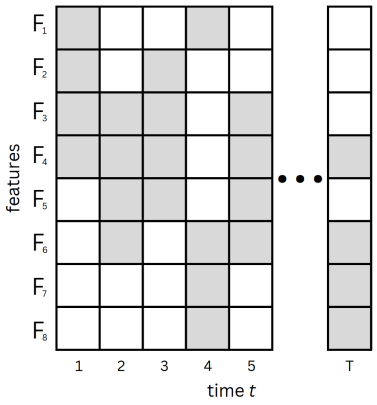


Figure 2.6: Simulated evaluation of Varying Feature Spaces, gray boxes indicate that a feature is available at time t.

of T. The removal ratio is set to $r = 0.5$ in this case. It can be observed that the feature space initially encompasses the first four features and then evolves, eventually encompassing all features at a later point.

Running Example. In the weather sensor network, not all sensors provide continuous measurements over time: some sensors may break down temporarily or permanently, while new ones may be deployed. This situation characterizes a *varying feature space*, where features both appear and vanish as the stream progresses.

Consider a setting with four sensors: temperature, humidity, wind speed, and precipitation. At the beginning of the stream ($t = 1$), the available feature vector might be

$$x_1 = (Temp. = 28°\text{C}, Humidity = 60\%), \quad y_1 = \textit{Mild},$$

where only temperature and humidity are observed.

At the next step ($t = 2$), wind speed measurements become available, but humidity data are missing due to a temporary sensor failure:

$$x_2 = (Temp. = 30°\text{C}, Humidity = ?, Wind = 12 \text{ km/h}), \quad y_2 = \textit{Mild}.$$

Later in the stream ($t = 3$), precipitation sensors are deployed, while

wind speed readings vanish:

$$x_3 = (Temp. = 32°\text{C}, Humidity = 55\%, Precipitation = 2 \text{ mm}), \quad y_3 = Hot.$$

This reflects realistic conditions in sensor networks, where hardware failures, maintenance, or gradual network expansion lead to dynamically changing input spaces. The evaluation framework, as shown in Figure 2.6, captures these dynamics by controlling the fraction of available features over the course of the simulated data stream.

2.4.2 Evaluation Metrics

In the context of evaluating online learning algorithms, the concepts of accuracy, cumulative error, and cumulative error rate provide a framework for quantifying and interpreting model performance as data arrives incrementally [Cesa-Bianchi and Lugosi, 2006]. These metrics are particularly relevant to gauge performance when the data-generating process is continuous and evolving.

Accuracy is one of the most straightforward performance metrics. For a given set of predictions, $\{\hat{y}_t\}_{t=1}^{T}$ and corresponding ground-truth labels $\{y_t\}_{t=1}^{T}$, accuracy at a specific time T is defined as the proportion of instances up to time T for which the model's prediction is equivalent to the actual label. More formally, we can define an indicator function $\mathbb{I}(\hat{y}_t = y_t)$ that is one if the model's prediction is correct and zero otherwise. Thus, the accuracy over the first T instances is:

$$Accuracy(T) = \frac{1}{T} \sum_{t=1}^{T} \mathbb{I}(\hat{y}_t = y_t) \tag{2.4.5}$$

When accuracy is computed incrementally as instances arrive, it reflects how consistently the model makes correct predictions over time. A rising or stable high accuracy as $T \to \infty$ suggests the model generalizes well and adapts effectively to new data [Hoi et al., 2021a].

Cumulative error is an indicator of the absolute errors made by the model. For a given set of predictions $\{\hat{y}_t\}_{t=1}^{T}$ and corresponding ground-truth labels

$\{y_t\}_{t=1}^{T}$, cumulative error at a specific time T is defined as the total amount of instances up to time T for which the model's prediction is *not* equivalent to the true label. Hereby, we use the inverse indicator function $\mathbb{I}(\hat{y}_t \neq y_t)$ that is one if $\hat{y}_t \neq y_t$ and zero otherwise.

$$CE(T) = \sum_{t=1}^{T} \mathbb{I}(\hat{y}_t \neq y_t) \tag{2.4.6}$$

The cumulative error was used in earlier works on varying feature spaces [Beyazit et al., 2019]. However, the downside becomes apparent as it is required to mentally set it in relation to T to gain an understanding of the model's performance. Hence, in more recent research, such as He et al. [2021b], the cumulative error rate was used.

Cumulative error rate describes proportional errors made by the model in relation to the observed instances [Cesa-Bianchi and Lugosi, 2006]. For a given set of predictions $\{\hat{y}_t\}_{t=1}^{T}$ and corresponding ground-truth labels $\{y_t\}_{t=1}^{T}$, the cumulative error rate at a specific time T is defined as proportion of instances up to time T for which the model's prediction is *not* equivalent to the actual label.

$$CER(T) = \frac{1}{T} \sum_{t=1}^{T} \mathbb{I}(\hat{y}_t \neq y_t) \tag{2.4.7}$$

Essentially, the cumulative error rate can be seen as the inverse of the accuracy. Rather than providing a positive interpretation of the outcome, it offers insight into the potential risk associated with a particular prediction.

Running Example. In the weather sensor network, suppose the task is to classify each 5-minute interval as *Hot* or *Mild*. Consider the first $T = 5$ observations in the stream:

Time t	Ground-truth y_t	Prediction $\hat{y}_t$	$\mathbb{I}(\hat{y}_t = y_t)$
1 (12:00)	Hot	Hot	1
2 (12:05)	Mild	Hot	0
3 (12:10)	Mild	Mild	1
4 (12:15)	Hot	Mild	0
5 (12:20)	Hot	Hot	1

From these outcomes:

- **Accuracy:** Out of five intervals, three predictions are correct, giving

$$Accuracy(T = 5) = \frac{3}{5} = 0.6.$$

- **Cumulative error:** Two predictions are wrong (at $t = 2$ and $t = 4$), so

$$CE(T = 5) = 2.$$

- **Cumulative error rate:** Proportionally,

$$CER(T = 5) = \frac{2}{5} = 0.4,$$

  which is simply the inverse of the accuracy.

As more data arrive, these metrics are updated continuously, reflecting how well the online model adapts to the evolving sensor data.

Part II

Contributions

Chapter 3

Learning from Monotonically Increasing Feature Spaces

In this chapter, we present a Hoeffding tree extension called Dynamic Fast Decision Tree (DFDT). The proposed approach can handle monotonically increasing feature spaces by continuously restructuring the decision tree, moving more important nodes to the top. This yields the advantage that the tree is more reflective of the feature importance of the given task and is faster to adapt, which leads to better performance.

We proceed by first presenting the approach in Section 3.1. We conduct this in a top-down manner, first presenting the overall algorithm for building the Dynamic Fast Decision Tree and then providing a detailed description of the required components. Subsequently, we relate this approach to two relevant research areas: Learning from monotonically increasing feature spaces and Hoeffding tree pruning methods in Section 3.2 and evaluate it with regard to emerging features in Section 3.3. This chapter concludes with a discussion in Section 3.4, which puts the requirements (cf. Section 1.2) into perspective.

3.1 Dynamic Fast Decision Tree

The Dynamic Fast Decision Tree learning algorithm was published in Schreckenberger et al. [2020]. This approach utilizes the concept of Hoeffding trees in the same manner as the established VFDT algorithm (cf. Section 2.2.3),

Parameter	Description
δ	probability of not choosing the right feature to split on
ρ	probability of choosing the wrong test in pruning
τ	tie-breaking for two similarly beneficial splits
n_{min}	split evaluation interval
m_{min}	restructure interval

Table 3.1: Parameters used in the DynamicFastDecisionTree Algorithm (cf. Algorithm 3.1).

but provides additional tools to enable it to handle monotonically increasing feature spaces. This is done by dynamic restructuring and efficient pruning of the tree. In the following, we will introduce the notation used in the algorithm and then describe the components and algorithms required for restructuring and pruning.

3.1.1 Notation Dynamic Fast Decision Tree

In our notation, we denote a tree by $\mathcal{T}$, with its left and right subtrees represented as $\mathcal{T}_{\text{left}}$ and $\mathcal{T}_{\text{right}}$, respectively. A split decision node that splits on feature F_i is denoted by S_{F_i}, and the collection of all split decision nodes is written as $\mathbf{S}$. Similarly, a single leaf node is denoted by L, while $\mathbf{L}$ represents the collection of leaf nodes.

As detailed in Section 2.2.3, Hoeffding tree approaches utilize the n_{ijk} matrix to track statistics, where the indices i, j, and k correspond to the i-th feature, the j-th value, and the k-th class, respectively. Although this notation does not fully align with our earlier definitions of referring to the d-th feature with d instead of i, it is the established standard in the context of Hoeffding trees, and we adopt it in this approach for compliance. Additionally, we denote the n_{ijk} matrix associated with a specific leaf node L by $L_{n_{ijk}}$.

3.1.2 Learning with DFDT

The list of parameters is given in Table 3.1. The input parameters δ, n_{min}, and τ are the same as for VFDT. They adjust the probability of not choosing the right feature to split on after a certain number of observed instances is evaluated for splits, and a tie-breaking mechanism is used for two

Algorithm 3.1 Learning with DynamicFastDecisionTree

1: Initialize root node L_0 in $\mathcal{T}$
2: Initialize n_{ijk} in L_0
3: **for** $t = 1, 2, \ldots, T$ **do**
4: receive instance (x_t, y_t)
5: pass (x_t, y_t) down the tree until a leaf L is reached
6: update statistics in n_{ijk}
7: **if** data instances seen since last restructure $\geq m_{min}$ **then**
8: restructure and prune tree if required
9: **end if**
10: **if** more than n_{min} instances arrived since last evaluation in L **then**
11: $F^* \leftarrow$ feature with the highest split criterion $IG(L_{n_{ijk}}, F_i)$.
12: $F^{**} \leftarrow$ feature with the second-highest $IG(L_{n_{ijk}}, F_i)$.
13: Compute the Hoeffding bound:

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}$$

14: **if** $IG(F^*) - IG(F^{**}) > \epsilon$ **then**
15: Replace former leaf L with split node S_{F_i}
16: Create child nodes L_{left}, L_{right}, attach to S_{F_i}
17: Initialize sufficient statistics n_{ijk} for both child nodes
18: **end if**
19: **end if**
20: **end for**

equally good splits, respectively. In addition, ρ determines the probability of choosing the wrong test for pruning, and a restructure interval m_{min} has to be specified. If the restructure interval m_{min} is set to infinity, both approaches, VFDT and DFDT, are essentially the same as restructuring never takes place. If m_{min} is set to a finite number, the tree is first restructured and then pruned, yielding an optimized tree that can be used for further training. In the description and the evaluation, we assume Boolean features.

Algorithm 3.1 provides a high-level overview of the learning algorithm. After a root node and its respective n_{ijk} matrix are initialized, data instances are fed to the learner. The data instances are then passed down the tree until a leaf node is reached, where the counts n_{ijk} are being updated (cf. Section 2.2.3 for an explanation of n_{ijk}). A trivial but necessary step in extending the Hoeffding tree algorithm for handling monotonically increasing feature spaces is to adapt the n_{ijk} matrix to the new features. This can be done

trivially by adding a new feature column in the i-axis of the matrix to then account for the emerged feature. Subsequently, it is checked if the tree needs to be restructured, which is checked periodically after m_{min} observed instances. Once n_{min} instances are seen at a leaf node, an evaluation based on the Hoeffding bound is carried out to determine whether it can be split to grow the tree. This is repeated every time n_{min} data instances are seen in a leaf node.

Algorithm Roadmap The proposed approach for handling emerging features is to restructure the decision tree in such a way that split nodes yielding high information gain are placed close to the root, as described in Section 3.1.2.2. Split nodes promising low information gain, on the other hand, should be close to the leaves. Before the restructuring of a (sub-)tree can take place, Section 3.1.2.1 illustrates why the information gain has to be calculated locally and how this is done.

After the restructuring process, the tree is pruned, i.e., decision nodes are revisited, and a leaf node replaces those yielding very little to no information gain. After pruning, it may be required to merge the leaf nodes. This component is introduced in Section 3.1.2.3.

3.1.2.1 Information Gain

Calculating a feature's information gain once and storing it globally is insufficient because the same feature can yield different information gains in different subtrees. This variability often arises from dependencies among features or from differences in the data aggregated at each leaf.

Consider the motivating example in Table 3.2. Hereby, a n_{ijk} matrix is given for a binary classification problem with two Boolean features. We calculate the different information gains for F_1 to showcase the need for determining information gain locally.

If no split has occurred yet and all data instances with label counts $(55, 55)$ are in one leaf node, the information gain is

$$IG(n_{ijk}, F_1) = \mathrm{H}(55, 55) - \frac{6}{11}\mathrm{H}(30, 30) - \frac{5}{11}\mathrm{H}(25, 25) = 0 \tag{3.1.1}$$

because the entropy of $(55, 55)$ does not decrease with splits into $(30, 30)$ and $(25, 25)$.

Label Counts ($c = 1, c = 2$)	$F_1 = \textbf{True}$	$F_1 = \textbf{False}$	$\sum$
$F_2 = \textbf{True}$	$(15, 15)$	$(25, 0)$	$(40, 15)$
$F_2 = \textbf{False}$	$(15, 15)$	$(0, 25)$	$(15, 40)$
$\sum$	$(30, 30)$	$(25, 25)$	$(55, 55)$

Table 3.2: Counts for labels ($c = 1, c = 2$) with dependent features

Taking now into account the dependence between the features, the same split conditional to $F_2 = \textbf{True}$ yields

$$IG(n_{ijk}, F_1 | F_2 = \textbf{True}) = \mathrm{H}(40, 15) - \frac{6}{11}\mathrm{H}(15, 15) - \frac{5}{11}\mathrm{H}(25, 0) \approx 0.3. \tag{3.1.2}$$

Therefore, it would be wrong to assume F_1 yields an information gain of zero anywhere in the tree. To avoid working with incorrect information gain, it is calculated when required using statistics held in the leaf nodes of the corresponding subtree, as shown in Algorithm 3.2.

The algorithm requires an array of all leaf nodes $\mathbf{L}$ of the (sub-)tree as input. As a first step, we initialize an array $\mathbf{G}$ to collect the information gains we calculate for each feature that occurs in the given array of leaf nodes $\mathbf{L}$. For each feature F_i, we then initialize a variable $\bar{n}_{ijk}$. This variable is used to aggregate the statistics for this feature F_i that is present in the leaf nodes. After the aggregation, we can then calculate the information gains for each feature and store them in the $\mathbf{G}$ array. After completing these computations, we finally return the array with the information gains of the respective features.

3.1.2.2 Restructuring Decision Trees

The goal of the restructuring process is to create a new arrangement of edges and nodes in the decision tree, with decreasing information gains from the root node to the leaf nodes, thereby representing feature importance more accurately. After restructuring, the tree should produce the same predictions as before. As long as each leaf node preserves the split nodes with their respective decisions in its path from the root, the model's outputs remain unchanged. The restructured tree may then have the advantage of pruning

Algorithm 3.2

Input:	**L**	array of all leaf nodes of a tree or subtree
Output:		information gains on all data instances in leafs in **L**

LocalInformationGain(**L**)

```
1: initialize array G for information gain of each feature
2: for each feature F_i represented in L do
3:    initialize n̄_ijk to aggreagte statistics
4:    for each leaf node L ∈ L do
5:       n̄_ijk ← n̄_ijk + L_{n_ijk}
6:    end for
7:    G[i] ← IG(n̄_ijk, F_i)
8: end for
9: return G
```

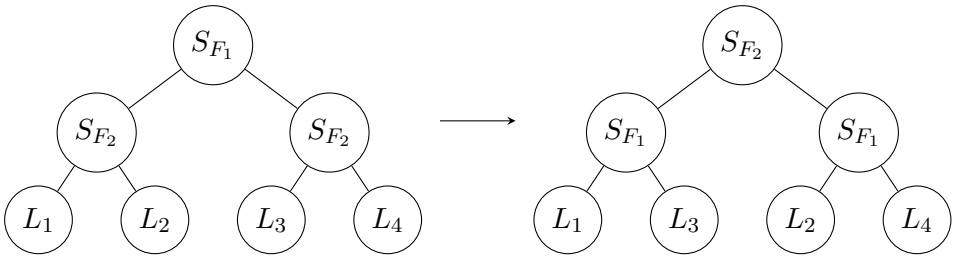


Figure 3.1: Tree Restructuring - All leaves with same split decisions $\mathbf{S} = \{S_{F_1}, S_{F_2}\}$

nodes that are no longer required (cf. Section 3.1.2.3). In simple cases, this restructuring is essentially a matter of rearranging nodes. However, more complex trees can be reorganized to push certain splits as high as possible, resulting in a generally more efficient structure.

In each of the following cases, split nodes S_{F_i} split on the same feature F_i. For simplicity, the higher the index of the feature, the closer it should be to the root; for example, F_2 is a more important feature than F_1. The *split decision test set* **S** (sometimes referred to simply as the *split decisions*) of a leaf node refers to the set of features that are used in the path from root to the leaf.

The simplest restructuring case occurs when the same split decisions are present for all leaf nodes (Figure 3.1). This way, any order of features can be achieved. After rearranging the decision nodes, the leaves can be reattached in the appropriate place. The approach works on trees of any

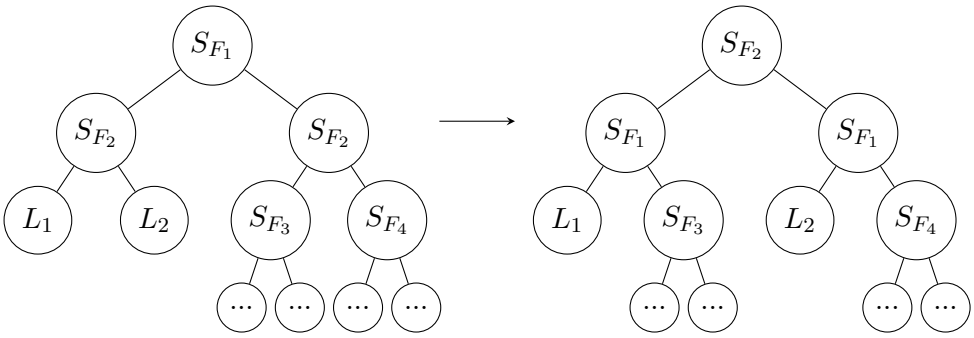


Figure 3.2: Tree Restructuring - Subtrees are treated as leaves if split decision sets are not equal

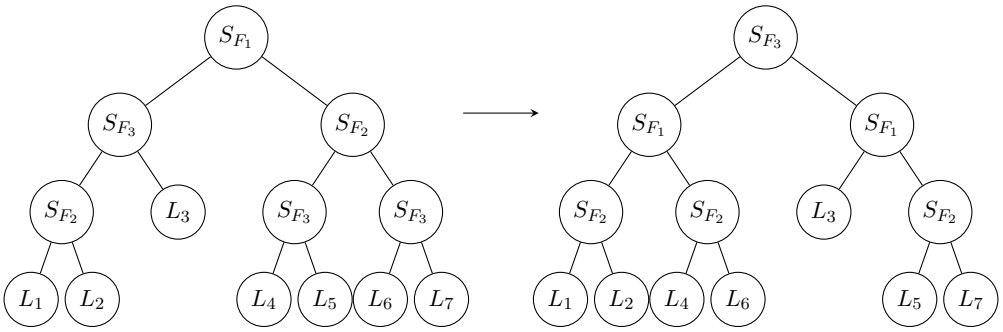


Figure 3.3: Tree Restructuring - For each leaf where F_3 is in the path from the root, S_{F_3} is pushed up to root.

size, provided that the leaf nodes share the same split decision set. However, this condition is often not met in practice, so not all split nodes can be freely repositioned. In cases where a split node cannot be placed as close to the root as intended, it is pushed as far up the tree as possible. This is done by subdividing the restructuring task into smaller subproblems, effectively treating certain subtrees as leaves and then recursively restructuring them. (Figure 3.2).

As previously noted, important features eventually emerge in monotonically increasing feature spaces. Consequently, many leaf nodes are converted into split nodes, using those features as the basis for the split decisions. Upon restructuring, such split nodes must be moved to the root of its subtree or, in extreme cases, to the root of the entire tree. As long as that node is part of every split decision set, it can be pushed up to the corresponding (sub-)tree root (Figure 3.3).

To realize the restructuring process, we extract relevant information from

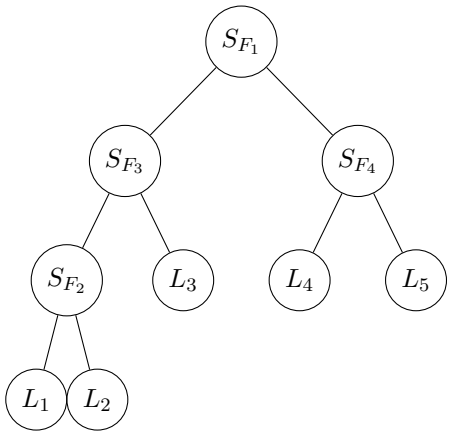
Algorithm 3.3

Input:	$\mathcal{T}$	decision tree with k leaves
Output:	$\mathbf{M}$	decision matrix representing all decision nodes
	$\mathbf{L}$	array holding all k leaves of $\mathcal{T}$

GetDecisionMatrix($\mathcal{T}$)

1: Initialize 2-dimensional decision matrix $\mathbf{M}$
2: $\mathbf{L} \leftarrow$GetLeafNodes($\mathcal{T}$)
3: **for** each leaf node L_k in $\mathbf{L}$ **do**
4: $\quad\mathbf{S} \leftarrow$ GetSplitDecisionSet($\mathcal{T}, L_k$)
5: $\quad$**for** each split decision S_{F_i} in $\mathbf{S}$ **do**
6: $\quad\quad\mathbf{M}[k, i] \leftarrow$ outcome of S_{F_i}
7: $\quad$**end for**
8: **end for**
9: **return** $\mathbf{M}, \mathbf{L}$

the current tree and rebuild it based on that information. Given the tree $\mathcal{T}$ as input, in a first step, we initialize a matrix $\mathbf{M}$, which holds information on the split decision sets for each leaf. In a second step, we retrieve all the leaf nodes in the tree and store them in the $\mathbf{L}$ array. We then use this array to iterate over each leaf node L_k. For each leaf node L_k, we get the split decision set $\mathbf{S}$, which contains the split nodes S_{F_i} from the root to this leaf node L_k. We can then fill the already initialized matrix $\mathbf{M}$. The rows k refer to the k-th leaf node in the decision matrix. The i-th column corresponds to feature F_i and indicates the outcome of the split node S_{F_i} in the path to the leaf node L_k. In the decision matrix, an entry of 1 at the $[k, i]$-th position indicates that the leaf node L_k is in the left subtree of the split node S_{F_i}, and a 0 that it is in the right subtree. If a split decision on the feature F_i was not installed in the path from root to the leaf, the entry is -1. The algorithm for obtaining that matrix is given in Algorithm 3.3. The decision matrix $\mathbf{M}$ has exactly as many rows as the tree has leaf nodes, and exactly as many columns as the tree has unique split decision nodes installed. Figure 3.4 gives an example of a tree with its decision matrix. During the restructuring process, it must remain unambiguous which matrix row each leaf corresponds to, ensuring that the correct decision path is consistently tracked. The decision matrix can then be used to rebuild the tree in a recursive manner, as shown in Algorithm 3.4. All columns with no entry equal to -1 represent possible candidates to be installed at the root of the



$$M = \begin{pmatrix} 1 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & -1 & 0 & -1 \\ 0 & -1 & -1 & 1 \\ 0 & -1 & -1 & 0 \end{pmatrix}$$

Figure 3.4: Decision tree with the corresponding decision matrix

tree or a subtree because leaves without that split node cannot be included in the subtree. If there is only one such column, the corresponding split node can be installed. If that is not the case, one of the columns satisfying that condition must be chosen. With respect to the objective, that split nodes yielding high information gains should be installed close to the root, for all those split nodes, the information gain is calculated as presented in Section 3.1.2.1 and the best split node is selected. After that, the children of the split node have to be set up. This can be done similarly, as they are trees themselves. The decision matrix is divided based on the column of the split node that was picked. All rows with entry 0 in that column form submatrix $\mathbf{M}_0$, and all rows with entry 1 form submatrix $\mathbf{M}_1$. Together with the corresponding leaf nodes, which are divided into $\mathbf{L}_0$ and $\mathbf{L}_1$ respectively, they can then be passed on to the children to build the subtrees. This procedure is repeated until only one leaf node is passed down to a child, which can then be installed directly.

After restructuring, the tree continues to make the same predictions as before, but with a re-ordered sequence of split nodes. If, at any step, there is only a single choice of which split node to install next, the tree effectively remains unchanged, indicating that reordering was not possible. Meaningful split nodes (those with high information gain in many subtrees) tend to appear more frequently and are thus pushed higher up, while less important split nodes end up near the leaves. As a result, pruning can remove these unimportant decisions, leading to a smaller and more interpretable tree overall.

Algorithm 3.4

Input:	$\mathbf{M}$	decision matrix
	$\mathbf{L}$	array of leaf nodes belonging to $\mathbf{M}$
	$\mathcal{T}$	empty tree $\mathcal{T}$
Output:	$\mathcal{T}$	decision tree based on $\mathbf{L}$ and $\mathbf{M}$

BuildTree($\mathbf{M}$, $\mathbf{L}$, $\mathcal{T}$)

1: **if** number of nodes in $\mathbf{L}$ is equal to 1 **then**
2: **return** the only node in $\mathbf{L}$
3: **end if**
4: $F^* \leftarrow \underset{F_i}{\text{argmax}}$ LocalInformationGain($\mathbf{L}$)
5: Attach new split node S_{F^*} to $\mathcal{T}$
6: $\mathbf{M}_0, \mathbf{M}_1 \leftarrow$ rows of $\mathbf{M}$, with column of F^* equal to 0 or 1, respectively
7: $\mathbf{L_0}, \mathbf{L_1} \leftarrow$ respective nodes to rows in $\mathbf{M}_0$ and $\mathbf{M}_1$
8: Attach BuildTree($\mathbf{M}_1$,$\mathbf{L_1}$, Init($\mathcal{T}_{\text{left}}$)) as left child to $\mathcal{T}$
9: Attach BuildTree($\mathbf{M}_0$,$\mathbf{L_0}$, Init($\mathcal{T}_{\text{right}}$)) as right child to $\mathcal{T}$
10: **return** $\mathcal{T}$

3.1.2.3 Pruning

Offline tree induction (cf. Section 2.2) is generally stopped by a criterion that leaves room for some overfitting. Once the tree induction is concluded, *pruning* takes place to make the tree more generalizable Rokach and Maimon [2008]. Based on specific rules, subtrees are replaced by leaf nodes, which yields a smaller and less complex tree. Due to the potentially infinite nature of monotonically increasing feature spaces, it is required to perform pruning in an online manner, which enables the tree to continue learning.

Once the tree has been restructured, decision nodes with little or no information gain are inserted near the leaf nodes. In some cases, this renders some subtrees redundant. An extreme case is displayed in Figure 3.5. In this case, it is clear that by replacing the subtree with one of its leaves, the entire tree retains its predictive power while decreasing in size. However, no general rule can be deducted as to when it is appropriate to replace a subtree with a leaf, nor which leaf to replace it with. Hence, it is crucial to apply appropriate rules to the pruning process. If pruned too rigorously, meaningful split nodes might be dropped, effectively reducing the trees' predictive power. It must be noted that all statistics in a leaf node are discarded once it is converted into a decision node, and, in accordance with Requirement 1, cannot be retrieved again at a later point in time. Therefore,

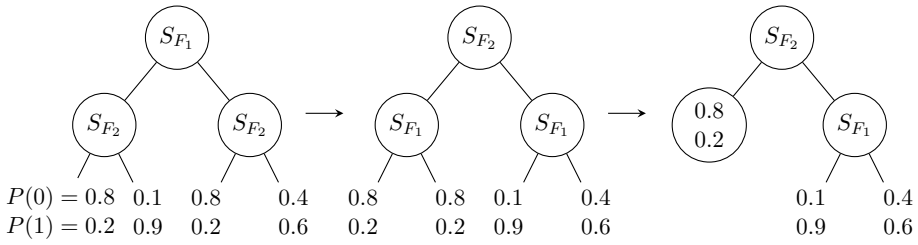


Figure 3.5: Tree Restructuring - By pushing S_{F_2} up to the root, the left subtree does not yield additional information and can be pruned.

dropping split nodes and then reinstalling them later reduces the number of observations for further tree induction.

On the other hand, if pruned too cautiously, redundant split nodes might not be detected, and the tree becomes unnecessarily large, or no pruning takes place at all, hence lessening the interpretability (Requirement 6). As a result, the restructuring would effectively be rendered useless, and the resulting tree would not differ from the tree obtained by the baseline VFDT algorithm.

Pruning aims to identify decision nodes that split on features providing no information gain, particularly when both children are leaf nodes, and replace those nodes with leaves. This reduces the tree's size and complexity without sacrificing its predictive accuracy.

Pruning is only permitted if both leaves in the subtree predict the same label. In that case, the replacement leaf adopts that label, ensuring that any instance arriving at this new leaf receives the same prediction the subtree would have produced.

To ensure that only those split nodes that are likely not to yield any information gain are dropped, the Hoeffding bound is again used to reassess the split nodes' information gain. It essentially checks whether, given the current samples, the split node would now be installed. This time, the bound is applied less strictly, which is achieved in two ways. Firstly, a higher probability bound ρ is used. A bigger ρ in

$$P(|\mu - 0| > \epsilon) = P(|\mu| > \epsilon) \leq \rho \tag{3.1.3}$$

results in a lower ϵ. Secondly, the information gain calculated from the samples is compared to an information gain of zero, rather than to the

Algorithm 3.5

Input:	$\mathcal{T}$	decision tree
	ρ	upper limit to probability of choosing the wrong test
	m_{min}	minimum of samples seen in leafs to allow pruning
Output:	$\mathcal{T}$	pruned decision tree

PRUNE($\mathcal{T}$, ρ, m_{min})

1: Obtain $\mathcal{T}_{\text{left}}, \mathcal{T}_{\text{right}}$ child subtrees from $\mathcal{T}$
2: **if** $\mathcal{T}_{\text{left}}$ is not a leaf **then**
3: $\quad\mathcal{T}_{\text{left}} \leftarrow$ PRUNE($\mathcal{T}_{\text{left}}$)
4: **end if**
5: **if** $\mathcal{T}_{\text{right}}$ is not a leaf **then**
6: $\quad\mathcal{T}_{\text{right}} \leftarrow$ PRUNE($\mathcal{T}_{\text{right}}$)
7: **end if**
8: **if** $\mathcal{T}_{\text{left}}$ is leaf **and** $\mathcal{T}_{\text{right}}$ is leaf **then**
9: $\quad$**if** SAMPLECOUNT($\mathcal{T}_{\text{left}}$) $\geq m_{min}$ **and** SAMPLECOUNT($\mathcal{T}_{\text{right}}$) $\geq m_{min}$ **then**
10: $\quad\quad\epsilon \leftarrow$ Hoeffding bound based on ρ and n
11: $\quad\quad$Obtain i from split decision feature F_i from root of $\mathcal{T}$
12: $\quad\quad$**if** LOCALINFORMATIONGAIN($[\mathcal{T}_{\text{left}}, \mathcal{T}_{\text{right}}]$)$[i] \leq \epsilon$ **then**
13: $\quad\quad\quad\mathcal{T} \leftarrow$ MERGELEAVES($[\mathcal{T}_{\text{left}}, \mathcal{T}_{\text{right}}]$)
14: $\quad\quad$**end if**
15: $\quad$**end if**
16: **end if**
17: **return** $\mathcal{T}$

second-best split. It is, therefore, evaluated whether the split node is better than no split at all. If $\mu \leq \epsilon$, the split node is pruned.

Another measure to avoid rigorous pruning is to set a threshold. Only if both leaves hold more samples than the predefined limit can pruning take place. This makes sense because low counts in the statistics of the leaves indicate that the split node has been installed only recently. The older a leaf node is, the more samples it should have accumulated, and the larger the volume of data used to decide as to whether or not to drop a split node, the more reliable the outcome. To prune the entire tree, a recursive approach that takes this into account is presented in Algorithm 3.5. A subtree that is replaced by a leaf may result in a new decision node with two leaves as children. For this reason, for any decision node, both children are pruned first, and then the node itself is investigated. If both leaves are children, the conditions to replace the node are checked. The sample counts of both leaves

Algorithm 3.6

Input:	**L**	array of leaves to be merged
Output:	$\bar{L}$	aggregated leaf node

MergeLeaves(**L**)

1: $\bar{L} \leftarrow$ empty leaf
2: **for** leaf L in **L do**
3: $\quad \bar{L}_{n_{ijk}} \leftarrow \bar{L}_{n_{ijk}} + L_{n_{ijk}}$
4: **end for**
5: **return** $\bar{L}$

have to be above the threshold, and the information gain of the test has to be below ϵ for the decision node to be replaced. The data instances stored in the subtree can still be used for further induction. Instead of simply dropping the whole subtree, a leaf to replace it is generated by calling MergeLeaves on the children, which aggregates all samples seen since the initial split node installation. Algorithm 3.6 presents the approach to aggregating the statistics of the leaf nodes. The algorithm takes the array of the two child nodes as input. Then, a new leaf $\bar{L}$, to aggregate the statistics, is initialized and subsequently filled with the statistics given by the n_{ijk} matrices of the leaf nodes. This leaf is returned and attached to the tree.

3.1.3 Predicting with DFDT

Predicting with DFDT requires traversing the model built during training. Given the current structure of the tree, an observation x is fed into the tree at the root node. The split decision, i.e., the feature-threshold combination, at the root is then evaluated by comparing the instance's value against it. Given the outcome of this comparison, the instance is propagated down the tree to the next node. This procedure is repeated until the instance reaches a leaf node. The leaf node can then predict the class. This retraceable traversal of the decision trees justifies its interpretability, as the decision path clearly outlines the decisions made by the model that led to the prediction.

3.2 Related Work

The presented approach can be related to two research areas. The first area is learning with monotonically increasing feature spaces, where no methods,

especially no interpretable methods, exist that adjust decision tree learning algorithms to handle an increasing feature space. The second area is concerned with the pruning of Hoeffding trees; as shown in Section 3.1.2.3, we utilize pruning to regrow a tree more efficiently based on the Hoeffding bound. Generally, if we solely restructured the tree without pruning, there would be no difference in the performance of the tree compared to the VFDT algorithm. However, an additional advantage is to have more important features at the top of the tree. This replicates the behavior of traditional decision trees, which have their splits arranged from highest to lowest importance, thereby improving interpretability. In the following two sections, we present some work from the area of learning with monotonically increasing feature spaces and pruning of Hoeffding trees.

3.2.1 Learning with Monotonically Increasing Feature Spaces

As described earlier, the initial formulation of the problem of monotonically increasing feature spaces was done in Zhang et al. [2016]. In their work, they present a linear model, OLSF, that utilizes a projection onto an L_1 ball to promote sparsity and a truncation step to perform feature selection on the fly. In Beyazit et al. [2018], they propose a single hidden layer feedforward network with shortcut connections. If the domain to be learned is complex and requires non-linearity, the presented approach is capable of identifying this. The layers in their approach are adjusted to handle the growing feature space, while applying a weight-based pruning procedure to maintain a low runtime. The pruning procedure limits the trainable parameters linearly to the size of the input space. Another approach is presented by Alagurajah et al. [2020], hereby the focus is on monotonically increasing feature spaces that have features with arbitrary scaling. Due to the nature of data streams, it may be impossible to scale features appropriately. Therefore, they proposed two algorithms based on linear models to handle the problem. These algorithms track various parameters in the learning process, such as the cumulative sum of squared gradients and the maximum feature value encountered, to address the posed problem.

All three approaches share the commonality that they are difficult to interpret, as is often the case with linear models and neural networks. Another downside that can be considered is that the approaches of Zhang et al.

[2016] and Beyazit et al. [2018], OLSF, need to track the scaling of features additionally.

3.2.2 Hoeffding Tree Pruning

In the work of Yang and Fong [2011a], Moderated-VFDT is presented. The main contributions of this work are an adaptive tie threshold and an incremental pruning mechanism. The presented pruning method can be considered a pre-pruning method, as it restricts the tree's growth by applying one of two proposed mechanisms: strict pruning and loose pruning. In Yang and Fong [2011b], another pre-pruning method is proposed that limits the growth of the tree by comparing the number of samples that fall into existing leaves against those that do not. If more samples fall into existing leaves, then according to the proposed pruning method, there is no need to update the tree structure. The approach by Barddal and Enembreck [2019] can be loosely classified as a pre-pruning method. They propose to regularize the growth of the feature by splitting multiple times on the same feature, leading to a more diverse decision tree.

The pruning method of the presented DFDT algorithm can be considered more as a post-pruning method. It takes the existing tree structure and minimizes it in a certain interval of observed instances. It is hard to say which is more optimal. However, in cases where we have uncertainty regarding features that emerge later in the process, we argue that it may be beneficial to first grow the tree and then prune it later on.

3.3 Evaluation

In an experimental evaluation, the behavior of DFDT is analyzed. In a first step, we describe the synthetic data we generate to evaluate DFDT in Section 3.3.1. We generate three datasets to evaluate the parameters (Section 3.3.2) and three datasets to evaluate the performance of DFDT (Section 3.3.3). Due to the lack of decision trees built to handle monotonically increasing feature spaces, and because it is highly related, we modified the VFDT algorithm Domingos and Hulten [2000] to handle emerging features, using it as a baseline for comparison. Essentially, the two approaches work similarly, except for the pruning, which is exclusively done by DFDT.

Finally, we present results on the UCI benchmark datasets proposed by Beyazit et al. [2019] in Section 3.3.4. These UCI datasets are also used in Chapter 4 as one of the benchmark cases to evaluate our methods for varying feature spaces.

3.3.1 General Setting

To better understand the algorithm and its advantages and disadvantages, we use synthetic data in the evaluation. To generate a data set with T instances, D features, and two classes, first, a $T \times D$ matrix with random entries of 0 and 1 is generated, yielding X. To determine the class, each column F_d is then multiplied by $g(d)$; for example, this function could be $g(d) = (-1)^d * d$. The sum of each row is then used to determine the class of a data instance by using quantiles to split it into two groups of the same size. Specifically, the bottom quantile is assigned class one, and the top quantile is assigned class two, which ultimately yields Y. The process of generating the data gives us dataset (X, Y) of length T with D features and a balanced class distribution.

A monotonously increasing feature space can now be constructed by extracting data instances one row at a time. Initially, only a limited number of columns are included, and as the number of data instances increases, more columns are added to the rows extracted. As presented in Section 2.4, we can simulate the monotonously increasing feature space by $\{(x_t, y_t)\}_{t=1}^{T}$ where $x_t \in \mathbb{R}^{D_t}$ with $D_t \leq D_{t+1}$. At some point, all features D are available.

The datasets used in the synthetic evaluation are shown in Table 3.3. The first three datasets are used to identify the impact of the parameters on both the accuracy and the size of the trees. Datasets four to six were used for the performance evaluation. In the parameter evaluation, the attributes emerge according to their index d, so with an increasing importance to better measure the effects of the parameters, as this represents the best case for the DFDT. For the performance evaluation, except for the two initial features, the features emerge randomly with no specific order, which means that the most important features do not necessarily emerge at the end. The generation is seeded, and while the impact of each feature may differ in terms of its contribution to the final class, the features emerge in the same pattern for datasets four to six.

Data Set	Feature Count			Instances	$g(i)$	Class Ratio
	Total	Start	End			
ParamEval_1	25	10	20	500000	$(-1)^i * i$	1 : 1
ParamEval_2	25	10	20	500000	$(-1)^i * \log(i)$	1 : 1
ParamEval_3	25	10	20	500000	$(-1)^i * i^2$	1 : 1
PerfEval_1	50	2	50	2000000	$(-1)^i * i$	1 : 1
PerfEval_2	50	2	50	2000000	$(-1)^i * \log(i)$	1 : 1
PerfEval_3	50	2	50	2000000	$(-1)^i * i^2$	1 : 1

Table 3.3: Datasets used for evaluation

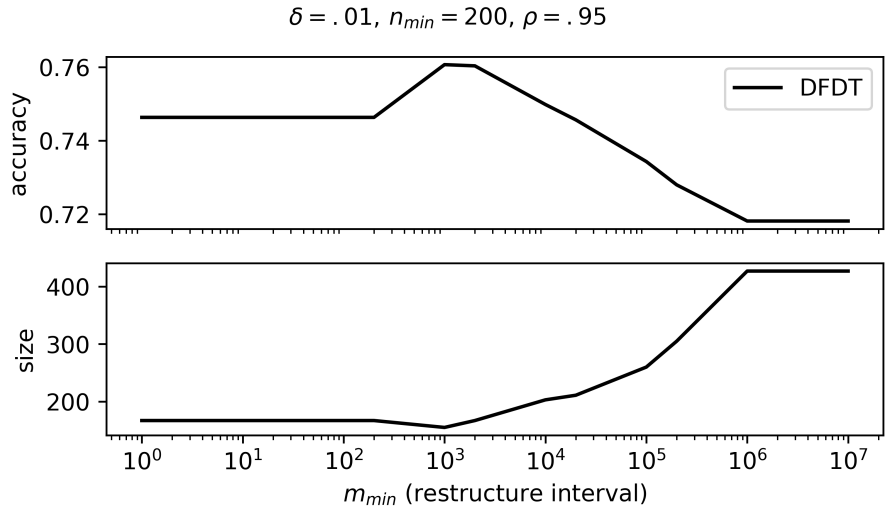


Figure 3.6: Effects of varying parameter m_{min} on ParamEval_1 as presented in Schreckenberger et al. [2020].

3.3.2 Effects of parameters

In a first step, the effects of varying input parameters are compared for both the DFDT algorithm and the VFDT algorithm. This is performed on three of the synthetically generated datasets as described. The restructure interval m_{min} sets the number of samples that have to be fed into the tree as a whole before the tree is restructured again. This is combined with pruning, so the tree may be smaller after this operation. Doing this very often results in more nodes being dropped; therefore, there is a clear correlation between low values for m_{min} and low tree size.

Again, there seems to be a limit from which point a varying m_{min} still results in the same tree (Figure 3.6). This may be explained by the fact

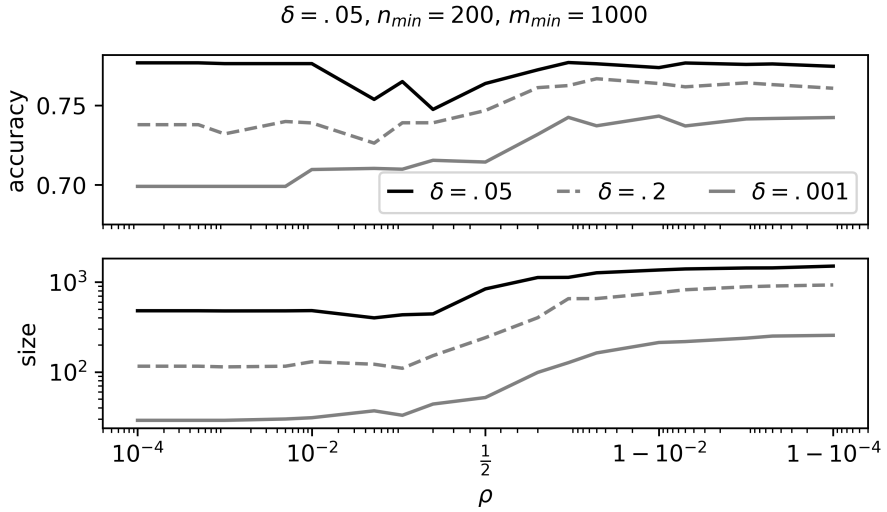


Figure 3.7: Effects of varying parameter ρ on ParamEval_2 as presented in Schreckenberger et al. [2020].

that values below that limit all restructure at any given points at which the operation changes the tree. This is clearly equivalent to m_{min} being one, and after each data instance is seen, the tree is restructured.

Restructuring at a very high frequency increases the likelihood of pruning too harshly, which could ultimately result in dropping important decision nodes. This, in turn, would then lead to lower accuracy. On the other hand, a very high restructure interval might miss the opportunity to prune unnecessary nodes early on, resulting in bigger trees. Once the parameter exceeds the number of examples the tree will ever see, restructuring never takes place, and the algorithm is effectively identical to the VFDT algorithm. When pruning, the parameter ρ is used to calculate the Hoeffding bound and reevaluate a split. In order not to be pruned, a decision node does not have to yield as high an information gain to be kept as for induction itself because here, the Hoeffding bound only compares it to a theoretical information gain of zero. This check is also less strict because, generally, the values for ρ are higher than the values for δ.

The effect of this parameter on accuracy does not follow a clear rule, as shown in Figure 3.7, and it appears to be dependent on the dataset. It may also be dependent on other input parameters; therefore, the parameter is evaluated with two additional values for δ on one dataset. For those values,

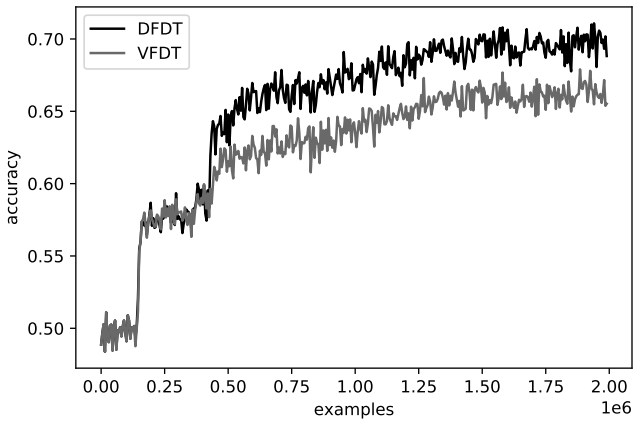


Figure 3.8: Performance of DFDT and VFDT on PerfEval_1 as presented in Schreckenberger et al. [2020].

the accuracy increases with a higher ρ.

In general, higher values for ρ have an advantage if the other parameters are chosen appropriately. This statement can be observed better at a later point. It may be caused by the fact that pruning less harshly is more effective in combination with more careful splitting, which in turn leads to fewer data instances being used up by splits that exist only temporarily. The effect on the size, however, is evident. The lower this parameter, the less likely a subtree is to be replaced by a leaf, resulting in bigger trees, which can be observed for all three datasets.

We present a more in-depth analysis of the parameters and their impact in Appendix A.1

3.3.3 Performance evaluation

To evaluate the algorithm's performance on emerging features, it is benchmarked against the VFDT algorithm. The trees are induced, and the accuracy is averaged every $1,000$ data instances. For each data set, 50 randomly chosen combinations of parameters are selected for both VDFT and DFDT algorithms. The possible values for each parameter are given in Table 3.4. The results for PerfEval_1, PerfEval_2, and PerfEval_3 are shown in Figure 3.8, Figure 3.9, and Figure 3.10 respectively. There is a general tendency for the DFDT algorithm to create smaller trees than the VFDT algorithm

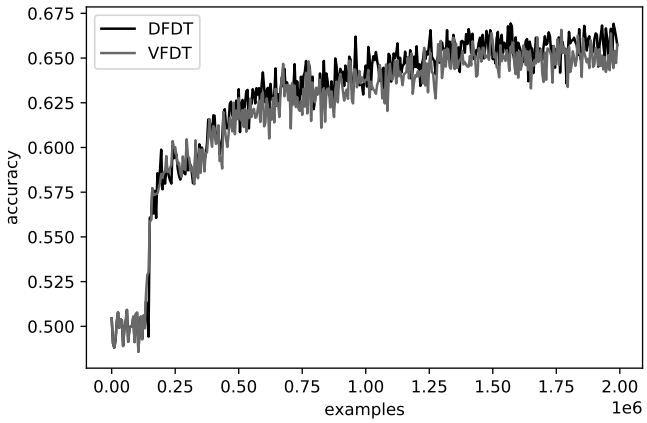


Figure 3.9: Performance of DFDT and VFDT on PerfEval_2 as presented in Schreckenberger et al. [2020].

Parameter	Values
δ	$\{.001, .01, .05, .1, .5\}$
n_{min}	$\{100, 200, 500, 1000\}$
m_{min}	$\{100, 200, 500, 1000\}$
ρ	$\{.01, .05, .1, .5, .9, .95, .99\}$

Table 3.4: Values to select from randomly for each parameter

does. The reason for this tendency is that once decision nodes become redundant due to new features, the DFDT algorithm can drop them, whereas the VFDT algorithm cannot. The difference is more substantial later on in the induction process. Initially, the DFDT algorithm often constructs larger trees. A combination of δ and n_{min} determines the growth of the tree, and because the DFDT algorithm can cope better with unnecessary splits, these combinations of the well-performing trees lead to faster growth.

For PerfEval_1 and PerfEval_3 the DFDT algorithm outperforms the VFDT algorithm significantly. Especially at roughly 500,000 examples, there is a huge jump in accuracy for the DFDT, which is unmatched by the VFDT. This can be explained based on the dataset's characteristics, as features that emerge have a stronger impact on the class than in PerfEval_2, where the impact on the class is nearly equal for every emerging feature. However, even in the case where emerging features are roughly equally im-

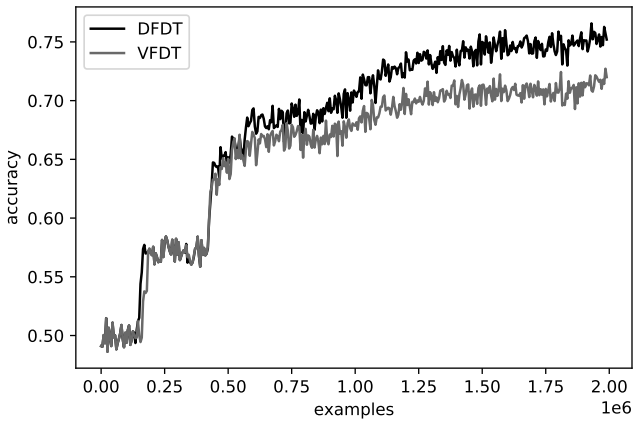


Figure 3.10: Performance of DFDT and VFDT on PerfEval_3 as presented in Schreckenberger et al. [2020].

Dataset	Obs.	Dim.	Dataset	Obs.	Dim.
wpbc	198	34	svmguide3	1,234	21
ionosphere	351	35	spambase	4,601	57
wdbc	569	31	magic04	19,020	10
wbc	699	10	imdb	25,000	7500
german	1,000	24	a8a	32,561	123

Table 3.5: The datasets used in the benchmark cases of the evaluation.

portant as the initial features, there is still an increase in performance for the DFDT compared to the VFDT, as shown in Figure 3.9. The smaller performance increase in PerfEval_2 for the DFDT compared to the two other cases can be explained by fewer opportunities to prune, as mentioned before, because the chance that a feature is significantly better than another, and therefore satisfying the constraint of ρ, is smaller.

3.3.4 UCI Benchmark

For the evaluation, we rely on the datasets used in the evaluation of varying feature spaces [Beyazit et al., 2019]. However, we omit the imdb dataset as this dataset is inherently representative of varying feature spaces, and therefore does not solely represent a monotonically increasing feature space. The datasets and their characteristics are depicted in Table 3.5.

Dataset	OLSF	DFDT
german	$.369 \pm .010$	$\mathbf{.300 \pm .001}$
ionosphere	$\mathbf{.243 \pm .017}$	$.360 \pm .002$
spambase	$\mathbf{.212 \pm .010}$	$.395 \pm .000$
magic04	$\mathbf{.321 \pm .003}$	$.352 \pm .000$
svmguide3	$.308 \pm .021$	$\mathbf{.239 \pm .001}$
wbc	$\mathbf{.054 \pm .013}$	$.352 \pm .002$
wpbc	$.428 \pm .027$	$\mathbf{.234 \pm .005}$
wdbc	$\mathbf{.221 \pm .018}$	$.374 \pm .002$
a8a	$.318 \pm .004$	$\mathbf{.204 \pm .000}$

Table 3.6: CER made by OLSF and DFDT on simulated data streams with monotonically increasing feature spaces.

The rival model we are benchmarking against is OLSF [Zhang et al., 2016]. For each dataset, we have run the experiments 20 times and report on the CER as well as its standard deviation. We reused the same parameters as for the synthetic data. The results can be seen in Table 3.6.

The comparison between OLSF and DFDT (Table 3.6) indicates that performance is strongly dataset-dependent. DFDT achieves lower CER on *german*, *svmguide3*, *wpbc*, and in particular on *a8a*, where it provides the largest improvement with a reduction from 0.318 to 0.204. In contrast, OLSF performs better on *ionosphere*, *spambase*, *magic04*, *wbc*, and *wdbc*, with its strongest advantage on *wbc*, achieving 0.054 compared to 0.352 for DFDT. Overall, DFDT tends to excel on larger, higher-dimensional datasets, while OLSF remains more effective on smaller datasets. However, it is worth noting that due to the small size of the datasets, excluding a8a, the restructuring and pruning by the DFDT algorithm did not occur, as it requires a few more examples to be observed than what some of the datasets provided. Hence, we argue that these UCI benchmark datasets can be seen as an indicator of the approach's validity; however, we caution against overinterpreting these results.

3.4 Discussion

In the following, we discuss the proposed approach DFDT with respect to the six requirements presented in Section 1.2.

3.4.1 Requirement 1: Online Processing

The DFDT algorithm is designed for data streams where each instance is processed immediately upon arrival. This is implemented by processing a data instance once by aggregating the feature statistics in the n_{ijk} matrix of the given leaf node. In practice, every observation is propagated through the tree in a linear fashion that scales with its depth, ensuring that incoming data are processed without delay and only inspected once. This design choice guarantees that the model remains responsive in real-time settings and facilitates Requirement 5.

3.4.2 Requirement 2: Finite Memory

Given that data streams can far exceed available memory, the method emphasizes a minimal memory footprint. The design ensures that each new feature increases the corresponding dimension of the n_{ijk} matrix linearly, thereby helping to control memory growth. However, because the algorithm stores 2^{depth} instances of these matrices at the leaf nodes, there is an inherent exponential potential that is mitigated by pruning. Especially in environments where more impactful features emerge, the DFDT algorithm is capable of reducing the tree size by first restructuring and then pruning the model. This careful balance between maintaining necessary statistics and tree structures ensures that the overall model remains feasible in resource-constrained environments.

3.4.3 Requirement 3: Finite Processing Time

The efficiency of processing each data point in terms of time is critical to the algorithm's real-time applicability. Each observation is integrated through a straightforward, incremental update of the n_{ijk} matrix, with computation scaling linearly both in terms of the number of features and the tree's depth. Additionally, restructuring and pruning operations are designed to run in linear time relative to the number of nodes. While not currently implemented, in scenarios where the tree grows too large, these maintenance tasks can be offloaded to run in the background, ensuring that the core online learning process remains efficient and capable of keeping pace with rapidly arriving data.

3.4.4 Requirement 4: Feature Space Adaptations

Adaptability to monotonically increasing feature spaces is central to the method's design. With every new feature, the algorithm seamlessly extends the n_{ijk} matrix, maintaining a linear relationship with the number of features. Although this step may seem trivial, it offers opportunities to move forward. In our case, this allows the model to prioritize more relevant features as the feature space expands dynamically. By additionally providing a method to restructure the tree, thereby allowing for pruning to take place, we provide the necessary tools to keep the tree smaller and more accurate, as shown in the evaluation.

3.4.5 Requirement 5: Anytime Predictions

An essential aspect of the DFDT algorithm is its ability to provide predictions at any moment based on the data observed so far. The online nature of the method, combined with incremental model updates, ensures that the model does not need to be relearned from scratch with each new observation. Even during restructuring, which takes place periodically, the model is designed to deliver immediate predictions. This continuous readiness is a direct result of the efficient decision tree-based structure.

3.4.6 Requirement 6: Interpretability

Interpretability remains a cornerstone of the approach, vital for user trust and compliance with regulatory demands [Molnar, 2020]. As motivated, it is especially important in dynamic environments, where traditional data inspection methods are challenging to apply. The underlying decision tree structure, which is continuously minimized by restructuring and pruning, provides traceable pathways to understand feature importance and decision outcomes. Although the model may undergo periodic restructuring, which could potentially erase some historical details, the overall design maintains a balance between model complexity and interpretability. As the memory footprint of the model is low, in high-stakes domains, it may be beneficial to take snapshots of the model to ensure post-hoc analysis.

Chapter 4

Learning from Varying Feature Spaces

In this chapter, which is based on our two publications Schreckenberger et al. [2022] and Schreckenberger et al. [2023], we present two approaches designed to handle varying feature spaces. The first approach, called Dynamic Forest (DynFo), is an incremental learning approach that generates decision stumps from a buffer of instances and regulates the impact of each decision stump as well as the ensemble size. The second approach Online Random Feature Forest for Feature space variabilities (ORF^{3}V) is an online learning approach that generates a so-called feature forest for each observed feature from a condensed t-digest data distribution representation and dynamically manages the size of the ensemble based on the Hoeffding bound. Both approaches can be considered a mixture of weighted random forests and random forests of stumps as introduced in Section 2.3.3.

We begin by presenting the crowdsense dataset in Section 4.1, which was collected during this thesis and published in Schreckenberger et al. [2023]. It can be seen as a real-world example of varying feature spaces and hence justifies this research. This dataset is later used in the evaluation. We then proceed to present the first approach DynFo in Section 4.2, and subsequently the second approach ORF^{3}V in Section 4.3. Subsequently, we relate both approaches to the current state-of-the-art in Section 4.4 and provide the evaluation in Section 4.5. The evaluation is comprised of the benchmark evaluation case, as introduced by Beyazit et al. [2019], the crowdsense eval-

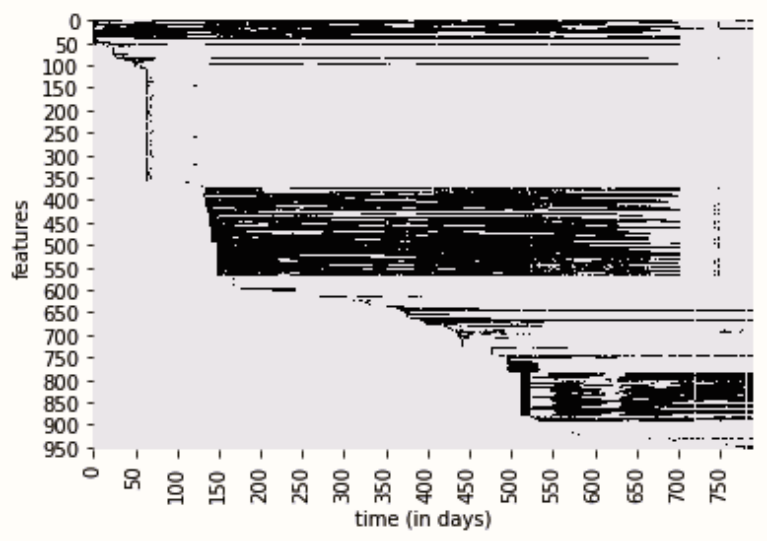


Figure 4.1: Visualization of how the feature space of the crowdsense dataset varies over 790 days, where each black horizontal line indicates the availability of a feature, as presented in Schreckenberger et al. [2023]

uation, and an experiment on the interpretability of the two approaches. Finally, we discuss the findings with respect to different perspectives in Section 4.6.

4.1 Crowdsense Dataset

The crowdsense dataset, published in Schreckenberger et al. [2023], is the first crowd-based sensor network dataset to represent the problems of varying feature spaces. We consider this dataset as an additional motivation, highlighting the need for algorithms that can adapt to a changing feature space. The dataset was collected during the COVID-19 pandemic. It contains environmental sensors that are labeled with the corresponding government response. The main idea behind this dataset is that governmental responses, such as the cancellation of public events, school closings, and lockdowns, had a measurable impact on the environment and, hence, are detectable by the sensors. In the following, we describe the data collection process.

The data originate from the SmartCitizen (SC) sensor network Camprodon et al. [2019], which is a crowd-sensing platform. The respective la-

bels come from the Oxford COVID-19 Government Response Tracker (GRT) Hale et al. [2021], which categorized the government responses for each country according to the strictness of the measure, e.g., restrictions of gatherings, which can be categorized from no restrictions, to a limit on the number of people that are allowed to gather, to no gatherings allowed. For the input data, we have selected sensors from two categories: noise pollution (sound pressure) and air quality (eCO_2 level and eTVOC level). Since the SC project originated in Spain, most of the sensors are located there. We focused on data from the 56 largest Spanish cities, resulting in 954 data streams across three sensor types. Hence, the classes we extracted are focused on the restrictions in Spain. The GRT begins on January 1, 2020, so we collected data from that date until February 28, 2022, covering 790 days of measurements. Because the GRT provides daily labels, we computed the mean of each data stream for each day in this range. Although this procedure would nominally yield 857,940 data points, the actual count is 152,798 due to variations in the feature space, which means that we have a data availability of approximately 18%.

Figure 4.1 displays the availability of each feature, illustrating how the data set's feature space changes over time. The GRT data are divided into eight cases, each containing multiple classes that describe the severity of governmental measures; the first class in each case indicates no response, and the classes are ordered by increasing stringency.

- $case_1$: School closing $(68, 480, 134, 108)$
- $case_2$: Workplace closing $(68, 144, 517, 61)$
- $case_3$: Cancel public events $(69, 721)$
- $case_4$: Restriction of gatherings $(69, 20, 13, 104, 584)$
- $case_5$: Close public transport $(704, 86)$
- $case_6$: Stay-at-home requirements $(253, 151, 386)$
- $case_7$: Movement restrictions $(339, 104, 347)$
- $case_8$: International travel $(69, 132, 493, 96)$

4.2 Dynamic Forest

Dynamic Forest was the first approach we introduced to tackle the problem of varying feature spaces with a method that dynamically adapts and provides interpretability. It was published in Schreckenberger et al. [2022]. The method was developed within the framework of random forests (cf. Section 2.3), where an ensemble of many weak learners forms a strong predictor by averaging the results of the weak learners.

We first introduce the notation used in DynFo, and then we proceed to explain the main parts of the algorithm, as well as the ideas and motivations behind them. We follow up by introducing the algorithm and its parameters, describing the main components in a top-down manner. Subsequently, we describe how the learning process works with regard to the weak learner used, the update strategy, and the adaptation of weak learners. Finally, we explain how to predict a given observation with the presented algorithm.

4.2.1 Notation Dynamic Forest

In our notation, $\mathcal{L}$ refers to the ensemble of all weak learners used in the algorithm. It is also used analogously to the function we want to approximate $\hat{f}$. Hence, we can use $\mathcal{L}(x_t) = \hat{y}_t$ to make a prediction. In our algorithm, we use decision stumps as weak learners Iba and Langley [1992]. For each weak learner $\mathcal{L}_b \in \mathcal{L}$, we have a corresponding weight of $w_b \in \mathbf{w}$ and an associated accepted feature space of $\mathcal{F}_b \subset \mathcal{F}_t$, where $\mathcal{F}_t$ are all the features seen so far by the algorithm at step t. The accepted feature space $\mathcal{F}_b$ of a weak learner $\mathcal{L}_b$ can be considered as bagging (cf. Section 2.3), as it selects a subset of the total feature space seen so far for each weak learner from which they can be learned. As decision stumps only make one split, we denote the feature F_d chosen for this split of a decision stump $\mathcal{L}_b$ as $\mathcal{L}_b^{F_d}$.

4.2.2 Learning with Dynamic Forest

In Algorithm 4.1, the steps for learning with DynFo are shown. In the following, we provide an introduction to the parameters and then offer a high-level explanation of the algorithm's steps, which we will elaborate on in more detail in subsequent sections.

The algorithm offers a large set of parameters (Table 4.1), enabling it to

Algorithm 4.1 Learning with DYNAMIC FOREST

Initialize ensemble $\mathcal{L}$
Initialize $\mathbf{r}$ of size M
for $t = 1, 2, \ldots, T$ **do**
  receive instance (x_t, y_t) and fill buffer $\mathcal{W}_t$ of size N
  $\mathcal{W}_t \leftarrow \left[(x_{t-N+1}, y_{t-N+1}), \ldots, (x_t, y_t)\right]$
  for $b = 1, 2, \ldots, B$ **do**
    $\mathcal{L}_b \leftarrow \mathcal{L}[b]$
    if $\mathcal{L}_b$ is not initialized **then**
      $\mathcal{L}_b \leftarrow$ INITDECISIONSTUMP($\mathcal{W}_t$)
      $w_b \leftarrow 1$
      $\mathbf{w}[b] \leftarrow w_b$
    end if
  end for
  $\mathcal{L} \leftarrow$ UPDATEACCEPTEDFEATURESPACE($\mathcal{L}$, δ, B)
  $\mathcal{L}, \mathbf{w}, \mathbf{r}, B, \gamma \leftarrow$INCREASEENSEMBLESIZE($\mathcal{L}$, $\mathbf{w}$, (x_t, y_t), $\mathbf{r}$, γ, B)
  $\mathbf{w} \leftarrow$ UPDATEWEIGHTS($\mathcal{L}$, $\mathbf{w}$, (x_t, y_t), α, ϵ, B)
  $\mathcal{L}$, $\mathbf{w}$, $B \leftarrow$ PRUNEENSEMBLE($\mathcal{L}$, θ_1, θ_2, B)
end for

be adaptable to many scenarios. The parameters α and ϵ are used for the weight update strategy, i.e., updating the weights of each weak learner of the ensemble. The parameters β, γ, as well as θ_1 and θ_2, are used to balance the number of weak learners used and the relearning of those, i.e., increasing and decreasing the number of classifiers used and replacing some if needed. The initial number of weak learners the ensemble starts with is determined by B. The parameter N defines the instance buffer size, and δ serves as a bagging parameter, determining the fraction of randomly selected features to consider for each weak learner and thereby forcing a diverse ensemble.

At the beginning, the algorithm sets up an ensemble of weak learners, $\mathcal{L}$, and allocates a result window $\mathbf{r}$ of a fixed size M to keep track of recent performance. The learning process then unfolds over T time steps, one for each incoming data instance.

Upon receiving a new instance (x_t, y_t) at time t, the algorithm fills a buffer $\mathcal{W}_t$ with the last N instances—this acts as a small reservoir of recent data. For each weak learner in $\mathcal{L}$, if it has never been initialized, the algorithm creates a decision stump (i.e., a shallow decision tree of depth one) by calling a dedicated initialization method on the current buffer $\mathcal{W}_t$. The new learner's weight is initialized with one before being added to the ensemble.

Parameter	Description
α	magnitude of impact on weight update
β	probability to keep weak learner in ensemble
δ	fraction of features to consider
ϵ	penalty if split of decision stump is not in the current instance
γ	performance tracking threshold
B	starting number of weak learners in the ensemble
N	buffer size of instances
M	size of result window
θ_1, θ_2	bounds for the update strategy

Table 4.1: Overview of input parameters for Algorithm 4.1.

Next, the system updates the feature space that each weak learner can use, governed by the parameter δ. This ensures that learners see only the features they are allowed to train on, creating a bagging-like effect. The algorithm then checks whether the ensemble's performance has declined to a certain degree, requiring the addition of a new weak learner. This happens in the IncreaseEnsembleSize routine, which reviews recent accuracy and, if necessary, expands the ensemble size.

After possible expansion, the algorithm updates the weights of all weak learners based on their performance on the latest instance. Good performance leads to an increase in weight, whereas failing to predict correctly or using a feature that is missing reduces a learner's weight. Finally, the PruneEnsemble procedure reviews each learner's weight and either retains, retrains, or removes it, aiming to maintain an efficient, up-to-date collection of weak learners.

This sequence of steps—buffering recent data, initializing new stumps when needed, adjusting feature availability, optionally adding another learner, updating weights, and pruning—repeats for each time step t until all T data points have been processed. By continuously adapting to incoming data and dynamically managing the ensemble size and feature usage, DynFo remains flexible in response to changing conditions throughout the entire training period.

Algorithm Roadmap The proposed approach has three major components. The first component is the selection and learning of the weak learners,

which is described in Section 4.2.2.1. The main idea of DynFo is to dynamically manage the weights associated with each weak learner in the ensemble. The process of those weight updates is described in Section 4.2.2.2. Generally, the idea is to increase the weights of good-performing weak learners and retain them in the ensemble, while decreasing the weight of weak learners and removing them if they fail to contribute positively. The latter part, i.e., managing the membership of the weak learners in the ensemble, is described in Section 4.2.2.3.

4.2.2.1 Weak Learners

The first design choice involved determining how to train the weak learners and which type to use as the base of the ensemble. We decided on decision stumps [Iba and Langley, 1992], which are decision trees of depth one, so each weak learner uses a single feature and a split point. This approach is straightforward in terms of determining whether a stump can be used, i.e., whether the necessary split decision is present in the instance's feature space. However, since we operate in an online learning setting, decision stumps are not inherently suited to streaming data. To address this, we employ a small instance buffer that collects recent data points in a sliding window. Whenever a new decision stump is learned, it is trained on these recent instances. This strategy ensures stable, consistent predictions from each stump as long as it remains part of the ensemble.

Each incoming instance is placed into the instance buffer, which uses a first-in-first-out policy to retain the last N instances of the data stream. When performance degrades or feature availability changes, new stumps can be learned from the buffer. During the weak learner initialization, we also decide on the accepted feature space $\mathcal{F}_b$ based on the δ parameter, which indicates a portion of all the features observed so far at time t with $\mathcal{F}_t$. Comparing this to the random forest (cf. Algorithm 2.3), this corresponds to line six of the referenced algorithm, where a subset of the available features is selected. As described, this introduces a form of bagging that increases ensemble diversity (cf. Section 2.3).

The DynFo algorithm begins with B weak learners, denoted $\mathcal{L}_b$, for $b = 1, \ldots, B$. Each weak learner is continuously retrained on the current instance buffer and their accepted feature space $\mathcal{F}_b$. Early on, frequent re-training

Algorithm 4.2

Input:	$\mathcal{L}$	ensemble with all weak learners
	δ	chance to add feature to the accepted feature space
	B	size of ensemble
Output:	$\mathcal{L}$	updated ensemble

UpdateAcceptedFeatureSpace($\mathcal{L}$, δ, B)

1: $\mathcal{F}_t^{new} \leftarrow \mathcal{F}_t \setminus \mathcal{F}_{t-1}$
2: **for** $b = 1, 2, \ldots, B$ **do**
3: $\quad \mathcal{L}_b \leftarrow \mathcal{L}[b]$
4: $\quad$ **for** each F_d contained in $\mathcal{F}_t^{new}$ **do**
5: $\quad\quad$ **if** Random() $< \delta$ **then**
6: $\quad\quad\quad$ Add F_d to accepted feature space $\mathcal{F}_b$ of $\mathcal{L}_b$
7: $\quad\quad$ **end if**
8: $\quad$ **end for**
9: $\quad \mathcal{L}[b] \leftarrow \mathcal{L}_b$
10: **end for**
11: **return** $\mathcal{L}$

allows the ensemble to stabilize. Later, retraining can occur less frequently unless changes in the feature space or performance prompt further updates.

Algorithm 4.2 introduces the required steps to adjust the accepted feature space for each weak learner. It begins by determining which features have appeared since the previous time step, storing them in $\mathcal{F}_t^{new}$. Each weak learner in the ensemble is then examined in turn. For every newly emerged feature $F_d \in \mathcal{F}_t^{new}$, the algorithm makes a decision with probability δ to include it into the accepted feature space. If the outcome is favorable, the feature F_d is added to that learner's accepted feature space, effectively allowing the learner to use F_d for future splits when relearning. Once all newly arrived features have been assessed for each learner, the updated ensemble is returned. This process ensures that evolving features have a chance to be incorporated into the model, thereby preserving flexibility as the data changes over time.

4.2.2.2 Weight updates

Following the idea of weighted random forests [Winham et al., 2013] (cf. Section 2.3.3). Each decision stump in the ensemble has an associated weight. The weight updates are performed in lines eleven and twelve of Algorithm

4.1, respectively. In the following, we will first discuss the weight update strategy and then elaborate on the need for a penalty.

The following function of $\mathbb{I}(y_t = \hat{y}_t)$ in Equation 4.2.1 is used to indicate if a prediction made by a weak learner was correct or not (as similarly used in the evaluation, cf. Section 2.4). If the weak learner's prediction is correct, the function returns a one; otherwise, it returns a zero.

$$\mathbb{I}(y_t = \hat{y}_t) = \begin{cases} 1 & \text{if } y_t = \hat{y}_t \\ 0 & \text{otherwise} \end{cases} \tag{4.2.1}$$

The indicator function can then be used to update the weights based on Equation 4.2.2. This equation updates a weight with respect to α by multiplying 2α with the indicator function and adding the current weight w_b to it. This is then divided by $\alpha + w_b$. As previously mentioned, the weight for each weak learner is initialized to one. If the prediction of the weak learner is correct, it will increase; otherwise, it will decrease. If the weight of a weak learner is below one, it increases faster and decreases more slowly. In contrast, a weight above one increases more slowly and decreases faster for correct and wrong predictions, respectively.

$$w_b = \frac{\mathbb{I}(y_t = \hat{y}_{b,t}) * 2\alpha + w_b}{1 + \alpha} \tag{4.2.2}$$

By increasing the weight of the weak learners with correct predictions and decreasing the weight of those with incorrect predictions, we can ensure that impactful weak learners remain in the ensemble and irrelevant ones are dropped. Due to the update mechanism, the weights will increase more slowly the higher the current weight of a weak learner is, and vice versa. This bounding of the weights enables good weak learners to be replaced more quickly with the proposed algorithm if they start to become obsolete, i.e., their individual performance deteriorates.

As described, we keep a weight w_b for every weak learner in our ensemble, which is initialized with a value of one. There is no upper bound on the weight of a weak learner in the ensemble. So, given the case that a weak learner made a split on a feature that is scarcely available, this weak learner would never get replaced based on the proposed update strategy, as this strategy only updates those weak learners that could make a prediction for

Algorithm 4.3

Input:	$\mathcal{L}$	ensemble with all weak learners
	$\mathbf{w}$	current weights
	(x_t, y_t)	current instance and class
	α	magnitude of weight update
	ϵ	penalty
	B	size of ensemble
Output:	$\mathbf{w}$	updated weights

UpdateWeights($\mathcal{L}$, $\mathbf{w}$, (x_t, y_t), α, ϵ, B)

for $b = 1, 2, \ldots, B$ **do**
  $w_b \leftarrow \mathbf{w}[b]$
  $\mathcal{L}_b \leftarrow \mathcal{L}[b]$
  if $\mathcal{L}_b^{F_d} \notin \mathcal{F}(x_t)$ **then**
    $w_b \leftarrow w_b - \epsilon$
  else
    $\hat{y}_{b,t} \leftarrow \mathcal{L}_b(x_t)$
    $w_b \leftarrow \frac{\mathbb{I}(y_t = \hat{y}_{b,t}) * 2\alpha + w_b}{1+\alpha}$
  end if
  $\mathbf{w}[b] \leftarrow w_b$
end for
return $\mathbf{w}$

the current instance. Consequently, this means that we need to apply a penalty to the weight of this weak learner so that, in the case it provides very low to no benefit, another decision stump will eventually replace it with a more reliable feature. This is done by subtracting a constant penalty ϵ from the weight of that weak learner as shown in Equation 4.2.3, if the split of the weak learner is not in the feature space of $\mathcal{F}(x_t)$, i.e. $\mathcal{L}_b^{F_d} \notin \mathcal{F}(x_t)$. Then, the weight is updated according to

$$w_b = w_b - \epsilon. \tag{4.2.3}$$

In Algorithm 4.3, the computation of weight updates is illustrated. The algorithm iterates over each weak learner in the ensemble, examining its current weight and determining how it should be updated based on the newly arrived instance (x_t, y_t). First, if the feature on which a learner splits is not present in the current instance's feature set, a penalty ϵ is subtracted from that learner's weight. Otherwise, the learner makes a prediction $\hat{y}_{b,t}$ for (x_t, y_t). If the prediction is correct, the learner's weight increases, and if

incorrect, it decreases—both according to a factor controlled by α. Specifically, the weight is adjusted by forming a fraction whose numerator considers the correctness term $\mathbb{I}(y_t = \hat{y}_{b,t})$ and whose denominator ensures the update magnitude remains bounded. After this process, the newly calculated weight is stored back into the ensemble.

4.2.2.3 Adaptation of Weak Learners

The last part of the learning strategy of DynFo is the dynamic adaptation of the number of decision stumps used as weak learners, i.e., adding or deleting weak learners.

As varying feature spaces are highly dynamic, there is a need to increase the number of weak learners if unknown or new features arise and the previous number of weak learners is insufficient to deal with the feature space at hand. Therefore, we added a mechanism to track the error rate over a sliding window. If the error rate surpasses the threshold γ, we add a new weak learner to the ensemble and also update γ according to the error rate. This step should detect if performance plummets based on the experienced performance so far, and then introduce further weak learners if needed.

Algorithm 4.4 introduces the necessary steps to increase the size of the ensemble. The procedure begins by using the current ensemble $\mathcal{L}$ to predict a label for the incoming instance (x_t, y_t). It records whether this prediction was correct by adding an indicator (1 or 0) to the sliding window $\mathbf{r}$, removing the oldest entry if necessary to keep $\mathbf{r}$ at a fixed size. The mean of $\mathbf{r}$, denoted e, serves as an estimate of the ensemble's recent accuracy. If e falls below the threshold γ, the procedure deems the ensemble's performance insufficient and increases the ensemble size B by creating a new weak learner $\mathcal{L}_B$. This learner is added to $\mathcal{L}$ with an initial weight of 1, and γ is updated to the new accuracy level e. If the average accuracy remains above γ, no additional learner is introduced. Finally, the algorithm returns the updated ensemble, weights, sliding window, ensemble size, and threshold, allowing the model to adapt if its performance starts to degrade.

On the other hand, we have a strategy in place to relearn as well as drop weak learners who are performing poorly. Hereby, we utilize two bounds defined by the parameters θ_1 and θ_2, where the value range is between zero and one. The general idea of this approach is that the good decision stumps

Algorithm 4.4

Input:	$\mathcal{L}$	ensemble with all weak learners
	$\mathbf{w}$	current weights
	(x_t, y_t)	current instance and class
	B	size of ensemble
	$\mathbf{r}$	sliding window of results
	γ	performance tracking threshold
Output:	$\mathcal{L}$	updated ensemble
	$\mathbf{w}$	updated weights
	$\mathbf{r}$	updated result window
	B	new ensemble size
	γ	adjusted γ

IncreaseEnsembleSize($\mathcal{L}$, $\mathbf{w}$, (x_t, y_t), B, $\mathbf{r}$, γ)

1: $\hat{y} \leftarrow \mathcal{L}(x_t)$
2: add $\mathbb{I}(y_t = \hat{y}_t)$ to $\mathbf{r}$ based on FIFO principle
3: $e \leftarrow \textsc{Mean}(\mathbf{r})$
4: **if** $e < \gamma$ **then**
5: $\quad B \leftarrow B + 1$
6: $\quad$ initialize new $\mathcal{L}_B$
7: $\quad \mathcal{L}[B] \leftarrow \mathcal{L}_B$
8: $\quad w_B \leftarrow 1$
9: $\quad \mathbf{w}[B] \leftarrow w_B$
10: $\quad \gamma \leftarrow e$
11: **end if**
12: **return** $\mathcal{L}$, $\mathbf{w}$, $\mathbf{r}$, B, γ

are kept in the ensemble, while the mediocre decision stumps are relearned by chance, and the worst decision stumps are dropped; hence the feature space that was chosen by the bagging parameter δ for this weak learner is dropped as well and if the performance plummets based on this decision, we get the chance to add new weak learners to the ensemble in accordance to line 15 of the learning algorithm (cf. Algorithm 4.1).

The algorithm, presented in Algorithm 4.5, begins by initializing an array $\mathbf{I}_{delete}$, which is supposed to keep track of the weak learners and their associated weights that are pruned. Given the current size of the ensemble B, we iterate for $b = 1, \ldots, B$ over each weak learner and their associated weight. For each learner $\mathcal{L}_b$ with weight w_b, it compares w_b to two thresholds. The upper threshold θ_2 is applied by taking the θ_2-quantile of all weights. Learners whose weights meet or exceed this quantile are considered strong

Algorithm 4.5

Input:	$\mathcal{L}$	ensemble with all weak learners
	B	size of ensemble
	θ_1	lower bound
	θ_2	upper relative bound
Output:	$\mathcal{L}$	updated ensemble
	$\mathbf{w}$	updated weights
	B	new ensemble size

PRUNEENSEMBLE($\mathcal{L}$, B, θ_1, θ_2)

initialize $\mathbf{I}_{delete}$
for $b = 1, 2, \ldots, B$ **do**
  $\mathcal{L}_b \leftarrow \mathcal{L}[b]$
  if $w_b \geq$ QUANTILE(θ_2, $\mathbf{w}$) **then**
    keep $\mathcal{L}_b$ in $\mathcal{L}$
  else if $\theta_1 \leq w_b <$ QUANTILE(θ_2, $\mathbf{w}$) **then**
    if RANDOM() $> \beta$ **then**
      RELEARN($\mathcal{L}_b$)
    end if
  else
    add b to $\mathbf{I}_{delete}$
  end if
end for
$\mathbf{w} \leftarrow$ DELETE($\mathbf{w}$, $\mathbf{I}_{delete}$)
$\mathcal{L} \leftarrow$ DELETE($\mathcal{L}$, $\mathbf{I}_{delete}$)
$B \leftarrow B - |\mathbf{I}_{delete}|$
return $\mathcal{L}$, $\mathbf{w}$, B

and remain in the ensemble unchanged. If a learner's weight falls between the lower bound θ_1 and the θ_2-quantile, it has borderline performance. In such cases, a random check (against a parameter β) determines whether the learner will be relearned. Finally, learners whose weight is below θ_1 are marked for removal. Finally, these underperforming learners are pruned from both the ensemble and the weight vector, and the ensemble size B is adjusted accordingly. This ensures that only the most relevant or potentially improvable learners persist, while weaker ones are eliminated.

4.2.2.4 Example

Figure 4.2 illustrates an example of the algorithm at time $t = n$. Here, we have a small ensemble $\{\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3\}$, where each learner is a decision stump.

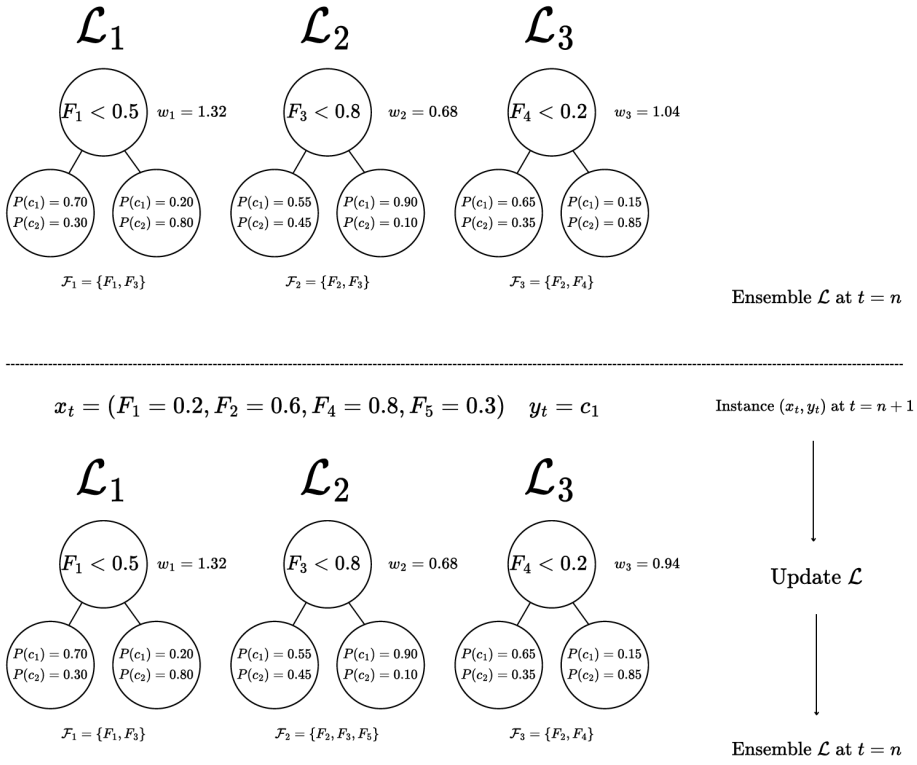


Figure 4.2: Example of weight updates and accepted feature space, based on a new data instance at $t = n + 1$.

Up to this point, four features have emerged, $\mathcal{F} = \{F_1, F_2, F_3, F_4\}$. Stump $\mathcal{L}_1$ splits on F_1 at 0.5, $\mathcal{L}_2$ splits on F_3 at 0.8, and $\mathcal{L}_3$ splits on F_4 at 0.2. Each stump also maintains a set of accepted features, determined by the δ parameter.

At time $t = n + 1$, the algorithm receives a new instance that contains a newly emerged feature F_5 but is missing feature F_3. This instance is then used to update the ensemble from $t = n$. The first step, according to the algorithm, is to refresh each learner's feature space. In our example, F_5 is added to $\mathcal{L}_2$ as a potential feature. Next, we check the value of γ to determine if a new learner should be added, which does not occur in this example.

Following the algorithm, any learner splitting on a feature absent in the current instance space is penalized by ϵ. Assuming $\epsilon = 0.01$, we lower $\mathcal{L}_2$'s weight accordingly because it splits on F_3, which is missing. As $\mathcal{L}_1$ and $\mathcal{L}_3$ split on features present in the instance space, no penalty applies to them.

Algorithm 4.6 Predicting with Dynamic Forest

Input: x instance to predict class for

Initialize class weights $\hat{\mathbf{w}}$ with $\hat{w}_c = 0$ for $c = 1, \ldots, C$
for $b = 1, 2, \ldots, B$ **do**
  if $\mathcal{L}_b^{F_d} \notin \mathcal{F}(x)$ **then**
    continue
  end if
  for $c = 1, \ldots, C$ **do**
    $p_{c,b} \leftarrow P_{\mathcal{L}_b}(y = c \mid x)$
    $\hat{w}_c \leftarrow \hat{w}_c + (p_{c,b} * w_b)$
  end for
end for
return $\underset{c \in \mathcal{Y}}{\operatorname{argmax}}\ \hat{\mathbf{w}}$

Nonetheless, we still update their weights according to Eq. 4.2.2. With $\alpha = 0.1$, $\mathcal{L}_1$ increases to a weight of 1.5 for making a correct prediction, whereas $\mathcal{L}_3$ drops to 0.94.

In the final step, the algorithm decides, based on θ_1 and θ_2, whether to retain, relearn, or remove any learner. Although this is not depicted, $\mathcal{L}_1$ would likely remain, $\mathcal{L}_2$ might be relearned or dropped depending on the threshold, and $\mathcal{L}_3$ would be relearned in line with β.

4.2.3 Predicting with Dynamic Forest

The idea behind predicting with DynFo is grounded in probability distribution summation Clark and Boswell [1991] (cf. Section 2.3.3). Conceptually, each model's distribution indicates how likely each class label is for a given instance. By adding up these distributions combined with their respective weights, we can obtain the final result with Equation 4.2.3.

$$\hat{y} = \underset{c \in \mathcal{Y}}{\operatorname{argmax}} \sum_{b=1}^{B} w_b * P_{\mathcal{L}_b}(y = c \mid x) \tag{4.2.4}$$

In Algorithm 4.6, we outline the steps taken to compute $\hat{y}$. To predict the class of a new instance x, the algorithm starts by initializing an array $\hat{\mathbf{w}}$ of size C, which is the number of classes. Each entry $\hat{w}_c$ is set to zero.

The ensemble of B weak learners is then processed one by one. For each learner $\mathcal{L}_b$, the algorithm first checks whether the feature it split on is present in x. If that feature is missing, the learner is skipped. Otherwise, the learner outputs a probability distribution over the C classes, denoted by $p_{c,b} = P_{\mathcal{L}_b}(y = c \mid x)$.

Next, the algorithm scales each probability $p_{c,b}$ by that learner's weight w_b and adds the result to the corresponding entry $\hat{w}_c$. In other words, it accumulates each learner's contribution into a running total per class. By the end of this loop, $\hat{w}_c$ holds a sum of weighted probabilities for class c, taken across all learners.

Finally, the algorithm selects the class $\hat{y}$ that maximizes $\hat{w}_c$. In doing so, it leverages summation over probability distributions: each learner provides a distribution over classes, and these distributions are combined via a weighted sum.

4.3 Online Random Feature Forests

Online Random Feature Forests for Feature Space Variabilities (ORF^{3}V) is our second approach, published in Schreckenberger et al. [2023], for handling varying feature spaces in a dynamically adaptive and interpretable way. Built on the random forest concept (cf. Section 2.3), this method maintains an ensemble of so-called feature forests, whose collective predictions, which are obtained by averaging their individual results, form a strong overall predictor.

We first introduce the notation used in ORF^{3}V. This is followed by a high-level description of the learning algorithm, along with the parameters used in the algorithm. Subsequently, we describe the learning process in terms of storing feature statistics, generating feature forests, updating the feature forests themselves, updating weights in the ensemble, and pruning feature forests. Finally, we describe how to predict a given observation with a learned model.

4.3.1 Notation ORF^{3}V

In our notation, $\mathcal{L}$ is used to refer to the ensemble of feature forests $\mathcal{L}_d$. Each feature forest $\mathcal{L}_d$ corresponds to one observed feature $F_d \in \mathcal{F}_t$, where $\mathcal{F}_t$ are

Parameter	Description
N	size of feature forest
α	magnitude of impact on weight update
$\rho_{replace}$	replacement strategy
r	replacement interval
λ	compression for feature statistics Φ
δ	probability for correct pruning decision

Table 4.2: Overview of input parameters for Algorithm 4.7.

all the features seen so far by the algorithm at step t, with $d = 1, \ldots, D_t$, where D_t is the dimensionality of $\mathcal{F}_t$. For each feature forest $\mathcal{L}_d \in \mathcal{L}$, we have a corresponding weight of $w_d \in \mathbf{w}$. The ensemble $\mathcal{L}$ is also used analogously to the function we want to approximate $\hat{f}$. Hence, we can use $\mathcal{L}(x) = \hat{y}$ to make a prediction. Furthermore, each feature forest can also be used to make a prediction based on an instance x with $\mathcal{L}_d(x) = \hat{y}_d$. As opposed to the sliding window approach used in the Dynamic Forest, we use Φ to approximate a feature's distribution, where Φ_d corresponds to said distribution of the feature F_d.

4.3.2 Learning with ORF^{3}V

The high-level steps for learning with ORF^{3}V are shown in Algorithm 4.7. In the following sections, we provide an introduction to the parameters used and elaborate on the specific parts of the algorithm in subsequent subsections.

The algorithm offers six parameters (cf. Table 4.2) that can be adjusted, making it flexible and adaptable to the expected data characteristics. The first parameter is N ($N > 0$), which determines the number of decision stumps used in each feature forest. The parameter α determines the impact on the weight update for each feature forest in the ensemble. The value range is $\alpha \in (0; 1]$, where a lower α resembles a less aggressive weight update strategy, and a higher α resembles a more aggressive update strategy. Furthermore, the replacement strategy needs to be determined by $\rho_{replace}$. The replacement strategy can either be $\rho_{replace} = oldest$ or $\rho_{replace} = random$. The replacement interval r determines the number of instances we must observe before applying the replacement strategy, with $r > 0, r \in \mathbb{N}$ being a

possible value. The parameter λ sets the compression for the data storage, with $\lambda > 0$. Finally, $\delta \in (0; 1]$ determines the probability of making a correct pruning decision. The first step is to loop over all time steps T. In each time step t, we start by receiving the labeled instance of the time step (x_t, y_t).

The stored feature statistics Φ are then updated according to the observed features and their values in x_t. It is then checked if some feature forests $\mathcal{L}_d$ in the ensemble need to be pruned according to the pruning strategy.

Subsequently, we iterate over every feature F_d in the universal feature space $\mathcal{F}_t$. Based on the stored feature stats Φ, we then generate a feature forest $\mathcal{L}_d$ containing N decision stumps for each feature F_d. Each new feature forest $\mathcal{L}_d$ is then added to the ensemble $\mathcal{L}$, and a corresponding weight $w_d = 1$ is initialized and stored in the weights $\mathbf{w}$.

Subsequently, we loop over every feature F_d in x_t and use the corresponding feature forest $\mathcal{L}_d$ to update the weights. Hereby, we use each feature forest $\mathcal{L}_d$ and make a prediction based on the corresponding value. If the prediction is correct, the respective feature forest $\mathcal{L}_d$ gains weight, and if the prediction is false, it loses weight according to the weight update strategy.

In the final step, we check whether we have reached the replacement interval r. If this is the case, we update each feature forest $\mathcal{L}_d$ present in the ensemble $\mathcal{L}$, according to the replacement strategy $\rho_{replace}$ and the stored feature stats Φ.

The ORF^{3}V approach differs from the DynFo in its reliance on feature-level statistics instead of instance storage for learning. In the DynFo algorithm, a buffer stores recent instances to support incremental updates and retraining of weak learners, making it sensitive to memory constraints and requiring direct access to raw data. In contrast, the ORF^{3}V algorithm aggregates feature values over time into statistical summaries (Φ_d) for each feature, allowing it to operate without explicitly storing instances. This approach reduces memory usage and simplifies adaptation to evolving feature spaces, as decisions are driven by feature-level approximations rather than raw instance data.

Algorithm Roadmap The ORF^{3}V approach has five core components. In the following, we will start by elaborating on the storage of the feature

Algorithm 4.7 Learning with ORF^{3}V

for $t = 1, 2, ...T$ **do**
  receive instance (x_t, y_t)
  $\Phi \leftarrow$ UpdateFeatureStatistics(Φ, (x_t, y_t), λ)
  for each d of feature F_d contained in $\mathcal{F}(x_t)$ **do**
    if CheckPrune($\mathcal{L}_d$, δ) **then**
      $\mathcal{L} \leftarrow \mathcal{L} \setminus \{\mathcal{L}_d\}$
    end if
  end for
  for each feature F_d observed in $\mathcal{F}_t$ with sufficient instances **do**
    $\mathcal{L}_d \leftarrow$GenerateFeatureForest(Φ_d)
    $\mathcal{L} \leftarrow \mathcal{L} \cup \{\mathcal{L}_d\}$
    $w_d \leftarrow 1$
    $\mathbf{w}[d] \leftarrow w_d$
  end for
  for each featue F_d contained in $\mathcal{F}(x_t)$ **do**
    $\mathbf{w} \leftarrow$ UpdateWeights($\mathcal{L}$, $\mathbf{w}$, α, (x_t, y_t))
  end for
  if $t \bmod r == 0$ **then**
    $\mathcal{L} \leftarrow$ Replace($\mathcal{L}$, $\rho_{replace}$, Φ)
  end if
end for

statistics in Section 4.3.2.1. Subsequently, we explain how feature forests are generated from the stored feature statistics in Section 4.3.2.2. After generating the feature forests, it is necessary to update them periodically; this procedure is explained in Section 4.3.2.3. In Section 4.3.2.4, we explain the weight updates for each of the feature forests in the ensemble. Finally, in Section4.3.2.5, we showcase how the ensemble employs pruning of feature forests to promote sparseness.

4.3.2.1 Storage of Feature Statistics

To efficiently store the feature values, we choose t-digest Dunning and Ertl [2019]. The t-digest algorithm is a data structure that enables online updates by estimating the cumulative distribution function (CDF) of the data seen so far, which can then be used to estimate rank statistics. According to Dunning [2021], t-digest can provide them with orders of magnitude better accuracy than previous algorithms. The size and the accuracy of the quantile estimation of a t-digest are determined by the compression for each t-digest.

Algorithm 4.8

Input:	Φ	feature statistics
	(x_t, y_t)	labeled instance
	λ	compression for feature statistics
Output:	Φ	updated feature statistics

UpdateFeatureStatistics(Φ, (x_t, y_t), λ)

1: **for** each d of feature F_d contained in $\mathcal{F}(x_t)$ **do**
2: $c \leftarrow$ obtain class from y_t
3: $\Phi_{c,d} \leftarrow \Phi[c, d]$
4: **if** $\Phi_{c,d}$, is not initialized **then**
5: $\Phi_{c,d} \leftarrow$ InitT-Digest(λ)
6: **end if**
7: $v_{t,d} \leftarrow x_t[d]$
8: $\Phi_{c,d} \leftarrow$ Insert($\Phi_{c,d}$, $v_{t,d}$, y_t)
9: $\Phi[c, d] \leftarrow \Phi_{c,d}$
10: **end for**
11: **return** Φ

In line 3 of Algorithm 4.7, we update the statistics for the features in the current instance x_t. Hereby, we use one t-digest for each class seen in a feature. The t-digests are updated with the observed value for each feature-class combination. Storing the data in this manner also has the advantage that the algorithm can easily handle multi-class learning problems, as it only needs to initialize a new t-digest for each observed class. Furthermore, this also allows dealing with classes that are not known beforehand, as t-digest can be generated and updated on the fly.

Algorithm 4.8 updates the feature-level statistical summaries Φ by incorporating information from a new labeled instance (x_t, y_t). It ensures that feature distributions are maintained separately for each class label c, supporting class-conditional analysis.

For each feature F_d present in the instance's feature set $\mathcal{F}(x_t)$, the algorithm identifies the class c carried by the instance in y_t. It retrieves the current statistical summary $\Phi_{c,d}$ for feature F_d under class c from Φ. If no summary exists for this combination, the algorithm initializes it using a t-Digest structure configured with the compression parameter λ.

The algorithm then extracts the value $v_{t,d}$ of feature F_d from the instance x_t. This value, along with the label y_t, is inserted into the summary $\Phi_{c,d}$, updating the class-conditional statistics for the feature. Once updated, the

Algorithm 4.9

Input:	Φ_d	feature statistics for feature F_d
	N	ensemble size
Output:	$\mathcal{L}_d$	feature forest for feature F_d

GenerateFeatureForest(Φ, N)

1: initialize **S** as array
2: $v_{min}, v_{max} \leftarrow min(\Phi_d), max(\Phi_d)$
3: **for** $i = 1, \ldots, 2N$ **do**
4: $\quad \theta_i \leftarrow$ Random(v_{min},v_{max})
5: $\quad \mathcal{T}_{d,\theta_i} \leftarrow$ InitDecisionStump(θ_i)
6: $\quad \mathbf{S}[i] \leftarrow (\mathcal{T}_{d,\theta_i}, AGI(\mathcal{T}_{d,\theta_i}))$
7: **end for**
8: $\mathcal{L}_d \leftarrow$ GetTopN(**S**)
9: **return** $\mathcal{L}_d$

modified summary is stored back into $\Phi[c, d]$. After processing all features in $\mathcal{F}(x_t)$, the algorithm returns the updated Φ.

4.3.2.2 Generation of Feature Forests

Based on the stored feature statistics, ORF^{3}V randomly generates multiple decision stumps, in total N, for each feature, which together form a feature forest. In the following, we elaborate on the generation of a single feature forest.

The generation of the feature forests is executed in line ten of Algorithm 4.7. Hereby, we generate a feature forest for every new feature after having seen sufficient instances for the observed features at time step t (cf. Algorithm 4.9).

In the algorithm, $2N$ decision stumps are generated randomly based on the stored feature statistics and temporarily stored in the array **S**. The splits for each decision stump in the feature forest are randomly generated in the interval of the minimum and maximum value seen for a feature. The N best decision stumps are retrieved from **S** according to the approximate Gini impurity. The Gini impurity for a decision stump can be approximated based on the statistics stored in the respective t-digests of the feature, as described below.

Let $\mathcal{T}_{d,\theta}$ denote a decision stump thresholding feature F_d at value θ. Let N_c denote the number of instances of class c observed so far, and let $N =$

$\sum_{c=1}^{C} N_c$ be the overall number of observed instances. The *approximated Gini impurity* (AGI) is computed as the weighted Gini impurity of both branches of the decision stump:

$$\mathrm{AGI}(\mathcal{T}_{d,\theta}) = 1 - \left(\frac{N_B}{N} g_B + \frac{N_A}{N} g_A \right). \tag{4.3.1}$$

Here, N_B and N_A are the estimated number of samples where the values of feature F_d are *below* or *above* the threshold θ, respectively. For online learning, the training samples are not present as a batch, and thus, these counts cannot be directly measured. Instead, they can be computed as the expectation of the number of samples below (or above) the threshold θ:

$$N_B = \sum_{c=1}^{C} \tilde{P}(v < \theta \mid c)\, N_c, \tag{4.3.2}$$

$$N_A = \sum_{c=1}^{C} \tilde{P}(v \geq \theta \mid c)\, N_c. \tag{4.3.3}$$

Here, $\tilde{P}(v < \theta \mid c)$ and $\tilde{P}(v \geq \theta \mid c)$ express the estimated probability that values of feature F_d are above (or below) a threshold θ, given that the sample belongs to class c. The probabilities can be estimated based on the cumulative distribution functions, which are given by the online data storage t-digest. Hereby, $\Phi_{c,d}(\theta) = \tilde{P}(v < \theta \mid c)$ and $1 - \Phi_{c,d}(\theta) = \tilde{P}(v \geq \theta \mid c)$, where the distribution $\Phi(\theta)$, gives us the probability of values v in a feature below a threshold θ. This approximation of the probability allows us to compute N_B and N_A.

The Gini impurity g_B of one of the branches of the decision stump is computed as follows (similar computations apply for g_A):

$$g_B = \sum_{c=1}^{C} P(c \mid F_d < \theta)^2, \tag{4.3.4}$$

where $P(c \mid F_d < \theta)$ is the class posterior for samples where F_d is *below* the threshold θ of decision stump $\mathcal{T}_d$, and can be computed using Bayes' theorem:

$$P(c \mid F_d < \theta) = \frac{P(F_d < \theta \mid c)\, N_c}{N_B}. \tag{4.3.5}$$

4.3.2.3 Update of Feature Forests

Since the algorithm is designed to operate in a dynamic environment where new data emerge constantly, there is a need to update the feature forests. As more data becomes available, more accurate decision stumps can be generated. The feature forests are updated in line 19 of Algorithm 4.7 by using the specified replacement strategy $\rho_{replace}$ and the stored feature statistics Φ. The update is only executed every r-th time step. This should be set according to the environment's characteristics. More dynamics require more frequent updates. To update a feature forest, we propose two replacement strategies ($\rho_{replace}$), i.e., *replace random* and *replace oldest*.

In the first strategy, $\rho_{replace} = random$, we propose to replace each decision stump in the feature forest based on a random chance. Hereby, we iterate over all decision stumps in the feature forest and replace them by chance. For each replacement, we only generate one new decision stump with a new random splitting point.

The second strategy, $\rho_{replace} = oldest$, aims to replace the oldest decision stump in the feature forest. This also requires keeping track of the age of each decision stump. When the oldest decision stump is determined, we reuse the N parameter to generate a pool of N decision stumps and then follow a similar procedure as in the generation step by picking the single best of the N generated decision stumps in the pool according to the approximate Gini gain.

An in-depth description of the first strategy in the update mechanism is given in Algorithm 4.10. The process begins by checking the replacement strategy specified by ρ_{replace}. The algorithm iterates over all features $d = 1, \ldots, D$. For each feature F_d, the algorithm retrieves its corresponding feature forest $\mathcal{L}_d$ and computes the minimum ($v_{\min}$) and maximum ($v_{\max}$) values from the feature statistics Φ_d. These values define the range for generating the splitting point.

Within the feature forest $\mathcal{L}_d$, the algorithm iterates over each decision stump $i = 1, \ldots, N$. For learners flagged for replacement, a new splitting point θ_i is randomly sampled from the range $[v_{\min}, v_{\max}]$. A new decision stump is initialized with this splitting point ($\mathcal{T}_{d,\theta_i}$) and replaces the corresponding decision stump in the feature forest $\mathcal{L}_d[i]$.

Once all replacements for a feature forest are complete, the updated

Algorithm 4.10

Input:	$\mathcal{L}$	ensemble of feature forests
	$\rho_{replace}$	replacement strategy
	Φ	feature statistics
	N	ensemble size
Output:	**w**	updated weights

UPDATE($\mathcal{L}$, $\rho_{replace}$, Φ, N)

1: **if** $\rho_{replace} = random$ **then**
2: **for** $d = 1, \ldots, D$ **do**
3: $\mathcal{L}_d \leftarrow \mathcal{L}[d]$
4: $v_{min}, v_{max} \leftarrow min(\Phi_d), max(\Phi_d)$
5: **for** $i = 1, \ldots, N$ **do**
6: **if** RANDOMREPLACE() **then**
7: $\theta_i \leftarrow$ RANDOM(v_{min},v_{max})
8: $\mathcal{T}_{d,\theta_i} \leftarrow$ INITDECISIONSTUMP(θ_i)
9: $\mathcal{L}_d[i] \leftarrow \mathcal{T}_{d,\theta_i}$
10: **end if**
11: **end for**
12: $\mathcal{L}[d] \leftarrow \mathcal{L}_d$
13: **end for**
14: **else**
15: ... //$\rho_{replace} = oldest$
16: **end if**
17: **return** $\mathcal{L}$

feature forest $\mathcal{L}_d$ overwrites the old feature forest in the ensemble $\mathcal{L}$. Finally, the algorithm returns the updated ensemble $\mathcal{L}$, reflecting the adjustments made based on the chosen replacement strategy.

4.3.2.4 Weight updates of the ensemble

The weight updates are performed in line 16 of Algorithm 4.7. A detailed description is given in Algorithm 4.11. For each feature forest $\mathcal{L}_d$, we have a respective weight w_d in the ensemble, which is initialized with a value of one. The indicator function of $\mathbb{I}(y_t, \hat{y}_{d,t})$ is used to indicate if a prediction made by the d-th feature forest was correct or not. The indicator function can then be used to update the weights based on Equation 4.3.6. This equation updates a weight with respect to α by multiplying 2α with the indicator function and adding the current weight w_d to it. This is then divided by $1+\alpha$. Similarly to DynFo, if the prediction of the feature forest is correct, it

Algorithm 4.11

Input:	$\mathcal{L}$	ensemble of feature forests
	$\mathbf{w}$	current weights
	(x_t, y_t)	current instance and class
	α	magnitude of weight update
Output:	$\mathbf{w}$	updated weights

UpdateWeights($\mathcal{L}$, $\mathbf{w}$, (x_t, y_t), α)

1: **for** $d = 1, \ldots, D$ **do**
2: $w_d \leftarrow \mathbf{w}[d]$
3: $\mathcal{L}_d \leftarrow \mathcal{L}[d]$
4: $\hat{y}_{b,t} \leftarrow \mathcal{L}_b(x_t)$
5: $w_d \leftarrow \frac{\mathbb{I}(y_t = \hat{y}_{d,t}) * 2\alpha + w_d}{1+\alpha}$
6: $\mathbf{w}[d] \leftarrow w_d$
7: **end for**
8: **return** $\mathbf{w}$

will increase; otherwise, it will decrease. If a weight is below one, it increases faster and decreases slower and vice versa for weights above one, for correct and wrong predictions, respectively.

$$w_d = \frac{\mathbb{I}(y_t = \hat{y}_{d,t}) * 2\alpha + w_d}{1 + \alpha} \tag{4.3.6}$$

Furthermore, the weight update function is bounded, restricting feature forests that were formed on features that emerged earlier than other features from gaining too much impact on the final prediction. This allows feature forests for features that emerge later, which are potentially beneficial, to catch up in terms of impact on the final prediction, as well as highlighting the importance of these features.

4.3.2.5 Pruning of Feature Forests

While one characteristic of data streams with varying feature spaces is that new data streams may emerge, another characteristic is that these data streams may also vanish. However, sometimes these data streams may produce missing data and provide additional values at a later point in time. To decide if a feature vanished completely, we propose to use the Hoeffding bound while tracking the general missing ratio of a feature as well as the ratio of missing values over the last m instances seen by the algorithm. This

has the advantage that we have a dynamic threshold for each feature, given that some may produce more sparseness than others.

The Hoeffding bound Hoeffding [1994] states that, with probability at least $1 - \delta$, the true mean μ of a random variable v, bounded in range R, satisfies

$$\mu \geq \bar{v} - \epsilon, \quad \text{where} \quad \epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}.$$

Here, $\bar{v}$ is the empirical mean over the last m observations. In our case, v is a binary variable indicating the availability of a given feature F_d: it is 1 if the feature is present in an instance, and 0 otherwise. Accordingly, we set $R = 1$.

Let $\bar{\Omega}(F_d)$ denote the average availability of feature F_d across all observed instances, and let $\bar{\Omega}_{\text{window}}(F_d)$ denote the average availability computed over a recent sliding window of size m. Define the deviation

$$\Delta\bar{\Omega} = \bar{\Omega}(F_d) - \bar{\Omega}_{\text{window}}(F_d).$$

Under the assumption that the availability of F_d is stationary, the Hoeffding bound guarantees that $\Delta\bar{\Omega} \leq \epsilon$ with probability at least $1-\delta$. Therefore, if $\Delta\bar{\Omega} > \epsilon$, this indicates that the feature has recently become less available than expected based on its historical presence. In such cases, we consider the feature as vanished and remove the corresponding feature forest $\mathcal{L}_d$ from the ensemble $\mathcal{L}$, along with its associated statistics stored in Φ.

4.3.3 Predicting with ORF^{3}V

Essentially, we follow the same idea as introduced by the DynFo for predicting with the ORF^{3}V (cf. Section 4.2.3). The approach has its theoretical grounding in the summation of probability distributions Clark and Boswell [1991] (cf. Section 2.3.3). Conceptually, each feature forest's distribution indicates how likely each class label is for a given instance. By adding up these distributions combined with their respective weights, we can obtain the final result with Equation 4.3.3.

$$\hat{y} = \underset{c \in \mathcal{Y}}{\operatorname{argmax}} \sum_{d=1}^{D} w_d * P_{\mathcal{L}_d}(y = c \mid x) \tag{4.3.7}$$

Algorithm 4.12 Predicting with ORF^{3}V

Input: x instance to predict class for

Initialize class weights $\hat{\mathbf{w}}$ with $\hat{w}_c = 0$ for $c = 1, \ldots, C$
for $d = 1, 2, \ldots, D$ **do**
  if $F_d \notin \mathcal{F}(x)$ **then**
    continue
  end if
  for $c = 1, \ldots, C$ **do**
    $p_{c,d} \leftarrow P_{\mathcal{L}_d}(y = c \mid x)$
    $\hat{w}_c \leftarrow \hat{w}_c + (p_{c,d} * w_d)$
  end for
end for
return $\underset{c \in \mathcal{Y}}{\operatorname{argmax}}\ \hat{\mathbf{w}}$

The ORF^{3}V prediction algorithm, as given in Algorithm 4.12, determines the class label for an input instance x by aggregating predictions from individual feature forests. It begins by initializing a set of class weights $\hat{\mathbf{w}} = \{\hat{w}_1, \hat{w}_2, \ldots, \hat{w}_C\}$ to zero, where each $\hat{w}_c$ corresponds to a class c.

For each feature F_d in the total feature space $\mathcal{F}$, the algorithm first checks whether F_d is present in the input instance x. If F_d is missing, it skips the corresponding feature forest $\mathcal{L}_d$. Otherwise, the feature forest $\mathcal{L}_d$ provides a probability distribution $P_{\mathcal{L}_d}(y = c \mid x)$ over the classes. For each class c, the algorithm calculates a weighted contribution to $\hat{w}_c$ by multiplying the probability $p_{c,d}$ with the forest's weight w_d. This value is then added to $\hat{w}_c$, accumulating evidence for class c across all feature forests.

Once all relevant feature forests have contributed to the class weights, the algorithm selects the class c with the highest $\hat{w}_c$ as the final prediction. This process effectively combines probabilistic predictions from individual feature forests, weighted by their importance, to make a decision.

4.4 Related Work

The two approaches, DynFo and ORF^{3}V, are closely related to two distinct areas of research. The first area addresses feature space variability, which both methods are designed to handle by adapting to evolving feature sets in dynamic environments. The second area focuses on the model class,

specifically tree-based and tree-ensemble approaches in an online learning setting. At the time these approaches were published, no other methods leveraging this model class had been introduced to address the challenges of feature space variability in online learning scenarios.

4.4.1 Online Learning in Varying Feature Spaces

In online learning, a learning algorithm incrementally builds a prediction model from sequentially arriving data instances. Online learning algorithms are generally categorized into first-order and second-order approaches [Zhang et al., 2016]. First-order algorithms rely on first-order information, such as gradients, to update the model parameters. Examples include the Perceptron algorithm [Rosenblatt, 1958, Freund and Schapire, 1999] and Online Gradient Descent [Zinkevich, 2003]. In contrast, second-order algorithms leverage the underlying structure between features, incorporating additional information, such as feature correlations, to guide the learning process [Zhang et al., 2014].

However, traditional online learning methods are not equipped to handle varying feature spaces, as they assume the feature space remains constant throughout the learning process. To address this limitation, early studies focused on scenarios with a monotonically increasing feature space [Gomes et al., 2013, Zhang et al., 2015, 2016]. The key challenge in this setting was to initialize the learning weights of emerged features in a way that provides an informed starting point, allowing the online learner to achieve faster convergence compared to random initialization.

Subsequent research further generalized this setting to account for open feature spaces [Hou et al., 2017, Hou and Zhou, 2017, Beyazit et al., 2019, He et al., 2019b, Zhang et al., 2020, Hou et al., 2021a,b, He et al., 2021b]. In these scenarios, new features can emerge dynamically, while previously observed features may disappear at any time. Some of these studies include further constraints on the evolution of the feature space. The core idea of these approaches is to establish relationships among features, enabling the learner to reconstruct missing features and leverage their previously learned weights. This reconstruction allows the learner to maintain predictive performance, even when some features are no longer observable. Unfortunately, all these methods prescribed linear classifiers and hence tend to yield inferior

performance when linearity does not hold.

To handle data streams with more complex patterns, more recent studies [He et al., 2021a, Liu et al., 2022b, Lian et al., 2022] have proposed methods that capture non-linear feature interactions within a low-dimensional latent space. While effective at modeling intricate relationships, these approaches inherently compromise the interpretability of the resulting models. This is because each latent variable represents a combination of multiple original features, making it difficult to trace the contributions of individual features. Consequently, these methods are unsuitable for sensitive domains where model interpretability is essential for trust and accountability.

4.4.2 Online Learning with Trees and Tree-Ensembles

Single-tree and tree-ensemble models strike a reasonable balance between interpretability and predictive performance. Single-tree models, such as ID.3 [Quinlan, 1986] and C4.5 [Quinlan, 1993], are highly interpretable as they provide clear, rule-based decision paths. However, they typically underperform compared to ensemble methods, such as Random Forests [Breiman, 2001] and XGBoost [Chen and Guestrin, 2016], which leverage the power of multiple trees to achieve higher accuracy. Despite their strengths, neither single tree models nor tree ensembles are inherently designed for online learning settings.

To address the challenges of streaming data, early research introduced the Hoeffding Tree [Domingos and Hulten, 2000]. Building on this foundation, ensemble methods such as Ensembles of Restricted Hoeffding Trees [Bifet et al., 2012], Adaptive Random Forests [Gomes et al., 2017], and Dynamic Streaming Random Forest [Abdulsalam et al., 2008] were developed to increase learning capacity.

However, these ensemble approaches come with significant drawbacks. Their interpretability diminishes as they rely on multiple deep trees developed in a bagging fashion, making it difficult to track the impact of individual features on predictions. Additionally, the Hoeffding Tree and its variants are memory-intensive, as they require storing observed data for each feature until an optimal split decision is reached. This makes them inapplicable for scenarios involving varying feature spaces, as none of these methods are equipped to deal with this dynamic.

Our approaches improve upon these online tree and tree ensemble models in the sense that they are interpretable at any time and manage memory overhead, as pruning is applied in both ORF^{3}V and DynFo, ensuring the model size remains contained.

4.5 Evaluation

In the following, we present the empirical evaluation of the DynFo and ORF^{3}V algorithms by illustrating their performance in three different scenarios. The first scenario is a reproduction of the benchmark as used in Beyazit et al. [2019]. Hereby, three cases are evaluated: a simulation of varying feature spaces, a simulation of monotonically increasing feature spaces, and an experiment on a real-world dataset of varying feature spaces. The second scenario is an evaluation based on the real-world dataset crowdsense, as presented in our work Schreckenberger et al. [2023]. Hereby, we evaluate the performance of OLVF [Beyazit et al., 2019], OVFM [He et al., 2021b], DynFo and ORF^{3}V. Finally, we also present an experiment on the interpretability of DynFo and ORF^{3}V based on a non-linear additive dataset from Ahmed M. Alaa [2019]. In the following, we provide a brief description of the rival models used in the evaluation, along with their corresponding parameters. With the best set of parameters, we repeat the experiments 20 times. The scores are then reported as the cumulative error rate of the 20 runs with the respective standard deviation.

Other Models We compare with the state-of-the-art online learners tailored for MIFS and VFS, as follows.

- OLSF [Zhang et al., 2016] was the first method for monotonically increasing feature spaces, using a linear classifier that initializes new feature coefficients through margin-based optimization.
- OLVF [Beyazit et al., 2019] extends the MIFS setting to VFS by introducing a feature-space classifier that leverages the co-occurrence patterns of unobserved and newly emerging features to enhance discriminative power.
- OVFM [He et al., 2021b] extends OLVF by leveraging feature correlations, allowing new features to be initialized with a best guess (similar

to OLSF) and enabling the reconstruction of missing features.

DynFo Parameters For the N values, i.e. the buffer size, we used for all datasets, except the *magic04* and the *a8a* dataset, a value of ten percent of the instances to avoid learning from the whole dataset towards the end of the simulated data stream, as some datasets are rather small. The N values for the *magic04* and the *a8a* dataset were both set to 500, which is less than 10% of the instances. The two bounding parameters theta were set to θ_1=0.05, and θ_2=0.75, while the rest of the parameters were determined from a random grid search.

ORF^{3}V Parameters The parameters of the ORF^{3}V are tuned by running each possible combination of the following parameters $N = \{1, 3, 5, 10\}$, $\alpha = \{0.01, 0.1, 0.3, 0.5, 0.9\}$, $rs = \{$"*oldest*", "*random*"$\}$. Furthermore, we fixed some parameters as their impact is rather small on these datasets: $\delta = 0.01$ and $\lambda = 1000$. The replacement interval was adjusted according to the dataset size, for the smaller datasets we used $r = 1$ and for the larger datasets $r = 50$ (more than 10,000 instances).

Rival models parameters For the OLVF algorithm, as well as the OLSF [Zhang et al., 2016] algorithm, we use the parameters as presented by Beyazit et al. [2019]. For OVFM [He et al., 2021b], we use the parameters as presented by them.

4.5.1 Benchmark

According to Beyazit et al. [2019], three benchmark cases are presented to evaluate their algorithm. The first case are experiments on varying feature spaces on nine UCI ML repository datasets[1], the second case are experiments on monotonically increasing feature spaces with the same datasets (cf. Section 2.4.1), and the third case is an evaluation on the real-world case of the imdb dataset [Maas et al., 2011]. The characteristics, in terms of features and instances for each dataset, are shown in Table 3.5. They are identical to the datasets used for the monotonically increasing feature spaces (Chapter 3.3.4) with the addition of the imdb dataset.

[1] https://archive.ics.uci.edu/ml/index.php

4.5.1.1 Experiment with Varying Feature Spaces

In the first benchmark case of Beyazit et al. [2019], varying feature spaces are simulated by randomly removing a portion of the features for each instance in the respective datasets. The removing ratios (Rem.) are set to 0.25, 0.50, and 0.75. The results for this benchmark case are reported in Table 4.3.

Overall, DynFo emerges as the most frequent winner, achieving the lowest cumulative error rate in 12 cases, followed closely by ORF^{3}V, which wins in 11 cases. OLVF, while competitive in specific scenarios, is the weakest overall, securing the lowest error rate only 6 times.

When analyzing performance by dataset, we see that DynFo dominates in ionosphere, magic04, wbc, and wdbc, whereas ORF^{3}V consistently outperforms the other methods in german, svmguide3, wpbc, and a8a. OLVF shows some strength in spambase, but it does not dominate in any of the other datasets. This suggests that DynFo and ORF^{3}V excel in varied datasets with complex patterns.

Examining performance trends across different removal ratios, DynFo performs best at Rem. $= 0.25$ and Rem. $= 0.5$, securing the lowest error rate in four and five datasets, respectively. However, at Rem. $= 0.75$, ORF^{3}V takes the lead, emerging as the best performer in five datasets, showing that it maintains greater stability as more data is removed. OLVF performs well only at lower Rem. values but quickly deteriorates as the removal ratio increases.

Examining consistency across removal ratios within the same dataset, ORF^{3}V is the most stable, consistently maintaining the best performance across increasing removal ratios. DynFo, while strong, sometimes sees performance degradation at Rem. $= 0.75$, making it slightly less consistent than ORF^{3}V. On the other hand, OLVF struggles with consistency in its performance, particularly in terms of volatility.

4.5.1.2 Experiment with Monotonically Increasing Feature Spaces

As a second evaluation case, the benchmark also encompasses experiments on monotonically increasing feature spaces. Hereby, we not only benchmark the ORF^{3}V against OLVF, but also against OLSF, an algorithm specifically for monotonically increasing feature spaces. As in the previous case, we follow the evaluation setup provided by Beyazit et al. [2019]. In this

Dataset	Rem.	OLVF	DynFo	ORF^{3}V
german	0.25	.333 ± .009	**.302 ± .004**	.304 ± .002
	0.5	350.9 ± .008	.307 ± .007	**.305 ± .003**
	0.75	.365 ± .004	.325 ± .017	**.309 ± .002**
ionosphere	0.25	.220 ± .020	**.191 ± .015**	.226 ± .001
	0.5	.226 ± .021	**.216 ± .016**	.246 ± .001
	0.75	**.227 ± .018**	.293 ± .021	.287 ± .028
spambase	0.25	**.143 ± .003**	.192 ± .013	.265 ± .004
	0.5	**.188 ± .004**	.204 ± .008	.264 ± .003
	0.75	.299 ± .005	**.244 ± .007**	.277 ± .005
magic04	0.25	.323 ± .003	**.275 ± .005**	.280 ± .003
	0.5	.356 ± .001	**.278 ± .004**	.289 ± .003
	0.75	.375 ± .000	**.298 ± .003**	.341 ± .003
svmguide3	0.25	.280 ± .009	.232 ± .007	**.227 ± .004**
	0.5	.298 ± .009	.239 ± .009	**.234 ± .006**
	0.75	.301 ± .009	.261 ± .018	**.244 ± .004**
wbc	0.25	**.036 ± .002**	.060 ± .008	.093 ± .008
	0.5	.087 ± .008	**.077 ± .009**	.096 ± .005
	0.75	.176 ± .005	**.133 ± .013**	.175 ± .012
wpbc	0.25	.447 ± .029	.278 ± .021	**.265 ± .005**
	0.5	.456 ± .026	.306, ±.034	**.268 ± .008**
	0.75	.544 ± .039	.360 ± .036	**.269 ± .006**
wdbc	0.25	**.071 ± .006**	.089 ± .008	.118 ± .007
	0.5	.097 ± .009	**.097 ± .009**	.122 ± .011
	0.75	.357 ± .013	**.137 ± .012**	.148 ± .015
a8a	0.25	.276 ± .001	.241 ± .000	**.233 ± .001**
	0.5	.294 ± .002	.240 ± .001	**.236 ± .001**
	0.75	.382 ± .002	**.237 ± .002**	.239 ± .001
imdb	-	.392 ± .003	.341 ± .023	**.286 ± .003**

Table 4.3: CER made by OLVF, DynFo, and ORF^{3}V on simulated data streams with varying feature spaces with removing ratios (Rem.) of 0.25, 0.5, and 0.75.

Dataset	OLSF	OLVF	DynFo	ORF^{3}V
german	.369 ± .010	.356 ± .009	.303 ± .012	**.303 ± .006**
ionosphere	.243 ± .017	**.166 ± .010**	.201 ± .015	.230 ± .013
spambase	**.212 ± .010**	.253 ± .003	.235 ± .010	.269 ± .010
magic04	.321 ± .003	.337 ± .004	**.276 ± .004**	.283 ± .007
svmguide3	.308 ± .021	.355 ± .026	.246 ± .018	**.237 ± .005**
wbc	.054 ± .013	**.045 ± .003**	.071 ± .009	.110 ± .011
wpbc	.428 ± .027	.445 ± .056	.270 ± .028	**.267 ± .005**
wdbc	.221 ± .018	.131 ± .018	**.123 ± .009**	.123 ± .016
a8a	.318 ± .004	.354 ± .004	.241 ± .000	**.231 ± .003**

Table 4.4: CER made by OLSF, OLVF, DynFo, and ORF^{3}V on simulated monotonically increasing feature spaces.

setup, monotonically increasing feature spaces are simulated by splitting the dataset into ten chunks, where for each chunk, the feature space is increased by ten percent (cf. Section 2.4.1). The results are presented in Table 4.4

Overall, the best-performing method is ORF^{3}V, which achieves the lowest error rates in four datasets. DynFo achieves the best performance in german, magic04, wdbc, while ORF^{3}V closely matches DynFo in german, magic04, and wdbc. OLVF performs well in ionosphere and wbc, while OLSF is the weakest in most cases but wins in spambase.

Overall, our methods not only match but often surpass the performance of OLSF and OLVF across a majority of the datasets. This stability and consistency highlight the adaptability and effectiveness of DynFo and ORF^{3}V in handling monotonically increasing feature spaces.

4.5.1.3 Experiment with Real-World Varying Feature Spaces

As their real-world evaluation task, they included the imdb movie reviews dataset Maas et al. [2011], where the goal is to perform a sentiment analysis. The reviews are represented as bag-of-words, leading to every instance of the dataset having its own representation. We feed the dataset row by row to the two learning algorithms, which simulates online learning from data streams. Due to each instance having its own feature set, we have data streams with varying feature spaces. Furthermore, the problem can also be

considered a monotonically increasing feature space, where we assume that non-existing features are missing; that is why the OLSF algorithm is also applied. The size of the feature space per instance is roughly 100. For the best performing parameter combination, we performed 10 runs, shuffling the instances within each run.
The results are shown in the bottom line of Table 4.3. Hereby, we can see that ORF^{3}V outperforms OLVF and DynFo. The best settings for the ORF^{3}V algorithm were $N = 1, \alpha = 0.1, rs = oldest$. The parameter settings are consistent with the results of the first evaluation case. For larger datasets, it seems to be beneficial to use a smaller α and, therefore, a less aggressive update strategy for the weights in the ensemble. For DynFo we fixed $N = 1000$, the rest of the parameters as determined are: $\alpha = 0.5$, $\beta = 0.3$, $\delta = 0.01$, $\epsilon = 0.001$, $\gamma = 0.7$, $\theta_1 = 0.05$, $\theta_2 = 0.6$ and $M = 1000$. An interesting observation that can be made is that DynFo is using more aggressive weight updates with a higher α compared to ORF^{3}V on this dataset.

4.5.2 Experiments on the crowdsense dataset

The results for the crowdsense dataset, as introduced in Section 4.1, are shown in Table 4.5. They reveal clear trends across different experimental settings, including binary classification with original data order (BC/O), binary classification with shuffled data order (BC/S), and multiclass classification with both original (MC/O) and shuffled (MC/S) data orders. We shuffle the data order to generate additional evaluation cases while prioritizing the results from the original data order since they accurately reflect the actual dynamics of the feature space.

In binary classification (BC/O and BC/S), ORF^{3}V outperforms all other methods in eleven out of 16 cases. This is particularly evident in $case_1$ (0.022), $case_2$ (0.022), $case_3$ (0.018), and $case_8$ (0.019), where it significantly outperforms competitors with original data order. DynFo shows competitive performance in BC settings and wins in some of the shuffled binary classification cases. OVFM performs moderately well, frequently achieving lower error rates than OLVF but still failing to match ORF^{3}V's and DynFo's performance. Meanwhile, OLVF struggles significantly by consistently producing the highest error rates, making it the least reliable method.

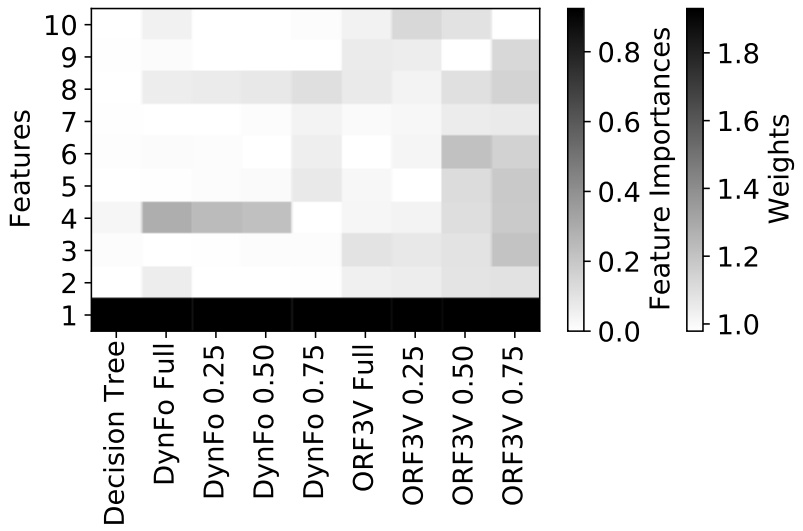


Figure 4.3: Interpretability experiment for DynFo and ORF^{3}V indicating feature importance.

For multiclass classification (MC/O and MC/S), the results show a different trend. ORF^{3}V remains dominant in MC/O, achieving the lowest error rates in $case_1$ (0.070), $case_2$ (0.096), $case_4$ (0.085), $case_6$ (0.093), and $case_8$ (0.056). However, DynFo performs poorly in multiclass classification, often exhibiting significantly high error rates, such as 0.915 in $case_8$ and 0.819 in $case_4$, making it unsuitable for MC tasks. OVFM, on the other hand, shows its strength in MC/S, where it frequently outperforms ORF^{3}V, particularly in $case_2$ (0.111), $case_4$ (0.127), $case_6$ (0.220), and $case_8$ (0.287). This suggests that OVFM is better adapted to shuffled multiclass settings, while ORF^{3}V remains superior in multiclass classification with original data order.

In conclusion, ORF^{3}V is the best overall method for binary classification and multiclass classification. DynFo is a reliable choice for binary classification but struggles in MC settings, where OVFM proves to be an alternative. OLVF generally underperforms across all settings.

4.5.3 Interpretability experiment

To assess the interpretability of the proposed approach, we generated a nonlinear additive dataset Ahmed M. Alaa [2019]. With this data, we learned an interpretable model (a decision tree) and observed the feature importance,

crowdsense	Setting	OLVF	OVFM	DynFo	ORF^{3}V
$case_1$	BC/O	.318	.252	.044	**.022**
	BC/S	.271 ± .016	.309 ± .011	**.041 ± .006**	.083 ± .001
	MC/O	-	.261	.813	**.070**
	MC/S	-	.315 ± .008	.331 ± .045	**.302 ± .007**
$case_2$	BC/O	.318	.191	.053	**.022**
	BC/S	.523 ± .017	.106 ± .001	**.043 ± .007**	.084 ± .002
	MC/O	-	.212	.737	**.096**
	MC/S	-	**.111 ± .001**	.350 ± .029	.221 ± .008
$case_3$	BC/O	.318	.100	.037	**.018**
	BC/S	.522 ± .015	.159 ± .004	**.039 ± .005**	.083 ± .001
	MC/O	-	-	-	-
	MC/S	-	-	-	-
$case_4$	BC/O	.318	.098	.035	**.022**
	BC/S	.522 ± .015	.101 ± .001	**.039 ± .008**	.083 ± .001
	MC/O	-	.109	.819	**.085**
	MC/S	-	**.127 ± .003**	.458 ± .211	.227 ± .004
$case_5$	BC/O	.272	.193	.144	**.039**
	BC/S	.389 ± .022	.214 ± .007	.119 ± .012	**.117 ± .001**
	MC/O	-	-	-	-
	MC/S	-	-	-	-
$case_6$	BC/O	.202	.165	.210	**.061**
	BC/S	.406 ± .011	.299 ± .004	.190 ± .028	**.166 ± .007**
	MC/O	-	.180	.624	**.093**
	MC/S	-	**.220 ± .009**	.407 ± .025	.283 ± .004
$case_7$	BC/O	.202	.211	.190	**.058**
	BC/S	.379 ± .16	.138 ± .014	.233 ± .052	**.136 ± .010**
	MC/O	-	**.065**	.510	.215
	MC/S	-	.202 ± .011	.337 ± .062	**.182 ± .011**
$case_8$	BC/O	.318	.298	.034	**.019**
	BC/S	.522 ± .015	.264 ± .005	**.039 ± .004**	.084 ± .001
	MC/O	-	.215	.915	**.056**
	MC/S	-	**.287 ± .010**	.896 ± .018	.315 ± .006

Table 4.5: CER made by OLVF, OVFM, DynFo, and ORF^{3}V on the crowd sense dataset.

which we compared to our approaches DynFo and ORF^{3}V. Both methods are learned on the complete data (no varying feature space) and simulated VFS data with removal ratios of 0.25, 0.5, and 0.75, respectively, as shown in Figure 4.3.

In ORF^{3}V, weights close to 1 are to be interpreted as having a low importance. It can be observed that the weights of the models behave analogously to the feature importance of the decision tree for both the complete data scenario and the varying feature space scenarios. For DynFo the feature importance can be calculated by the normalized amount a feature was chosen for a split in a decision stump, scaled by the weight of the decision stump in the ensemble. We can then infer the feature importances of DynFo inherently. The figure shows that DynFo rightfully indicates the first feature of the dataset as the most important one. Hence, we can conclude that both approaches are capable of depicting feature importances and are, therefore, interpretable.

4.6 Discussion

In the following, we discuss the proposed approaches ORF^{3}V and DynFo with respect to the six requirements presented in Section 1.2.

4.6.1 Requirement 1: Online Processing

Online processing dictates that each incoming instance is to be handled immediately upon arrival, without requiring multiple passes. In ORF^{3}V, feature statistics are aggregated efficiently so that each example is processed exactly once. In contrast, DynFo employs a sliding window mechanism. While this allows for the incorporation of new examples to update decision stumps, it may result in some instances being revisited during the formation of the ensemble. Thus, ORF^{3}V more strictly adheres to the inspect-once policy, whereas DynFo 's design, although still online in nature, circumvents the requirement slightly by buffering data in the sliding window. Depending on whether the environments allow for relaxation of this requirement, it may therefore be infeasible to use DynFo.

4.6.2 Requirement 2: Finite Memory

Since data streams can be massive, minimizing memory usage is crucial. ORF^{3}V leverages a compact data structure by initializing a new t-digest for each emerging feature, ensuring that memory consumption increases linearly as features are added. This controlled expansion allows ORF^{3}V to operate efficiently even under strict memory constraints, such as in edge computing environments. Furthermore, ORF^{3}V employs a mechanism to prune vanished features based on the Hoeffding bound. On the other hand, DynFo relies on a sliding window, which means that the buffer size must be carefully managed. The window's capacity directly affects memory usage, and a larger window, which is necessary for more reliable statistics, may risk higher memory consumption. The model size is managed dynamically, allowing for decision stumps to be initialized or pruned. However, we argue that for memory-constrained applications, ORF^{3}V has an advantage.

4.6.3 Requirement 3: Finite Processing Time

Both methods are designed with linear scaling in mind. ORF^{3}V aggregates feature statistics in a single online pass, ensuring that processing time per instance is minimal. Its use of t-digest supports rapid updates without compromising runtime. DynFo also scales linearly with the feature space and the number of parameters. However, the additional overhead incurred by the sliding window buffer may introduce slight delays in processing, particularly when instances are repeatedly examined for decision stump generation. Empirical runtime tests are suggested to quantify this difference, though both approaches maintain overall scalability.

4.6.4 Requirement 4: Feature Space Adaptations

Varying feature spaces, where features appear and vanish, are handled differently by the two models. ORF^{3}V employs the Hoeffding bound to detect and phase out vanished features. Once a feature is deemed obsolete, its associated feature forest is dropped immediately. In contrast, DynFo does not explicitly remove decision stumps when features vanish. Instead, it gradually phases out such features as they leave the sliding window and ignores decision stumps based on outdated features during prediction. For new fea-

tures, DynFo incorporates new instances into the window, while ORF^{3}V initializes a new t-digest to integrate the feature into the model quickly. Both models, therefore, meet the requirement for feature space adaptation but via different mechanisms.

4.6.5 Requirement 5: Anytime Predictions

Anytime predictions require that the model can generate an output at any moment, even if only a subset of features is present. ORF^{3}V 's true online aggregation facilitates rapid updates to the model as new data arrive. DynFo, although capable of dynamic updates, may exhibit minor delays due to its sliding window and periodic processing. Nevertheless, both approaches update their ensembles incrementally to avoid full retraining whenever an observation arrives, thereby supporting real-time predictions.

4.6.6 Requirement 6: Interpretability

As for monotonically increasing feature spaces, interpretability is also critical in varying feature spaces for building user trust and ensuring compliance with regulatory standards [Molnar, 2020]. Both ORF^{3}V and DynFo provide mechanisms for extracting feature importance. ORF^{3}V 's use of weights assigned to each feature forest yields transparent insights into which features drive predictions. DynFo, while also offering interpretability through its decision stumps, may be slightly less clear, as we may have multiple decision stumps with respective weights for a given feature. Despite this, both approaches strike a balance between predictive performance and transparency. This makes them suitable for being employed in critical yet highly dynamic environments.

Part III

Outlook

Chapter 5

Open Challenges and Future Directions

In this chapter, we explore five areas that indicate open challenges and future directions for learning in open feature spaces, as presented in our survey ([He et al., 2023]). We begin by examining issues related to label imbalance, scarcity, and noise—factors that compromise the quality of data and the accuracy of predictions. We then address the problem of concept drift, where shifting data distributions necessitate continuous model adaptation. The chapter also explores the open-world crisis, highlighting the challenges of managing unknown classes as new data emerges. Finally, we address concerns regarding security, privacy, and algorithmic fairness, which are crucial for protecting sensitive information and ensuring unbiased decisions.

5.1 Robustness in the Face of Label Imbalance, Scarcity, and Noise

Learning from open feature spaces is based mainly on the assumption that we have perfectly labeled data, but in the real world, we often encounter more complex scenarios. Key challenges include label imbalance, scarcity, or noise [He et al., 2019a]. In many practical applications, the overwhelming majority of data belongs to a single class, while minority classes remain underrepresented. This imbalance leads to models that are overly confident

in predicting the majority class, with traditional metrics like accuracy failing to capture the nuances of the minority data. Moreover, the labeling process can be both expensive and labor-intensive. Relying on human annotators not only drives up costs but also introduces errors. Human labelers are inherently fallible, which can result in noisy labels that further skew model performance [Snow et al., 2008].

The difficulty increases when the feature space itself is changing. With unlabeled data often arriving in streams, there is little opportunity for supervised correction. Robust learning, therefore, must focus on extracting meaningful relationships from sparse, imbalanced, and sometimes erroneous data. Instead of relying on models that require complete and accurate labels, innovative approaches are needed. Techniques such as semi-supervised or active learning could be combined with noise-tolerant methods to build decision boundaries that remain resilient even when the majority of inputs are either unlabeled or imprecisely annotated. Additionally, extending these methods to regression tasks poses further challenges, requiring the learning algorithm to handle a continuous prediction target. Addressing these issues is critical for advancing this field of research to real-world applications where data imperfection is the norm.

5.2 Adapting to Concept Drift in Open Feature Spaces

Concept drift refers to the phenomenon where the statistical properties of data evolve over time, even though the set of features remains unchanged. In many applications, the conditional distributions shift gradually or abruptly due to changes in user behavior, external environmental factors, or technological advancements. This drift can render established decision boundaries obsolete, resulting in a degradation of model performance [Lu et al., 2018]. Approaches for learning from open feature spaces generally assume a non-shifting environment. While some methods periodically refresh their parameters, they are not explicitly designed to cope with the complexities introduced by drift.

The challenge increases with the reliance on parametric estimators that assume a fixed feature set. When new features emerge or the relationships

among existing features evolve, these estimators, such as Bayesian density estimators, may fail and provide incorrect probability estimates. Addressing concept drift in open feature space learning, therefore, requires adaptive mechanisms capable of dynamically recalibrating the model. This may involve employing non-parametric drift detection methods, adaptive learning rates, or continuous re-training schedules [Webb et al., 2017]. A robust solution must balance quick adaptations with the stability of predictions to ensure that the model retains good quality despite the inherent uncertainty in evolving data streams.

5.3 Open World Crisis

The open-world crisis in online learning emerges when models must simultaneously contend with both known and unknown classes, as well as an evolving feature space. Traditional systems rely on static test sets and fixed output spaces, but in dynamic environments, new classes can appear unexpectedly during the learning process. Open-world learning (OWL), as coined by Jafarzadeh et al. [2020], extends the paradigm of learning from open feature spaces by addressing not only the variability in input features but also the potential expansion of the output label space. This dual challenge requires systems to robustly manage imbalanced and noisy data while also detecting and accommodating the emergence of new classes [Zhu et al., 2024].

A central difficulty in open-world learning is delineating the boundary between known and novel classes. Many current approaches attempt to gauge the volume of the region spanned by each known class so that any sample falling outside can be flagged as belonging to a new category [Zhu et al., 2024]. However, these methods are inherently tied to the assumption of a fixed feature space, and once that space begins to evolve, the notion of a static volume becomes untenable. Moreover, misclassifying a novel instance as a mere outlier can have severe consequences in critical applications. To overcome these challenges, advanced outlier detection techniques and adaptive boundary definitions are required [Zhu et al., 2024]. Models should be capable of incorporating new class information incrementally without requiring complete retraining, ensuring a seamless transition as the environment changes. Ultimately, integrating learning from open feature

spaces with OWL could provide a highly flexible system that can manage the uncertainty of both feature space variability and output.

5.4 Security & Privacy

The rise of online learning in data-intensive applications brings significant security and privacy challenges. Conventional open feature space learning methods centralize data from different sources, e.g., from sensors, for processing on a single server. This inherently creates the risk of sensitive data leaks, as data ownership is transferred to a single responsible instance [Mothukuri et al., 2021]. In sectors such as healthcare, the exposure of sensitive data can lead to catastrophic breaches. Distributed methods dealing with open feature spaces, inspired by federated learning approaches, could offer a promising alternative by partitioning the data processing task among multiple parties, allowing each party to retain its own data. This decentralization mitigates the risk associated with a single point of failure while providing a more resilient framework [Mothukuri et al., 2021].

Despite its advantages, federated open feature space learning introduces new complexities, particularly when the feature sets across different parties are not aligned. In scenarios similar to vertical federated learning [Liu et al., 2022a], distinct data sources might capture various aspects of the same phenomenon, hence complicating the integration process. Moreover, maintaining data privacy during interparty communications is essential. Security measures should be implemented to prevent any possibility of reconstructing original data from shared updates. Even when the protocols are robust, the inherent risk of data leakage through indirect inference remains a challenge Liu et al. [2022a]. Thus, designing secure and privacy-preserving distributed open feature space learning methods requires additional thought and attention in future research. While the current application areas used to benchmark the methods are less critical, this will certainly gain importance when this learning paradigm finds application in sensitive areas.

5.5 Algorithmic Fairness and Mitigating Bias

When online learning systems are deployed in increasingly critical contexts, ensuring algorithmic fairness becomes essential. Bias can inadvertently seep

into models through correlations between target labels and sensitive features such as gender, ethnicity, or socioeconomic status [Barocas et al., 2023]. Another risk can originate from missing features. In applications such as crowd sensing, the lack of access to modern technology (e.g., smartphones or sensors) may serve as an indirect marker of socioeconomic disadvantage, resulting in skewed predictions that perpetuate systemic inequalities [Hino et al., 2018].

In open feature spaces, fairness can be negatively impacted, as even minor discrepancies in feature representation can compound over time as the model continuously updates its parameters [Barocas et al., 2023]. Static fairness metrics, designed for fixed datasets, may fail to capture the dynamic nature of bias in online learning environments. Therefore, it would be beneficial to develop adaptive fairness evaluation frameworks that assess both immediate and long-term impacts on disadvantaged groups. Measures should address both the presence and especially the absence of key features, ensuring that superficial correlations or missing data do not lead to unfair outcomes. Robust, fair, and transparent models are crucial for mitigating bias and promoting equitable outcomes in a rapidly evolving and transparent data landscape.

Chapter 6

Conclusion

In conclusion, this thesis addresses the challenge of online learning in open feature spaces by establishing a robust framework that requires stringent real-time processing requirements, finite memory constraints, interpretability, and anytime prediction. Initially, the work outlines the core motivation, problem statement, and a detailed set of requirements that pave the way for the subsequent contributions in Chapter 1. Subsequently, the foundations are introduced in Chapter 2. The novel methods targeted at solving the outlined issues are introduced in Chapter 3 and Chapter 4, which target monotonically increasing feature spaces and varying feature spaces, respectively. Finally, the outlook in Chapter 5 identifies several open challenges and future directions. Therefore, the work advances the technical state of online learning in open feature spaces and presents a path for future research in this field.

In the following, we summarize the contributions of this thesis and provide some final thoughts regarding the open challenges and future directions.

6.1 Summary of Contributions

The thesis makes a significant contribution by addressing the challenging problem of online learning in environments where feature spaces are not static but continuously evolving. The research is structured around two main contributions that together enhance the applicability and efficiency of decision tree-based models in open feature space settings.

The first significant contribution is the development of the Dynamic Fast Decision Tree. Decision tree methods for learning from data streams assume a fixed set of features, which limits their effectiveness in real-world scenarios characterized by the constant addition of new features. To overcome this limitation, the Dynamic Fast Decision Tree incorporates novel techniques for managing a monotonically increasing feature space. This is achieved by dynamically restructuring the tree and implementing a pruning strategy to ensure that the model remains efficient despite continuous growth in feature dimensions. We demonstrate that the approach meets essential requirements, such as online processing, finite memory usage, and bounded processing time, while also providing interpretable, anytime predictions.

The second contribution extends these ideas to more complex scenarios involving varying feature spaces, where the feature space not only grows but may also potentially shrink. We address this by introducing two ensemble-based approaches: the Dynamic Forest and the ORF^{3}V algorithm. The Dynamic Forest employs an ensemble of decision stumps. The ORF^{3}V algorithm, conversely, offers an alternative ensemble strategy in which feature forests are formed around a single feature's data stream. Both approaches have been evaluated through extensive experiments, with results indicating that they are competitive while maintaining interpretability in the complex field of varying feature spaces.

In summary, by advancing both single-tree and ensemble-based methodologies, this thesis contributes methods for online learning in open feature spaces. These contributions bridge significant gaps in online machine learning and motivate future research.

6.2 Final Thoughts

We want to dedicate our final thoughts to reflecting on the open challenges that remain. It is clear that, while significant progress has been made in this thesis and other research in this field, the landscape of online learning in open feature spaces still presents many opportunities for further exploration and innovation.

One of the primary challenges is achieving robust performance in the face of label adversity. As data streams grow, ensuring that models maintain high accuracy without being overly sensitive to imperfect data is a concern that

future work should address. The notion of an "open world crisis" connects to this by emphasizing the unpredictable nature of data stream learning. Generally, it is advisable that systems, which are supposed to operate in this area, should be capable of incorporating the unknown unknowns.

The challenges of handling concept drift are already established in the general field of online learning. However, it has not yet received attention in open feature space learning research. This calls for the further development of methods that can transition between different states of feature spaces and different data distributions within the features to ensure reliability in ever-changing environments.

Security and privacy are becoming increasingly critical as more sensitive data is processed in real-time. Future research directed towards this topic could integrate security measures and privacy-preserving techniques. Inspirations from adjacent fields provide a good starting point to propel this stream of research in open feature space learning. Alongside these technical challenges, the ethical dimensions of algorithmic fairness should not be overlooked. As society becomes increasingly dependent on machine learning, considering the resulting models are becoming more and more interwoven into decision-making processes, it is essential to develop methods that mitigate bias and promote fairness across all demographics.

The future directions highlighted by this work can pave the way for enabling online learning to be not only adaptive and efficient but preferably also secure, fair, and resilient. Addressing these challenges will be key to advancing the theory and practice of open feature space machine learning.

References

Hanady Abdulsalam, David B Skillicorn, and Patrick Martin. Classifying evolving data streams using dynamic streaming random forests. In *International Conference on Database and Expert Systems Applications*, pages 643–651. Springer, 2008. 101

Charu C Aggarwal. *Data streams: models and algorithms*, volume 31. Springer, 2007. 4

Mihaela van der Schaar Ahmed M. Alaa. Demystifying black-box models with symbolic metamodels. In *Neural Information Processing Systems*, 2019. 102, 108

Jeevithan Alagurajah, Xu Yuan, and Xindong Wu. Scale invariant learning from trapezoidal data streams. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 505–508, Brno Czech Republic, March 2020. ACM. ISBN 978-1-4503-6866-7. doi: 10.1145/3341105.3375775. URL `https://dl.acm.org/doi/10.1145/3341105.3375775`. 62

Amirah Alharthi, Charles C Taylor, and Jochen Voss. Forests of stumps. *Archives of Data Science, Series A*, 5(1):30, 2018. 37

Jean Paul Barddal and Fabrício Enembreck. Learning regularized hoeffding trees from data streams. *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019. URL `https://api.semanticscholar.org/CorpusID:142503707`. 63

Solon Barocas, Moritz Hardt, and Arvind Narayanan. *Fairness and machine learning: Limitations and opportunities*. MIT press, 2023. 119

Ege Beyazit, Matin Hosseini, Anthony Maida, and Xindong Wu. Learning Simplified Decision Boundaries from Trapezoidal Data Streams. In Věra Kůrková, Yannis Manolopoulos, Barbara Hammer, Lazaros Iliadis, and Ilias Maglogiannis, editors, *Artificial Neural Networks and Machine Learning – ICANN 2018*, volume 11139, pages 508–517. Springer International Publishing, Cham, 2018. ISBN 978-3-030-01417-9 978-3-030-01418-6. doi: 10.1007/978-3-030-01418-6_50. Series Title: Lecture Notes in Computer Science. 62, 63

Ege Beyazit, Jeevithan Alagurajah, and Xindong Wu. Online Learning from Data Streams with Varying Feature Spaces. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:3232–3239, July 2019. ISSN 2374-3468, 2159-5399. doi: 10.1609/aaai.v33i01.33013232. URL `https://aaai.org/ojs/index.php/AAAI/article/view/4192`. 5, 17, 42, 45, 64, 69, 73, 100, 102, 103, 104

Albert Bifet, Eibe Frank, Geoff Holmes, and Bernhard Pfahringer. Ensembles of restricted hoeffding trees. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 3(2):1–20, 2012. 101

Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006. 13, 15

Max Bramer. Avoiding overfitting of decision trees. *Principles of data mining*, pages 119–134, 2007. 25

Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001. 33, 34, 38, 101

Leo Breiman. *Classification and regression trees*. Routledge, 2017. 23, 32

Guillem Camprodon, Óscar González, Víctor Barberán, Máximo Pérez, Viktor Smári, Miguel Ángel de Heras, and Alejandro Bizzotto. Smart citizen kit and station: An open environmental monitoring system for citizen participation and scientific experimentation. *HardwareX*, 6:e00070, 2019. 74

Nicolo Cesa-Bianchi and Gábor Lugosi. *Prediction, learning, and games*. Cambridge university press, 2006. 17, 39, 44, 45

Sanjay Chakraborty and Lopamudra Dey. Applications of multi-objective, multi-label, and multi-class classifications. In *Multi-objective, Multi-class and Multi-label Data Classification with Class Imbalance: Theory and Practices*, pages 135–164. Springer, 2024. 14

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016. 101

Peter Clark and Robin Boswell. Rule induction with cn2: Some recent improvements. In *Machine Learning—EWSL-91: European Working Session on Learning Porto, Portugal, March 6–8, 1991 Proceedings 5*, pages 151–163. Springer, 1991. 37, 87, 98

Adele Cutler, D Richard Cutler, and John R Stevens. Random forests. *Ensemble machine learning: Methods and applications*, pages 157–175, 2012. 36

Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012. 15

Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80, 2000. 16, 28, 30, 31, 63, 101

Ted Dunning. The t-digest: Efficient estimates of distributions. *Software Impacts*, 7:100049, 2021. 91

Ted Dunning and Otmar Ertl. Computing extremely accurate quantiles using t-digests. *arXiv preprint arXiv:1902.04023*, 2019. 91

Floriana Esposito, Donato Malerba, and Giovanni Semeraro. A further comparison of pruning methods in decision-tree induction. In *Pre-proceedings of the Fifth International Workshop on Artificial Intelligence and Statistics*, pages 211–218. PMLR, 1995. 25

Thomas Bartz-Beielstein Eva Bartz. *Online Machine Learning*. Springer Nature Singapore, 2024. 16

Maddalena Favaretto, Eva De Clercq, Christophe Olivier Schneble, and Bernice Simone Elger. What is your definition of big data? researchers' understanding of the phenomenon of the decade. *PloS one*, 15(2):e0228987, 2020. 16

Yoav Freund and Robert E Schapire. Large margin classification using the perceptron algorithm. *Machine learning*, 37(3):277–296, 1999. 100

Michael Friedewald and Oliver Raabe. Ubiquitous computing: An overview of technology impacts. *Telematics and Informatics*, 28(2):55–65, 2011. 3

João Gama and Mohamed Medhat Gaber. *Learning from data streams: processing techniques in sensor networks*. Springer, 2007. 4

Joao Gama, Ricardo Rocha, and Pedro Medas. Accurate decision trees for mining high-speed data streams. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 523–528, 2003. 31

Joao Gama, Raquel Sebastiao, and Pedro Pereira Rodrigues. On evaluating stream learning algorithms. *Machine learning*, 90:317–346, 2013. 38, 39

Heitor M Gomes, Albert Bifet, Jesse Read, Jean Paul Barddal, Fabrício Enembreck, Bernhard Pfharinger, Geoff Holmes, and Talel Abdessalem. Adaptive random forests for evolving data stream classification. *Machine Learning*, 106(9):1469–1495, 2017. 101

Joao Bartolo Gomes, Mohamed Medhat Gaber, Pedro AC Sousa, and Ernestina Menasalvas. Mining recurring concepts in a dynamic feature space. *IEEE Trans. on Neural Networks and Learning Systems*, 25(1): 95–110, 2013. 100

Maissae Haddouchi and Abdelaziz Berrado. A survey and taxonomy of methods interpreting random forest models, 2024. URL `https://arxiv.org/abs/2407.12759`. 38

Thomas Hale, Noam Angrist, Rafael Goldszmidt, Beatriz Kira, Anna Petherick, Toby Phillips, Samuel Webster, Emily Cameron-Blake, Laura Hallas, Saptarshi Majumdar, et al. A global panel database of pandemic policies (oxford covid-19 government response tracker). *Nature Human Behaviour*, 5(4):529–538, 2021. 75

Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 3rd edition, 2011. ISBN 9780123814791. 20, 21, 23, 25, 26, 27, 31

Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009. 21, 33, 34, 36, 38

Elad Hazan. Introduction to online convex optimization, 2023. URL `https://arxiv.org/abs/1909.05207`. 17

Wenwu He, Fumin Zou, and Quan Liang. Online learning with sparse labels. *Concurrency and Computation: Practice and Experience*, 31, 2019a. URL `https://api.semanticscholar.org/CorpusID:65585717`. 115

Yi He, Baijun Wu, Di Wu, Ege Beyazit, Sheng Chen, and Xindong Wu. Online learning from capricious data streams: a generative approach. In *IJCAI*, pages 2491–2497, 2019b. 100

Yi He, Jiaxian Dong, Bo-Jian Hou, Yu Wang, and Fei Wang. Online learning in variable feature spaces with mixed data. In *ICDM*, pages 181–190. IEEE, 2021a. 5, 101

Yi He, Xu Yuan, Sheng Chen, and Xindong Wu. Online learning in variable feature spaces under incomplete supervision. In *AAAI*, volume 35, pages 4106–4114, 2021b. 42, 45, 100, 102, 103

Yi He, Christian Schreckenberger, Heiner Stuckenschmidt, and Xindong Wu. Towards utilitarian online learning-a review of online algorithms in open feature space. In *Proc. of International Joint Conferences on Artificial Intelligence Organization*, 2023. 18, 19, 115

Miyuki Hino, Elinor Benami, and Nina Brooks. Machine learning for environmental monitoring. *Nature Sustainability*, 1(10):583–588, 2018. 119

Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, 20(8): 832–844, 1998. 33

Wassily Hoeffding. Probability inequalities for sums of bounded random variables. In *The Collected Works of Wassily Hoeffding*, pages 409–426. Springer, 1994. 28, 98

Steven C.H. Hoi, Doyen Sahoo, Jing Lu, and Peilin Zhao. Online learning: A comprehensive survey. *Neurocomputing*, 459:249–289, 2021a. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2021.04.112. URL `https://www.sciencedirect.com/science/article/pii/S0925231221006706`. 44

Steven CH Hoi, Doyen Sahoo, Jing Lu, and Peilin Zhao. Online learning: A comprehensive survey. *Neurocomputing*, 459:249–289, 2021b. 3

Bo-Jian Hou, Lijun Zhang, and Zhi-Hua Zhou. Learning with feature evolvable streams. In *NeurIPS*, pages 1417–1427, 2017. 5, 18, 100

Bo-Jian Hou, Yu-Hu Yan, Peng Zhao, and Zhi-Hua Zhou. Storage fit learning with feature evolvable streams. In *AAAI*, 2021a. 100

Bo-Jian Hou, Lijun Zhang, and Zhi-Hua Zhou. Prediction with unpredictable feature evolution. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–10, 2021b. 100

Chenping Hou and Zhi-Hua Zhou. One-pass learning with incremental and decremental features. *IEEE transactions on pattern analysis and machine intelligence*, 40(11):2776–2792, 2017. 100

Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 97–106, 2001. 31

Wayne Iba and Pat Langley. Induction of one-level decision trees. In *Machine Learning Proceedings 1992*, pages 233–240. Elsevier, 1992. 37, 76, 79

Mohsen Jafarzadeh, Akshay Raj Dhamija, Steve Cruz, Chunchun Li, Touqeer Ahmad, and Terrance E Boult. Open-world learning without labels. *arXiv preprint arXiv:2011.12906*, 2020. 117

Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, and Jonathan Taylor. Statistical learning. In *An introduction to statistical learning: With applications in Python*, pages 15–67. Springer, 2023. 3, 14, 21

Been Kim. An introduction on interpretable machine learning. *International Journal of Innovative Technology and Exploring Engineering*, 2020. URL `https://api.semanticscholar.org/CorpusID:243165405`. 5

Richard Brendon Kirkby. *Improving hoeffding trees*. PhD thesis, The University of Waikato, 2007. xiii, 5, 6, 8

Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. Estimating mutual information. *Physical review E*, 69(6), 2004. 20

Vrushali Y Kulkarni and Pradeep K Sinha. Pruning of random forest classifiers: A survey and future directions. In *2012 International Conference on Data Science & Engineering (ICDSE)*, pages 64–68. IEEE, 2012. 36

Daniel Leite, Pyramo Costa, and Fernando Gomide. Evolving granular neural networks from fuzzy data streams. *Neural Networks*, 38:1–16, 2013. 4

Tatsiana Levina, Yuri Levin, Jeff McGill, and Mikhail Nediak. Dynamic pricing with online learning and strategic consumers: an application of the aggregating algorithm. *Operations research*, 57(2):327–341, 2009. 4

Heng Lian, John Scovil Atwood, Bojian Hou, Jian Wu, and Yi He. Online deep learning from doubly-streaming data. In *ACM Multimedia*, 2022. 101

Ji Liu, Jizhou Huang, Yang Zhou, Xuhong Li, Shilei Ji, Haoyi Xiong, and Dejing Dou. From distributed machine learning to federated learning: A survey. *Knowledge and Information Systems*, 64(4):885–917, 2022a. 118

Yanfang Liu, Xiaocong Fan, Wenbin Li, and Yang Gao. Online passive-aggressive active learning for trapezoidal data streams. *IEEE Transactions on Neural Networks and Learning Systems*, 2022b. 101

Gilles Louppe. Understanding random forests: From theory to practice. *arXiv preprint arXiv:1407.7502*, 2014. 15

Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. Learning under concept drift: A review. *IEEE transactions on knowledge and data engineering*, 31(12):2346–2363, 2018. 116

Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 4768–4777, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964. 32

Andrew Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, pages 142–150, 2011. 103, 106

Christoph Molnar. *Interpretable machine learning*. Lulu. com, 2020. 72, 112

Viraaji Mothukuri, Reza M Parizi, Seyedamin Pouriyeh, Yan Huang, Ali Dehghantanha, and Gautam Srivastava. A survey on security and privacy of federated learning. *Future Generation Computer Systems*, 115:619–640, 2021. 118

Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, Cambridge, MA, 2012. ISBN 9780262018029. 14, 21, 27

Silvia Nittel. Real-time sensor data streams. *SIGSPATIAL Special*, 7(2): 22–28, 2015. 4

Tommaso Paladini, Martino Bernasconi de Luca, Michele Carminati, Mario Polino, Francesco Trovò, and Stefano Zanero. Advancing fraud detection systems through online learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 275–292. Springer, 2023. 4

Juan Pardo, Francisco Zamora-Martínez, and Paloma Botella-Rocamora. Online learning algorithm for time series forecasting suitable for low cost wireless sensor networks nodes. *Sensors*, 15(4):9277–9304, 2015. 4

Lionel S Penrose. The elementary statistics of majority voting. *Journal of the Royal Statistical Society*, 109(1):53–57, 1946. 20

J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986. 20, 23, 101

J Ross Quinlan. C 4.5: Programs for machine learning. *The Morgan Kaufmann Series in Machine Learning*, 1993. 23, 25, 27, 101

Sebastian Raschka. Model evaluation, model selection, and algorithm selection in machine learning. *arXiv preprint arXiv:1811.12808*, 2018. 14, 15

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Model-agnostic interpretability of machine learning. *ArXiv*, abs/1606.05386, 2016. URL `https://api.semanticscholar.org/CorpusID:8561410`. 5

Lior Rokach. Ensemble-based classifiers. *Artificial intelligence review*, 33: 1–39, 2010. 37

Lior Rokach and Oded Z Maimon. *Data mining with decision trees: theory and applications*, volume 69. World scientific, 2008. 32, 58

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. 100

Christian Schreckenberger, Tim Glockner, Heiner Stuckenschmidt, and Christian Bartelt. Restructuring of hoeffding trees for trapezoidal data streams. In *2020 International Conference on Data Mining Workshops (ICDMW)*, pages 416–423. IEEE, 2020. xiv, 49, 65, 66, 67, 68, 69

Christian Schreckenberger, Christian Bartelt, and Heiner Stuckenschmidt. Dynamic forest for learning from data streams with varying feature spaces. *Lecture Notes in Computer Science*, 13591:95–111, 2022. 73, 76

Christian Schreckenberger, Yi He, Stefan Lüdtke, Christian Bartelt, and Heiner Stuckenschmidt. Online random feature forests for learning in varying feature spaces. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 4587–4595, 2023. xiv, 73, 74, 88, 102

Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014. 17

Shai Shalev-Shwartz et al. Online learning and online convex optimization. *Foundations and trends in Machine Learning*, 4(2):107–194, 2011. 4

Qi Shi and Mohamed Abdel-Aty. Big data applications in real-time traffic operation and safety monitoring and improvement on urban expressways. *Trans. Res. Part C: Emerging Technologies*, 58:380–394, 2015. 4

Rion Snow, Brendan O'connor, Dan Jurafsky, and Andrew Y Ng. Cheap and fast–but is it good? evaluating non-expert annotations for natural language tasks. In *Proceedings of the 2008 conference on empirical methods in natural language processing*, pages 254–263, 2008. 116

Tanapon Tantisripreecha and Nuanwan Soonthomphisaj. Stock market movement prediction using lda-online learning model. In *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 135–139. IEEE, 2018. 4

Vladimir Vapnik. Principles of risk minimization for learning theory. *Advances in neural information processing systems*, 4, 1991. 14, 15

M. Virgolin, Eric Medvet, Tanja Alderliesten, and Peter A. N. Bosman. Less is more: A call to focus on simpler models in genetic programming for interpretable machine learning. *ArXiv*, abs/2204.02046, 2022. URL `https://api.semanticscholar.org/CorpusID:247957926`. 5

Geoffrey I. Webb, Loong Kuan Lee, François Petitjean, and Bart Goethals. Understanding concept drift. *ArXiv*, abs/1704.00362, 2017. URL `https://api.semanticscholar.org/CorpusID:15696884`. 117

Stacey J Winham, Robert R Freimuth, and Joanna M Biernacka. A weighted random forests approach to improve predictive performance. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 6(6):496–505, 2013. 37, 38, 80

Hang Yang and Simon Fong. Moderated vfdt in stream mining using adaptive tie threshold and incremental pruning. In *Data Warehousing and Knowledge Discovery: 13th International Conference, DaWaK 2011, Toulouse, France, August 29-September 2, 2011. Proceedings 13*, pages 471–483. Springer, 2011a. 63

Hang Yang and Simon Fong. Optimized very fast decision tree with balanced classification accuracy and compact tree size. In *The 3rd interna-*

tional conference on data mining and intelligent information technology applications, pages 57–64. IEEE, 2011b. 63

Hao Yu, Michael Neely, and Xiaohan Wei. Online convex optimization with stochastic constraints. In *Advances in Neural Information Processing Systems*, pages 1428–1438, 2017. 4

Peng Zhang, Chuan Zhou, Peng Wang, Byron J Gao, Xingquan Zhu, and Li Guo. E-tree: An efficient indexing structure for ensemble models on data streams. *IEEE Transactions on Knowledge and Data engineering*, 27(2):461–474, 2014. 100

Qin Zhang, Peng Zhang, Guodong Long, Wei Ding, Chengqi Zhang, and Xindong Wu. Towards mining trapezoidal data streams. In *2015 IEEE International Conference on Data Mining*, pages 1111–1116. IEEE, 2015. 5, 18, 41, 100

Qin Zhang, Peng Zhang, Guodong Long, Wei Ding, Chengqi Zhang, and Xindong Wu. Online learning from trapezoidal data streams. *IEEE Transactions on Knowledge and Data Engineering*, 28(10):2709–2723, 2016. 5, 18, 41, 62, 70, 100, 102, 103

Yu Zhang, Peter Tino, Ales Leonardis, and Ke Tang. A survey on neural network interpretability. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 5(5):726–742, 2021. 15, 33

Zhen-Yu Zhang, Peng Zhao, Yuan Jiang, and Zhi-Hua Zhou. Learning with feature and distribution evolvable streams. In *ICML*, pages 11317–11327, 2020. 100

Lina Zhou, Shimei Pan, Jianwu Wang, and Athanasios V Vasilakos. Machine learning on big data: Opportunities and challenges. *Neurocomputing*, 237: 350–361, 2017. 3

Zhi-Hua Zhou. *Machine learning*. Springer nature, 2021. 26

Fei Zhu, Shijie Ma, Zhen Cheng, Xu-Yao Zhang, Zhaoxiang Zhang, and Cheng-Lin Liu. Open-world machine learning: A review and new outlooks. *arXiv preprint arXiv:2403.01759*, 2024. 117

Martin Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the 20th international conference on machine learning (icml-03)*, pages 928–936, 2003. 100

Appendix A

Additional Insights

A.1 Dynamic Fast Decision Tree

In a first step, the effects of varying input parameters are compared for both the DFDT algorithm, and the VFDT algorithm. This is performed on four data sets, three of which are generated synthetically as described. An overview for the characteristics of the datasets used is given in Table A.1.

Data Set	Feature Count			Data Instances	$f(i)$	Class Ratio
	Total	Start	End			
ParamEval_1	25	10	20	500000	$(-1)^i * i$	1 : 1
ParamEval_2	25	10	20	500000	$(-1)^i * i \log(i)$	1 : 1
ParamEval_3	25	10	20	500000	$(-1)^i * i^2$	1 : 1

Table A.1: Datasets used for evaluation of parameters

The different input parameters are evaluated by keeping all other parameters constant and inducing multiple trees for a wide range of different values of the target parameter. For parameters that are not specific to the DFDT algorithm, the same is done for the VFDT algorithm. The performance and size of the trees are measured after all examples are seen and the results then give an idea of how the different parameters influence the tree. The size of a decision tree refers to the number of leaf nodes and the performance is measured by accuracy, depending on the data set.

To limit the probability of choosing not the best test, δ is used to calcu-

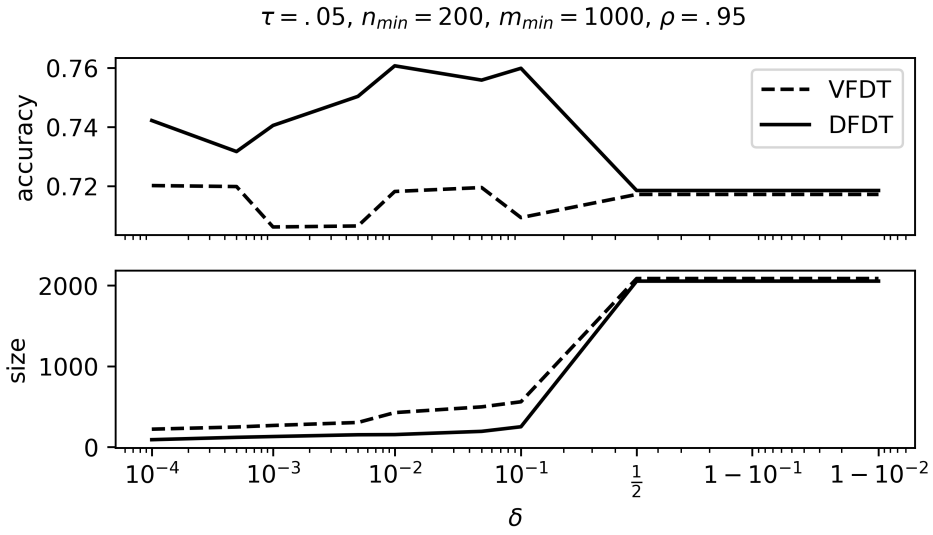


Figure A.1: Effects of varying parameter δ on ParamEval_1

late the Hoeffding bound. An increasing δ clearly results in larger trees for either algorithm (Figure A.1). It allows splits to occur earlier on, resulting in fewer data instances required per split, and at the same time, more data instances available for future splits. The effect on the accuracy, however, is not as clear. A low value for δ stops the tree from installing helpful tests, resulting in less accurate trees, whereas a high value may result in an overfitted tree. The wrong choice of this parameter can easily lead to overfitting or underfitting.

For all data sets, once a certain limit is reached, accuracy and size stay constant for any other value of δ above that limit. Such high δ result in such low values for ϵ that the split requirement is instantly satisfied. Therefore, each leaf turns into a decision node after n_{min} data instances are seen. Because all splits are the same, resulting trees are identical as well. The DFDT algorithms can cope better with this situation because it can undo unnecessarily induced splits.

To avoid recalculation of the information gains every time a new sample is seen in a leaf, only after n_{min} samples seen, it is recalculated. As stated before, this parameter increase should have similar effects as a lower δ would have, and indeed, there is a clear decrease in size as n_{min} increases (Figure A.2) until eventually the tree size is one. This happens if less than n_{min}

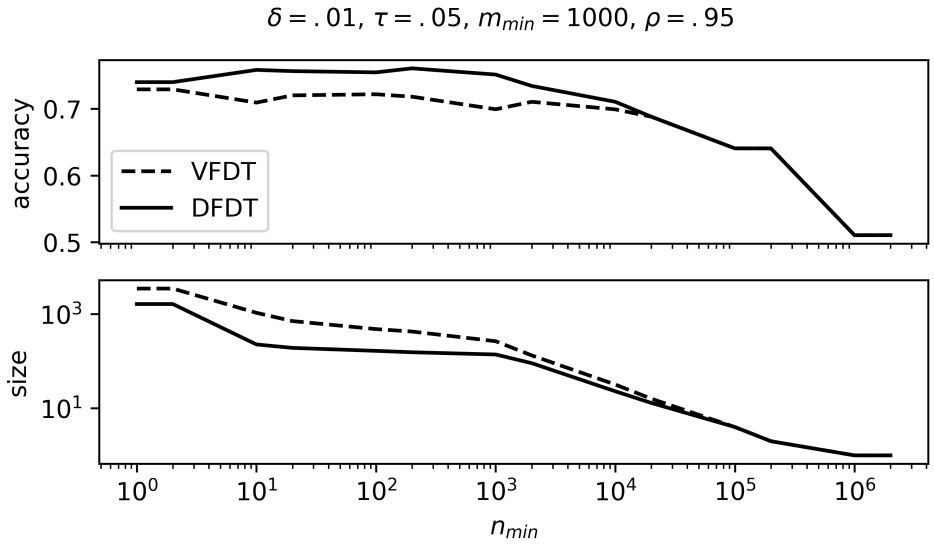


Figure A.2: Effects of varying parameter n_{min} on ParamEval$_1$

data instances are accumulated in the root node. There is an area where the DFDT algorithm can particularly benefit from a higher n_{min}; however, at some point, the accuracy gradually declines as the trees become excessively small. Besides the effects on accuracy, a higher value for n_{min} with the same accuracy is preferable due to computational advantages. The parameter τ determines if and when a tie is reached, so a test is installed even though the difference between the best and second-best test is smaller than ϵ.

In theory, there is a threshold for τ below which the choice of it does not matter. In practice, it might not be determinable, but the constant lines for size and accuracy do suggest a value for that limit - at least for the splits made within these trees so far (Figure A.3). A τ this low results in fewer splits, because it may take a lot of examples to determine the better of two tests. That in turn leads to smaller trees and possibly underfitting.

Similarly, there exists an upper limit above which the actual value of the parameter results in the same tree. Such τ results in a split as soon as it is evaluated, that is, when n_{min} data instances arrived at the leaf. The best test based on the current data is then installed.

DFDT seems to be coping slightly better with a higher τ. The reason for this may be that it can benefit from early splits based on relatively few data instances, provided they improve performance. At the same time, if

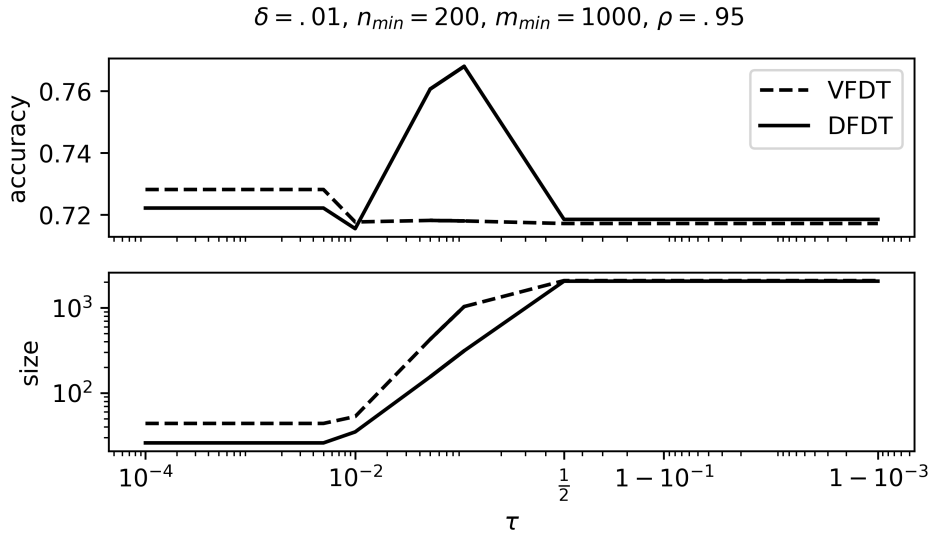


Figure A.3: Effects of varying parameter τ on ParamEval$_1$

some decision nodes turn out not to be helpful, they can be eliminated later on. What can be observed for both algorithms is that a high value for this parameter results in earlier splits and bigger trees, similar to high values of δ. The restructure interval m_{min} sets the number of samples that have to be fed into the tree as a whole, before the tree is restructured again. This is combined with pruning so that the tree may be smaller after this operation. Doing this very often results in more nodes being dropped; therefore, there is a clear correlation between low values for m_{min} and low tree size.

Again, there seems to be a limit from which point a varying m_{min} still results in the same tree (Figure A.4). This may be explained by the fact that values below that limit all restructure at any given points at which the operation changes the tree. This is clearly the same as m_{min} being one, and after each data instance seen, the tree is restructured.

Restructuring at a very high frequency increases the probability of pruning too harshly, which eventually would also result in dropping important decision nodes. This, in turn, would lead to lower accuracy. On the other hand, a very high restructure interval might miss the opportunity to prune unnecessary nodes early on, resulting in bigger trees. Once the parameter is higher than the number of examples the tree would ever see, restructuring never takes place, and the algorithm is effectively identical to the VFDT

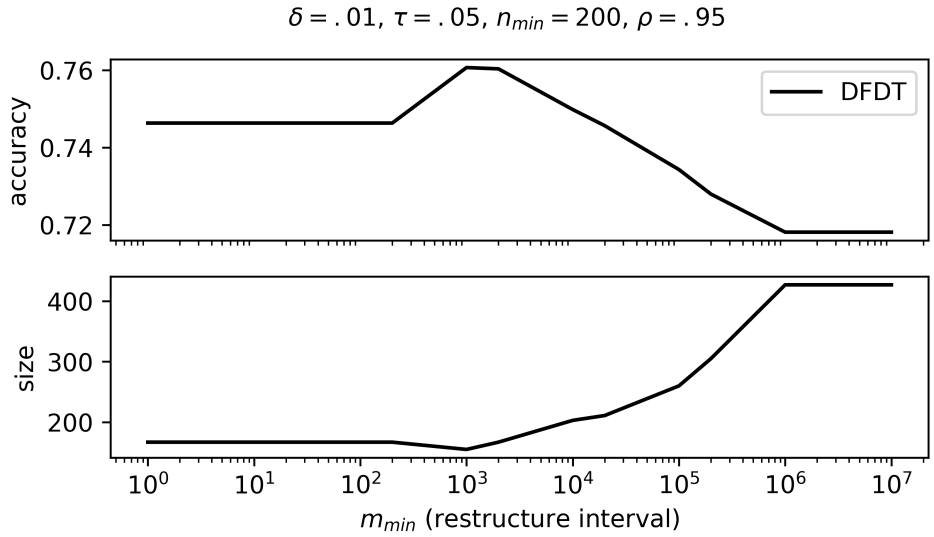


Figure A.4: Effects of varying parameter m_{min} on ParamEval_1

algorithm. The relatively low accuracy of all datasets once this point is reached should not be used to conclude that the DFDT algorithm is superior, because different parameters would be chosen for the VFDT algorithm, possibly leading to a different accuracy.
When pruning, the parameter ρ is used to calculate the Hoeffding bound and reevaluate a split. In order not to be pruned, a decision node does not have to yield as high of an information gain to be kept as for induction itself, because here, the Hoeffding bound only compares it to an information gain of zero. This check is also less strict, because generally, the values for ρ are higher than the values for δ.

The effect of this parameter on the accuracy does not follow a clear rule (Figure A.5), and it seems to be dependent on the data set. It may also be dependent on the other input parameters, so additionally, the parameter is evaluated with another two values for δ on one dataset. For those values, the accuracy increases with a higher ρ.

In general, higher values for ρ have an advantage if the other parameters are chosen appropriately. This statement can be observed better at a later point. It may be caused by the fact that pruning less harshly is better in combination with more careful splitting, which in turn leads to less data instances to be used up by splits that exist only temporarily.

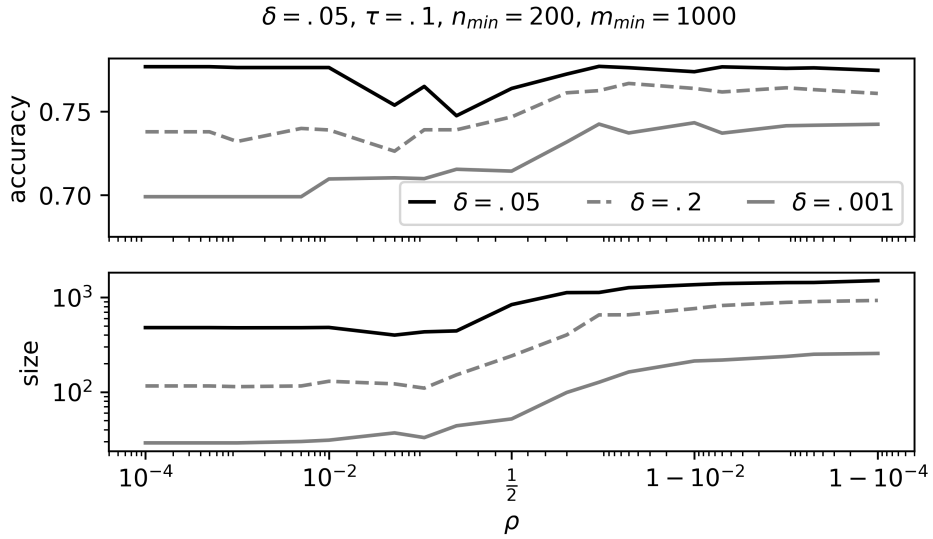


Figure A.5: Effects of varying parameter ρ on ParamEval$_2$

The effect on the size, however, is very clear. The lower this parameter the less likely a subtree is to be replaced by a leaf resulting in bigger trees which can be observed for all four datasets.

A.2 Dynamic Forest

In this section, we present experiments and an analysis of the various parameters used in the DynFo algorithm. Hereby, we evaluate the impact of the parameters α, β, and δ. The impact of the parameters is analyzed on the *german* dataset by simulating varying feature spaces with a removal ratio of 0.25. We report on the relearning operations, which are counted every time we reach line 8 of Algorithm 4.5 and the total error for the respective values of the inspected parameter. The relearning can be considered the most expensive operation in our proposed algorithm and should therefore be minimized while managing the trade-off with the errors.

A.2.1 Weight impacts of alpha

The α parameter impacts the weight updates for the respective weak classifiers. In the experiments with different values of α it showed that for a

lower α value of 0.1, we have less relearns than for a high α value of 1.0. However, while the change in the number of relearns is consistent, the impact is relatively small. For the lowest α value, we had 1494 relearning operations, while we had 1323 operations for the highest α value. In terms of performance, it shows that a lower α value is beneficial over a higher α value.

A.2.2 Relearning with beta

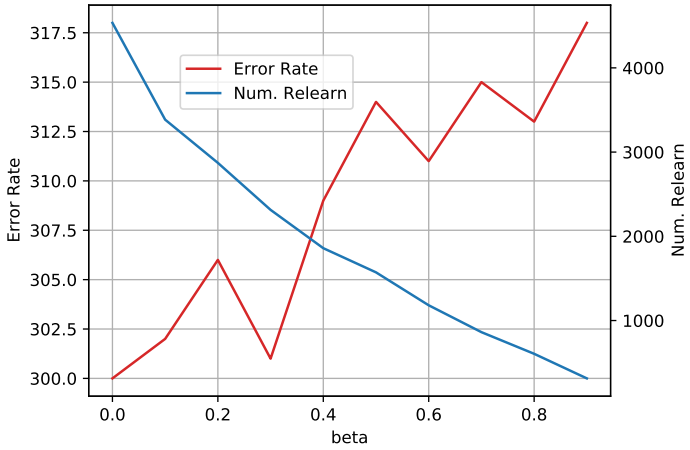


Figure A.6: Influence of the value of β on the total errors and the number of relearns on the *german* dataset with a removal ratio of 0.25.

The parameter β is the probability of keeping a weak learner despite not being in the top bracket of the weights. The impact of β on the error rate and the amount of relearning operations for a simulated varying feature space with a removal ratio of 0.25 is shown in Figure A.6. It can be seen that with an increasing value for β, the error number increases as well. However, the additional performance comes at the cost of roughly 4000 relearns for a low β value, which can be decreased to about 500 relearns for a β value of 0.9. Further experiments have shown that the impact of β for higher removal ratios, i.e., varying feature spaces that have a higher sparsity, is not as significant as for the presented removal ratio of 0.25. Therefore, depending on the sparsity, we recommend a lower β value for dense data streams and a higher β value for sparser data streams. Setting β to one means no relearning operations are performed; the algorithm's dynamic would then only come

from the dropping and the addition of new learners based on γ and θ_2.

A.2.3 Feature diversity for delta

The δ parameter determines the accepted feature space for a weak learner. Hence, it serves as a bagging parameter. We evaluated the parameter δ with respect to the feature diversity it introduces as well as the relearns it forces. We can observe that with an increasing value of δ, the number of relearns increases. While the impact is not as big as it is for β, it increased from 1271 to 2389 for $\delta = 0.1$ and $\delta = 1.0$ respectively. While for $\delta = 0.1$, all features of the dataset were used in the final ensemble, for $\delta = 1.0$, only a small subset of four features were actually used in the final ensemble. This means that δ can be considered to impact the feature diversity of the model. Considering that in varying feature spaces, not every feature is always present at a given time step, it is desirable to increase the diversity with a rather low value for δ. The actual value that should be used depends on the expected feature space, which is usually not known beforehand in the setting of varying feature spaces.

A.3 ORF^{3}V

We present the high-level calculations for the space and time complexity of the proposed ORF^{3}V algorithm.

A.3.1 Space Complexity

The ORF^{3}V algorithm allocates space for two main components: the feature forests and the t-digests. At any time step t, let $|\mathcal{F}_t|$ denote the number of features encountered so far. Since there is one feature forest per feature and each forest contains N decision stumps, the space required for the feature forests is $\mathcal{O}(|\mathcal{F}_t| \cdot N)$. Because N is a small constant, this space complexity is effectively linear in $|\mathcal{F}_t|$.

The second component, the t-digests, stores feature statistics. Each t-digest is bounded by the parameter λ, yielding a constant space requirement of $\mathcal{O}(\lambda)$ per digest. Since ORF^{3}V maintains one t-digest for every combination of feature and class, and if C represents the number of classes observed by time step t, then the total space needed for all t-digests is $\mathcal{O}(|\mathcal{F}_t| \cdot C \cdot \lambda)$.

With both λ and C being small constants in most scenarios, the overall space complexity for storing the t-digests is also linear in $|\mathcal{F}_t|$.

A.3.2 Time Complexity

The time complexity of the ORF^{3}V algorithm is influenced by five key components, all executed sequentially (c.f. Algorithm 4.7): updating feature statistics, pruning feature forests, generating feature forests, updating weights, and replacing decision stumps.

First, updating the feature statistics involves iterating over all features present in the instance x_t and updating the corresponding t-digests. Since each instance belongs to one class, a single pass over its feature space suffices, resulting in a complexity of $\mathcal{O}(|\mathcal{F}(x_t)|)$.

Next, pruning feature forests is performed by checking each forest in $\mathcal{L}$ to determine if a feature has vanished. Given that the number of forests at time t is at most $|\mathcal{F}_t|$, the pruning step has a complexity of $\mathcal{O}(|\mathcal{F}_t|)$.

Generating new feature forests for each new feature $F_d \in \mathcal{F}_t$ similarly incurs a complexity of $\mathcal{O}(|\mathcal{F}_t|)$.

For weight updates, a prediction is computed for every feature forest corresponding to a value in x_t, which again results in a complexity of $\mathcal{O}(|x_t|)$.

Lastly, updating the feature forests (executed every r-th step) also depends on the number of features, yielding a complexity of $\mathcal{O}(|\mathcal{F}_t|)$.

Combining these, the overall time complexity is

$$\mathcal{O}(|\mathcal{F}_t| + |\mathcal{F}_t| + |\mathcal{F}_t| + |\mathcal{F}(x_t)| + |\mathcal{F}_t|).$$

Since $|\mathcal{F}_t| > |\mathcal{F}(x_t)|$, this simplifies to $\mathcal{O}(5 \cdot |\mathcal{F}_t|)$, which is equivalent to $\mathcal{O}(|\mathcal{F}_t|)$. Therefore, the learning process scales linearly with the number of observed features at time step t.

A.4 Additional Results

The results depicted in Table A.2 and Table A.3 show the results of the benchmark evaluation data set as introduced in 4.5. Instead of displaying the cumulative error rate, the total errors made per dataset are shown here.

Dataset	Rem.	*OLVF*	DynFo	ORF^{3}V
german	0.25	333.4 ± 9.7	$\mathbf{302.0 \pm 3.6}$	304.4 ± 1.5
	0.5	350.9 ± 7.8	307.1 ± 7.3	$\mathbf{305.4 \pm 3.0}$
	0.75	365 ± 3.6	325.5 ± 17.2	$\mathbf{309.2 \pm 2.0}$
ionosphere	0.25	77.2 ± 7.1	$\mathbf{67.1 \pm 5.2}$	79.6 ± 2.4
	0.5	79.5 ± 7.4	$\mathbf{75.9 \pm 5.6}$	86.4 ± 2.1
	0.75	$\mathbf{79.7 \pm 5.9}$	102.9 ± 7.2	100.8 ± 9.7
spambase	0.25	$\mathbf{659.8 \pm 14.5}$	885.4 ± 61.4	$1220. \pm 20.7$
	0.5	$\mathbf{864 \pm 20.6}$	939.6 ± 36.1	1216 ± 14.6
	0.75	1375 ± 21.5	$\mathbf{1120.8 \pm 30.6}$	1275.4 ± 24.3
magic04	0.25	6152.4 ± 54.7	$\mathbf{5229.6 \pm 90.7}$	5323.8 ± 62.3
	0.5	6775 ± 27.4	$\mathbf{5295.2 \pm 80.5}$	5505.6 ± 53.4
	0.75	7136 ± 2.7	$\mathbf{5674.7 \pm 55.9}$	6489 ± 51.4
svmguide3	0.25	346 ± 11.6	286.1 ± 8.9	$\mathbf{280.4 \pm 5.5}$
	0.5	367.2 ± 11.9	295.7 ± 11.3	$\mathbf{288.2 \pm 7.0}$
	0.75	371 ± 11.7	322.1 ± 22.3	$\mathbf{300.6 \pm 4.8}$
wbc	0.25	$\mathbf{25.3 \pm 1.4}$	42.5 ± 5.6	65.2 ± 5.9
	0.5	60.6 ± 5.3	$\mathbf{53.6 \pm 6.6}$	67.2 ± 3.6
	0.75	123.1 ± 3.6	$\mathbf{92.8 \pm 9.0}$	122.2 ± 8.4
wpbc	0.25	88.5 ± 5.8	55.4 ± 4.2	$\mathbf{52.5 \pm 1.0}$
	0.5	90.2 ± 5.1	$60.6, \pm 6.7$	$\mathbf{52.6 \pm 1.6}$
	0.75	107.7 ± 7.7	71.2 ± 7.2	$\mathbf{53.2 \pm 1.1}$
wdbc	0.25	$\mathbf{40.8 \pm 3.5}$	50.6 ± 4.7	67.6 ± 4.2
	0.5	55.2 ± 5.4	$\mathbf{55.1 \pm 4.9}$	69.8 ± 6.8
	0.75	202.85 ± 7.5	$\mathbf{78.0 \pm 7.0}$	84.4 ± 8.4
a8a	0.25	8993.8 ± 40.3	7834.4 ± 14.3	$\mathbf{7585.6 \pm 28.5}$
	0.5	9585.8 ± 53.8	7805.15 ± 22.3	$\mathbf{7683 \pm 24.3}$
	0.75	12453 ± 74.4	$\mathbf{7709.6 \pm 58.9}$	7781.2 ± 18.4
imdb	-	9804.2 ± 65.9	8535.6 ± 584.9	$\mathbf{7144.1 \pm 70.0}$

Table A.2: Cumulative error by OLVF, DynFo, and ORF^{3}V on simulated data streams with varying feature spaces with removing ratios (Rem.) of 0.25, 0.5, and 0.75.

Dataset	OLSF	OLVF	DynFo	ORF^{3}V
german	369.0 ± 10.9	356.3 ± 9.1	$\mathbf{303.1 \pm 12.2}$	$\mathbf{303.4 \pm 6.1}$
ionosphere	85.3 ± 5.8	$\mathbf{58.3 \pm 3.4}$	70.6 ± 5.1	80.8 ± 4.6
spambase	$\mathbf{977.6 \pm 47.4}$	1163.5 ± 14.9	1080.0 ± 44.5	1237 ± 44.2
magic04	6109.5 ± 62.1	6408.5 ± 78.0	$\mathbf{5242.6 \pm 68.4}$	5390.8 ± 134.1
svmguide3	379.8 ± 26.0	438.3 ± 31.8	303.1 ± 21.6	$\mathbf{292.0 \pm 5.7}$
wbc	37.9 ± 9.1	$\mathbf{31.6 \pm 2.2}$	50.2 ± 6.4	77.4 ± 7.8
wpbc	84.8 ± 5.4	88.1 ± 11.0	53.5 ± 5.6	$\mathbf{52.8 \pm 0.9}$
wdbc	125.5 ± 10.1	74.4 ± 10.5	$\mathbf{70.2 \pm 5.2}$	70.2 ± 9.3
a8a	10356.4 ± 140.1	11522.3 ± 119.6	7852.6 ± 10.2	$\mathbf{7534.4 \pm 90.6}$

Table A.3: Average number of errors made by OLSF, OLVF, and ORF^{3}V in the simulated trapezoidal data streams, and on the real-world IMDB dataset

Appendix B

Disclosures

B.1 Contributions

Publication	**My Contributions**
Restructuring of Hoeffding trees for trapezoidal data streams.	- Developed research question - Supervised thesis/Guided development of approach - Reimplemented code in GoLang for performance - Ran additional experiments - Wrote manuscript based on thesis
Dynamic forest for learning from data streams with varying feature spaces.	- Developed research question - Developed approach - Implemented code in Python - Ran experiments - Wrote manuscript
Online random feature forests for learning in varying feature spaces	- Developed research question - Idea and Collection of crowdsense data - Developed approach - Implemented code in GoLang - Ran experiments - Wrote draft - Finalized manuscript based on feedback of co-authors
Towards utilitarian online learning–a review of online algorithms in open feature space	- Initiated literature review project - Conducted the literature search - Collaboration on research questions - Gathered and prepared datasets - Ran some of the experiments - Contributed to draft - Contributed to the final manuscript

Table B.1: Summary of contributions to publications.

B.2 Tool Disclosures

Generative AI-based tools were used to support the preparation of this dissertation. ChatGPT (Model 4o and Model 5) was utilized for drafting, language refinement (clarity), and stylistic suggestions, while Grammarly was employed in a final pass to correct grammatical errors and further refine language.