Universidade Estadual de Londrina

VICTOR GUILHERME TURRISI DA COSTA

STRICT VERY FAST DECISION TREE: A MEMORY CONSERVATIVE ALGORITHM FOR DATA STREAM CLASSIFICATION

Londrina
2019

VICTOR GUILHERME TURRISI DA COSTA

STRICT VERY FAST DECISION TREE: A MEMORY CONSERVATIVE ALGORITHM FOR DATA STREAM CLASSIFICATION

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Sylvio Barbon Junior

Londrina
2019

Ficha de identificação da obra elaborada pelo autor, através do Programa de Geração Automática do Sistema de Bibliotecas da UEL

Costa, Victor Guilherme Turrisi.
Strict Very Fast Decision Tree : A memory conservative algorithm for data stream classification / Victor Guilherme Turrisi Costa. - Londrina, 2019.
91 f.

Orientador: Sylvio Barbon Junior.
Dissertação (Mestrado em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Programa de Pós-Graduação em Ciência da Computação, 2019.
Inclui bibliografia.

1. Machine Learning - Tese. 2. Data Stream Mining - Tese. 3. Memory Conservative Algorithm - Tese. I. Barbon Junior, Sylvio . II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação. III. Título.

VICTOR GUILHERME TURRISI DA COSTA

STRICT VERY FAST DECISION TREE: A MEMORY CONSERVATIVE ALGORITHM FOR DATA STREAM CLASSIFICATION

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Mestre em Ciência da Computação.

BANCA EXAMINADORA

Orientador: Prof. Dr. Sylvio Barbon Junior
Universidade Estadual de Londrina – UEL

Prof. Dr. Bruno Bogaz Zarpelão
Universidade Estadual de Londrina – UEL

Prof. Dr. Rafael Gomes Mantovani
Universidade Tecnológica Federal do Paraná – UTFPR

Londrina, 27 de março de 2019.

ACKNOWLEDGEMENTS

Primeiramente, gostaria de agradecer aos meus pais, Marcia e Felinto. Não faltam motivos pelos quais eu deveria agradecê-los.

Gostaria de agradecer também a ambas as minhas avós, Teresa e Maria. Infelizmente você partiu cedo demais, vó Teresa, mas tenha certeza que você contribuiu muito para quem eu sou hoje.

Agradeço a minha namorada e companheira, Raissa, por esses quatro anos. Obrigado por toda a influência que você tem na minha vida.

Agradeço também ao restante da minha família.

Agradeço ao meu orientador, professor e amigo, Sylvio, por todo o apoio e orientação no decorrer desse mestrado.

Agradeço ao meu orientador de TCC, professor e amigo, Bruno.

Agradeço a todos os meus amigos!

Agradeço aos meus companheiros de laboratório e estudos.

Por fim, agradeço pelo apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

*"Great passions are incurable diseases." -
Johann Wolfgang von Goethe*

COSTA, V. G. T. **Árvore de decisão estrita e muito rápida: um algoritmo conservador de memória para classificação de fluxo de dados.** 2019. 91 f. Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2019.

RESUMO

Lidar com restrições de memória e tempo são desafios presentes quando deseja-se aprender com fluxos massivos de dados. Diversos algoritmos foram propostos para lidar com essas dificuldades, entre eles, a Very Fast Decision Tree (VFDT). Apesar da VFDT ser amplamente utilizada para a mineração de fluxos de dados, nos últimos anos, diversos autores sugeriram mudanças para melhorar a capacidade preditiva desse algoritmo, mas ignorando os altos custos adicionais de memória e tempo ocasionados por essas modificações. Além disso, mais recentemente, a maioria dos algoritmos para mineração de fluxos de dados são baseados em ensembles, sendo bem custosos considerando os recursos de memória e tempo. Esse estudo apresenta um novo algoritmo baseado na VFDT chamado de Strict VFDT (SVFDT) que reduz custos de memória mantendo capacidade preditiva similar a VFDT. Além disso, como as árvores de decisão geradas são mais rasas, a SVFDT acaba reduzindo o tempo computacional necessário para se processar um fluxo de dados. Esse algoritmo aplica medidas heurísticas para bloquear divisões de folhas que não resultam em um aumento significativo na capacidade preditiva. Foram realizados experimentos comparando a SVFDT com a VFDT em 26 fluxos de dados. Após realizar diversas análises, cada qual focando em algum dos múltiplos aspectos dos algoritmos, foi possível observar que o algoritmo proposto é capaz de reduzir drasticamente o consumo de memória enquanto possui capacidade preditiva similar e sendo mais rápido em muitos casos.

Palavras-chave: Aprendizado de máquina. Mineração de fluxos de dados. Algoritmo com baixo custo de memória. Algoritmo baseado em árvore para classificação.

COSTA, V. G. T. **Strict Very Fast Decision Tree: a memory conservative algorithm for data stream classification.** 2019. 91 p. Dissertation (Master's Deegre of Science in Computer Science) – Universidade Estadual de Londrina, Londrina, 2019.

ABSTRACT

Memory and time constraints are current challenges when learning from massive data streams. Many algorithms have been proposed to handle these difficulties, among them, the Very Fast Decision Tree (VFDT). Although it has been widely used in data stream mining, in the last years, several authors have suggested modifications to increase its predictive performance, putting aside the high memory and time demands of these modifications. Besides, recently, most data stream mining solutions have been centred around ensembles, which are very costly from a memory and time point of view. This study presents a novel algorithm based on the VFDT called Strict VFDT (SVFDT) that reduces memory costs while maintaining predictive performance. Moreover, since it creates much shallower trees than VFDT, the SVFDT can achieve a shorter computational time. It works by blocking splits which would not yield a significant increase in predictive performance using heuristic measures. Experiments were carried out comparing the SVFDT with the VFDT in 26 data stream datasets. By performing different analysis focusing on multiple aspects of the algorithms, the proposed algorithm was able to greatly reduce memory consumption while yielding similar predictive performance and being faster in many cases.

Keywords: Machine learning. Data stream mining. Memory conservative algorithm. Tree-based classification algorithm.

LIST OF FIGURES

Figure 1 – Example of a binary classification problem. 19
Figure 2 – Example of a first level split of a decision tree in a binary classification problem. 19
Figure 3 – Example of a first level split of a decision tree in a binary classification problem. 20
Figure 4 – Example of a unnecessary splits on a binary classification problem. . . 22
Figure 5 – Types of Concept Drifts. Image from [1]. 24
Figure 6 – Behaviour of the HB function with $R = \log_2 2$ and $\delta = 10^{-4}$. 26
Figure 7 – Example of a Gaussian numeric estimator for a feature in a binary problem. Image adapted from [2]. 28
Figure 8 – Diagram of the VFDT. 30
Figure 9 – Diagram of the SVFDT. Image created by the author. 42
Figure 10 – Prequential evaluation. 48
Figure 11 – Boxplots of the performance metrics of the VFDT, SVFDT-I and SVFDT-II. 55
Figure 12 – Heatmaps of each algorithm for each possible GP and τ configuration using entropy and IG. Rows correspond to different metrics. Red colours represent better performance (higher accuracy or lower memory or time consumption), while blue colours the opposite. 57
Figure 13 – Heatmaps of each algorithm for each possible GP and τ configuration using GI. Rows correspond to different metrics. Red colours represent better performance (higher accuracy or lower memory or time consumption), while blue colours the opposite. 58
Figure 14 – Heatmaps of the performance of each algorithm when considering accuracy, memory and time efficiency for each possible GP and τ. Red colours represent better performance, while blue colours the opposite. . 59
Figure 15 – Accuracy Nemenyi for algorithms using entropy and IG ($\alpha = 0.05$). . . 60
Figure 16 – Kappa M Nemenyi for algorithms using entropy and IG ($\alpha = 0.05$). . . 60
Figure 17 – Memory consumption Nemenyi for algorithms using entropy and IG ($\alpha = 0.05$). 60
Figure 18 – Time costs Nemenyi for algorithms using entropy and IG ($\alpha = 0.05$). . 60
Figure 19 – Accuracy Nemenyi for algorithms using GI ($\alpha = 0.05$). 61
Figure 20 – Kappa M Nemenyi for algorithms using GI ($\alpha = 0.05$). 61
Figure 21 – Memory consumption Nemenyi for algorithms using GI ($\alpha = 0.05$). . . 61
Figure 22 – Time costs Nemenyi for algorithms using GI ($\alpha = 0.05$). 61

Figure 23 – Performance metrics of all algorithms by dataset for $\tau = 0.01$, $GP = 800$, H and IG (worst case). The metrics are scaled by dividing by the largest value for each dataset. Larger values are better for accuracy and Kappa M, and lower values are better for memory and time. 62
Figure 24 – Performance metrics of all algorithms by dataset for $\tau = 0.20$, $GP = 100$, H and IG (best case). The metrics are scaled by dividing by the largest value for each dataset. Larger values are better for accuracy and Kappa M, and lower values are better for memory and time. 64
Figure 25 – Performance metrics of all algorithms by dataset for $\tau = 0.01$, $GP = 100$ and GI (worst case). The metrics are scaled by dividing by the largest value for each dataset. Larger values are better for accuracy and Kappa M, and lower values are better for memory and time. 65
Figure 26 – Performance metrics of all algorithms by dataset for $\tau = 0.20$, $GP = 200$ and GI (best case). The metrics are scaled by dividing by the largest value for each dataset. Larger values are better for accuracy and Kappa M, and lower values are better for memory and time. 67
Figure 27 – Performance metrics of all algorithms by dataset for $\tau = 0$, $GP = 100$, H and IG. The metrics are scaled by dividing by the largest value for each dataset. Larger values are better for accuracy and Kappa M, and lower values are better for memory and time. 69
Figure 28 – Performance metrics of all algorithms by dataset for $\tau = 0$, $GP = 100$ and GI. The metrics are scaled by dividing by the largest value for each dataset. Larger values are better for accuracy and Kappa M, and lower values are better for memory and time. 70

LIST OF TABLES

Table 1 – Summary of related work. 38
Table 2 – Summary of the datasets used in the experiment. 47
Table 3 – Hyperparameters explored in the experiments 49
Table 4 – Performance metrics of all algorithms for entropy and IG. 52
Table 5 – Performance metrics of all algorithms for GI. 54
Table 6 – Possible combinations for the constraints and skip mechanisms for the SVFDTs. 71
Table 7 – Performance metrics for different combinations of constraints and skip mechanisms using for $\tau = 0.05$, $GP = 1000$, entropy and IG. 72
Table 8 – Performance metrics for different combinations of constraints and skip mechanisms using for $\tau = 0.05$, $GP = 1000$ and GI. 73
Table 9 – Performance metrics of all algorithms by dataset for $\tau = 0.01$, $GP = 800$, entropy and IG (worst case). 83
Table 10 – Performance metrics of all algorithms by dataset for $\tau = 0.20$, $GP = 100$, entropy and IG (best case). 84
Table 11 – Performance metrics of the algorithms by dataset for $\tau = 0.01$, $GP = 100$ and GI (worst case). 85
Table 12 – Performance metrics of all algorithms by dataset for $\tau = 0.20$, $GP = 200$ and GI (best case). 86
Table 13 – Performance metrics of all algorithms by dataset for $\tau = 0$, $GP = 100$, entropy and IG. 88
Table 14 – Performance metrics of all algorithms by dataset for $\tau = 0$, $GP = 100$ and GI. 89

LIST OF ABBREVIATIONS AND ACRONYMS

AI	Artificial Intelligence
ANB	Adaptive Naive Bayes
CART	Classification and Regression Tree
CD	Concept Drift
CVDFT	Concept-adapting Very Fast Decision Tree
EVFDT	Enhanced Very Fast Decision Tree
GI	Gini Impurity
GP	Grace Period
HB	Hoeffding Bound
HOT	Hoeffding Option Tree
H	Entropy
ID3	Iterative Dichotomise 3
IoT	Internet of Things
IG	Information Gain
MC	Most Common
ML	Machine Learning
NB	Naive Bayes
OVFDT	Optimised-Very Fast Decision Tree
RAM	Random Access Memory
RBF	Radial Basis Function
SVFDT	Strict Very Fast Decision Tree
VFDT-GTD	Very Fast Decision Tree Genuine Tie Detection
VFDT-IWT	Very Fast Decision Tree Increase Wait Time
VFDT	Very Fast Decision Tree

CONTENTS

1 INTRODUCTION . . . 14
1.1 Hypothesis . . . 15
1.2 Objectives and Contributions . . . 16
1.3 Outline . . . 16

2 THEORETICAL FOUNDATION . . . 17
2.1 Machine Learning . . . 17
2.1.1 Decision Tree Induction Algorithms . . . 18
2.1.2 Heuristic Metrics . . . 19
2.1.3 Pruning Techniques . . . 21
2.1.4 Naive Bayes . . . 22
2.2 Data Stream Mining . . . 22
2.2.1 Data Streams . . . 22
2.2.2 Very Fast Decision Tree . . . 25
2.2.3 Hyperparamaters . . . 30

3 RELATED WORK . . . 34
3.1 Direct modifications to the VFDT algorithm . . . 34
3.2 New algorithms based on the VFDT . . . 36

4 STRICT VERY FAST DECISION TREE . . . 39
4.1 Memory and Time Complexities . . . 41

5 MATERIALS AND METHODS . . . 45
5.1 Datasets . . . 45
5.2 Evaluation Metrics . . . 47
5.3 Experimental Setup . . . 48

6 RESULTS . . . 50
6.1 Analysis of the influence of the hyperparameters . . . 50
6.2 Friedman's statistical test and post-hoc Nemenyi analysis . . . 59
6.3 Best and worst results according to hyperparameters . . . 61
6.4 Can the SVFDTs produce trees smaller than the smallest VFDT? . . . 68
6.5 Exploring the impact of each heuristic . . . 70

7 CONCLUSION . . . 74

BIBLIOGRAPHY 76

APPENDIX 81

APPENDIX A – WORST AND BEST POSSIBLE SCENARIOS FOR THE SVFDTS 82

APPENDIX B – CAN THE SVFDTS PRODUCE TREES SMALLER THAN THE SMALLEST VFDT? 87

Works Published by the Author 90

1 INTRODUCTION

Recently, due to technological advances, Machine Learning (ML) is becoming more pervasive in real-world applications. ML algorithms are being employed for several tasks, e.g., detecting spam and malware [3], driving autonomous vehicles [4], detecting diseases [5], identifying objects in images [6], performing image segmentation [7], sentiment analysis [8], language to language translation [9], and artistic style transfer for images [10], to name a few.

Traditional ML algorithms work by modelling knowledge from static datasets. In other words, data from a problem of interest is collected during some period of time and then used by an algorithm to induce a model. Nowadays, massive amounts of data are available, but traditional ML algorithms are not capable of taking advantage of this and so, a new field of study emerged, called Data Stream Mining [11, 12]. It consists of algorithms capable of dealing with very large volumes of data, which usually comes in the form of continuous streams [1, 13].

Unlike learning from static data, learning from data streams assumes that new data can arrive at any time, which can make the current model outdated. Additionally, Concept Drifts (CD), which are related to the change of data distribution in the problem space over time, are also possible in this scenario. Therefore, learning from data streams requires continuous model updates. An additional challenge posed by learning from these streams is the demand to perform accurate predictions at any time [1, 11]. This means that algorithms need reasonable performance even with a low number of instances, i.e., algorithms with fast convergence are desired. Also, model updating must be fast so it does not become a bottleneck for solutions and the memory available can be very limited, depending on where this model is updated (for example, in an Internet of Things (IoT) device). A good algorithm is capable of efficiently dealing with processing time and memory space [14, 11, 1].

Many learning algorithms have been proposed to cope with some of these aspects. Among them, the Very Fast Decision Tree (VFDT) [14] is one of the most well-known algorithms for data stream classification, being capable of constructing a decision tree (DT) in an online fashion by taking advantage of a statistical property called Hoeffding Bound (HB). By doing so, the VFDT obtains a predictive performance similar to conventional DTs created by traditional algorithms applied to static datasets [14]. Although the VFDT is memory-friendly when compared to ML algorithms for batch data, learning from data streams can lead to unnecessary tree growth, increasing memory usage, compromising its application on memory-scarce scenarios and even making it perform poorly.

In the last years, modifications to the VFDT algorithm have been proposed to improve it [15, 16, 17, 18]. However, this came at the cost of a substantial increase in memory usage without any substantial predictive performance increase. Moreover, according to Krawczyk et al. [1], data stream researchers are shifting their focus to ensemble-based solutions, which are algorithms with a better predictive performance by nature. Ensemble solutions combine multiple ML algorithms (usually named base learners in this situation) in a specific fashion to create a model with higher predictive performance. The predictive performance of these solutions depends on the strength (predictive performance) of the base learners and the statistical correlation between them. An optimal ensemble has base learners with high predictive performance and low statistical correlation. However, in practice, increasing the predictive performance of each base learner will result in a higher statistical correlation. In this sense, ensembles with only weak learners, i.e., base learners with relatively low predictive performance, will have very similar predictive performance as ensembles with strong base learners, as long as the correlation between weak learners is low [19]. Nonetheless, ensembles are very costly, since the memory and time costs of each learner are stacked, which hinders the use of ensembles.

Here, a new modification of the VFDT algorithm called Strict VFDT (SVFDT) is proposed to reduce the VFDT's memory costs without compromising predictive performance. It works by allowing growth only in branches of the tree that will likely lead to a high increase in predictive performance. To do so, we introduced four constraints that block splits in the following situations:

1. Leaves in which the classification problem is already easy.

2. Splits which do not contribute to solving the problem.

3. Leaves where only a small amount of instances are present.

Furthermore, a second version is also proposed, introducing skip mechanisms which ignore the proposed constraints when the classification problem is *too hard* or splitting will solve it. Both versions were extensively evaluated in 26 datasets considering predictive performance, memory costs and training time. Results show that both SVFDTs significantly reduce memory costs while only decreasing predictive performance by a small margin. Likewise, they are also faster than the VFDT in multiple scenarios.

1.1 Hypothesis

Creating shallower DTs result in a significant boost in their generalisation capabilities [20, 21, 22]. To do so, split nodes from the DTs are turned into leaves in a process called pruning. In batch scenarios, it is possible to evaluate the impact of pruning and

undo it if the resulting DT performs poorly. However, in data streams, undoing pruned nodes is not possible due to time limitations and the fact that instances are not stored. Nonetheless, our hypothesis is the following:

"It is possible to create a much shallower DT, reducing its memory costs while keeping predictive performance, in a data stream environment, without undoing pruned nodes"

1.2 Objectives and Contributions

This work proposes a modification to the Very Fast Decision Tree called Strict Very Fast Decision Tree to significantly reduce memory costs while maintaining predictive performance. Additionally, the SVFDT is also faster than the VFDT in many cases.

The main contributions of this work can be summarised as:

1. Proposes the SVFDT which uses significantly less memory in comparison to the VFDT, reaching similar predictive performance. Two versions were designed, one focusing on using the least possible amount of memory and another with skip mechanisms to yield better predictive performance.
2. Experimental evaluation of the SVFDTs against the VFDT in 26 benchmark datasets.
3. Evaluation of the impact of each individual heuristic and their possible combinations to guide the creation of new heuristics.

1.3 Outline

The remainder of this work is divided in the following manner: Chapter 2 presents a theoretical foundation about traditional ML, concepts regarding data streams and ML on them, and the VFDT algorithm. Chapter 3 presents, as related work, algorithms that modified the VFDT to improve it. In Chapter 4, the SVFDT algorithm is presented. Materials and methods are discussed in Chapter 5. The results are presented in Chapter 6. Chapter 7 presents the conclusions.

2 THEORETICAL FOUNDATION

This study coverts ML algorithms applied to data streams, often called Data Stream Mining. In this way, it is important to first define ML to then move on to the definition of data streams. Two ML algorithms are also presented, the DT and Naive Bayes (NB), since they are broadly used and modified in the Data Stream Mining field.

2.1 Machine Learning

Machine Learning is a subarea of Artificial Intelligence (AI) which comprises to the capacity to improve performance in a task using past experience [23, 24]. Furthermore, ML can also be defined as the process of inducing a hypothesis or approximating a function from previous experiences [23]. Mitchel [24] gives the following, more formal, definition:

> "*A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*"

Traditional ML tasks are divided into two main categories: supervised (predictive) or unsupervised (descriptive). Supervised tasks are divided primarily into classification and regression tasks. Flach [22] defines a classification task as creating a **classifier** $\hat{c}$ that is a mapping $\hat{c} : X \rightarrow Y$, which better approximates the real mapping $c : X \rightarrow Y$. X is the instance space. $Y = \{y_1, y_2, y_3, ..., y_m\}$ is a finite set of class labels. And each instance (example) is presented in the form of $(\mathbf{x}, c(\mathbf{x}))$, where $\mathbf{x} = [x_1, x_2, x_3, ..., x_f] \in X$, f being the number of features, and $c(\mathbf{x}) \in Y$. Learning a classifier consists of trying to create the function $\hat{c}$ which best matches c on not just the training set, but ideally on the whole instance space X. A regression task consists of creating a **regressor** $\hat{r}$ which is a mapping $\hat{r} : X \rightarrow \mathbf{R}$. In this case, $\hat{r}$ maps the instance space X to the real space and instances are presented in the format of $(\mathbf{x}, r(\mathbf{x}))$, $r(\mathbf{x}) \in R$.

Unsupervised tasks are mostly clustering (grouping) tasks. Given the available data $D \subseteq X$, a clustering model is a mapping $\hat{q} : D \rightarrow C$, where $C = \{c_1, c_2, c_3, ..., c_p\}$ is simply the cluster of a given instance [22]. Instances are grouped into clusters given a similarity measure, i.e., the Euclidean distance.

To better illustrate the differences between the three tasks, consider the following scenario, described in [23]. Assume that there is a dataset containing information about patients in a hospital. In this dataset, the information about each patient are characteristics (features) which describe their main aspects. These features can be the age, height and weight of a patient, as well as, the results according to some given clinical test. There

are two types of features: quantitative (numeric) and qualitative (nominal). Quantitative features refer to continuous features which can assume an infinite number of values, e.g., results from measures, such as weight and height of a patient. Differently, qualitative features assume discrete values, e.g., binary (for example, if a patient has a given condition) or nominal values (for example, in which month the patient felt sick). Now, consider that we would like to *predict* one of these features. For example, using the age, height, weight and some clinical test results from a patient, we would like to *predict* if that patient will test positive for some disease (without actually performing the laboratory test). In this case, we have a **classification task**, since prediction goal (target) is a discrete value: if the patient will test positive for a disease. On the other hand, if we tried to predict a continuous variable, the problem would become a **regression task**, for example, the concentration of red blood cells. In descriptive tasks, the goal is to explore and better describe data and no target variables exist. Using the same previous example, we may want to group patients based on their similarities into clusters, e.g., patients with similar weights and heights that tested positive to some disease.

Since this work focuses solely on classification tasks, algorithms will be described focusing only on it. Nonetheless, the techniques presented here can also be applied to other ML tasks, although outside the scope of this work.

2.1.1 Decision Tree Induction Algorithms

DT algorithms are frequently used for supervised learning. Multiple implementations have been proposed throughout the years, e.g., Classification and Regression Tree (CART) [25], Iterative Dichotomiser 3 (ID3) [26] and C4.5 [27]. As defined by Gama et al. [23] and Mitchel [24], decision tree algorithms, apart from some particularities, employ a divide and conquer approach to recursively build a decision scheme in the format of a tree. A complex problem is recursively divided into smaller and easier problems. Then, the solution for these smaller problems are combined to solve the original problem. Considering that instances are scattered in a hyperspace (formed by the m features), a DT divides this space into subspaces. This is also called the learning process.

To better visualise the learning process employed by DT, consider the binary problem represented in Figure 1. At first, a DT will try to simplify this problem by separating it into two problems (since these algorithms usually perform binary splits). To do so, it will divide the hyperplane. One possible division is represented in Figure 2, where the DT algorithm found out that it was best to separate instances with feature $f_1 \leq 20$ from instances with feature $f_1 > 20$ (represented by the line in orange). At this point, for the true branch, the problem is already simple enough, since only instances from class 1 are presented. Nonetheless, the problem on the other branch is still not simple enough, and so, the divide and conquer procedure will continue for that leaf. This leaf is

divided in instances with feature $f_1 \leq 40$ and instances with feature $f_1 > 40$, resulting in a division in the hyperspace and the DT represented in Figure 3. At that point, the training process terminates. To perform predictions, simply sort (allocate) the new instances to their respective nodes and attribute them to the majority class of instances in that leaf.

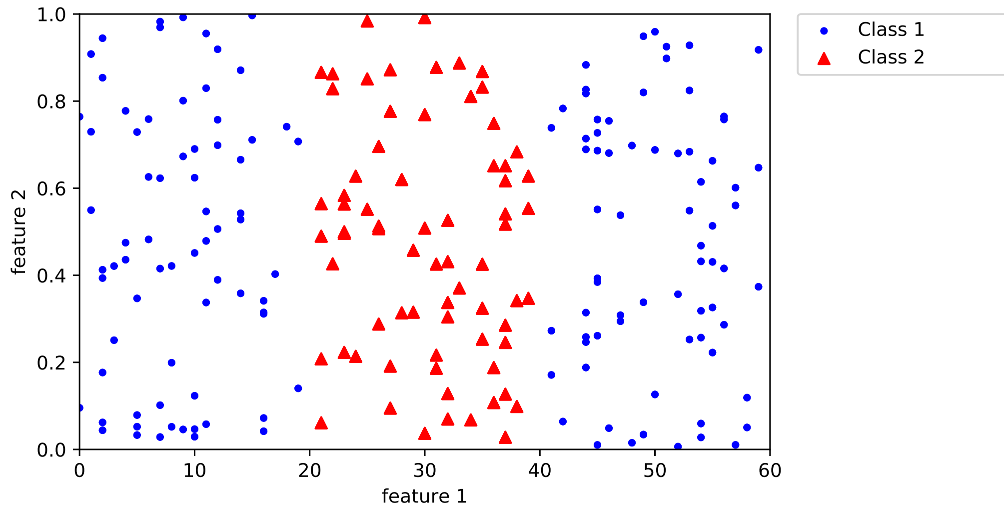


Figure 1 – Example of a binary classification problem.

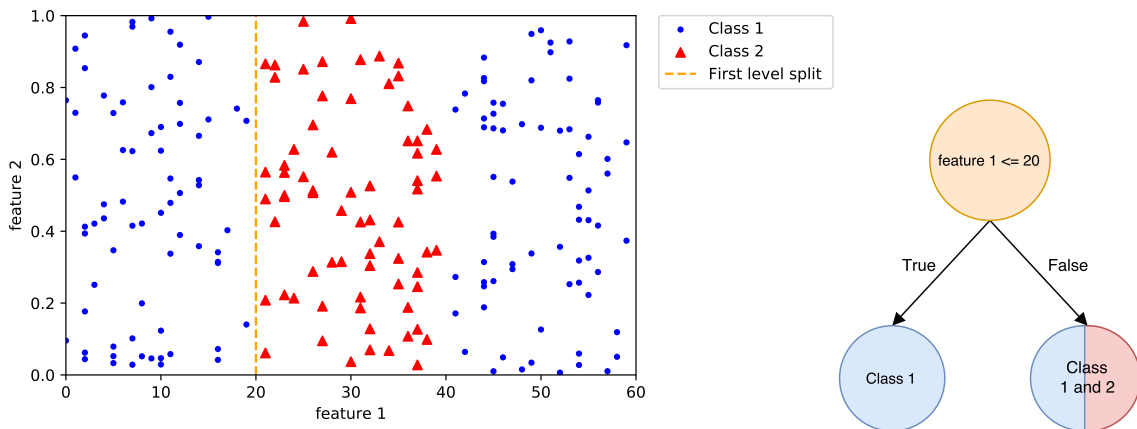


Figure 2 – Example of a first level split of a decision tree in a binary classification problem.

In this example, finding the best split points was a trivial task, but in reality, this is not. To do this, algorithms usually employ heuristic metrics to measure the "goodness of a split" [24, 28, 21, 23, 22]. Different metrics were proposed in the literature, with the majority of them agreeing on fundamental points. Splits which maintain a similar class proportion among all resulting subsets have no utility, while splits that result in subsets which contain only instances of one class have maximum utility [23].

2.1.2 Heuristic Metrics

The two most commonly used metrics [21, 23, 22] are the Information Gain (IG) and Gini Impurity (GI). To compute the IG of a given feature, a metric called entropy

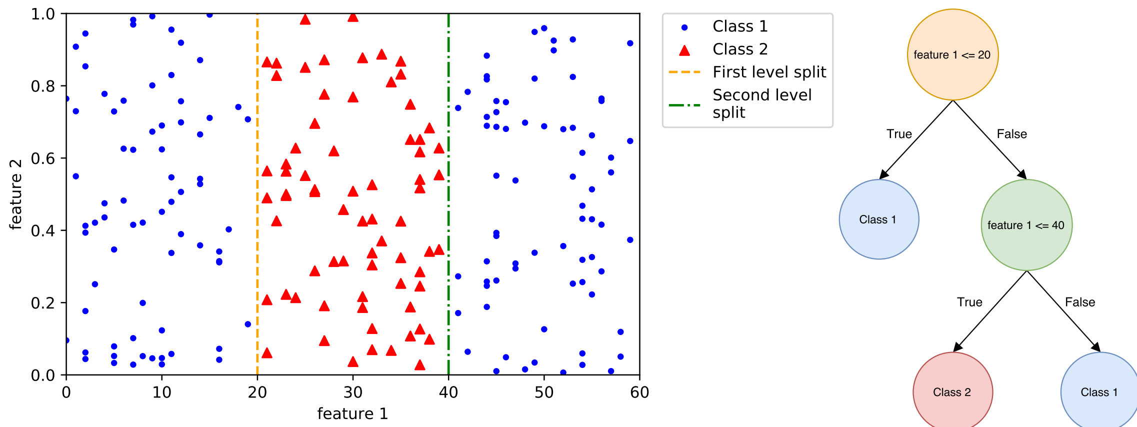


Figure 3 – Example of a first level split of a decision tree in a binary classification problem.

(H) needs to be computed first. Intuitively, in this scenario, H measures how difficult it is to be right when randomly guessing the class of instances present at that leaf. If all classes have a similar number of instances, this probability would be the lowest, yielding a higher H value. On the other hand, when classes distribution is highly imbalanced, H would be closer to 0. Given a leaf l, the H of that leaf is computed as:

$$H(l) = -\sum_{i=1}^{C} P_l(i)\,\log_2 P_l(i), \tag{2.1}$$

where C is all classes of a problem and $P_l(i)$ referes to the probability of that given class according to the instances observed at l.

After that, assume that a split feature f is being tested with a branch factor of two, where the first branch will contain all instances with a f value lower than or equal to a split point p and the other will contain the remainder instances (with a f value greater than p). Note that the split factor of branches for nominal features is usually equal to the number of possible values (one branch for each value) with no numeric split point. By defining the first branch as l_{le} and the second as l_{gt}, $H(l_{le})$ and $H(l_{gt})$ are computed. Then, let $\widetilde{H}(l,f,p)$ be the weighted average of $H(l_{le})$ and $H(l_{gt})$. Note that the weights of $H(l_{le})$ and $H(l_{gt})$ are equal to the number of instances in l_{le} and l_{gt} respectively. Finally, the IG of a split in the feature f at split point p on that leaf l is computed as:

$$IG(l,f,p) = H(l) - \widetilde{H}(l,f,p). \tag{2.2}$$

If the branch factor is larger than two, the H values of each branch are computed independently in the same manner and their weighted average is computed in $\widetilde{H}(l,f,p)$. The IG equation presented would remain the same.

On the other hand, the GI of a leaf l is given by the following equation:

$$GI(l) = 1 - \sum_{i=1}^{C} P_l(i)^2, \tag{2.3}$$

where $P_l(i)$ holds the same meaning as in the equation for H (Equation 2.1). Then, assuming the same feature f, split point p, and resulting branches l_{le} and l_{gt}, compute $GI(l_{le})$ and $GI(l_{gt})$ and compute their weighted average as $\widetilde{GI}(l, f, p)$. Then, the goodness of a split according to GI is computed as:

$$GI(l, f, p) = 1 - \widetilde{GI}(l, f, p), \tag{2.4}$$

Note that to find the best split point for numeric features, DTs usually test for all possible split points and select the one that yields the largest IG or GI. By applying any of those metrics, a DT algorithm selects the best features and their respective split points. Additionally, it also naturally performs feature selection since only the best features are used, according to a heuristic measure.

2.1.3 Pruning Techniques

Pruning techniques can also be applied to increase the generalisation capabilities of DTs. As defined in [20], it is a strategy to simplify a DT, improving its performance by removing error-prone components and further facilitating the analysis of the resulting model. Note that pruning should never eliminate predictive parts of a classifier, requiring a mechanism to determine where it should be performed. Pruning methods can be divided into pre-pruning and post-pruning. Pre-pruning consists of “pruning” in advance by suppressing the growth of a branch if it would not increase the predictive performance of the tree. Literally, no pruning occurs, since unnecessary nodes will never exist. On the other hand, post-pruning methods consist of taking a fully grown tree and cutting off superfluous branches that do not improve or even jeopardise predictive performance.

These techniques also help to combat overfitting. Overfitting occurs when a model is adjusted to the noise in the training dataset and is not capable of generalisation [21, 22]. In other words, the predictive performance in the training dataset is high but low in the testing dataset. When a model is *too complex*, a small dataset may lead to overfitting [21, 29]. However, this can be observed by monitoring the performance in the testing dataset [21, 22]. Pruning is used in DTs to avoid creating too complex models.

To illustrate one of the many possible scenarios where pruning is desired, consider the example presented in Figure 4. Possibly, the instance from class 2 next to the instances from class 1 is either an outlier or noise in the dataset. In this sense, letting the divide and conquer procedure execute until only one class of instances are present at the leaves would result in overfitting the data. Note that other ML algorithms may also suffer from overfitting and deal with it in different ways.

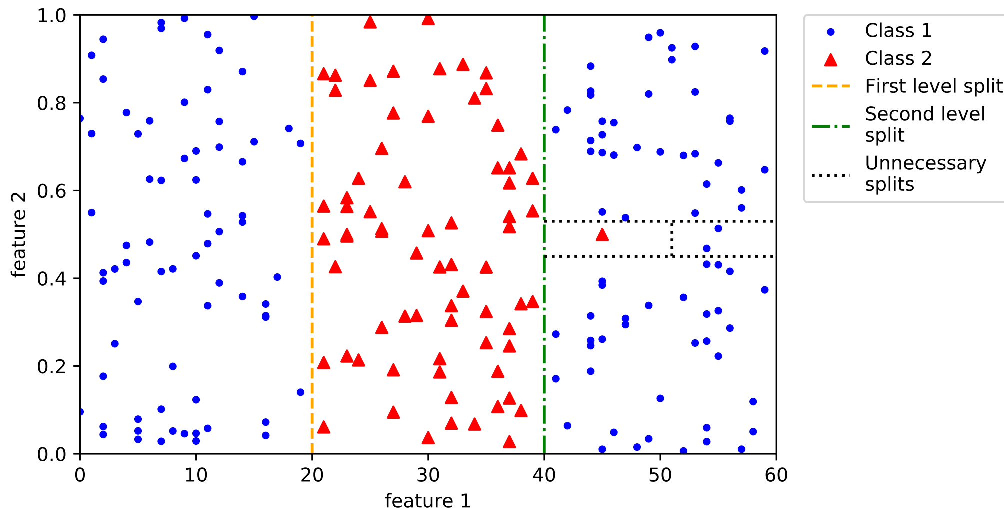


Figure 4 – Example of a unnecessary splits on a binary classification problem.

2.1.4 Naive Bayes

The Naive Bayes (NB) is a ML algorithm based on the Bayes's theorem that works by making the naive assumption that all features are independent, given the value of a class label.

It computes the frequency of occurrences of each feature for each class, and then combines all these frequencies with the frequency of the classes themselves, resulting in a set of probabilities [30, 11, 31]. The NB will classify an instance with the class value c which maximises the *a posterior* conditional probability given by:

$$P(c|\mathbf{x}) \propto P(c) \prod_{i=0}^{f} P(x_i|c), \tag{2.5}$$

where $\mathbf{x}$ is an instance, f is the total number of features in that instance, $P(c)$ is the probability of class c and $P(x_i|c)$ is the probability of occurrence of the value present in the i-th feature of $\mathbf{x}$ considering the class c. Note that numerical values are usually discretised.

2.2 Data Stream Mining

Data Stream Mining is a subarea of ML focused on creating and applying techniques from the broad area of ML to data streams. To better comprehend it, we need to first define data streams.

2.2.1 Data Streams

In traditional ML, data is usually collected and stored during an amount of time and then used to extract and generate knowledge. However, recently, data is being gener-

ated with high frequency and volume. These are the main characteristics of a data stream, i.e., data streams are composed by data generated continuously and indefinitely [1, 11].

As discussed in Krawcyzk et al. [1] and Gama et al. [11], data streams are potentially unbounded, meaning that they can be infinite, and instances are related in a time-ordered way. Additionally, the arrival time between instances is also variable.

The main characteristics of a data stream can be summarised as follows:

1. The size of the stream is huge and potentially infinite.
2. Instances are time ordered.
3. Instances arrive continuously, at high speed, and with different time intervals between each of them.
4. The streams are susceptible to change.
5. Labelling is very costly.

A data stream S is defined as $S = \{Z_1, Z_2, Z_3, ..., Z_n, ..., Z_\infty\}$. When considering a supervised scenario with one target variable, $Z = (\mathbf{x}, y)$, with $\mathbf{x} = [x_1, x_2, x_3, ..., x_f]$, where f is the number of features, and y as the target variable. If more than one target was present, then y would be a target vector. On the other hand, in unsupervised scenarios, $Z = \mathbf{x}$.

Due to the amount of data, storing them all is not possible. Additionally, usually, it is only possible to process each instance once. Even though algorithms like neural networks and NB are incremental by nature, this is rarely sufficient to deal with data streams, since the first needs many iterations to yield good predictive performance and the latter usually has low predictive performance. All data stream characteristics pose a need for specially tailored algorithms, that are capable of dealing with memory and time constraints while also being incremental with only one scan over each instance [1, 11].

Additionally to the limitations presented, the phenomena called concept drift (CD) may be present in data streams. CDs are usually defined with a probabilistic view. As defined in [1, 32], first consider that each instance arrives at a time t and is generated by a joint probability distribution $P^t(X, y)$, where $\mathbf{x}$ corresponds to the feature vector and y to the class label. If the same joint probability distribution generates all instances in the stream, then the concepts in data are stable, i.e., data is stationary. However, assume two distinct instances, arriving at times t and $t + z$, generated by $P^t(\mathbf{x}, y)$ and $P^{t+z}(\mathbf{x}, y)$. If $P^t(\mathbf{x}, y) \neq P^{t+z}(\mathbf{x}, y)$, then a CD occurred. Particularly, when a CD occurs, either one or both of the following change:

1. Prior probabilities of classes $P(y)$;

2. Class conditional probabilities $P(\mathbf{x}|y)$.

Nonetheless, a change in $P(y)$ or $P(\mathbf{x}|y)$ is not a guarantee of change in $P(y|\mathbf{x})$, defined as $P(y|\mathbf{x}) = \frac{P(y)\,P(\mathbf{x}|y)}{P(\mathbf{x})}$.

There are two main types of drifts: virtual or real drifts. Real drifts are defined as changes in $P(y|\mathbf{x})$, whereas a virtual drift occurs when changes happen in $P(y)$ or $P(\mathbf{x}|y)$ without affecting $P(y|\mathbf{x})$. CD is the data stream learning counterpart of dataset shift from batch learning [33]. Further, different types of CD exist, with the most common types [34, 35, 33, 1] presented in Figure 5.

1. Incremental drifts consist of a sequence of small and not severe changes, where instances slowly change their values over time.

2. Gradual drifts occur when a new concept begins to slowly take over the existing concept. Assume that C_1 and C_2 are the old and new concepts and that their probability of occurrence is $P(C_1)$ and $P(C_2)$. Then, prior to this drift, $P(C_1) = 1$ and $P(C_2) = 0$. During the drift, $P(C_1)$ would slowly decrease, while $P(C_2)$ would increase, reaching 1.

3. Sudden drifts occur when at some instance t $P(C_1)$ turns into 0 and $P(C_2)$ into 1.

4. Reoccurring drifts may happen in some domains in which concepts reappear after some time. Notice that this reoccurrence can be cyclic, meaning that two or more concepts reappear in a specific order, or not.

It is important to notice that a stream may present multiple CD types, separately or at the same time. Lastly, real CD usually concerns changes in all instances, but these changes may occur in a particular subspace of the hyperspace.

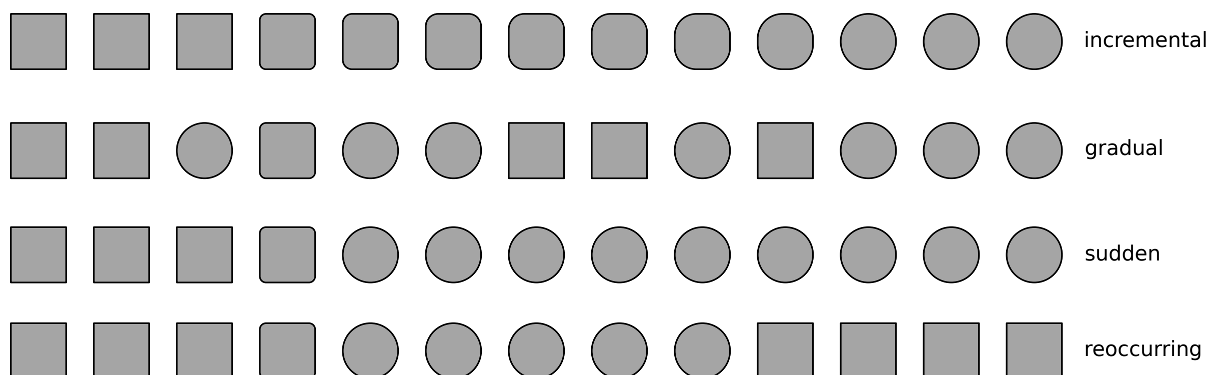


Figure 5 – Types of Concept Drifts. Image from [1].

Further, a data stream may still present noise and blips (which corresponds to rare events, or even outliers) [34, 1]. In this sense, an ideal data stream algorithm needs to be able to deal with all the aforementioned restrictions.

Like in traditional ML, many works that propose new algorithms deal with fully labelled data streams, meaning that the ground truth for each instance is available immediately or after a small time to the algorithm. However, for data streams, some researchers also consider three different scenarios [1]. In the first, labels come with a delay, meaning that the algorithm will only receive the ground truth for an instance after some time, e.g., after receiving q instances. One real-world example of this would be a financial fraud detection, where fraudulent transactions are informed with a long delay. In the second scenario, only portions of the stream are labelled. Lastly, a data stream can have only its initial instances labelled or even be fully unlabelled. The three last cases can be seen as an example of high labelling costs. In this work, following the approach taken by different authors [14, 36, 37, 38, 39, 40], only fully supervised scenarios were considered.

2.2.2 Very Fast Decision Tree

The Very Fast Decision Tree, proposed by Domingos and Hulten [14], is a tree-based algorithm for data streams designed around the principles of the Hoeffding Bound (HB) [41] theorem. The HB theorem, as applied in [42, 14], states the following. Suppose a continuous variable v, whose values are bounded by the interval $[v_{min}, v_{max}]$, with a range of values $R = v_{max} - v_{min}$. Additionally, presume that this variable was independently observed n times and the computed mean, according to these observations, is $\overline{v}$. Thus, the HB theorem states that this variable has an expected mean $\overline{v_{expected}}$ (when $n \to \infty$) bounded by the interval $[\overline{v} - \epsilon, \overline{v} + \epsilon]$ with statistical probability $1 - \delta$, where

$$\epsilon = \sqrt{\frac{R^2\, ln(\frac{1}{\delta})}{2\,n}}. \tag{2.6}$$

When training, the VFDT applies the HB to evaluate if a given leaf should be split. To do so, the VFDT first ranks the importance of all features considering the instances that had fallen into that leaf. To measure the importance of each feature f, consider a hypothetical function $G(.)$. In practice, two different functions are used as $G(.)$, the IG and the GI. Lastly, note that the HB function uses the range of values R. For the IG, $R = \log_2 C$, with C being the number of classes in that classification problem and for GI $R = 1.0$. Although, as stated by [43], applying the HB to metrics, such as IG or GI, is mathematically incorrect due to factors like their non-linearity, this has occurred in many previous works and, in practice, works well as a heuristic procedure. The authors also presented a mathematically correct bound to be used for the GI that can substitute the right part of Equation 2.6. Here, the HB is used as a heuristic procedure. Nonetheless, future adaptations can be made to employ the newly proposed bound.

The VFDT computes $\{G(f), \forall f; f \in F\}$, where F corresponds to all features in the problem, and sorts these values from highest to lowest. Then, consider that the features with highest and second highest values of $G(.)$ are $\mathbf{x}_b$ and $\mathbf{x}_{sb}$ and let $\Delta G = G(\mathbf{x}_b) - G(\mathbf{x}_{sb})$.

Using the HB, the lowest possible value of ΔG would be $\Delta G - \epsilon$. If $\Delta G > \epsilon$, where ϵ was computed at that given evaluation moment, then X_{b} would hold as the best feature if $n \to \infty$, with probability $1 - \delta$. This affirmation is easier to visualise if we give values for $G(\mathbf{x}_b)$, $G(\mathbf{x}_{sb})$ and ϵ. Suppose $G(\mathbf{x}_b) = 1.0$, $G(\mathbf{x}_{sb}) = 0.8$, $\Delta G = 0.2$, and $\epsilon = 0.2$. Then, ΔG would be at least 0 and the best feature would remain the best (since this difference is equal or greater than 0).

It is interesting to observe the decay behaviour of the HB function. Supposing that $R = \log_2 2$ and $\delta = 10^{-4}$, and n is varied, the resulting behaviour of the HB is presented in Figure 6. As it is possible to observe, the HB value quickly decreases with few instances seen. For example, the value almost halves from 100 to 300 instances and, in this sense, receiving a few more hundred instances can lead to satisfying the HB. Likewise, it has a behaviour similar to an exponential decay function, where the differences become smaller the bigger the n values.

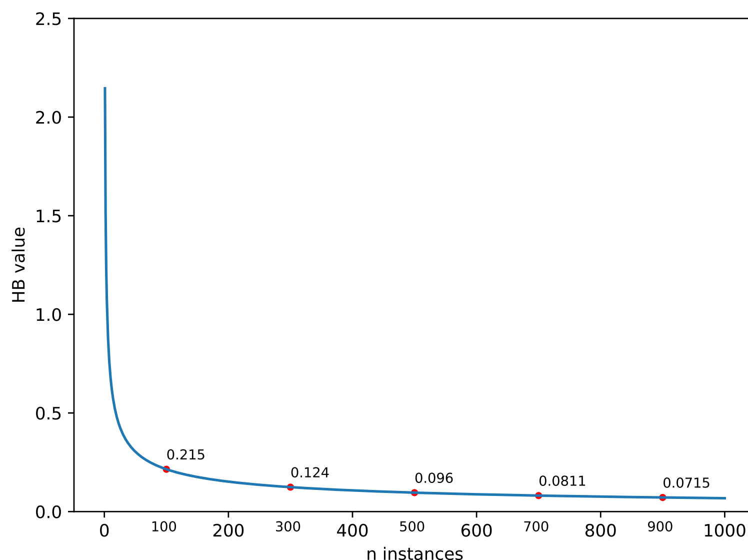


Figure 6 – Behaviour of the HB function with $R = \log_2 2$ and $\delta = 10^{-4}$.

To compute IG or GI, the VFDT needs to keep track of the relation $\mathbf{x} \to y$ per leaf. To handle nominal features, a simple counting of the number of occurrences of each value given a class is sufficient. Considering a nominal feature with three possible values in a problem with two classes, then the VFDT could have three vectors (one for each value) with two positions (one for each class). For each new instances falling into a leaf, the corresponding value vector in the position of that instance's class would be incremented. However, this is not possible for numeric features.

For numeric features, a naive method would be to store all instances seen and compute the probabilities using them, but that would impact in a memory cost of the same size of the stream, which is unfeasible. In this sense, many methods were proposed, such as discretising numeric features, using an exhaustive binary tree [44] or a Gaussian

numeric estimator, derived from [45]. For the first method, simply create a histogram with b bins where each bin corresponds to a range of possible values. For example, consider that the possible values for a feature are in the interval $[0, 10]$ and 10 bins are created. Then, all values from 0 to 1 would fall into the first bin, values from 1 to 2 in the second and so on. Each bin would have a class distribution vector associated with it, which is used to compute the IG or GI values. When evaluating a split in that feature, simply use as possible split points the values which divide bins and count the resulting class distributions. However, this method does not work well in practice and requires prior knowledge of the range of possible values.

The second method, the exhaustive binary tree, consists of adding each possible numeric value v to a binary search tree and storing the total number of instances of each given class for values smaller than v and larger than v. Suppose a binary classification problem and that the value 3.1 is added to the binary tree. Then, at the number 3.1 node, two vectors with two positions would be created. In one vector, the number of occurrences of class 1 or 2 for values smaller or equal to 3.1 would be stored and, in the other, the same would happen for values larger than 3.1. When evaluating splitting at that feature, simply go through the binary tree and compute the IG or GI according to each node. This method was mainly proposed to avoid sorting all possible numeric values to discover the resulting class distributions of each possible split point. In the worst case, memory costs stay the same as storing all instance seen.

The third method, the Gaussian numeric estimator, outperforms the other methods while maintaining low memory and time costs, being used as the default estimator in recent works [2]. The Gaussian numeric estimator first assumes that the values of the numeric features follow a normal distribution [2]. Then, it fits C normal distributions for each feature, with C being the number of classes in that problem. These normal distributions are created in an incremental way by updating the mean and standard deviation values of the features. To do so without introducing precision loss the following function is used to compute the mean [46, 47, 48]:

$$\overline{X}_n = \overline{X}_{n-1} + \frac{1}{n}\left(x_n - \overline{X}_{i-n}\right), \tag{2.7}$$

where n is the number of value seen and x_n n-th value. The standard deviation is updated according to [46, 47, 48]:

$$S_n = S_{n-1} + (x_n - \overline{X}_{n-1}) * (x_n - \overline{X}_n) \tag{2.8}$$

$$\sigma_n^2(X) = \frac{S_n}{n-1} \tag{2.9}$$

$$\sigma_n(X) = \sqrt{\sigma_n^2(X)}, \tag{2.10}$$

where S_n is an intermediate variable and $\sigma^2(X)_n$ is the variance of the sample. The Gaussian estimator then uses these normal distributions to compute the IG or GI of a given feature. This process is represented in Figure 7. At top of the image, there are two normal distributions (one for each class in this binary problem). Under the distributions, there are ten possible split points and the respective resulting class distributions if that leaf split at that point. On the bottom of the image, there are the IG values for all tested split points and, in this example, the chosen split point would be E, since it presents the higher IG.

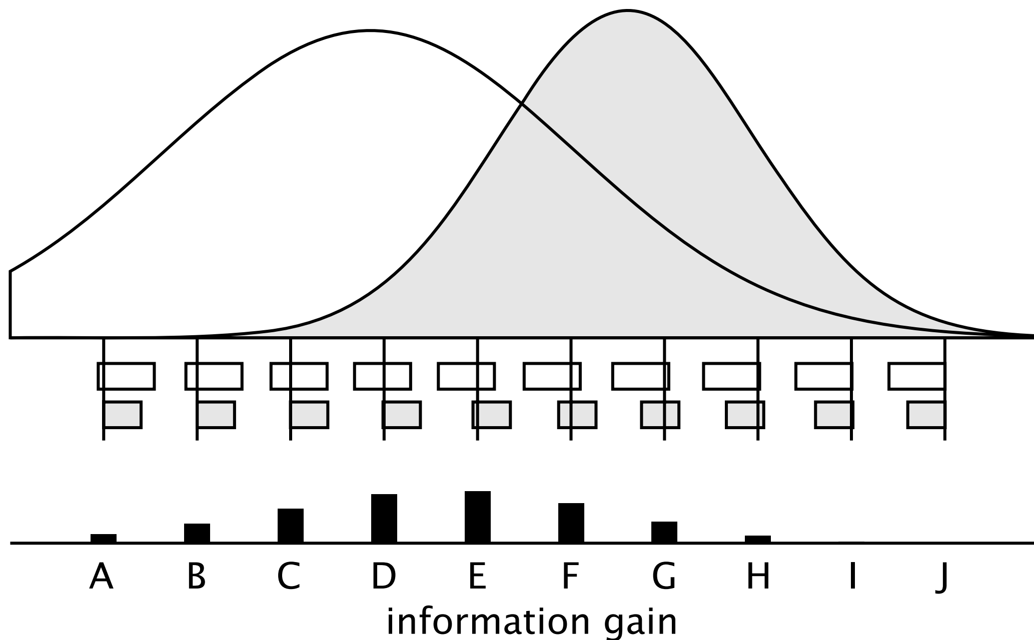


Figure 7 – Example of a Gaussian numeric estimator for a feature in a binary problem. Image adapted from [2].

Using the techniques presented before, the VFDT is able to learn from a single instance at a time with limited computational memory and time resources. Additionally, under realistic assumptions, it has the same asymptotic performance as a DT produced by a standard batch algorithm [11]. It is also worth highlighting that the VFDT, unlike batch DTs, is capable of performing predictions at any time.

To avoid unnecessarily checking the split condition, this procedure is only executed if the leaf has an impure class distribution, i.e., there is more than one class of instances that felt on that leaf. Likewise, with the same goal, this checking is performed in intervals of n instances that felt into that leaf [14]. This is referred by many authors as Grace Period (GP). Lastly, a tiebreak value τ was introduced to support tree growth when ΔG is very low. The VFDT checks if $\Delta G < \epsilon < \tau$ holds true and, if so, ignores the HB condition [14]. It is worth highlighting that a high value of τ may lead to tree size explosion and even completely ignoring the HB condition, e.g., when learning from a stream with two classes,

using IG, $n = 200$ and $\delta = 10^{-7}$, if $t \geq 0.201$ the HB condition will never be checked.

Instead of using a traditional most common (MC) approach, i.e., the majority class, prediction at the leaves, Gama et al. [44] introduced the idea of functional leaves to increase the predictive performance of the VFDT. These leaves use a NB or an Adaptive NB (ANB) algorithm to further increase predictive performance [44]. This means that, when a new instance is sorted to a leaf, one of the three strategies can be used (MC, NB or ANB). If these nodes use a MC prediction, which is usually employed in batch DT algorithms, then the class that appeared more times will be the prediction of that leaf. On the other hand, a leaf can use a NB prediction to compute the class probabilities of those instances. Lastly, when using a ANB prediction, the leaf will switch between performing a MC or a NB prediction, choosing the most accurate prediction at that given time in the leaf making the prediction.

To present a more general view of the algorithm, a diagram of the VFDT algorithm is provided in Figure 8. It abstracts the initialisation steps and feature selection of the algorithm and focuses more on how the tree building procedure works. First, instances flow from a data stream into the algorithm. Then, the next instance from the stream is selected, sorted to a leaf and used to update that leaf's statistics. After that, the algorithm checks if a split attempt should be made. If it should, the HB value is computed and the $G(.)$ values of the features are ranked. The leaf is split if possible, according to the HB or τ. This process is repeated until the stream ends or indefinitely.

The full pseudocode for training the VFDT is provided in Algorithms 1, 2, and 3. In Algorithm 1, the main function is presented. First, the tree is initialised with just the root. The root contains additional statistics used to perform the GP interval check and a set of features which are not candidates for a split at that given leaf. Then, each instance of the stream is sorted into its respective leaf (in the beginning all instances will fall into the root). This is done by travelling through the tree according to the split nodes. This instance is then added to that leaf and, if the class distribution is impure and there have been GP instances since that leaf last tried to split, the HB is computed and all features are ranked according to their $G(.)$ values. If that leaf validates the conditions according to the function **CanSplitVFDT**, it turns into a split node using the feature with the highest $G(.)$ value and its children nodes are initialised. Otherwise, the **FeatureSelection** function is executed. The **CanSplitVFDT** function is detailed in Algorithm 2. It works by checking the HB and the $G(.)$ values of the features according to the manner described before. Lastly, in Algorithm 3, the feature selection procedure is defined. This is done in the same way described in [14] and the idea behind it is to remove all features that validated the HB when compared with the highest $G(.)$ value. Recall that the HB is used in the VFDT to check if the difference between the highest and second highest $G(.)$ values differ enough. In this sense, when applied to the remaining features, it is possible to see

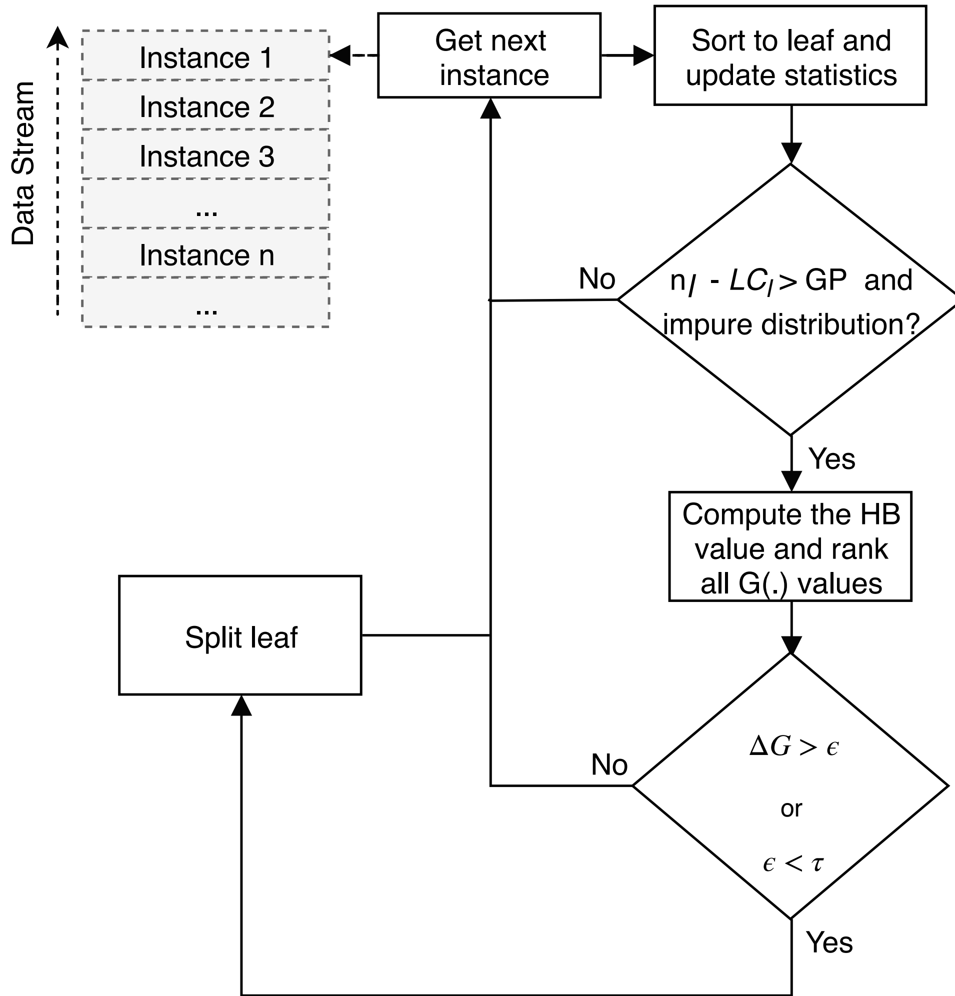


Figure 8 – Diagram of the VFDT.

if they will not become the best feature even if the VFDT observes more instances.

2.2.3 Hyperparamaters

Lastly, a more detailed discussion about the hyperparameters and their effects is provided.

1. GP: when increasing it, the algorithm will perform less split tests, which will result in smaller trees, using less memory and performing fewer computations, decreasing time costs. On the other hand, smaller GP values will make the tree adapt faster to the data stream. This results in higher predictive performance, at the cost of time, since it performs more computations, and memory, given the fact that the resulting tree will likely be larger.

2. τ: when increasing it, at each split attempt, the tree will need to see fewer instances to split (in the extreme case, when using very large values of τ, a split will occur whenever an attempt to split happens). This may increase predictive performance,

but if we consider the extreme case and its interaction with a very low GP, the produced tree can even overfit the data stream and perform poorly. Additionally, memory and time costs are greatly increased. On the other hand, very low τ values may hold tree growth, resulting in a small, low predictive performance tree.

3. δ: it is used for the HB computation. When δ values are larger, ϵ will be smaller, having the same resulting effect as increasing τ, whereas a small δ results in larger ϵ values and produces the same effect as decreasing τ.

4. $G(.)$: the heuristic measure used to decide the best feature. Both IG and GI can be used, yielding very similar results.

5. Drop poor features: an option whether or not to ignore poor features using the HB. This is used by default to minimise computational resources. Varying it may result in a slightly better predictive performance since a different feature may be chosen. Nonetheless, the trade-off is not favourable.

Algorithm 1 The VFDT training algorithm.

Input:

S: the stream of instances
GP: the grace period
δ: the error probability
τ: the tiebreak value

Output:

$VFDT$: a trained Very Fast Decision Tree

```
1: procedure VFDT(S, GP, δ, τ)
2:     Let VFDT ← l_root                                          ▷ The root
3:     Let n_{l_root} ← 0                          ▷ Number of elements seen at l_root
4:     Let LC_{l_root} ← 0               ▷ Number of elements on last split check at l_root
5:     Let F_{l_root} ← ∅                          ▷ Set of features removed from comparison
6:     for ((X, y) in S) do        ▷ X is the feature vector of an instance of class y in S
7:         Sort (X, y) to its leaf l
8:         Let n_l ← n_l + 1
9:         Update feature estimators and class distribution at l according to (X, y)
10:        if (class distribution at l is impure ∧ n_l − LC_l > GP) then
11:            Compute ϵ and G(.) of features in l ∉ F_l
12:            Let rank ← Sorted G(.) computed
13:            if (CanSplitVFDT(rank, ϵ, τ)) then
14:                Replace leaf l with a split node
15:                for each branch of the split do
16:                    Let l_new ← new leaf
17:                    Initiate all the feature estimators on l_new
18:                    Let class distribution on l_new ← post-split distribution of l_new
19:                    Let n_{l_new} ← sum of class distribution on l_new
20:                    Let LC_{l_new} ← n_{l_new}
21:                    Let F_{l_new} ← ∅
22:                end for
23:            else
24:                Let LC_l ← n_l
25:                FeatureSelection(rank, ϵ, F_l)
26:            end if
27:        end if
28:    end for
29:    return SVFDT
30: end procedure
```

Algorithm 2 The split check algorithm for VFDT.

Input:
$rank$: rank of $G(.)$ per feature
ϵ: the Hoeffding's Bound value
τ: tiebreak value
Output:
Boolean value

1: **procedure** CanSplitVFDT($rank, \epsilon, \tau$)
2: Let G_{best} and $G_{second_best} \leftarrow$ the highest and second highest $G(.)$ values
3: **if** ($G_{best} - G_{second_best} > \epsilon \vee \epsilon < \tau$) **then**
4: **return True**
5: **end if**
6: **return False**
7: **end procedure**

Algorithm 3 The feature selection algorithm.

Input:
$rank$: rank of $G(.)$ per attribute
ϵ: the Hoeffding's Bound value
F: features removed from comparison

Output:
Nothing

1: **procedure** FeatureSelection($rank, \epsilon, F$)
2: Let G_{best} be the highest $G(.)$ value
3: **for** feature in $rank$ **do**
4: Let G_{feature} be the $G(.)$ of the current feature
5: **if** ($G_{\text{best}} - G_{\text{feature}} > \epsilon$) **then**
6: Add the current and the rest of features to F
7: **Break**
8: **end if**
9: **end for**
10: **end procedure**

3 RELATED WORK

Several works proposed modifications to the VFDT algorithm. Here, they are divided into two groups: those which directly change the original VFDT algorithm and those that use the fundamental ideas of the VFDT to propose new algorithms.

3.1 Direct modifications to the VFDT algorithm

The Genuine Tie Detection (VFDT-GTD) [15] uses a simple mechanism to automatically choose τ during training. The authors assume that in a genuine tie situation the difference between the best candidate features is much smaller than the difference between the second best and worst candidate features. In this sense, assuming the same notation previously used for the VFDT algorithm, the VFDT-GTD replaces the normal tie breaking procedure by checking if $\alpha\,(G(X_b) - G(X_{sb})) < G(X_{sb}) - G(X_w)$ holds true, with X_w being the feature with the lowest value of $G(.)$ and α a hyperparameter to control the number of times smaller this difference needs to be. Despite the VFDT simplification by removing the τ hyperparameter, an α hyperparameter was added. Furthermore, predictive performance decreased in all datasets used in the experiments. In the same work, the authors also proposed another method which *increases waiting time* for ties to be broken. For simplicity, when referred here, this method will be referenced as VFDT-IWT. It takes a different approach to reduce the number of ties that occur when they are being broken too often. It works by requiring more instances to break a tie at each subsequent time a tie is broken. To understand it, recall Figure 6. Supposing that we chose a $\tau = 0.0716$, at 900 instances $\epsilon < \tau$ would hold true and a split would be performed. The method proposed to add more p instances for a tiebreak to happen in a child of this split node. In this way, even if $\epsilon < \tau$ holds true at 900 instances in one of the children of this node, this child would have to wait more p instances for a split to happen. If only tiebreaks are performed, these p values are stacked along the children of these split nodes. However, when a split without tiebreak happens, the additional instances p is reset to the initial value (0 in this example). Unlike the VFDT-GTD, this presented a somewhat similar predictive performance as the original VFDT. For both algorithms, memory and time costs were not explored.

Yang et al. [16] proposed the Moderated-VFDT (MVFDT) to deal mainly with noisy scenarios. They observed that the fluctuation of HB values intensifies with nosier streams. The authors argue that if they can manage to reduce this fluctuation they would be able to increase predictive performance. For this, they implemented a different splitting process by substituting the τ hyperparameter with a dynamic τ. The mechanism is very

simple and works by storing the mean HB values of each leaf. A split is only performed if either $\Delta G > \epsilon$ (normal verification for the HB) or $\Delta G \geq \overline{\epsilon}_l$, where $\overline{\epsilon}_l$ is the mean HB value of a leaf l, holds true. They also presented two pruning mechanisms, one stricter and one looser. They compared their four versions against the VFDT, using a τ value that ignores the HB in most cases, in five datasets. There was only an increase in accuracy for one dataset, and although the authors argue that there was a decrease in memory consumption, this is directly linked to using a very high τ value in the VFDT during their experiments that effectively ignores the HB.

Yang et al. [17] proposed the Optimised-VFDT (OVFDT), where the goal was to increase accuracy while avoiding tree size explosion, substituting τ by a dynamic threshold value according to statistics about the HB and the leaf's current accuracy. By doing so, they also removed a hyperparameter. To compute this dynamic threshold value, the OVFDT stores the mean value of all ϵ computed ($\overline{\epsilon_l}$) and the number of truly and falsely predicted (T_l and F_l) instances per leaf l. Then, at a split attempt, if the HB does not hold true, but either:

1. $\Delta G < \overline{\epsilon_{max}}$ and the accuracy of that leaf decreased since the last check or
2. $\Delta G \geq \overline{\epsilon_{min}}$ and $T_l - F_l < 0$

holds true, where $\overline{\epsilon_{min}}$ and $\overline{\epsilon_{max}}$ is the lowest and highest mean values of ϵ across all leaves, that leaf is split. OVFDT was compared with three algorithms: VFDT (with multiple τ values); Genuine Tie Detection; and Hoeffding Option Tree (HOT) [37]. When compared with VFDT with $\tau = 0.05$ (VFDT-0.05), OVFDT obtained a small accuracy improvement (3%) at the cost of creating a tree, on average, 2.4 times larger. Additionally, it must be highlighted that none of the compared algorithms focuses on reducing tree size.

The Enhanced VFDT (EVFDT), proposed by [49], implements a dynamic tiebreak based on previous statistics from the HB and an additional pruning mechanism. For the dynamic tiebreak T, it stores a list of sorted ϵ values. Then, at each split attempt, it computes the mean difference between ϵ values in this list given by:

$$T = \frac{\sum_{i=1}^{m-1} \epsilon_{i+1} - \epsilon_i}{m}, \tag{3.1}$$

where m is the number of ϵ values in the list. After that, considering that the position of this new ϵ value in the list is n, the EVFDT computes v using the following equation:

$$v = \frac{\epsilon_{n+1} - \epsilon_{n-1}}{2}. \tag{3.2}$$

If $v \geq T$ than this new ϵ value is added to the list. T is used in the same manner as τ in the VFDT. The pruning mechanism works by removing leaves where less than a percentage of the last GP instances fall in them. Although the algorithm presented interesting results

regarding accuracy, tree size remained the same as the VFDT. Furthermore, no benchmark datasets were employed and only one self-generated dataset was used.

Losing et al. [50] presented a modification of the VFDT to predict the number of instances needed by a given leaf to satisfy the HB or a tiebreak, reducing computational costs. Then, they only performed a split attempt when that number of instances arrived at a given leaf. By doing so, they achieved speedups of 2.64 and 1.42 times on two GP values (50 and 200) while having relatively the same predictive performance. However, memory costs were not evaluated, but were likely the same as the VFDT, as the modifications only reduce the number of split attempts performed and not the real number of splits.

3.2 New algorithms based on the VFDT

The Concept-adapting VFDT (CVFDT) algorithm [51], keeps secondary trees in memory and constantly assess them to check if they have a higher predictive performance than the original tree. Likewise, the CVFDT also uses a sliding window to discard old instances. Their experiments revolved around a single synthetic dataset with CD and a variable the number of features. Although the error rate decreased by around 20%, time costs increase by 5.7 times. In this single dataset, the memory used by the CVFDT is lower than the VFDT, but in the absence of CDs and in other datasets, the additional memory costs to store secondary trees makes CVFDT less efficient than VFDT-0.05, as shown in [17]. Lastly, in CD scenarios, CVFDT predictive performance is much lower than those of ensemble-based solutions [1].

The Hoeffding Option Tree (HOT) [37] introduces the concept of option nodes instead of normal split nodes. An option node is essentially a split node which tests for multiple conditions at the same time. When a new instance arrives at an option node, it travels along all children nodes where the conditions are true. During training, at each option node that an instance travels through, if that option node can still support more options (given a hyperparameter *maxOptions*), the $G(.)$ value of all features used for the splits are recomputed. Likewise, the $G(.)$ values of all unused features are also computed. If $G(b) - G(b_{used}) > \epsilon$, where $G(b)$ and $G(b_{used})$ are the $G(.)$ values of the best unused and the best used feature, a new split option is added to that node. If the number of existing splits is greater than *maxOptions*, the estimation statistics in that split node can be cleaned. When reaching a leaf, the split occurs in the same way as in the VFDT. The HOT performs a prediction by averaging the prediction probabilities given by all leaves that the instance felt into. This algorithm presented predictive performance higher than the VFDT (around 1.2%), at the cost of significant memory increase (consumed 3.5 times more memory on average). Additionally, cleaning statistics in split nodes takes longer than in the VFDT, and a poorly parameterised HOT can even hold statistics indefinitely. This also impacts on the HOT being 2 times slower than the VFDT. It is

also possible to categorise this algorithm as an ensemble since it practically uses multiple trees to perform each prediction. However, considering only predictive performance, other ensemble solutions outperform this algorithm [1].

These previous modifications to VFDT mostly aimed at providing better predictive performance at the cost of increasing memory and/or processing time. The SVFDT is focused on greatly reducing memory costs of the VFDT while maintaining a competitive predictive performance. Table 2 presents a summary of the related work.

Table 1 – Summary of related work.

Authors	Algorithm	Main aspects	Findings
Homes et al. [15]	VFDT-GTD	Substitutes τ with a dynamic value based on the G values of all features.	Decreased predictive performance in all datasets evaluated. Memory and time costs were not explored.
	VFDT-IWT	Makes children of nodes which split due to tie breaking wait for more instances before allowing another tie break. Subsequent tie breaks will increase the awaiting time.	Similar predictive performance in all datasets evaluated. Memory and time costs were not explored.
Yang et al. [16]	MVFDT	Substitutes τ with the mean of HB values computed on that leaf and adds two pruning strategies.	Increased accuracy in only one dataset.
Yang et al. [17]	OVFDT	Substitutes τ by a dynamic threshold value using the HB and the leaf's current accuracy	3% increase in accuracy at the cost of producing trees 2.4 times larger.
Latif et al. [49]	EVFDT	Substitutes τ by a dynamic threshold using statistics about the HB values. Also presents a pruning mechanism that removes leaves with a low number of recent instances falling into them.	Good accuracy results but no impact in memory costs. No benchmark dataset was used.
Losing et al. [50]	-	Modifies the VFDT to predict when a split will occur, reducing the number of split attempts performed.	Speedups of 2.64 and 1.42 times with the same accuracy. Memory was not evaluated but likely stayed the same.
Hulten et al. [51]	CVFDT	Keeps secondary trees in memory, discarding the original tree in the presence of a concept drift.	Increases accuracy at the cost of being 5.7 times slower. In other datasets, greatly increases memory size. This algorithm is outperformed by ensembles.
Pfahringer et al. [37]	HOT	Introduces the idea of options nodes, making an instance travel through multiple paths.Predictions are done by combining responses from all leaves reached by that instance.	Increases accuracy by 1.2% on average at the cost of consuming 3.5 times more memory and being 2 times slower.

4 STRICT VERY FAST DECISION TREE

This chapter describes the proposed algorithm, the Strict Very Fast Decision Tree. The SVFDT modifies the VFDT by controlling tree growth without degrading predictive performance. Note that the strategy proposed here is a pre-pruning mechanism. Two versions of the algorithm, SVFDT-I and SVFDT-II, are proposed, both of which are guided by the following assumptions:

1. A leaf node should split only if there is a minimum uncertainty of class assumption associated with the instances, according to previous and current statistics. Consider a heuristic metric which measures class assumption uncertainty m (e.g., H or GI), then splits will occur on leaves which have a *high* m value (high is set according to the m values observed throughout growth).

2. All leaf nodes should observe a similar number of instances to be turned into split nodes. Leaves which are not receiving many instances will likely not contribute too much to increase predictive performance.

3. The feature used for splitting should have a minimum relevance according to previous statistics. In some cases, m will be high and many elements will be falling into a leaf, nonetheless, a split should not occur if the uncertainty in class assumption is not *significantly* reduced. Note that significant is considered according to other observed reductions in class assumption uncertainty occasioned by splits.

In more details, for the first assumption, consider that, at a given leaf, there are 50 instances of class 0 and 50 of class 1. Here, uncertainty is the highest. For DT, class uncertainty is usually given by H or GI. The SVFDTs will hold back splits in leaves which have less uncertainty than what has been observed in other parts of the hyperplane. Note that this assumption does not stop growth in leaves which have higher uncertainty than what has been observed. The second assumption holds back growth in leaves which only a small amount of instances are falling into. By focusing on splitting leaves with more instances, the SVFDT can better use memory resources to increase its predictive performance. Additionally, it also makes the tree grow more evenly. This occurs since leaves wait for a similar number of instances before splitting. By using these two assumptions, the SVFDT holds growth where it is not needed (class uncertainty is low or only a small amount of instances fall into that leaf), but sometimes leaves may need to split but all available possible splits points will not be likely to help in increasing predictive performance. For these cases, the last assumption holds back growth in regions where class uncertainty would not decrease. To do so, the SVFDTs, using measures such as IG or GI,

only allow splits where this gain is at least as high as gains observed in other parts of the tree throughout training.

These assumptions are employed using a φ function defined as:

$$\varphi(x, X) = \begin{cases} \text{True}, & \text{if } x \geq \overline{X} - \sigma(X) \\ \text{False}, & \text{otherwise} \end{cases} \tag{4.1}$$

Where X is a set of observed values, $\overline{X}$ is their mean, $\sigma(X)$ is their standard deviation, and x is a new observation.

A leaf can satisfy the VFDT split conditions (according to the HB or τ value) and, when this happens, statistics corresponding to these occurrences are marked with an underscored *satisfiedVFDT*. These statistics are the ones used by the φ functions. At each leaf l, the following constraints are employed every time there is a split attempt to enforce assumptions 1, 2 and 3:

1. $\varphi(Imp_l, \{Imp_{l_0}, Imp_{l_1}, ..., Imp_{l_L}\})$, where the former parameter is the current impurity (H or GI) of l and the latter is a set of all impurities of all current leaves L in the tree, including l (Assumption 1);

2. $\varphi(Imp_l, \{Imp_{satisfiedVFDT_0}, Imp_{satisfiedVFDT_1}, ..., Imp_{satisfiedVFDT_S}\})$, where the latter parameter corresponds to the impurities computed at all S times a leaf satisfied the VFDT split conditions (Assumption 1).

3. $\varphi(G_l, \{G_{satisfiedVFDT_0}, G_{satisfiedVFDT_1}, ..., G_{satisfiedVFDT_S}\})$, where G_l is the $G(.)$ value of the best split feature at l and the latter parameter is a set of the $G(.)$ values computed all S times a leaf satisfied the VFDT split conditions (Assumption 3).

4. $n_l \geq \overline{\{n_{satisfiedVFDT_0}, n_{satisfiedVFDT_1}, ..., n_{satisfiedVFDT_S}\}}$, where the former parameter corresponds to the number of elements seen at l and the latter to the average number of elements observed at all S times a leaf satisfied the VFDT split conditions (Assumption 2).

The φ function was not employed in the last constraint since it is always possible to satisfy it by waiting for more instances to be assigned to that leaf. On the contrary, the other constraints are not so easily satisfied in the same way, which may cause deadlocks that even learning a large number of instances would not resolve. For example, consider a scenario where the impurity of leaves are very high at the beginning of training but then greatly decrease. The average value for this impurity would be very high and would block many split attempts until this average decreases greatly, which would take many instances. Adding the standard deviation term allows the function to address this scenario.

Additionally to the φ function, SVFDT-II has a skipping mechanism to speed-up growing by ignoring all previously presented constraints using the following function:

$$\varpi(x, X) = \begin{cases} \text{True}, & \text{if } x \geq \overline{X} + \sigma(X) \\ \text{False}, & \text{otherwise} \end{cases} \tag{4.2}$$

At a split attempt, if either

1. $\varpi(Imp_l, \{Imp_{satisfiedVFDT_0}, Imp_{satisfiedVFDT_1}, ..., Imp_{satisfiedVFDT_S}\})$ (skip mechanism 1) or

2. $\varpi(G_l, \{G_{satisfiedVFDT_0}, G_{satisfiedVFDT_1}, ..., G_{satisfiedVFDT_S}\})$ (skip mechanism 2)

holds true, then all the other constraints are ignored. In this way, the SVFDT-II ignores all constraints to hold back growth when class uncertainty or the reduction in class uncertainty occasioned by a split is too high.

Note that both φ and ϖ functions rely on the mean and standard deviation values of each metric, which are computed using Equations 2.7, 2.8, 2.9 and 2.10 in Section 2.

Figure 9 uses a flow chart to illustrate how the VFDT was modified to create the SVFDT-I and SVFDT-II (parts highlighted in blue indicate modifications). When a split can be made, according to the VFDT, the SVFDT computes the necessary values for the φ and ϖ functions. After that, it updates all statistics using the values previously computed. If the SVFDT-II is being used, it checks if one of the ϖ functions hold true and split if one does. Otherwise, it checks if all φ functions hold true, in a last effort to split this leaf.

Algorithm 4 shows the pseudocode for training the SVFDT (both versions are contained). The majority of the algorithm remained unchanged from the traditional VFDT algorithm. $Imp_{\text{statistics}}$, $G_{\text{statistics}}$, $n_{\text{statistics}}$ and LH correspond to the additional statistics that are used to validate $\varphi(.)$ and $\varpi(.)$ operations. Algorithm 5 implements the function that checks whether a given leaf should be split. In addition to the VFDT split check, variables ϱ, ξ, κ and ψ were added to denote constraints 1, 2, 3, and 4, respectively. It is worth reiterating that all statistics are updated when the VFDT split conditions are satisfied. The procedure of feature selection, invoked in line 29 of Algorithm 4, remained the same as in Algorithm 3.

4.1 Memory and Time Complexities

The memory costs added to the VFDT to compute the constraints 2, 3 and 4 are $O(1)$. Complementary, the memory cost of Constraint 1 is $O(L_{max})$, with L_{max} being the maximum number of leaves observed during the tree induction.

Regarding time complexity, the first constraint has a cost of $O(L_{max})$, while the others have $O(1)$ complexity. These costs corresponds to a single operation and so, the time complexity added to the whole induction process are $O(t_{satisfiedVFDT} * L_{max})$ and $O(t_{satisfiedVFDT})$, respectively, where $t_{satisfiedVFDT}$ is the number of times a leaf satisfied the VFDT split conditions. For SVFDT-II, there is an additional time cost of $O(t_{satisfiedVFDT})$ for each mechanism. Although there are these additional time costs, due to the fact that tree size is significantly reduced, the SVFDTs train faster or as fast as the VFDT.

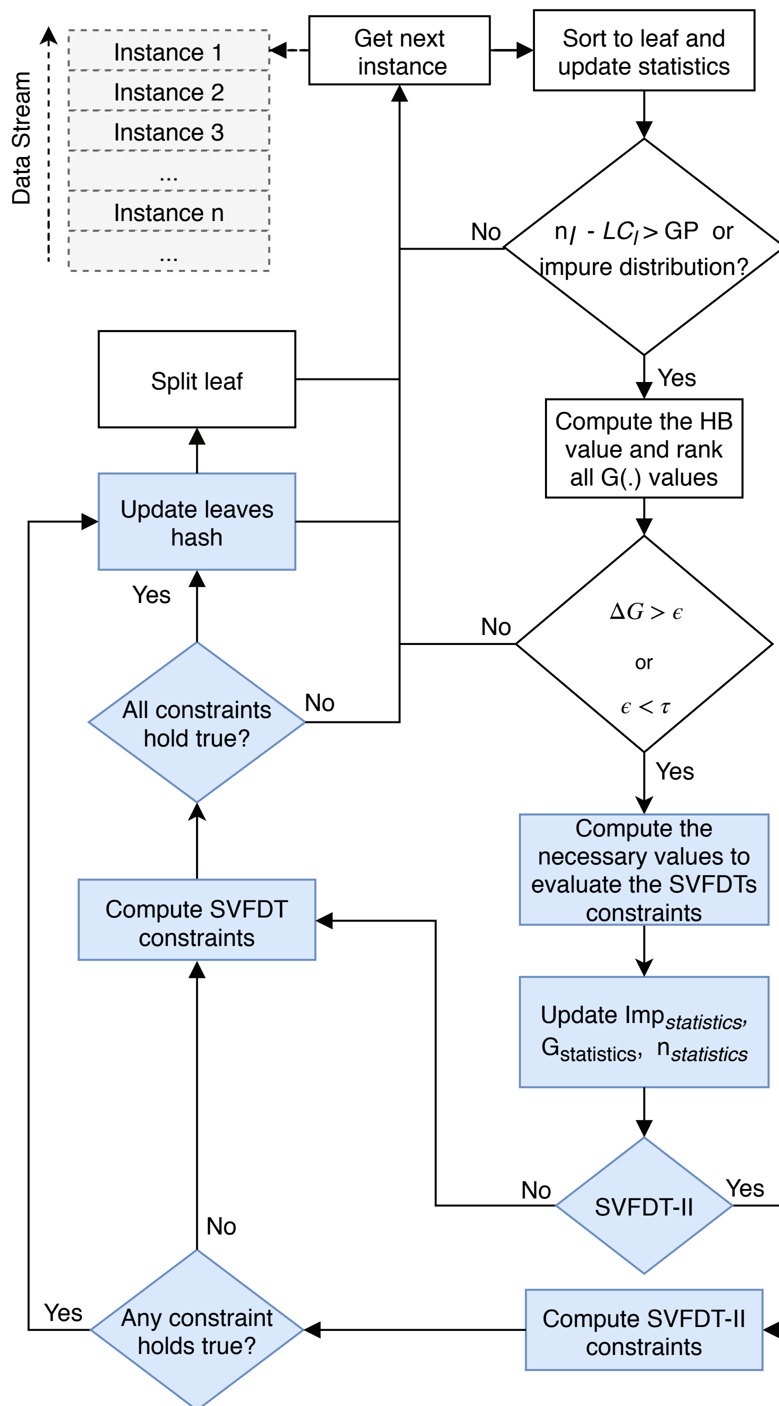


Figure 9 – Diagram of the SVFDT. Image created by the author.

Algorithm 4 The SVFDT algorithm.

Input:
S: the stream of instances
GP: the grace period
δ: the error probability
τ: the tiebreak value
Output:
$SVFDT$: a trained Strict Very Fast Decision Tree

1: **procedure** SVFDT(S, GP, δ, τ)
2: Let SVFDT $\leftarrow l_{root}$ ▷ The root
3: Initiate H_{stats}, G_{stats} and n_{stats} for φ and ϖ equations
4: Let LH be the hash of leaves
5: Let $n_{l_{root}} \leftarrow 0$ ▷ Number of elements seen at l_{root}
6: Let $LC_{l_{root}} \leftarrow 0$ ▷ Number of elements on last split check at l_{root}
7: Let $F_{l_{root}} \leftarrow \emptyset$ ▷ Set of features removed from comparison
8: **for** $((X, y)$ in $S)$ **do** ▷ X is the feature vector of an instance of class y in S
9: Sort (X, y) to its leaf l
10: Let $n_l \leftarrow n_l + 1$
11: Update feature estimators and class distribution at l according to (X, y)
12: **if** (class distribution at l is impure $\wedge$ $n_l - LC_l > GP$) **then**
13: Compute ϵ and $G(.)$ of features in $l \notin F_l$
14: Let $rank \leftarrow$ Sorted $G(.)$ values
15: **if** (***CanSplitSVFDT***$(rank, \epsilon, \tau, l, LH, H_{stats}, G_{stats}, n_{stats})$) **then**
16: Remove leaf l from LH
17: Replace leaf l with a split node
18: **for** each branch of the split **do**
19: Let $l_{new} \leftarrow$ new leaf
20: Initiate all the feature estimators on l_{new}
21: Let class distribution on $l_{new} \leftarrow$ post-split distribution of l_{new}
22: Let $n_{l_{new}} \leftarrow$ sum of class distribution on l_{new}
23: Let $LC_{l_{new}} \leftarrow n_{l_{new}}$
24: Let $F_{l_{new}} \leftarrow \emptyset$
25: Add leaf l_{new} to LH
26: **end for**
27: **else**
28: Let $LC_l \leftarrow n_l$
29: ***DropFeatures***$(rank, \epsilon, F_l)$
30: **end if**
31: **end if**
32: **end for**
33: **return** SVFDT
34: **end procedure**

Algorithm 5 The split check algorithm for SVFDT.

Input:

$rank$: rank of $G(.)$ per feature
ϵ: the Hoeffding's Bound value
τ: tiebreak value
l: the current leaf node
LH: the hash of leaves
Imp_{stats}: statistics about impurity values
G_{stats}: statistics about G(.) values
n_{stats}: statistics about the number of elements seen values

Output:

Boolean value

1: **procedure** CanSplitSVFDT($rank, \epsilon, \tau, l, LH, Imp_{stats}, G_{stats}, n_{stats}$)
2: Let G_{best} and G_{second_best} ← the highest and second highest $G(.)$ values
3: **if** ($G_{best} - G_{second_best} > \epsilon \vee \epsilon < \tau$) **then**
4: Compute $\overline{Imp}_{LH}$ and $\sigma(Imp_{LH})$ using LH
5: Compute $\overline{Imp}$ and $\sigma(Imp)$ using Imp_{stats}
6: Compute $\overline{G}$ and $\sigma(G)$ using G_{stats}
7: Compute $\overline{n}$ using n_{stats}
8: Let Imp_l and n_l ← impurity and number of elements seen at l
9: Update Imp_{stats}, G_{stats} and n_{stats} with Imp_l, G_{best} and n_l, respectively
10: Let $svfdt_ii_constraints \leftarrow Imp_l \geq \overline{Imp} + \sigma(Imp) \wedge G_{best} \geq \overline{G} + \sigma(G)$
11: **if** ($svfdt_ii_constraints$) **then** ▷ SVFDT-II version only
12: **return True**
13: **end if**
14: Let $\varrho \leftarrow Imp_l \geq \overline{Imp}_{LH} - \sigma(Imp_{LH})$ ▷ Constraint 1
15: Let $\xi \leftarrow Imp_l \geq \overline{Imp} - \sigma(Imp)$ ▷ Constraint 2
16: Let $\kappa \leftarrow G_{best} \geq \overline{G} - \sigma(G)$ ▷ Constraint 3
17: Let $\psi \leftarrow n_l \geq \overline{n}$ ▷ Constraint 4
18: Let $svfdt_constraints \leftarrow \varrho \wedge \xi \wedge \kappa \wedge \psi$
19: **if** ($svfdt_constraints$) **then**
20: **return True**
21: **end if**
22: **end if**
23: **return False**
24: **end procedure**

5 MATERIALS AND METHODS

This chapter presents the datasets used in the experiments, the metrics employed to evaluate the different aspects of the algorithms, and details about the experimental setup.

5.1 Datasets

A total of 26 public datasets were selected to compare the performance of both versions of the SVFDT (SVFDT-I and SVFDT-II) and the VFDT:

- **Agrawal dataset** [52, 13]: generates data according to classification functions using six numeric (salary, commission, age, hvalue, hyears, loan) and three categorical (elevel, car, zipcode) features. This dataset was generated using the default settings in the MOA framework [53] (function 1 and perturbation fraction of 0.05) with a total of 1,000,000 instances.
- **Airline dataset (airline)** [53]: this dataset contains 539,383 instances with 7 features, where each instance corresponds to a flight from one airport to another. The instances in this dataset have 2 class values, which describes if that flight was delayed or not, given the information of the scheduled departure.
- **Forest Cover Type dataset (covType)** [53, 40]: represents forest cover types for 30 x 30 meters cells of forest. Each class corresponds to a different cover type. This data set contains 581,012 instances, 54 features (10 numeric and 44 binary), and 7 imbalanced class labels, where each class corresponds to a different cover type.
- **CTU datasets (ctu_1, ctu_2, .. ctu_13)** [54]: contains 13 different network scenarios that vary in quantity and type of botnet. Each scenario contains legitimate, background and malicious traffic in the form of labelled bidirectional network flows. The first two differ in the sense that the former is traffic between confirmed uninfected machines and trustful sources, i.e., a Google's server, while the latter is traffic that was not confirmed as legitimate nor botnet. Here we considered only two classes: background or botnet.
- **Electricity Pricing dataset (elec)** [53]: this dataset was collected from the Australian New South Wales Electricity Market where electricity prices are not fixed. The prices are affected by the demand and supply of the market and updated every five minutes. The dataset contains 45,312 instances, 7 features (6 numeric and 1 categorical), and 2 class labels, which identify the changes on the price (2 possible

classes: up or down) relative to a moving average of the last 24 hours. Additionally, this dataset exhibits temporal dependencies among instances.

- **Led datasets** with 10% and 20% noise composed of 1,000,000 instances (**led_10** and **led_20**) [55]: for each instance in these datasets, there are 7 boolean features, representing the segments of one digit in a 8-bit display, and 17 useless random features. The datasets have 10 classes (one for each digit from 0 to 9).

- **Poker-hand dataset (poker)** [53]: consists of 892,200 instances with 10 features each. Each instance of this dataset is an example of a player's hand and has five cards drawn from a standard deck of 52. Each card is described by two attributes, its suit and rank. In this way, this dataset has 5 numeric features and 5 categorical features. The target class describes the ten possible poker games.

- **Random RBF datasets**: 1 million instances with 10 features; 500 thousand instances with 10 features; and 250 thousand instances with 50 features (**rbf_1kk, rbf_500k, and rbf_250k(50)**) [55, 40]. These datasets are generated using the radial basis function (RBF) generator. The generator creates n centroids (here $n = 50$) at random positions and associates them with a standard deviation value, a weight and a class label. To create each instance, a centroid is selected at random, where those with higher weights have a higher probability of being selected and its values are set according to a random direction chosen to offset the centroid. The extent of the displacement is drawn from a Gaussian distribution according to the standard deviation associated with the given centroid.

- **SEA dataset (sea)** [56]: this dataset has 60,000 instances, 3 features and 3 classes. All features are numeric between 0 and 10, with only the first two being relevant. There are four concepts, with 15,000 instances each. During each concept the threshold θ for relevant_feature1 + relevant_feature2 $> \theta$ then class $= 0$ changes. Threshold values are 8, 9, 7, and 9.5 and the dataset has about 10% of noise.

- **Usenet dataset (usenet)** [57]: simulation of news filtering with a CD related to the change of interest of a user over time. Each instance has 659 binary features describing the presence or absence of a respective word in the piece of text. The two classes indicate if a virtual user has an interest in this kind of news or not. The CD occurs by making this virtual user lose interest in some types of news and gain in others. The dataset contains 5,931 instances.

A summary of the datasets is presented in Table 2.

<table>
<tr><th rowspan="2">Dataset</th><th rowspan="2"># instances</th><th colspan="3"># features</th><th rowspan="2"># classes</th><th rowspan="2">% majority class</th></tr>
<tr><th>numeric</th><th>binary</th><th>categorical</th></tr>
<tr><td>agrawal</td><td>1,000,000</td><td>6</td><td>0</td><td>3</td><td>2</td><td>0.672</td></tr>
<tr><td>airlines</td><td>539,383</td><td>3</td><td>0</td><td>4</td><td>2</td><td>0.555</td></tr>
<tr><td>covType</td><td>581,012</td><td>10</td><td>44</td><td>0</td><td>7</td><td>0.488</td></tr>
<tr><td>ctu_1</td><td>2,824,636</td><td rowspan="13">9</td><td rowspan="13">0</td><td rowspan="13">2</td><td rowspan="13">2</td><td>0.985</td></tr>
<tr><td>ctu_2</td><td>1,808,122</td><td>0.988</td></tr>
<tr><td>ctu_3</td><td>4,710,638</td><td>0.994</td></tr>
<tr><td>ctu_4</td><td>1,121,076</td><td>0.997</td></tr>
<tr><td>ctu_5</td><td>129,832</td><td>0.993</td></tr>
<tr><td>ctu_6</td><td>558,919</td><td>0.991</td></tr>
<tr><td>ctu_7</td><td>114,077</td><td>0.999</td></tr>
<tr><td>ctu_8</td><td>2,954,230</td><td>0.998</td></tr>
<tr><td>ctu_9</td><td>2,753,884</td><td>0.911</td></tr>
<tr><td>ctu_10</td><td>1,309,791</td><td>0.919</td></tr>
<tr><td>ctu_11</td><td>107,251</td><td>0.924</td></tr>
<tr><td>ctu_12</td><td>325,471</td><td>0.993</td></tr>
<tr><td>ctu_13</td><td>1,925,149</td><td>0.979</td></tr>
<tr><td>elec</td><td>45,312</td><td>6</td><td>0</td><td>1</td><td>2</td><td>0.575</td></tr>
<tr><td>hyper</td><td>250,000</td><td>10</td><td>0</td><td>0</td><td>2</td><td>0.500</td></tr>
<tr><td>led_10</td><td rowspan="2">1,000,000</td><td rowspan="2">0</td><td rowspan="2">24</td><td rowspan="2">0</td><td rowspan="2">10</td><td rowspan="2">0.100</td></tr>
<tr><td>led_20</td></tr>
<tr><td>poker</td><td>829,200</td><td>5</td><td>0</td><td>5</td><td>10</td><td>0.501</td></tr>
<tr><td>rbf_500k</td><td>500,000</td><td rowspan="2">10</td><td rowspan="3">0</td><td rowspan="3">0</td><td rowspan="3">2</td><td></td></tr>
<tr><td>rbf_1kk</td><td>1,000,000</td><td>0.536</td></tr>
<tr><td>rbf_250k(50)</td><td>250,000</td><td>50</td><td></td></tr>
<tr><td>sea</td><td>60,000</td><td>3</td><td>0</td><td>0</td><td>2</td><td>0.627</td></tr>
<tr><td>usenet</td><td>5930</td><td>0</td><td>658</td><td>0</td><td>2</td><td>0.504</td></tr>
</table>

Table 2 – Summary of the datasets used in the experiment.

5.2 Evaluation Metrics

When evaluating traditional ML algorithms, strategies, such as hold-out or leave one out are employed [21, 22]. For data stream algorithms, the same strategies cannot be applied. For these algorithms, one possible strategy is called prequential evaluation [11, 14, 1], where the error of a model is computed from the sequence of examples. For each instance in the stream, the algorithm being evaluated makes a prediction based on the features of that instance. After this, that prediction is stored and used to compute the desired performance metrics. When all statistics are updated, the true label of that instance is presented to the algorithm, which is then updated using this new instance. To better understand it, Figure 10 depicts the prequential evaluation method in three different time points.

We considered predictive performance, memory and time costs to measure the performance of the proposed algorithm against the VFDT. To evaluate predictive performance, we used accuracy. Likewise, in data stream tasks, κ_m (Kappa M) [58] was proposed to deal with unbalanced datasets, measuring how a classifier compares with a classifier

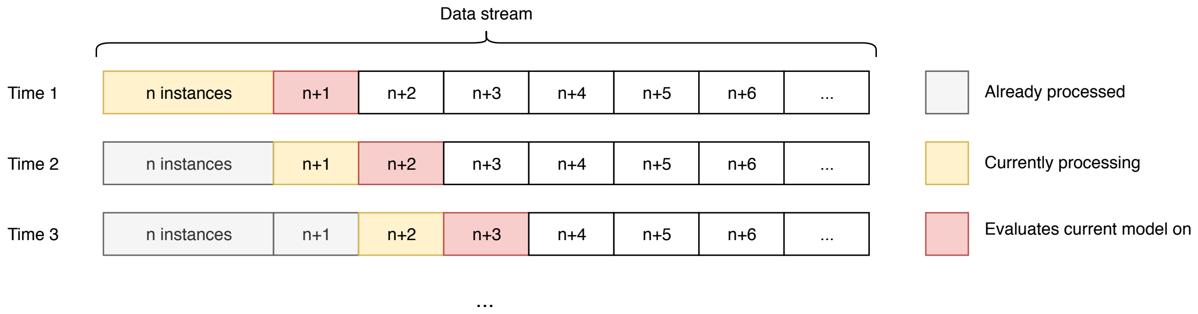


Figure 10 – Prequential evaluation.

that always predicts the majority class. Kappa M is computed as follows:

$$\kappa_m = \frac{\text{accuracy} - p_{maj}}{1 - p_{maj}}, \tag{5.1}$$

where p_{maj} is the percentage of the majority class, in other words, the accuracy of a majority class classifier. Memory efficiency was measured using the amount of RAM (in MB) used by each algorithm. In Python, the *sys* standard library offers the method *getsizeof* which computes the memory costs of an object and the references to its attributes. Here, we summed this cost with the costs returned by all attributes of this object. Furthermore, note that for complex default types (e.g. lists and dictionaries) *getsizeof* does not account for the costs of its elements even in situations where they are primitive types, such as integers or floats. To handle this, *getsizeof* is also called for each element in the list. When computing the cost of a tree, we recursively employ this strategy sum all the costs of all complex objects. Note that this is highly dependent on implementation details, such as programming language and paradigm. Nonetheless, considering that implementations were done following the same design patterns and in the same programming language, it is an accurate measure to compare memory costs of multiple techniques. Additionally, the number of splits performed is also considered. Time costs were measured considering the training time (in seconds) of each algorithm.

5.3 Experimental Setup

Since memory and time costs are highly dependent of implementation details, all modules and algorithms were implemented in Python 3.7[1] [59] and optimised with Cython[2], a super set of the Python language, which allows code to be written in Python and automatically converted to standard C. After that, C extensions are compiled and easily used by any Python code. Some implementation details were based on MOA's[3] implementation [53]. Experiments were performed on a Intel® Xeon Gold 6128 CPU at

[1] https://www.python.org/
[2] http://cython.org/
[3] https://moa.cms.waikato.ac.nz/

3.40 GHz in the Intel® AI DevCloud[4] and on an Intel® i7-6700 CPU at 3.40GHz. To avoid problematic time measures, all algorithms were executed sequentially and on the same machine. Only the experiments in the Subsections 6.4 and 6.5 were executed on the i7-6700. The source code is publicly available[5].

The algorithms' hyperparameters are presented in Table 3. GP and τ were varied to evaluate the behaviour of the SVFDTs when subjected to them. The experiments were repeated 10 times to reduce the variance of time costs. The other performance metrics are deterministic and so, executing multiple experiments yield the same performance.

Table 3 – Hyperparameters explored in the experiments

Hyperparamater	Values
GP	$(100, 200, 400, 800, 1000)$
τ	$(0.01, 0.05, 0.10, 0.15, 0.20)$
δ	$1e - 5$
Leaf Predictor	ANB
Split Criterion	(IG or GI)

[4] https://software.intel.com/ai-academy/tools/devcloud
[5] https://github.com/vturrisi/pystream

6 RESULTS

The results obtained from the experiments are presented in this section. First, results considering different hyperparameter values while using H and IG or GI are presented. Then, the statistical difference of the algorithms is assessed. After that, an in-depth analysis of the worst and best cases, given the hyperparameters, is considered. Then, we address the concern of whether or not the SVFDTs can produce trees smaller than the smallest VFDT. Lastly, the impacts of each heuristic and their possible combinations are explored.

6.1 Analysis of the influence of the hyperparameters

In Table 4, the mean performance metrics when considering all datasets for each possible combination of algorithm, GP and τ using H and IG are presented. First, it is possible to observe that accuracy (along with Kappa M) increases as τ increases. This is due to the fact that setting high values for τ result in larger trees (see memory consumption and number of splits), which will yield better predictive performance. Another interesting point is that as the trees increase, training time decreases. To better understand this, consider a $GP = 100$ and a stream of 1000 instances. If no splits are made, there would be ten split attempts (compute the $G(.)$ values for all features, sort these values and try to split). On the other hand, supposing that at 100 instances a split is made, the remaining 900 instances will fall into different leaves. This can result, for example, in 590 instances falling into one leaf, and the remaining 310 into the other. In this case, five split attempts would be made in the first leaf, and three on the second. Ten splits attempts were performed in the first scenario, while, on the second, only nine were performed. Larger trees will likely result in fewer split attempts in general, decreasing computations. When considering GP, larger values yield lower predictive performance due to smaller trees being produced. Nonetheless, training time decreases, since fewer split attempts are made, resulting in fewer computations. It is possible to see that the SVFDT-II yields better predictive performance than the SVFDT-I at the cost of more memory. Likewise, the VFDT yield better predictive performance than both at the cost of much more memory. The trade-off between predictive performance and memory consumption is not favourable for the VFDT when considering accuracy. In the best case for predictive performance for the VFDT when compared with the SVFDT-I, the difference in accuracy was only 0.96% ($\tau = 0.15$ and $GP = 800$ and $\tau = 0.20$ and $GP = 1000$) at the cost of trees larger by 81% and 80%. When compared with the SVFDT-II, the best difference in predictive performance achieved was 0.54% ($\tau = 0.20$ and $GP = 1000$) at the cost of trees larger by 62%. When considering Kappa M, the VFDT outperforms the SVFDT-I

by a large margin (around 0.10) in all hyperparameter setups. This is due to the fact the CTUs datasets are highly imbalanced (an analysis of the performances for each dataset is presented later), and the SVFDT-I seems to not handle this very well. However, when considering the SVFDT-II, the skip mechanisms introduced seem to greatly alleviate this problem, yielding Kappa M values very similar to the VFDT, with differences around 0.01. This indicates that the SVFDT-II is a better alternative than the SVFDT-I in highly imbalanced datasets. A similar situation occurs for time costs. For the SVFDT-I in comparison with the VFDT, the worst case scenario (where the SVFDT-I is slower than the VFDT) occurs for $\tau = 0.01$ and $GP = 100$, where the SVFDT-I is 2.69% (7 seconds) slower on average. The SVFDT-II is at most 2.74% (or 6 seconds) slower than the VFDT for $\tau = 0.15$ and $GP = 100$. In many cases, both SVFDTs were faster than the VFDT.

Table 5 is analogous to Table 4, but presents the results for the algorithms using GI. Like in the previous table, it is possible to observe that accuracy (along with Kappa M) increases when τ increases due to the same factors. Training time also decreases as the tree gets larger. When considering GP, larger values result in lower predictive performance, since smaller trees are produced. When comparing Table 4 and Table 5, it is possible to see that using GI as impurity and gain metrics results in larger trees for larger τ values. For $\tau = 0.01$, the algorithms using GI are slightly more lightweight. Likewise, for the same τ value, when using GI, the algorithms yield better predictive performance. Note that this is related to better splits since tree size is smaller. As τ gets larger, the trees produced when using GI consume almost double the memory of the trees produced when using H and IG. This also results in better predictive performance and smaller training times. One interesting point is that the largest tree created by the SVFDT-I when using H and IG was 4.648 MB whereas the largest tree created when using GI was 3.325 MB. In the first case, the SVFDT-I produced trees of 15% the size of the trees produced by the VFDT and, in the second case, the trees produced were only 6% of the original trees. This indicates that even though trees produced using GI are larger, the SVFDT-I sufficiently holds tree growth. This makes it a very good alternative when considering predictive performance and memory and time costs. Although memory reductions for the SVFDT-II are less perceptible for smaller τ values, they become much more clear for larger values. When the VFDT produces smaller trees, the majority of splits made by it are considered important, whereas larger trees are likely to have many unnecessary splits. The maximum difference in performance for the VFDT in comparison with the SVFDT-I occurs for $\tau = 0.15$ and $GP = 200$, where the former had 1.41% more accuracy than the latter. Nonetheless, the VFDT had to produce trees almost 14 times larger to achieve this predictive performance. The VFDT only outperformed the SVFDT-II for a maximum of 0.51% in accuracy (for $\tau = 0.15$ and $GP = 800$ and $\tau = 0.20$ and $GP = 1000$). When considering Kappa M, the VFDT also outperforms the SVFDT-I by a large margin (around 0.10) in all hyperparameter setups. Nonetheless, the SVFDT-II also

Table 4 – Performance metrics of all algorithms for entropy and IG.

τ	GP	Algorithm	Accuracy	Kappa M	Memory (MB)	Splits	Time (s)	Rel. Accuracy	Rel. Memory	Rel. Time
		VFDT	0.8797	0.6077	1.203	38.35 ± 44.78	252.61 ± 427.91	-	-	-
	100	SVFDT-I	0.8736	0.4731	0.513	11.58 ± 14.98	259.42 ± 446.68	0.9931	0.4263	1.0269
		SVFDT-II	0.8795	0.6060	1.050	34 ± 39.31	249.3 ± 429.81	0.9997	0.8735	0.9869
		VFDT	0.8783	0.5937	1.028	31.54 ± 33.84	245.73 ± 425.9	-	-	-
	200	SVFDT-I	0.8708	0.4665	0.445	9.65 ± 12.03	239.84 ± 430.5	0.9915	0.4334	0.9760
		SVFDT-II	0.8777	0.5891	0.924	28.27 ± 32.02	245.97 ± 429.26	0.9994	0.8991	1.0010
		VFDT	0.8778	0.5956	0.833	24.81 ± 27.26	240.65 ± 432.75	-	-	-
0.01	400	SVFDT-I	0.8723	0.4667	0.365	8.81 ± 10.13	238.27 ± 441.18	0.9937	0.4381	0.9901
		SVFDT-II	0.8783	0.5874	0.787	21.92 ± 22.56	240.01 ± 439.2	1.0005	0.9443	0.9973
		VFDT	0.8766	0.5817	0.711	20.77 ± 23.47	229.09 ± 424.75	-	-	-
	800	SVFDT-I	0.8717	0.4439	0.367	7.73 ± 9.49	227.01 ± 434.71	0.9944	0.5166	0.9909
		SVFDT-II	0.8775	0.5766	0.654	17.46 ± 17.66	230.35 ± 428.28	1.0010	0.9202	1.0055
		VFDT	0.8783	0.5805	0.715	20.08 ± 22.15	230.95 ± 433.81	-	-	-
	1000	SVFDT-I	0.8714	0.4522	0.276	6.19 ± 5.64	229.63 ± 440.89	0.9922	0.3856	0.9943
		SVFDT-II	0.8782	0.5758	0.642	18.08 ± 21.11	230.91 ± 434.09	0.9999	0.8979	0.9999
		VFDT	0.8893	0.6644	4.151	115.69 ± 127.26	222.38 ± 362.4	-	-	-
	100	SVFDT-I	0.8839	0.5497	1.963	31.69 ± 44.54	226.84 ± 383.12	0.9940	0.4729	1.0201
		SVFDT-II	0.8886	0.6538	3.259	77.38 ± 70.98	226.52 ± 373.28	0.9992	0.7851	1.0186
		VFDT	0.8894	0.6587	4.024	107.62 ± 122.97	217.83 ± 372.92	-	-	-
	200	SVFDT-I	0.8825	0.5435	1.489	25.35 ± 33.2	219.83 ± 389.55	0.9923	0.3700	1.0092
		SVFDT-II	0.8868	0.6437	2.791	63.31 ± 56.46	219.1 ± 369.58	0.9971	0.6936	1.0058
		VFDT	0.8883	0.6476	3.698	97.69 ± 117.83	213.74 ± 367.92	-	-	-
0.05	400	SVFDT-I	0.8820	0.5394	1.380	22.08 ± 27.13	215.6 ± 387.6	0.9928	0.3731	1.0087
		SVFDT-II	0.8867	0.6330	2.495	53.12 ± 50.39	218.34 ± 380	0.9982	0.6745	1.0215
		VFDT	0.8872	0.6373	3.451	92.27 ± 111.61	211.66 ± 371.76	-	-	-
	800	SVFDT-I	0.8813	0.5110	1.205	19.85 ± 22.37	214.29 ± 390.49	0.9933	0.3492	1.0124
		SVFDT-II	0.8862	0.6313	2.013	43.27 ± 39.11	212.75 ± 377.51	0.9989	0.5833	1.0051
		VFDT	0.8893	0.6363	3.275	87.42 ± 101.73	214.86 ± 379.72	-	-	-
	1000	SVFDT-I	0.8827	0.5200	0.878	17.31 ± 16.98	215.89 ± 389.8	0.9927	0.2680	1.0048
		SVFDT-II	0.8866	0.6290	1.799	38.54 ± 34.27	212.06 ± 374.06	0.9970	0.5492	0.9870
		VFDT	0.8930	0.7032	12.160	298.62 ± 383.44	219.81 ± 373.82	-	-	-
	100	SVFDT-I	0.8876	0.5277	2.987	52.42 ± 65.76	218.98 ± 369.19	0.9940	0.2457	0.9962
		SVFDT-II	0.8917	0.6959	7.554	159.54 ± 171.24	218.75 ± 362.9	0.9985	0.6212	0.9952
		VFDT	0.8916	0.6859	11.586	284.62 ± 379.06	221.87 ± 382.95	-	-	-
	200	SVFDT-I	0.8866	0.5535	2.529	44.88 ± 55.01	220.95 ± 390.47	0.9944	0.2183	0.9959
		SVFDT-II	0.8910	0.6728	5.780	124.08 ± 122.36	220.34 ± 382.34	0.9993	0.4989	0.9931
		VFDT	0.8926	0.6807	9.534	238.23 ± 305.61	214.5 ± 376.78	-	-	-
0.10	400	SVFDT-I	0.8863	0.5506	1.602	32.54 ± 35.56	209.79 ± 379.33	0.9929	0.1680	0.9781
		SVFDT-II	0.8911	0.6607	4.358	94.19 ± 94.69	212.31 ± 370.62	0.9983	0.4571	0.9898
		VFDT	0.8929	0.6676	6.412	166.38 ± 192.68	208.3 ± 373.01	-	-	-
	800	SVFDT-I	0.8860	0.5319	1.275	25.38 ± 29.08	209.43 ± 385.31	0.9922	0.1989	1.0054
		SVFDT-II	0.8899	0.6503	3.067	64.12 ± 61.67	201.77 ± 355.43	0.9965	0.4782	0.9686
		VFDT	0.8940	0.6598	5.543	142.46 ± 159.96	211 ± 371.85	-	-	-
	1000	SVFDT-I	0.8868	0.5388	1.249	24.96 ± 28.25	209.14 ± 376.35	0.9919	0.2253	0.9912
		SVFDT-II	0.8902	0.6339	2.367	51.27 ± 47.11	214.54 ± 385.54	0.9958	0.4271	1.0168
		VFDT	0.8973	0.7107	22.167	511.85 ± 691.87	219.06 ± 367.93	-	-	-
	100	SVFDT-I	0.8931	0.5706	4.663	74.85 ± 108.23	219.6 ± 373.74	0.9953	0.2103	1.0025
		SVFDT-II	0.8973	0.6992	11.561	236.65 ± 271.84	225.05 ± 371.02	1.0000	0.5215	1.0274
		VFDT	0.8969	0.7191	18.332	440.04 ± 582.43	212.93 ± 367.23	-	-	-
	200	SVFDT-I	0.8925	0.5164	2.575	48.12 ± 68.87	206.68 ± 366.89	0.9950	0.1405	0.9707
		SVFDT-II	0.8966	0.7061	8.572	182.46 ± 193.29	210.19 ± 352.96	0.9997	0.4676	0.9871
		VFDT	0.8956	0.6930	12.139	300.88 ± 356.99	215.37 ± 373.05	-	-	-
0.15	400	SVFDT-I	0.8883	0.5356	2.302	39.62 ± 56.17	207.86 ± 374.02	0.9918	0.1897	0.9651
		SVFDT-II	0.8931	0.6717	4.752	104.81 ± 103.71	215.04 ± 377.64	0.9972	0.3915	0.9984
		VFDT	0.8952	0.6719	7.278	188.96 ± 200.94	210.02 ± 362.9	-	-	-
	800	SVFDT-I	0.8866	0.5330	1.366	27.73 ± 33.75	209.08 ± 370.98	0.9904	0.1877	0.9955
		SVFDT-II	0.8911	0.6526	3.356	71.85 ± 71.47	206.57 ± 353.17	0.9954	0.4611	0.9836
		VFDT	0.8936	0.6588	6.360	163.5 ± 171.25	211.15 ± 368.2	-	-	-
	1000	SVFDT-I	0.8857	0.5363	1.483	30.08 ± 39.54	211.82 ± 378.7	0.9912	0.2332	1.0032
		SVFDT-II	0.8899	0.6330	2.638	58.77 ± 57.76	205.21 ± 358.6	0.9959	0.4148	0.9719
	100	VFDT	0.8957	0.7235	31.428	713.12 ± 976.69	221.22 ± 375.49	-	-	-
		SVFDT-I	0.8957	0.5769	4.648	81.46 ± 126.82	223.08 ± 373.67	1.0000	0.1479	1.0084
		SVFDT-II	0.8950	0.7028	12.354	276.12 ± 302.65	225.14 ± 374.66	0.9992	0.3931	1.0177
		VFDT	0.8964	0.7195	21.682	525.65 ± 653.14	221.98 ± 388.93	-	-	-
	200	SVFDT-I	0.8925	0.5154	3.322	64.92 ± 97.96	216.58 ± 387.82	0.9957	0.1532	0.9757
		SVFDT-II	0.8959	0.7053	9.286	205.85 ± 208.05	225.76 ± 394.65	0.9995	0.4283	1.0171
		VFDT	0.8960	0.6936	13.392	334.27 ± 371.49	214.89 ± 365.92	-	-	-
0.20	400	SVFDT-I	0.8887	0.5363	2.632	45.73 ± 72.54	208.22 ± 368.15	0.9918	0.1965	0.9690
		SVFDT-II	0.8929	0.6711	5.190	114.58 ± 119.06	218.03 ± 382.49	0.9965	0.3875	1.0146
		VFDT	0.8957	0.6728	8.376	217.92 ± 231.18	210.9 ± 368.53	-	-	-
	800	SVFDT-I	0.8888	0.5372	1.689	34 ± 53.02	202.92 ± 360.59	0.9923	0.2016	0.9622
		SVFDT-II	0.8927	0.6558	3.652	78.12 ± 81.57	207.68 ± 366.23	0.9967	0.4360	0.9847
		VFDT	0.8951	0.6616	7.323	189.15 ± 202.23	212.49 ± 374.25	-	-	-
	1000	SVFDT-I	0.8865	0.5379	1.493	30.58 ± 41.09	212.15 ± 383.02	0.9904	0.2039	0.9984
		SVFDT-II	0.8897	0.6325	2.781	61.88 ± 62.51	211.6 ± 378.22	0.9940	0.3798	0.9958

greatly alleviate this problem, yielding Kappa M values very similar to the VFDT, with differences around 0.02. Likewise, this came at the cost of producing trees 40% larger. The SVFDT-I was only 16.2% slower (31 seconds) for $\tau = 0.05$ and $GP = 100$ while the SVFDT-II was at most 14.5% slower (27 seconds) for $\tau = 0.20$ and $GP = 100$.

Figure 11 shows the boxplots for accuracy, Kappa M, memory consumption and time costs of the three algorithms. The boxplots were constructed using the mean metrics, computed across all datasets, of all possible hyperparameters setup, separately for H and IG and GI. First, considering H and IG, and focusing on accuracy, it is possible to see that the boxes present similar variance and median values. Although the SVFDTs performed worse, their median values are at most 0.01 less than the ones obtained by the VFDT. A similar pattern occurs for Kappa M. However, the difference in values is much more expressive for the SVFDT-I, with a median value of around 0.54, whereas the VFDT had a median value of around 0.66. Nonetheless, the SVFDT-II presents Kappa M values very similar to the VFDT. On the other hand, when evaluating memory consumption, the VFDT presented high variance (from around 1 MB to 23 MB) and a median value of around 7 MB. The SVFDT-I's box, inferior and superior limits, and its outlier are less than the median value of the VFDT. The SVFDT-II presented higher variance than the SVFDT-I but had a median value of around 3 MB. Likewise, its box and superior limit are less than the VFDT's box. Lastly, considering training time, both SVFDTs had higher values than the VFDT. Nonetheless, their differences are minimal in the median.

The accuracy boxes of the algorithms presented less variance and a higher median value when using GI than when using H and IG. Furthermore, the VFDT outperforms both SVFTDs by a very small margin. However, when for Kappa M, this margin is much larger for the SVFDT-I (around 0.15a similar pattern occurs. However, the difference in values is much more expressive for the SVFDT-I, with a median value of around 0.57, whereas the VFDT had a median value of around 0.68. Nonetheless, the SVFDT-II presents Kappa M values very similar to the VFDT as when using H and IG. When considering memory consumption, the VFDT presented a median value of around 8 MB whereas the SVFDT-I had a median value of around 1 MB and the SVFDT-II of around 5 MB. The variance for the SVFDT-I is almost non-existing when compared to the other algorithms. The SVFDT-II had higher variance but its box and superior limit are almost fitted inside the VFDT's box. Lastly, the VFDT is faster than both SVFDTs. Although the variance in the SVFDT-II is higher than the VFDT, their median values are very similar. The SVFDT-I presented a median value very superior to the VFDT, but this can be explained due to the fact that the SVFDT-I greatly reduced memory costs, which were possible due to many additional computations.

Figure 12 presents heatmaps for the performance metrics for each algorithm using H and IG while varying the GP value in the x-axis and the τ value in the y-axis. For

Table 5 – Performance metrics of all algorithms for GI.

τ	GP	Algorithm	Accuracy	Kappa M	Memory (MB)	Splits	Time (s)	Rel. Accuracy	Rel. Memory	Rel. Time
		VFDT	0.8814	0.5875	1.050	30.31 ± 39.41	232.15 ± 334.39	-	-	-
	100	SVFDT-I	0.8737	0.5048	0.426	10.88 ± 14.82	238.5 ± 352.2	0.9912	0.4058	1.0274
		SVFDT-II	0.8808	0.5846	1.026	27.62 ± 38.14	238.47 ± 345.67	0.9993	0.9777	1.0272
		VFDT	0.8806	0.5875	0.900	26.42 ± 30.85	209.05 ± 320.1	-	-	-
	200	SVFDT-I	0.8734	0.5043	0.378	9.23 ± 12.1	221.85 ± 347.28	0.9919	0.4193	1.0612
		SVFDT-II	0.8791	0.5837	0.842	23.73 ± 29.04	201.12 ± 316.22	0.9983	0.9350	0.9620
		VFDT	0.8808	0.5850	0.852	24.12 ± 27.14	205.87 ± 325.11	-	-	-
0.01	400	SVFDT-I	0.8745	0.5124	0.385	9.15 ± 9.82	204.74 ± 331.68	0.9929	0.4520	0.9945
		SVFDT-II	0.8786	0.5821	0.706	19.15 ± 20.2	210.34 ± 328.49	0.9974	0.8292	1.0217
		VFDT	0.8798	0.5888	0.757	21.19 ± 23.11	197.75 ± 325.13	-	-	-
	800	SVFDT-I	0.8702	0.4756	0.323	7.50 ± 7.36	200.69 ± 336.09	0.9892	0.4261	1.0149
		SVFDT-II	0.8770	0.5837	0.644	17.85 ± 17.75	197.62 ± 321.32	0.9968	0.8506	0.9994
		VFDT	0.8785	0.5800	0.706	19.88 ± 20.21	195.15 ± 319.5	-	-	-
	1000	SVFDT-I	0.8736	0.4754	0.359	8.15 ± 8.48	196.11 ± 329.4	0.9944	0.5086	1.0049
		SVFDT-II	0.8773	0.5756	0.561	15.65 ± 14.42	197.03 ± 326.14	0.9987	0.7943	1.0096
	100	VFDT	0.8973	0.6801	6.251	172.85 ± 194.65	194.66 ± 288.15	-	-	-
		SVFDT-I	0.8883	0.5756	1.547	30.27 ± 34.89	226.2 ± 321.96	0.9900	0.2474	1.1620
		SVFDT-II	0.8952	0.6750	5.179	137.92 ± 157.88	198.39 ± 293.8	0.9977	0.8286	1.0191
		VFDT	0.8970	0.6798	6.058	166.27 ± 187.9	190.3 ± 298.24	-	-	-
	200	SVFDT-I	0.8862	0.5626	1.809	28.85 ± 39.93	202.14 ± 319.72	0.9880	0.2986	1.0622
		SVFDT-II	0.8957	0.6707	4.876	129.12 ± 146.56	195.37 ± 304.04	0.9986	0.8050	1.0266
		VFDT	0.8968	0.6736	5.727	158.27 ± 179.18	183.31 ± 289.84	-	-	-
0.05	400	SVFDT-I	0.8872	0.5705	1.394	22.00 ± 30.65	187.62 ± 309.2	0.9893	0.2434	1.0235
		SVFDT-II	0.8948	0.6685	4.282	108.08 ± 129.4	185.53 ± 293.62	0.9978	0.7477	1.0121
		VFDT	0.8953	0.6667	5.210	142.92 ± 160.64	183.89 ± 293.67	-	-	-
	800	SVFDT-I	0.8845	0.5440	0.961	17.42 ± 19.9	186.57 ± 310.57	0.9879	0.1845	1.0146
		SVFDT-II	0.8930	0.6581	3.756	97.35 ± 117.05	183.76 ± 297.33	0.9974	0.7208	0.9993
		VFDT	0.8962	0.6696	4.785	129.46 ± 143.39	182.58 ± 291.18	-	-	-
	1000	SVFDT-I	0.8837	0.5213	1.118	20 ± 25.86	187.5 ± 306.95	0.9861	0.2336	1.0270
		SVFDT-II	0.8926	0.6516	3.138	74.38 ± 92.72	183 ± 296.8	0.9960	0.6560	1.0023
		VFDT	0.9028	0.7292	19.643	489.92 ± 609.17	184.55 ± 288.9	-	-	-
	100	SVFDT-I	0.8892	0.5586	2.061	34.19 ± 48.76	212.3 ± 315.85	0.9850	0.1049	1.1503
		SVFDT-II	0.8999	0.7172	16.197	381.08 ± 498.96	191.66 ± 294.9	0.9968	0.8246	1.0385
		VFDT	0.9019	0.7064	18.701	481.27 ± 594.91	183.67 ± 286.91	-	-	-
	200	SVFDT-I	0.8885	0.5676	1.844	33.12 ± 48.62	204.01 ± 313.64	0.9852	0.0986	1.1108
		SVFDT-II	0.9013	0.6854	13.810	315.46 ± 420.83	186.15 ± 290.14	0.9994	0.7384	1.0135
		VFDT	0.9013	0.7044	16.070	419.35 ± 543.9	183.89 ± 292.25	-	-	-
0.10	400	SVFDT-I	0.8896	0.5793	1.588	30.23 ± 35.71	185.35 ± 305.01	0.9870	0.0988	1.0079
		SVFDT-II	0.8968	0.6785	10.194	254.38 ± 345.7	185.51 ± 295.33	0.9949	0.6343	1.0088
		VFDT	0.8981	0.6937	9.841	262.12 ± 313.23	179.01 ± 285.65	-	-	-
	800	SVFDT-I	0.8872	0.5615	1.310	24.46 ± 29.6	184.02 ± 305.43	0.9879	0.1331	1.0280
		SVFDT-II	0.8930	0.6705	5.884	148.27 ± 195.73	183.62 ± 294.35	0.9943	0.5979	1.0258
		VFDT	0.8980	0.6868	7.979	213.08 ± 244.38	177.54 ± 286.65	-	-	-
	1000	SVFDT-I	0.8858	0.5372	1.171	21.92 ± 27.67	181.37 ± 304.16	0.9865	0.1468	1.0215
		SVFDT-II	0.8936	0.6572	4.825	119.85 ± 155.05	180.32 ± 292.41	0.9951	0.6047	1.0156
		VFDT	0.9003	0.7371	36.633	916.31 ± 1173.8	188.83 ± 286.71	-	-	-
	100	SVFDT-I	0.8922	0.5978	2.854	46.15 ± 73.42	211.7 ± 315.08	0.9910	0.0779	1.1211
		SVFDT-II	0.8980	0.7143	26.865	640.08 ± 895.37	201.54 ± 305.89	0.9975	0.7334	1.0673
		VFDT	0.9028	0.7372	31.551	804.92 ± 1081.94	185.22 ± 286.99	-	-	-
	200	SVFDT-I	0.8887	0.5456	2.260	33.42 ± 53.13	197.97 ± 308.18	0.9844	0.0716	1.0688
		SVFDT-II	0.9005	0.7188	20.227	489.12 ± 706.26	192.85 ± 300.75	0.9974	0.6411	1.0412
		VFDT	0.9020	0.7159	18.607	478.23 ± 596.18	177.86 ± 284.8	-	-	-
0.15	400	SVFDT-I	0.8882	0.5626	1.685	30.15 ± 38.15	186.07 ± 308.1	0.9848	0.0906	1.0461
		SVFDT-II	0.9005	0.6977	11.719	290.27 ± 397.09	181.08 ± 289.75	0.9984	0.6298	1.0181
		VFDT	0.8981	0.6937	9.841	262.12 ± 313.23	175.82 ± 286.71	-	-	-
	800	SVFDT-I	0.8872	0.5615	1.310	24.46 ± 29.6	182.8 ± 308.02	0.9879	0.1331	1.0397
		SVFDT-II	0.8930	0.6705	5.884	148.27 ± 195.73	178.34 ± 293.2	0.9943	0.5979	1.0143
		VFDT	0.8980	0.6868	7.979	213.08 ± 244.38	177.88 ± 284.64	-	-	-
	1000	SVFDT-I	0.8858	0.5372	1.171	21.92 ± 27.67	181.64 ± 302.02	0.9865	0.1468	1.0211
		SVFDT-II	0.8936	0.6572	4.825	119.85 ± 155.05	180.36 ± 290.91	0.9951	0.6047	1.0139
	100	VFDT	0.9016	0.7414	54.251	1351.81 ± 1964.98	189.60 ± 290.49	-	-	-
		SVFDT-I	0.8901	0.5959	3.325	50.46 ± 84.31	206.9 ± 312.9	0.9873	0.0613	1.0913
		SVFDT-II	0.9016	0.7261	33.490	758.69 ± 1290.81	217.16 ± 344.33	1.0000	0.6173	1.1454
		VFDT	0.9031	0.7417	35.035	884.85 ± 1149.03	177.84 ± 282.01	-	-	-
	200	SVFDT-I	0.8908	0.5488	2.293	40.46 ± 56.72	190.05 ± 309.67	0.9864	0.0654	1.0686
		SVFDT-II	0.9017	0.7217	20.765	498.88 ± 721.14	185.91 ± 292.04	0.9984	0.5927	1.0454
		VFDT	0.9020	0.7159	18.607	478.23 ± 596.18	187.41 ± 286.55	-	-	-
0.20	400	SVFDT-I	0.8882	0.5626	1.685	30.15 ± 38.15	194.59 ± 309.78	0.9848	0.0906	1.0383
		SVFDT-II	0.9005	0.6977	11.719	290.27 ± 397.09	191.71 ± 294.09	0.9984	0.6298	1.0230
		VFDT	0.8981	0.6937	9.841	262.12 ± 313.23	184.2 ± 295.37	-	-	-
	800	SVFDT-I	0.8872	0.5615	1.310	24.46 ± 29.6	190.69 ± 316.35	0.9879	0.1331	1.0352
		SVFDT-II	0.8930	0.6705	5.884	148.27 ± 195.73	185.96 ± 302.91	0.9943	0.5979	1.0096
		VFDT	0.8980	0.6868	7.979	213.08 ± 244.38	183.16 ± 292.92	-	-	-
	1000	SVFDT-I	0.8858	0.5372	1.171	21.92 ± 27.67	186.22 ± 310.27	0.9865	0.1468	1.0167
		SVFDT-II	0.8936	0.6572	4.825	119.85 ± 155.05	184.3 ± 296.77	0.9951	0.6047	1.0062

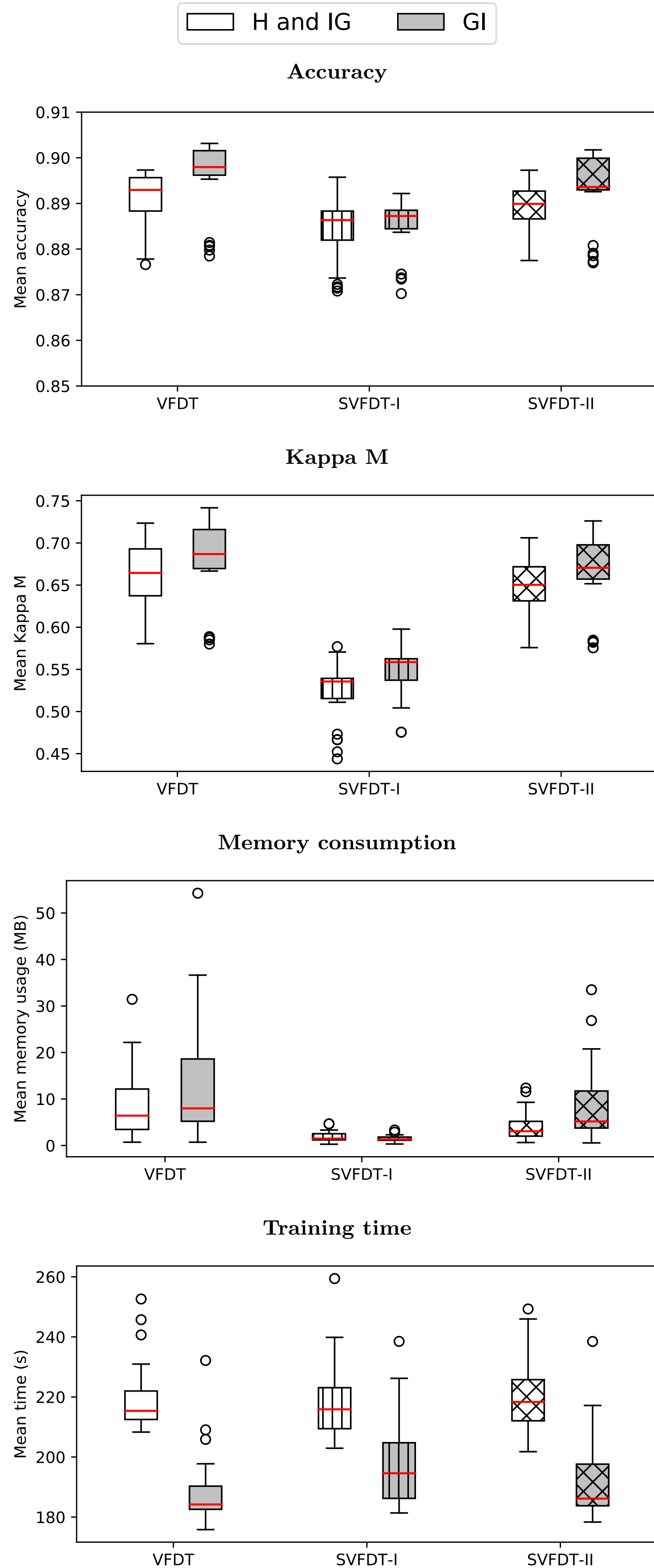


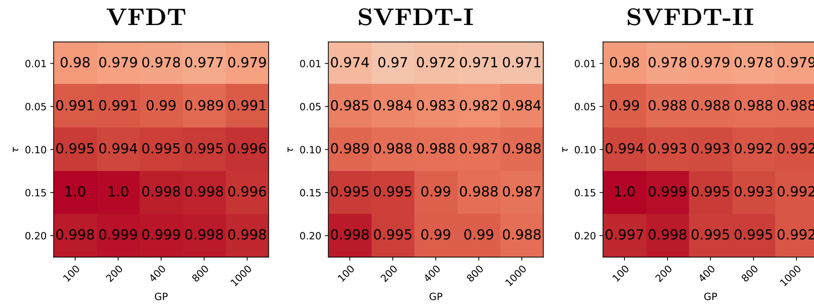
Figure 11 – Boxplots of the performance metrics of the VFDT, SVFDT-I and SVFDT-II.

memory and time, each value is computed by dividing the smallest value obtained across all algorithms by the value obtained by a given algorithm, GP and τ setting. So, the higher the obtained value the smaller it will be after this scaling process. To compute the values for accuracy, each value is divided by the highest value obtained across all algorithms and hyperparameter settings. In this sense, the values presented in the heatmap represent the percentage of the best obtained accuracy. The best accuracy values are concentrated for high τ values and low GP values. This is due to the fact that resulting trees are larger in those scenarios. The VFDT obtained the best accuracies, but both SVFDT-I for $\tau \in (0.15, 0.20)$ and $GP \in (100, 200)$ and SVFDT-II for $\tau \in (0.15, 0.20)$ and $GP \in (100, 200, 400, 800)$ also obtained values close to the best obtained accuracy. For Kappa M, the SVFDT-II also obtained very competitive performance for large values of τ and small values of GP. However, the SVFDT-I values significantly lower due to the highly imbalanced CTU datasets. When considering memory, high values of GP and low values of τ will result in smaller trees. For all possible GP values, when using $\tau = 0.01$ the SVFDT-I outperforms the other two algorithms in all scenarios. Lastly, considering time efficiency, high GP and τ values result in faster algorithms. The SVFDT-II is the fastest ($\tau = 0.10$ and $GP = 800$) followed by the SVFDT-I ($\tau = 0.20$ and $GP = 800$).

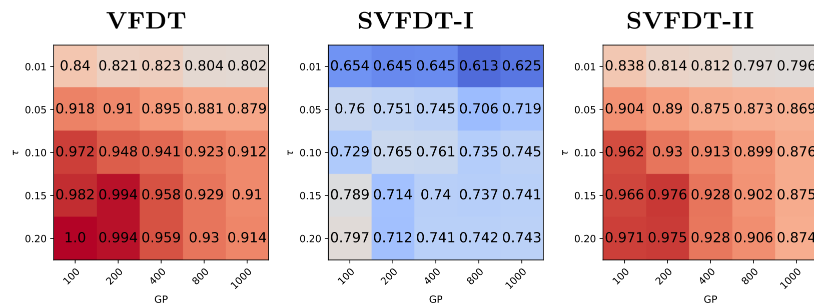
In the same way as Figure 12, Figure 13 presents heatmaps for each algorithm using GI The best accuracy values are concentrated around $\tau \in (0.10, 0.15, 0.20)$ and $GP \in (100, 200, 400)$, with the VFDT and SVFDT-II presenting the best values. Kappa M values follow a similar pattern but are significantly lower for the SVFDT-I due to the CTU datasets. Considering memory efficiency, for all GP values, the SVFDT-I with $\tau = 0.01$ outperforms the other algorithms in all cases. However, the best case here is for $\tau = 0.01$ and $GP = 800$, instead of $GP = 1000$. Lastly, a similar pattern for τ and GP considering time efficiency is present here. However, the VFDT outperforms both SVFDTs by a small margin.

The heatmaps in Figures 12 and 13 were overlaid to give each metric the same importance. This process is done by taking the mean of the heatmaps element-wise. These global heatmaps are presented in Figure 14. First, considering H and IG, it is possible to see that for $GP \in (400, 800, 1000)$, the SVFDT-I using $\tau = 0.01$ presents the best result among the three algorithms. The SVFDT-II, for $\tau = 0.01$ and $GP \in (800, 1000)$, also outperforms the VFDT by a small margin. Considering GI, a similar pattern repeats. The only difference is that the optimal τ and GP values are $\tau = 0.01$ and $GP = 800$ instead of $\tau = 0.01$ and $GP = 1000$ and that the SVFDT-I also outperforms the VFDT when using $GP = 200$ and $\tau = 0.01$. Likewise, the SVFDT-II outperforms the VFDT for $\tau = 0.01$ and $GP \in (800, 1000)$.

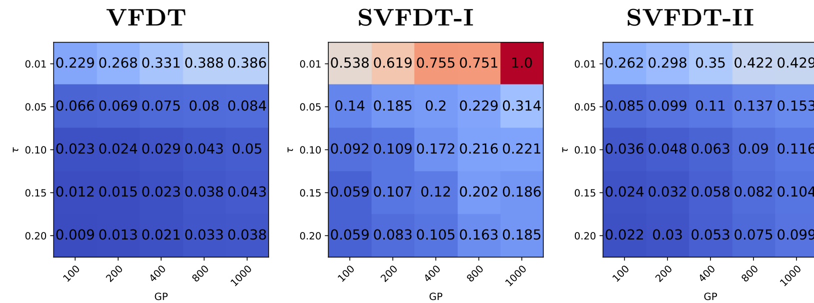
(a) Scaled accuracy according to GP and τ



(a) Scaled Kappa M according to GP and τ



(b) Scaled memory efficiency according to GP and τ



(c) Scaled time efficiency according to GP and τ

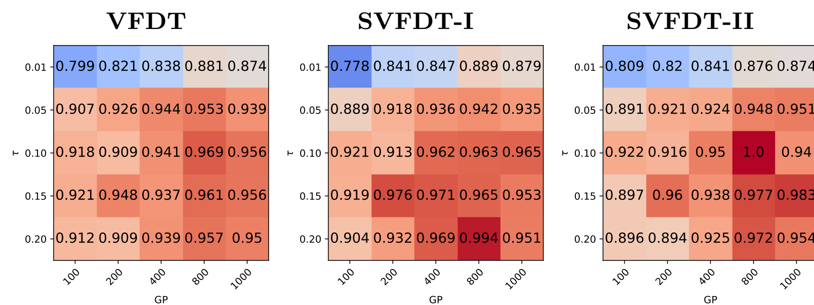
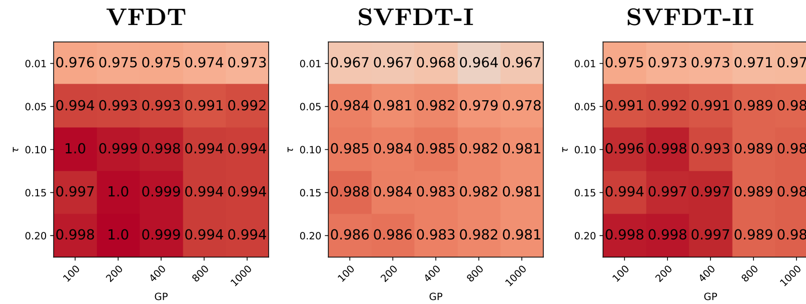
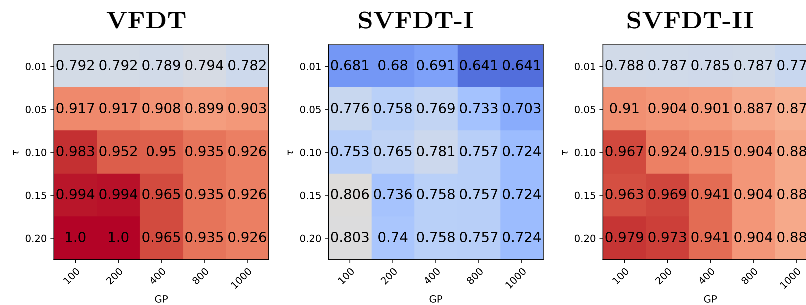


Figure 12 – Heatmaps of each algorithm for each possible GP and τ configuration using entropy and IG. Rows correspond to different metrics. Red colours represent better performance (higher accuracy or lower memory or time consumption), while blue colours the opposite.

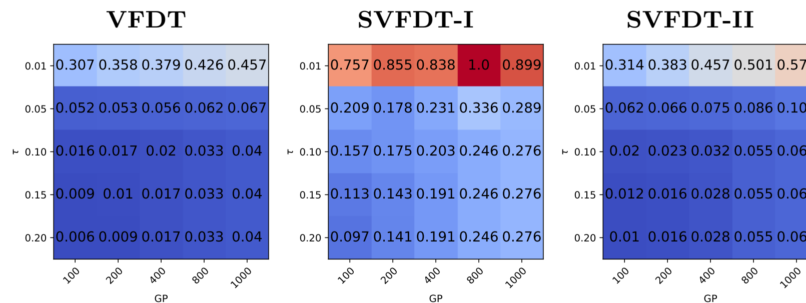
(a) Scaled accuracy according to GP and τ



(a) Scaled Kappa M according to GP and τ



(b) Scaled memory efficiency according to GP and τ



(c) Scaled time efficiency according to GP and τ

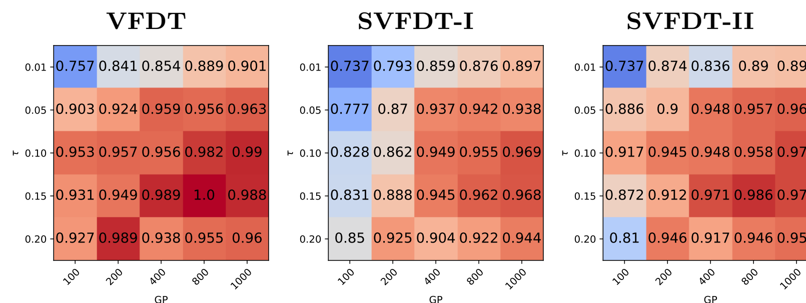
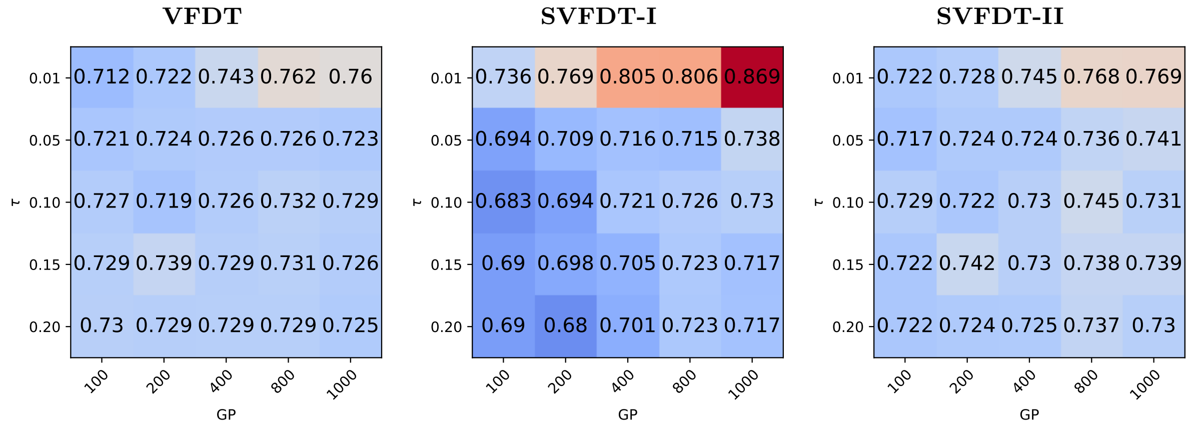


Figure 13 – Heatmaps of each algorithm for each possible GP and τ configuration using GI. Rows correspond to different metrics. Red colours represent better performance (higher accuracy or lower memory or time consumption), while blue colours the opposite.

(a) Entropy and IG



(b) GI

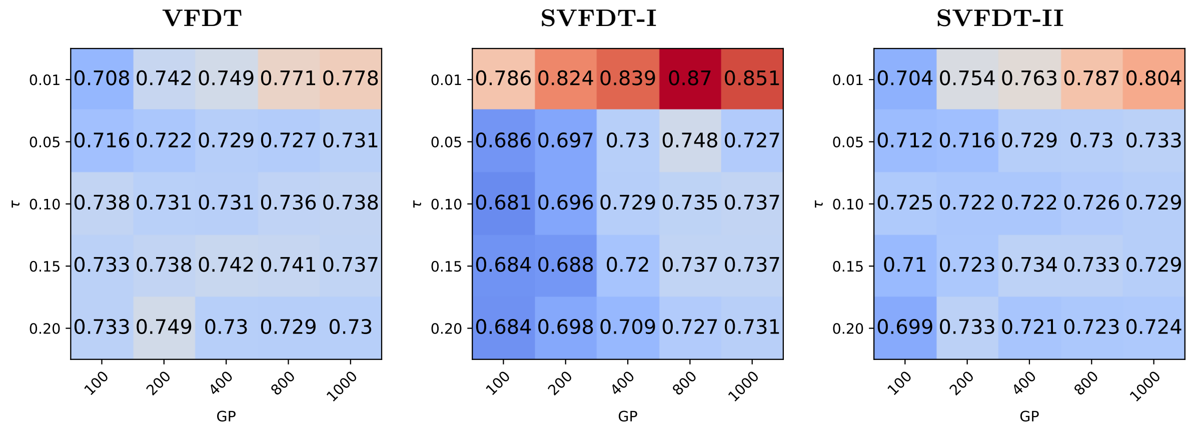


Figure 14 – Heatmaps of the performance of each algorithm when considering accuracy, memory and time efficiency for each possible GP and τ. Red colours represent better performance, while blue colours the opposite.

6.2 Friedman's statistical test and post-hoc Nemenyi analysis

The algorithms were statistically compared considering different τ and GP values using the Friedman's statistical test [60] and the post-hoc Nemenyi analysis [61]. A Critical Difference diagram is used to illustrate the results from these tests. As reported in [62], this test is used to compare multiple ML algorithms evaluated across multiple datasets. A Critical Difference diagram was constructed for each metric (accuracy, memory consumption and time costs) for the algorithms using H and IG. Likewise, diagrams were also constructed for the algorithms using GI, resulting in six diagrams. Note that the performance values obtained per dataset for each hyperparameter setting were used to employ the statistical tests. Recall that there are 25 different hyperparameter setups and 26 datasets, resulting in 650 performance values for each algorithm.

In Figures 15, 16, 17 and 18, the Critical Difference diagrams for accuracy, Kappa

M, memory consumption and time costs are presented for the algorithms using H and IG. First, considering accuracy and Kappa M, all algorithms are statistically different. The VFDT is the best, followed by the SVFDT-II and the SVFDT-I. In this sense, the VFDT outperforms both SVFTDs, which was expected. Nonetheless, these diagrams only measure if an algorithm is statistically better than other algorithms without considering the real difference values. Although there exists a statistical difference, it is minimal, as noted in Table 4, the boxplots in Figure 11 and the heatmaps in Figure 12. On the other hand, when considering memory consumption, the SVFDT-I is the best, followed by the SVFDT-II and the VFDT. Additionally, the differences are more substantial. Lastly, when considering time costs, the SVFDT-I is also significantly better, with the SVFDT-II and VFDT being statistically equivalent.

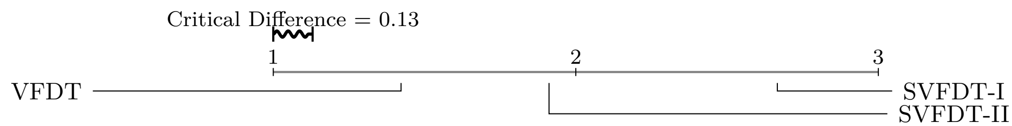


Figure 15 – Accuracy Nemenyi for algorithms using entropy and IG ($\alpha = 0.05$).

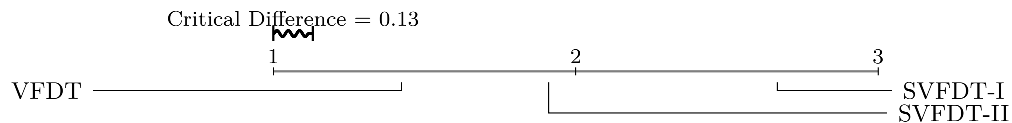


Figure 16 – Kappa M Nemenyi for algorithms using entropy and IG ($\alpha = 0.05$).

Figure 17 – Memory consumption Nemenyi for algorithms using entropy and IG ($\alpha = 0.05$).

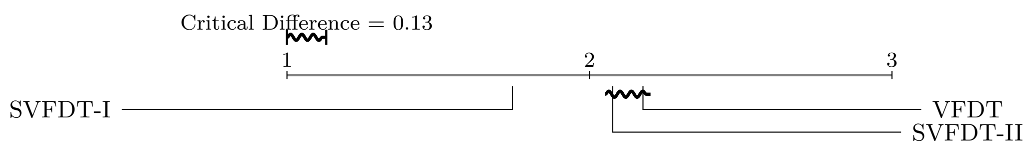


Figure 18 – Time costs Nemenyi for algorithms using entropy and IG ($\alpha = 0.05$).

The Critical Difference diagrams in Figures 19, 20, 21, 22 are analogous to the ones previously presented, but were constructed for the algorithms using GI. For accuracy, Kappa M and memory, the same pattern repeats, with the VFDT having the highest accuracy and highest memory consumption, the SVFDT-I having the worst accuracy but using the least memory, and the SVFDT-II being in the middle. However, when considering time costs, all algorithms are statistically different, with the VFDT being the

fastest, followed by the SVFDT-I. This is due to the fact that when using GI memory was further reduced then when using H and IG.

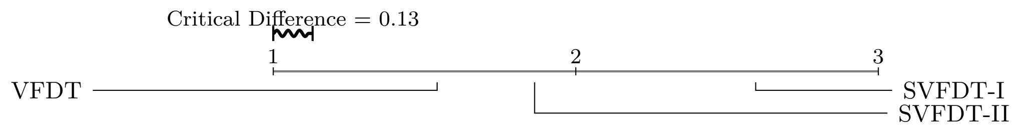


Figure 19 – Accuracy Nemenyi for algorithms using GI ($\alpha = 0.05$).

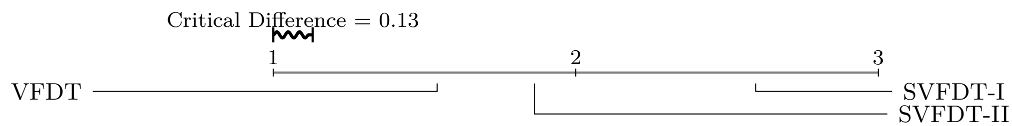


Figure 20 – Kappa M Nemenyi for algorithms using GI ($\alpha = 0.05$).

Figure 21 – Memory consumption Nemenyi for algorithms using GI ($\alpha = 0.05$).

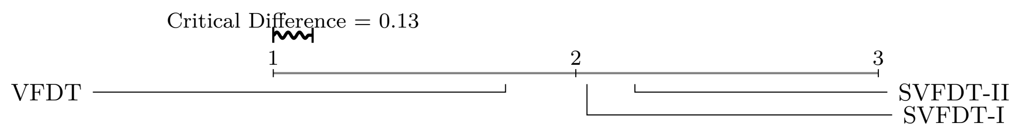


Figure 22 – Time costs Nemenyi for algorithms using GI ($\alpha = 0.05$).

6.3 Best and worst results according to hyperparameters

From now on, an analysis of the performance of the algorithms by fixing GP and τ is considered. Further, the discussion is carried out for H and IG or GI separately.

First, consider the worst case for the SVFDT-I, where memory reduction was minimal. To determine this case, the highest mean relative memory value was selected, i.e., the hyperparameter setup which made the SVFDTs the least effective. Considering H and IG, the worst case was for $\tau = 0.01$ and $GP = 800$. Figure 23 presents a comparison between the algorithms considering each individual dataset. The values presented are computed by dividing each performance metric by the largest value obtained in each dataset. In this sense, larger values are better when considering accuracy and Kappa M, and lower values are better when considering memory and time. Recall that, for accuracy, values closer to 1 represent better performance, whereas, for the other two metrics, lower values are more desirable. First, it is possible to see that the three algorithms achieved very similar accuracy values for all datasets. However, the SVFDT-I suffers from highly

imbalanced problems, such as the CTU datasets, obtaining significantly lower Kappa M scores for the *ctu_1*, *ctu_2*, *ctu_4*, *ctu_6*, *ctu_8*, *ctu_9*, *ctu_12* and *ctu_13* datasets. Nonetheless, the SVFDT-II is able to handle this, being a better alternative than the SVFDT-I for imbalanced datasets. When considering memory consumption, the SVFDT-I consumed more memory than the VFDT in the *covType*, *ctu_10* and *ctu_11* datasets. By holding tree growth, different splits were made for the *covType* and *ctu_10* datasets, which resulted in a larger tree with higher accuracy. For the *ctu_11* dataset, only 2 splits are performed by all algorithms. However, it is possible to observe that memory was reduced in other datasets. Considering the SVFDT-II, due to different splits, memory increased for the *covType*, *ctu_1* and *ctu_5* datasets. Although in some datasets memory consumption was not reduced, the SVFDT-II was able to reduce consumption for 16 out of the 26 datasets. Considering training time, the algorithms performed very closely, with the SVFDT-I being slightly faster in the majority of datasets.

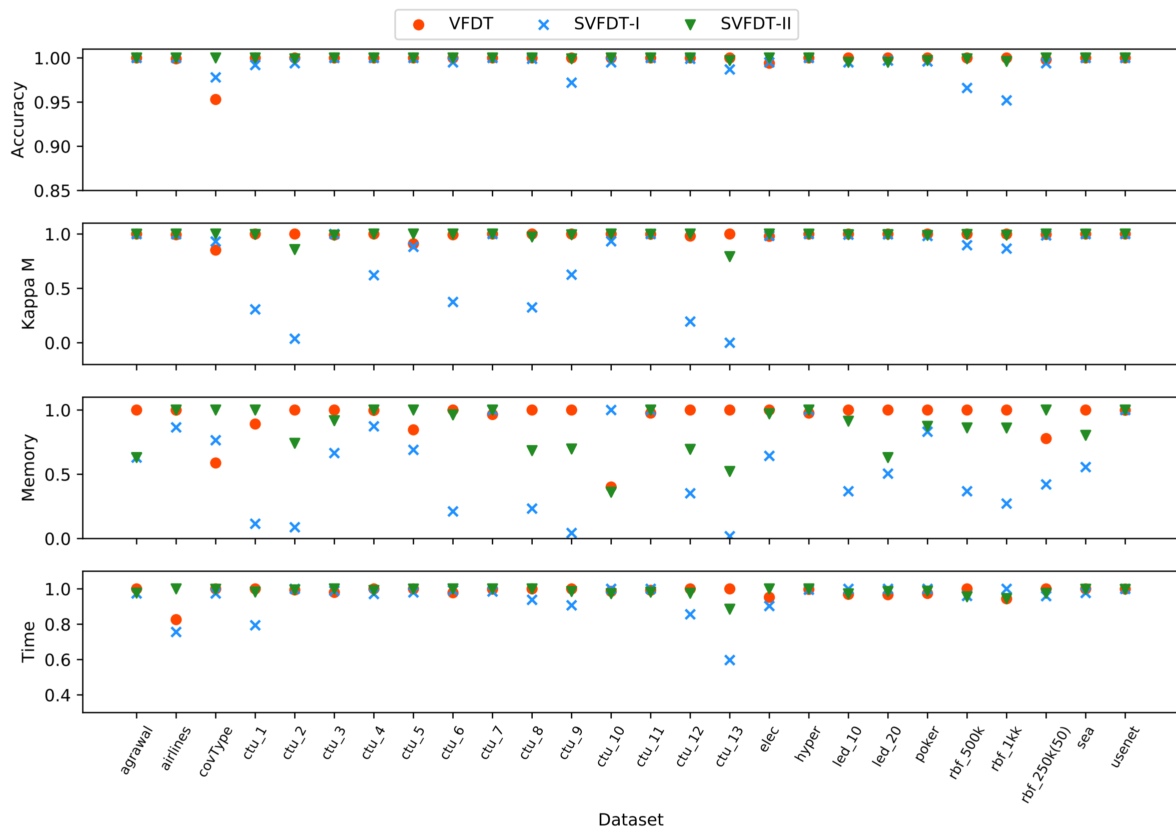


Figure 23 – Performance metrics of all algorithms by dataset for $\tau = 0.01$, $GP = 800$, H and IG (worst case). The metrics are scaled by dividing by the largest value for each dataset. Larger values are better for accuracy and Kappa M, and lower values are better for memory and time.

Table 9 presents the performance metrics for the algorithms for each dataset with this hyperparameter setup. Accuracy values obtained by the SVFDTs are very close to the ones of the VFDT. The minimal relative accuracy was 0.9515 for the SVFDT-I and 0.9951 for the SVFDT-II for the *rbf_1kk* and *led_10* datasets respectively. In only 4

(*ctu_9*, *ctu_13*, *rbf_500k* and *rbf_1kk*) out of the 26 datasets, the relative accuracy of the SVFDT-I was less than 0.99. However, in 5 datasets (*airlines*, *covType*, *ctu_11*, *elec*, *hyper*) the SVFDT-I obtained better accuracy than the VFDT. Likewise, the SVFDT-II also obtained better accuracy in 6 datasets (*airlines*, *covType*, *ctu_5*, *ctu_6*, *elec*, *rbf_250k(50)*). However, for Kappa M, the SVFDT-I obtained significantly lower performance for all CTU datasets, except for the *ctu_3*, *ctu_5*, *ctu_7*, *ctu_10* and *ctu_11*. It is interesting to highly that on *ctu_3* the SVFDT-I obtained a higher Kappa M even though this dataset is also highly imbalanced. This suggests that creating constraints based on predictive performance may enhance the SVFDT-I performance without the need to use the SVFDT-II skip mechanisms. Nonetheless, these mechanisms seem to be sufficient to handle this problem, since the SVFDT-II had similar Kappa M values in comparison to the VFDT even on the CTU datasets. When considering memory consumption, the SVFDT-I consumed more memory than the VFDT in 6 datasets (*covType*, *ctu_7*, *ctu_10*, *ctu_11*, *hyper*, *usenet*). For the *ctu_7*, *ctu_11*, *hyper*, *usenet* datasets, the extra memory costs are related to the fact that the resulting tree size was not minimised and so, the additional statistics stored by the SVFDT impacted memory consumption. However, for the *covType* and *ctu_10* datasets, memory increased due to the fact that splits in nominal features were made. By holding tree growth, nominal features surpassed the IG of other numeric features and were used to split, which result in non-binary splits. Lastly, considering time costs, the SVFDTs are faster than the VFDT in most cases. In the worst case, the relative times were 1.031 for SVFDT-I (*led_10*) and 1.21 for the SVFDT-II (*airlines*). However, for the best cases, they were 0.597 and 0.884 in the *ctu_13* dataset for the SVFDT-I and II.

Considering H and IG, memory was best reduced for $\tau = 0.20$ and $GP = 100$. Figure 26 presents the comparison between algorithms considering this hyperparameter configuration. Although the accuracy values are very similar, Kappa M values are also significantly lower for the same CTU datasets for the worst case. Memory consumption greatly decreased for all datasets, with the SVFDT-I achieving at least around 30% of reduction and the SVFDT-II around 15%. Furthermore, in some datasets, the SVFDT-I presented a reduction of more than 98% in memory consumption. Considering training time, there is no clear faster algorithm, with the SVFDT-I and the VFDT changing between the fastest and slowest algorithm depending on the dataset.

Table 10 presents the performance metrics for the algorithms with this hyperparameter setting. First, considering accuracy, the minimal relative accuracy was 0.9515 for the SVFDT-I and 0.9951 for the SVFDT-II for the *rbf_1kk* and *led_10* datasets. This comprehends to 2.4% and 1.31% less accuracy, respectively, which was obtained by saving 22.66 MB (99.17% of memory) and 1.717 MB (53.39% of memory). In average, the relative accuracy obtained by the SVFDTs was 0.9977 and 0.9999 for the SVFDT-I and II. In 6 datasets (*agrawal*, *airlines*, *hyper*, *poker*, *sea* and *usenet*) the SVFDT-I out-

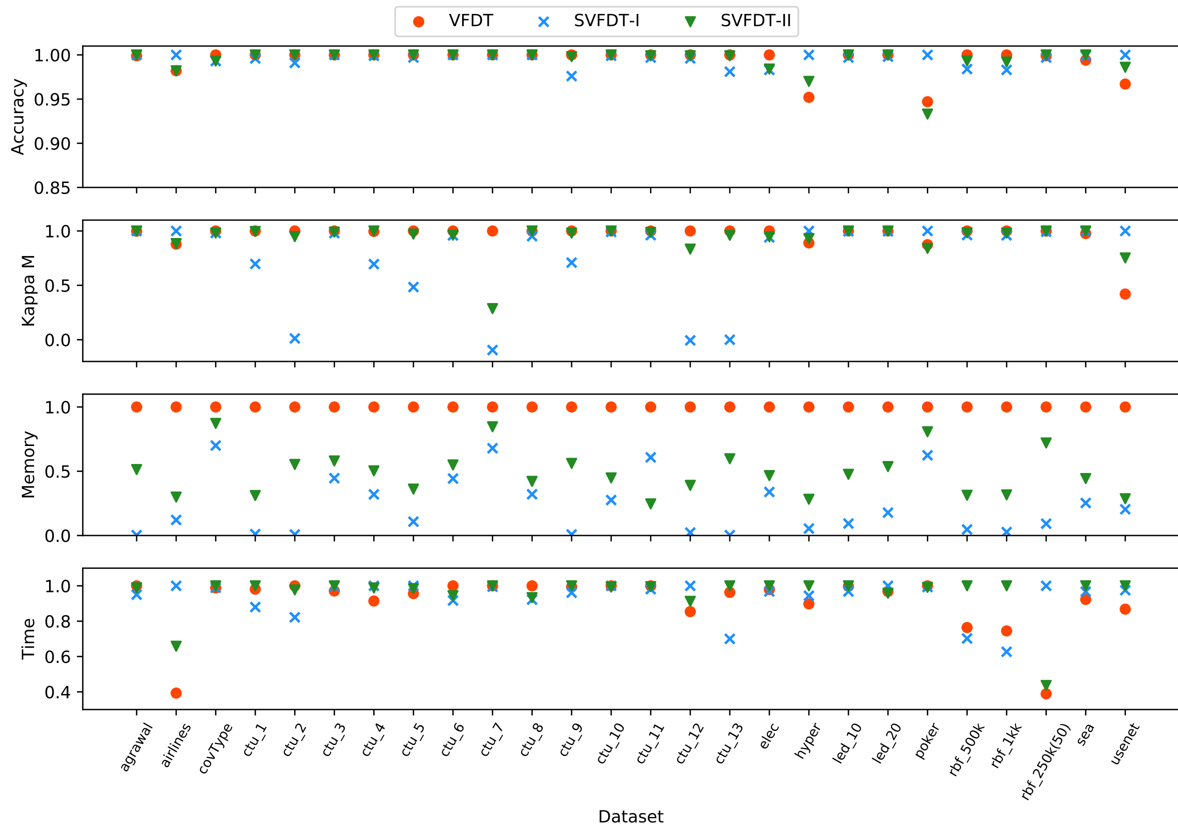


Figure 24 – Performance metrics of all algorithms by dataset for $\tau = 0.20$, $GP = 100$, H and IG (best case). The metrics are scaled by dividing by the largest value for each dataset. Larger values are better for accuracy and Kappa M, and lower values are better for memory and time.

performed the VFDT. Kappa M values are significantly lower for the SVFDT-I in the CTU datasets but the SVFDT-II presents a very similar performance to the VFDT in the same datasets. Likewise, the SVFDT-II had better accuracy in 6 datasets (*agrawal*, *airlines*, *hyper*, *rbf_250k(50)*, *sea* and *usenet*). The relative mean memory consumption of the SVFDT-I was 0.149, whereas, for the SVFDT-II, it was 0.4708. In the worst cases, the relative memory consumption of the SVFDTs was 0.624 and 0.807 for the SVFDT-I and II both in the *poker* dataset. However, in the best cases, the SVFDT-I had a relative memory consumption of 0.003 and the SVFDT-II of 0.246 for the *agrawal* and *ctu_11* datasets. Considering a scenario where all trees produced would be stored in memory, 820 MB would be needed for the VFDT, 120 MB for the SVFDT-I and 320 MB for the SVFDT-II. If an ensemble of 10 trees were used without any strategy reset the trees, memory costs would rise to 8 GB, 1.17 GB and 3.12 GB. Considering time costs, the SVFDT-I, apart from the *airlines* and *rbf_250k(50)* datasets, which had a relative time of 2.54 and 2.56, had the worst relative time of 1.17 (*ctu_12*). For the SVFDT-II, excluding the *airlines* dataset, which resulted in a relative time of 1.67, had in its worst case, a relative time of 1.34 for the *rbf_1kk* dataset. However, the mean relative times for the SVFDT-I and II were 0.99 and 1.02.

Now, considering when the algorithms were using GI, the worst case was for $\tau = 0.01$ and $GP = 100$. Figure 25 presents the comparison between algorithms considering this hyperparameter configuration. Accuracy values of the algorithms for all datasets are very similar. It is interesting to observe that using GI makes the SVFDT-I have much more similar Kappa M in the CTU datasets, having only performed very badly in 3 cases (*ctu_8*, *ctu_12* and *ctu_13*). Note, however, that although the Kappa M obtained in the *ctu_12* dataset is four times worst, the VFDT obtained a 0.0005 score whereas the SVFDT-I had a -0.0018 score, both performing very poorly. Differently from the worst case when using H and IG, the SVFDT-I was able to greatly reduce memory costs in 18 datasets. The SVFDT-II reduced memory in 13 datasets by a small margin. Training times are very similar, with the three algorithms performing very similar considering all datasets.

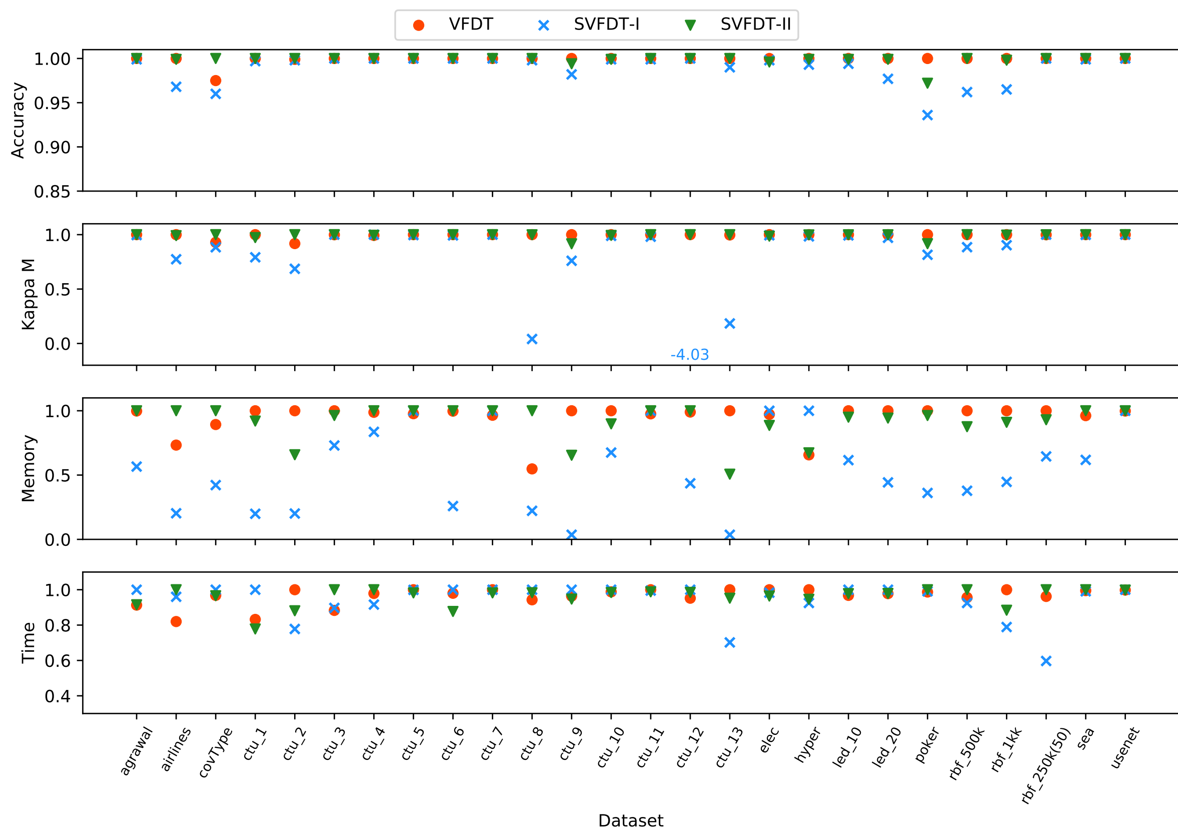


Figure 25 – Performance metrics of all algorithms by dataset for $\tau = 0.01$, $GP = 100$ and GI (worst case). The metrics are scaled by dividing by the largest value for each dataset. Larger values are better for accuracy and Kappa M, and lower values are better for memory and time.

Table 11, in Appendix B, presents the numerical values of the performance metrics Accuracy values of the VFDT and the SVFDTs are very similar for all datasets. In the worst case, the relative accuracy of the SVFDT-I was 0.936 for the *poker* dataset. However, this came at the cost of only spending around a third of memory. For the SVFDT-II, the worst case was 0.971, for the same dataset. In none of the cases did the SVFDTs outper-

form the VFDT, but they had a mean relative accuracy of 0.998 and 0.999. Considering Kappa M, the VFDT outperforms the SVFDT-I by at most 0.20 in the *ctu_9* if ignoring the *ctu_8* and *ctu_13* datasets since both correspond to outlier behaviours of poor performance of the SVFDT-I. For the SVFDT-II, the VFDT achieved at most 0.07 and 0.02 higher Kappa M values for the *ctu_9* and *ctu_1* datasets, respectively. Furthermore, the SVFDT-II outperforms the VFDT considering Kappa M in 5 datasets (*covType*, *ctu_2*, *ctu_3*, *ctu_4* and *ctu_13*). The SVFDT-I consumed more memory than the VFDT for *ctu_5*, *ctu_7*, *ctu_11*, *elec* and *usetnet* dataset due to the additional statistics stored by it. In the *hyper* dataset, by holding tree growth, the SVFDT-I produced different splits, resulting in a larger tree. Related to storing additional statistics, the SVFDT-II used more memory in the *agrawal*, *ctu_4*, *ctu_5*, *ctu_6*, *ctu_7*, *ctu_11*, *ctu_12*, *hyper*, *sea* and *usenet* datasets. However, due to different splits, resulting trees were larger for *airlines*, *covType* and *ctu_8* datasets. The mean relative memory consumption was 0.52 for the SVFDT-I and 0.98 for the SVFDT-II. Considering time costs, the relative time of the SVFDT-I was, in the worst case, 1.20 and, in the best case, 0.62, for the *ctu_1* and *rbf_250k(50)* datasets, respectively. For the SVFDT-II, the worst relative time was 1.22 for the *airlines* dataset and the best was 0.88 for the *ctu_2* dataset. In average, the relative times were 1.002 and 0.998 for the SVFDT-I and II.

For $\tau = 0.20$ and $GP = 200$ the SVFTDs presented the highest memory reduction. Figure 26 presents the comparison between algorithms for this hyperparameter configuration. A similar pattern in the accuracy values is also observed, with algorithms having similar accuracy values. However, unlike the worst case for GI, Kappa M performance decreased more significantly for all CTU datasets with the exception of *ctu_3*, *ctu_10* and *ctu_11*. For only the *ctu_12* dataset the SVFDT-II performed poorly considering Kappa M. Considering memory, the SVFDT-I greatly reduced consumption in all datasets by at least 50%. The SVFDT-II only presented the same memory consumption for the *ctu_7* dataset, reducing it by at least 15% in all the other datasets. Lastly, considering training time, the VFDT is the fastest in 13 out of 26 datasets, but with the SVFDT-I being faster in 5 datasets.

Table 12, in Appendix B, presents the numerical values of the performance metrics. Considering accuracy, the SVFDT-I had in the worst cases a relative accuracy of only 0.87 and 0.88 for the *covType* and *poker* datasets. However, for the other datasets, it was higher than 0.96, with the SVFDT-I outperforming the VFDT in 6 datasets (*agrawal*, *airlines*, *hyper*, *led_10*, *led_20* and *usenet*). In average, the SVFDT-I had a relative accuracy of 0.997. For the SVFDT-II, the worst case was 0.96 for the *poker* dataset. Its mean relative accuracy was 0.999. Additionally, it outperforms the VFDT in 8 datasets (*agrawal*, *airlines*, *hyper*, *led_10*, *led_20*, *rbf_500k*, *rbf_1kk* and *usenet*). For the CTU datasets, Kappa M performance is significantly lower for the SVFDT-I, but, as in the other hyperparameter configurations, the SVFDT-II is able to keep competitive performance

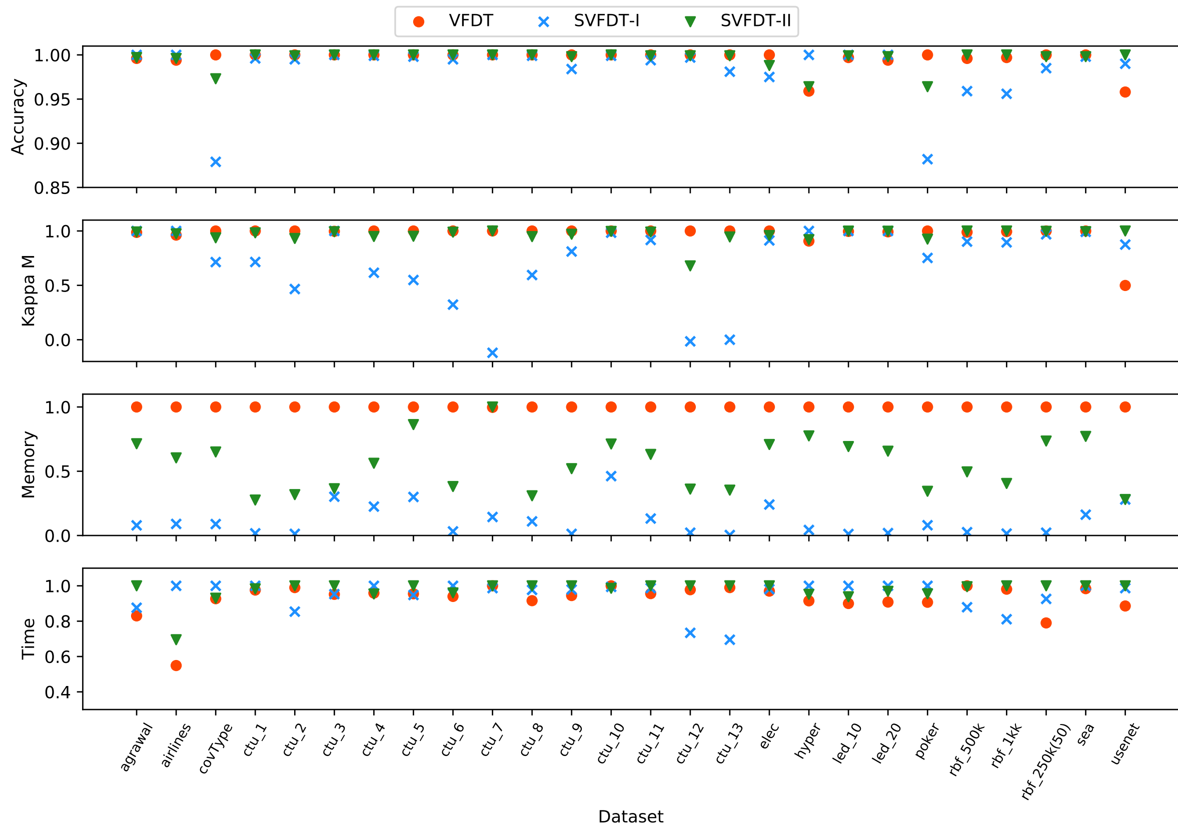


Figure 26 – Performance metrics of all algorithms by dataset for $\tau = 0.20$, $GP = 200$ and GI (best case). The metrics are scaled by dividing by the largest value for each dataset. Larger values are better for accuracy and Kappa M, and lower values are better for memory and time.

in these datasets. When considering memory consumption, the mean relative values of the SVFDT-I and II were 0.079 and 0.583 respectively. In the worst case, the SVFDT-I consumed only 46% of the memory used by the VFDT (*ctu_10*). Due to producing a tree of the same size, in *ctu_7*, the SVFDT-II consumed more memory than the VFDT. Excluding this case, the SVFDT-II produces trees of 86% of the size of the VFDT (*ctu_5*) or at most 77% of the size of the VFDT (*hyper* and *sea* datasets). If all trees were stored in memory, the VFDT would use 909 MB, whereas only 59 MB and 538 MB would be used by the SVFDT-I and II. Using an ensemble of 10 SVFDT-I trees would consume less memory than a single VFDT. Considering time costs, the SVFDTs were slower than the VFDT. Nonetheless, apart for the *airlines* dataset, the difference is very small, with the SVFDTs being faster than the VFDT in some cases. Lastly, the mean relative time costs were 1.03 for both SVFDTs.

6.4 Can the SVFDTs produce trees smaller than the smallest VFDT?

One question that may arise is if the SVFDTs are capable of reducing the memory size of the smallest possible VFDT (using $\tau = 0$). Although this is not used in real problems since it may stop growth, it is still worth seeing what is the impact of the SVFDTs in this situation.

Figures 27 and 28 presents the scaled performance metrics for the algorithms using H and IG, and GI, respectively. Considering H and IG, accuracy values for all algorithms across datasets is very similar. However, for Kappa M, the SVFDT-I performs very poorly in the *ctu_1*, *ctu_2*, *ctu_5*, *ctu_13* datasets. From the memory perspective, the SVFDT-I reduced memory in 17 datasets while the SVFDT-II reduced for 12 datasets. Finally, the SVFDT-I presents a slight time advantage in 12 datasets (when compared with the VFDT). The SVFDT-II is faster than the VFDT in 15 datasets. When using GI, accuracy values are also very close and the Kappa M performance seems to be less impacted since only in the ctu_13 dataset the SVFDT-I performed significantly worse. However, this is due to the fact that fewer splits are made in general when using GI and $\tau = 0$. When considering memory consumption, many deadlocks occurred, which made the trees perform no splits in 12 datasets (see Table 14). Nonetheless, the SVFDT-I was able to reduce memory consumption in 9 datasets. The SVFDT-II reduced memory in 7 datasets. Considering training time, the SVFDT-I is the fastest in 16 datasets (presenting an increase in speed of at least 15% in 6 datasets). The SVFDT-II is slightly faster than the VFDT in 15 datasets.

Tables 13 and 14, in Appendix C, present the performance of the algorithms using the same hyperparameter configuration. When using H and IG, the SVFDT-I had the lowest relative accuracy of 0.9737 for the *rbf_1kk* dataset and the highest of 1.0001 for the *ctu_10* dataset and 1 for the *ctu_4*, *ctu_7*, *ctu_8*, *ctu_11*, *hyper*, *led_10*, *led_20*, *rbf_250k(50)*, *sea* and *usenet*. Additionally, the mean relative accuracy obtained was 0.9949. On the other hand, for the SVFDT-II, the mean relative accuracy was 0.9994. Its lowest relative accuracy was 0.9844 for the *covType* dataset, with its highest value being 1.0042 for the *airlines* dataset. The Kappa M performance obtained by the SVFDT-I is also significantly impacted in the CTU datasets. From the memory perspective, due to deadlock situations introduced by not performing tiebreaks, no splits occurred in 7 datasets. However, the SVFDT-I was able to reduce memory consumption in 17 datasets. In only in the *airlines* dataset the SVFDT-I performed more splits, likely due to the fact that when holding tree growth, a different split was performed, which made subsequent split attempts validate the constraints more often. The SVFDT-II presented a similar pattern, reducing memory in 12 datasets. The mean relative memory consumption obtained by the SVFDT-I and II were 0.6258 and 0.9309 respectively (considering the 7 datasets

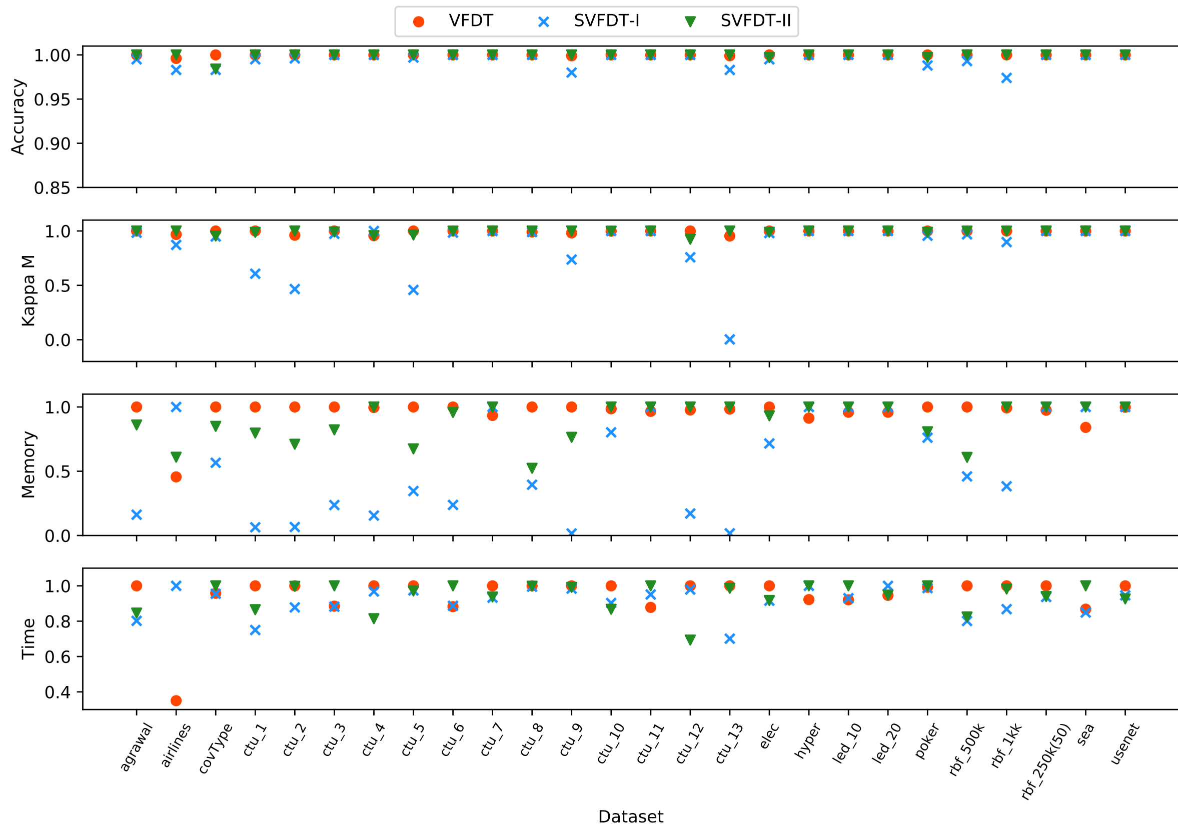


Figure 27 – Performance metrics of all algorithms by dataset for $\tau = 0$, $GP = 100$, H and IG. The metrics are scaled by dividing by the largest value for each dataset. Larger values are better for accuracy and Kappa M, and lower values are better for memory and time.

with deadlock situations). Lastly, the SVFDT-I and II are faster than the VFDT in 20 and 17 datasets, having mean relative time values of 1.0154 (due to the *airlines* dataset) and 0.9649. Considering GI, it is possible to see that the three algorithms present very similar accuracy values, with the SVFDT-I having the lowest relative accuracy of 0.9413 for the *poker* dataset and the highest of 1.0001 for the *rbf_500* dataset. The mean relative accuracy obtained by the SVFDT-I was 0.9955. The lowest relative accuracy obtained by the SVFDT-II was 0.9727 also for the *poker* dataset, achieving 2.5% higher relative accuracy in the *covType* data. Although the SVFDT-I was not so outperformed from the Kappa M perspective in the CTU datasets, this is due to the fact that in 7 datasets, 0 or at most 1 split was made. In 12 datasets, due to the deadlock, the number of splits were 0 for all trees. Likewise, for 2 datasets, only one split occurred. In this way, memory could not be reduced. For the *agrawal* and *airlines* datasets, the SVFDT-I performed additional splits, probably due to the tree selecting a different split attribute when the split conditions were met. The SVFDT-II performed more splits only in the *poker* dataset. Despite all these situations, the SVFDT-I reduced memory consumption in 8 datasets and the SVFDT-II reduced in 5 datasets. Lastly, considering time costs, the SVFDT-II ranked as the fastest in 21 datasets. The SVFDT-II was also faster than the VFDT for 17 datasets.

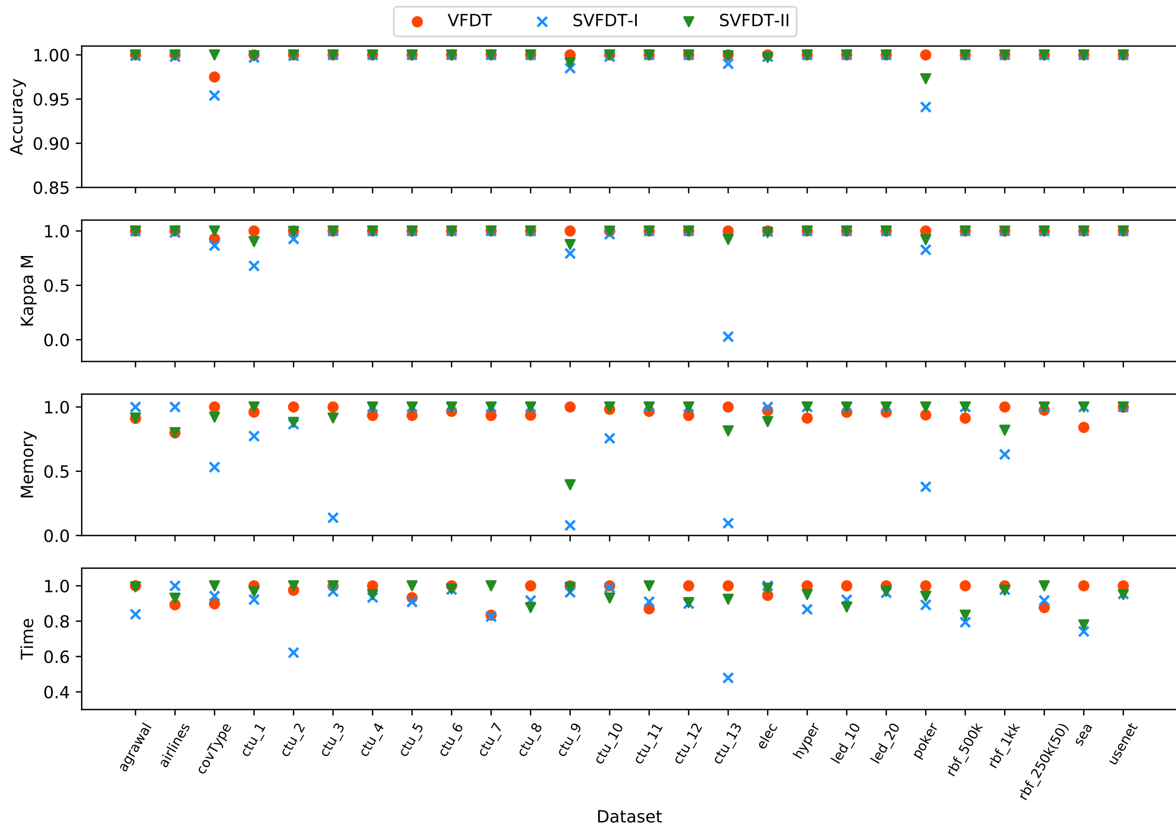


Figure 28 – Performance metrics of all algorithms by dataset for $\tau = 0$, $GP = 100$ and GI. The metrics are scaled by dividing by the largest value for each dataset. Larger values are better for accuracy and Kappa M, and lower values are better for memory and time.

By analysing these results, it is possible to see that the SVFDTs can reduce memory consumption further than tuning the hyperparameter values of the VFDT. Likewise, it also presents the same patterns of maintaining accuracy and training time.

6.5 Exploring the impact of each heuristic

Another question that may arise is: "which is the impact of each heuristic and their possible combinations?". To address this, the SVFDTs were evaluated on all the 26 datasets with all the possible combinations of constraints and skip mechanisms (presented in Table 6).

Recall that:

- Constraint 1: minimum impurity according to current leaves.
- Constraint 2: minimum impurity according to the whole tree creation process.
- Constraint 3: minimum relevance of split feature.

Table 6 – Possible combinations for the constraints and skip mechanisms for the SVFDTs.

Algorithm	Name	Constraint 1?	Constraint 2?	Constraint 3?	Constraint 4?	Skip mech. 1?	Skip mech. 2?
SVFDT-I	A	✓					
	B		✓				
	C			✓			
	D				✓		
	E	✓	✓				
	F	✓		✓			
	G	✓			✓		
	H		✓	✓			
	I		✓		✓		
	J			✓	✓		
	K	✓	✓	✓			
	L	✓	✓		✓		
	M	✓		✓	✓		
	N		✓	✓	✓		
SVFDT-II	O	✓	✓	✓	✓	✓	
	P	✓	✓	✓	✓		✓

- Constraint 4: minimum amount of instances.
- Skip mechanism 1: high impurity according to the whole tree creation process.
- Skip mechanism 2: high relevance of split feature.

Their performance metrics are presented in Tables 7 and 8.

First, it is possible to see that independently from the impurity and gain metrics used, using any constraint produces a tree smaller than the VFDT. Also, note that accuracy and time values are very similar and present a similar pattern as discussed in other experiments.

When using H and IG, and considering a single constraint (cases A-D), Constraint 4 seems to be the most impactful, followed by Constraints 1, 2 and 3. It is interesting to see that the constraints based on impurity impacted tree growth more than the ones based on reducing this impurity, although they are correlated. When using combinations of 2 constraints (cases E-J), all cases produced smaller trees than when using a single constraint. However, the combination of Constraints 2 and 3 (case H) resulted in a tree that is almost the same size as using Constraint 2 alone. An interesting phenomenon occurred when combining three constraints (cases K-N). When not using Constraint 4 (case K), tree sizes were greater than when using any pair combination of the constraints. The interaction between allowing only splits with minimum impurity according to current and past statistics and minimum relevance is not favourable from the memory perspective. Nonetheless, the combination of all constraints (SVFDT-I) results in the smallest trees. This indicates that all four constraints contribute to reducing memory consumption (although some are more important than others) and using all of them provides the best results. Using only one skip mechanism (cases O and P) also proved to be less efficient than combining the two (SVFDT-II), although they seem to have a very similar impact when used individually.

Considering GI, the rank of the impacts of using only one constraint is the same as when using H and IG and combining two constraints also produce smaller trees than when using a single constraint. However, the gains observed by combining two constraints are higher for all cases. Furthermore, case E (Constraints 1 and 2) performed worse than case H (Constraints 2 and 3). When using three constraints, case K is also the worst, and, although it is better than using a single constraint, combining two constraints resulted in lower memory consumption (cases G, I and J). Nonetheless, the combination of all four constraints also yields the best results (SVFDT-I). Employing a single skip mechanism (cases O and P) is also more ineffective than using both together, which indicates that they activate in different scenarios. Likewise, their individual impact is also very similar.

Table 7 – Performance metrics for different combinations of constraints and skip mechanisms using for $\tau = 0.05$, $GP = 1000$, entropy and IG.

Name	Accuracy	Kappa M	Memory (MB)	Splits	Time (s)
VFDT	0.8892	0.6571	3.267 ± 6.018	86.92 ± 101.75	60.95 ± 61.71
A	0.8886	0.6570	2.759 ± 5.863	60.92 ± 53.98	61.36 ± 61.66
B	0.8885	0.6572	2.882 ± 5.236	69.73 ± 65.58	62.23 ± 62.38
C	0.8897	0.6598	3.215 ± 5.981	83.69 ± 101.34	61.62 ± 61.22
D	0.8868	0.5992	2.214 ± 4.632	56.88 ± 77.63	60.88 ± 61.29
E	0.8880	0.6581	2.568 ± 5.199	56.69 ± 49.11	61.42 ± 62.39
F	0.8890	0.6599	2.757 ± 5.831	59.35 ± 54.48	61.35 ± 62.81
G	0.8846	0.5608	1.602 ± 4.472	27.92 ± 36.64	**60.76 ± 65.48**
H	0.8882	0.6569	2.833 ± 5.261	68.46 ± 66.03	61.42 ± 62.25
I	0.8847	0.5922	1.297 ± 2.413	25.96 ± 23.70	61.40 ± 61.97
J	0.8861	0.5945	1.934 ± 3.875	49.04 ± 72.48	62.83 ± 63.57
K	0.8876	0.6577	2.611 ± 5.246	57.46 ± 50.32	62.57 ± 62.48
L	0.8840	0.5602	**1.019 ± 2.330**	**19.15 ± 20.36**	61.64 ± 64.85
M	0.8832	0.5516	1.396 ± 3.701	23.19 ± 29.75	61.83 ± 64.07
N	0.8840	0.5901	**1.107 ± 1.895**	**23.00 ± 20.47**	64.01 ± 65.38
O	0.8856	0.6504	1.575 ± 2.959	31.77 ± 27.24	**60.61 ± 62.14**
P	0.8870	0.6467	1.409 ± 2.211	32.69 ± 26.44	**60.15 ± 60.36**
SVFDT-I	0.8832	0.5515	**0.857 ± 1.772**	**17.00 ± 16.41**	61.01 ± 62.84
SVFDT-II	0.8874	0.6542	1.783 ± 3.291	38.31 ± 34.16	61.12 ± 62.05

Table 8 – Performance metrics for different combinations of constraints and skip mechanisms using for $\tau = 0.05$, $GP = 1000$ and GI.

Name	Accuracy	Kappa M	Memory (MB)	Splits	Time (s)
VFDT	0.8962	0.6696	4.796 ± 7.080	129.46 ± 143.39	**59.53 ± 58.70**
A	0.8950	0.6769	3.876 ± 6.715	88.65 ± 99.39	60.82 ± 58.96
B	0.8930	0.6647	4.066 ± 6.049	105.96 ± 107.17	60.88 ± 59.46
C	0.8938	0.6391	4.256 ± 6.360	112.88 ± 123.11	62.12 ± 60.97
D	0.8944	0.6269	3.486 ± 5.424	91.23 ± 115.43	61.16 ± 60.38
E	0.8923	0.6722	3.546 ± 5.948	83.62 ± 85.04	62.04 ± 62.00
F	0.8918	0.6074	3.120 ± 5.954	66.62 ± 83.56	61.48 ± 61.73
G	0.8912	0.5817	2.178 ± 4.881	38.88 ± 63.54	60.02 ± 60.77
H	0.8899	0.6254	3.301 ± 5.129	82.77 ± 81.88	63.92 ± 63.92
I	0.8864	0.6119	1.857 ± 2.946	39.77 ± 40.68	61.96 ± 61.09
J	0.8905	0.6026	2.335 ± 3.677	58.58 ± 68.76	62.20 ± 59.25
K	0.8888	0.5964	2.707 ± 5.110	58.31 ± 65.13	63.19 ± 63.98
L	0.8853	0.5639	1.497 ± 2.930	27.19 ± 35.90	61.73 ± 63.82
M	0.8872	0.5394	**1.465 ± 3.292**	**24.27 ± 32.76**	61.32 ± 60.54
N	0.8853	0.5894	**1.421 ± 2.166**	**29.54 ± 29.09**	62.30 ± 59.65
O	0.8881	0.6483	2.094 ± 3.387	46.27 ± 44.25	61.61 ± 60.21
P	0.8884	0.5441	2.318 ± 4.329	50.58 ± 66.62	60.42 ± 60.67
SVFDT-I	0.8840	0.5218	**1.121 ± 2.126**	**20.46 ± 25.61**	**59.89 ± 60.11**
SVFDT-II	0.8925	0.6571	3.201 ± 5.204	75.69 ± 90.10	**59.45 ± 58.60**

7 CONCLUSION

This work presented two versions of a modification of the VFDT called SVFDT. The VFDT is a DT specifically build to data streams which is capable of learning incrementally using less computation resources and yielding high predictive performance. In general, the tree should not grow in parts where there are low class uncertainty, or if that split would not yield a significant reduction in this class uncertainty or only a small number of instances were seen. Additionally, skip mechanisms were also created, resulting in the SVFDT-II, which allows growth when class uncertainty is too high or will greatly reduce if a split is performed. The SVFDT modifies it by adding constraints to hold tree growth when it is not needed. While the SVFDT-II does not compromise predictive performance, the SVFDT-I seems to perform poorly on highly imbalanced datasets. In other scenarios, the SVFDT-I performs very similarly to the VFDT. In this sense, when considering a not highly imbalanced problem, using the SVFDT-I is the best alternative, whereas, in imbalanced problems, the SVFDT-II is a better solution.

The proposed algorithms were tested in 26 different benchmark datasets while varying hyperparameters and using different heuristic metrics. First, the algorithms were compared using the mean performance values obtained over multiple datasets for each hyperparameter setup. It was possible to see that the SVFDTs yield very similar predictive performance, being faster in many cases and greatly reducing memory consumption. Additionally, it also presented time gains in some scenarios. By creating boxplots of these metrics, the variation of accuracy, memory consumption and time costs were assessed. A heatmap of each performance metric was constructed considering GP and τ values, in which it is possible to see the better performing settings and their patterns. Likewise, by stacking all heatmaps together and giving each performance metric the same importance, optimal points emerged. In some scenarios, predictive performance or memory costs can have different weights. Although we did not explore this in this work, it is possible to address this scenario by creating a global heatmap using a weighted mean. Then, by using the Friedman test and post-hoc Nemenyi analysis, the statistical difference in each performance metric was tested. After that, The worst and best hyperparameter setups for the SVFDTs were considered and analysed more in-depth. In both cases, the SVFDTs reduce the memory used by the VFDT across datasets, and, in the best case, memory consumption was heavily reduced. Furthermore, the question of whether or not it could be possible to tune a VFDT to produce smaller trees than the SVFDTs was addressed. Despite deadlock situations, where no splits are made due to badly selected hyperparameter values, the latter was able to reduce memory by a large margin. Lastly, an in-depth analysis of the impact of the constraints and skip mechanisms proposed was performed.

It was possible to assert that each constraint and skip mechanism have their own impact on tree size, with the combination of all constraints (SVFDT-I) and all skip mechanisms (SVFDT-II) yielding the best results.

As future work, we will propose new constraints using predictive performance to better handle imbalanced datasets.

BIBLIOGRAPHY

[1] KRAWCZYK, B. et al. Ensemble learning for data stream analysis: A survey. *Information*, p. 1–86, 2017.

[2] PFAHRINGER, B.; HOLMES, G.; KIRKBY, R. Handling numeric attributes in hoeffding trees. In: *Proceedings of the 12th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*. Berlin, Heidelberg: Springer-Verlag, 2008. (PAKDD'08), p. 296–307. ISBN 3-540-68124-8, 978-3-540-68124-3.

[3] COSTA, V. G. T. da et al. Detecting mobile botnets through machine learning and system calls analysis. In: *2017 IEEE International Conference on Communications (ICC)*. [S.l.: s.n.], 2017. p. 1–6. ISSN 1938-1883.

[4] GEIGER, A. Are we ready for autonomous driving? the kitti vision benchmark suite. In: *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Washington, DC, USA: IEEE Computer Society, 2012. (CVPR '12), p. 3354–3361. ISBN 978-1-4673-1226-4. Disponível em: <http://dl.acm.org/citation.cfm?id=2354409.2354978>.

[5] LITJENS, G. et al. A survey on deep learning in medical image analysis. *Medical Image Analysis*, v. 42, p. 60 – 88, 2017. ISSN 1361-8415. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1361841517301135>.

[6] KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. USA: Curran Associates Inc., 2012. (NIPS'12), p. 1097–1105. Disponível em: <http://dl.acm.org/citation.cfm?id=2999134.2999257>.

[7] GIRSHICK, R. B. et al. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013. Disponível em: <http://arxiv.org/abs/1311.2524>.

[8] PANG, B.; LEE, L. Opinion mining and sentiment analysis. *Foundations and Trends® in Information Retrieval*, v. 2, n. 1–2, p. 1–135, 2008. ISSN 1554-0669. Disponível em: <http://dx.doi.org/10.1561/1500000011>.

[9] BAHDANAU, D.; CHO, K.; BENGIO, Y. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014. Disponível em: <http://arxiv.org/abs/1409.0473>.

[10] GATYS, L. A.; ECKER, A. S.; BETHGE, M. A neural algorithm of artistic style. *CoRR*, abs/1508.06576, 2015. Disponível em: <http://arxiv.org/abs/1508.06576>.

[11] GAMA, J. et al. Knowledge Discovery from Data Streams. *Web Intelligence and Security - Advances in Data and Text Mining Techniques for Detecting and Preventing Terrorist Activities on the Web*, p. 125–138, 2010. ISSN 1088-467X.

[12] GABER, M. M.; ZASLAVSKY, A.; KRISHNASWAMY, S. Mining data streams: A review. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 34, n. 2, p. 18–26, jun. 2005. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/1083784.1083789>.

[13] PIETRUCZUK, L. et al. How to adjust an ensemble size in stream data mining? *Information Sciences*, Elsevier Inc., v. 381, p. 46–54, 2017. ISSN 00200255. Disponível em: <http://dx.doi.org/10.1016/j.ins.2016.10.028>.

[14] DOMINGOS, P.; HULTEN, G. Mining high-speed data streams. *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, p. 71–80, 2000. ISSN 10844627.

[15] HOLMES, G.; RICHARD, K.; PFAHRINGER, B. Tie-breaking in Hoeffding trees. In: *Proceedings of the Second International Workshop on Knowledge Discovery from Data Streams*. [S.l.: s.n.], 2005.

[16] YANG, H.; FONG, S. Moderated vfdt in stream mining using adaptive tie threshold and incrementhttps://www.overleaf.com/project/5bc88ea9381492371c6d9968al pruning. In: CUZZOCREA, A.; DAYAL, U. (Ed.). *Data Warehousing and Knowledge Discovery*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 471–483. ISBN 978-3-642-23544-3.

[17] YANG, H.; FONG, S. Optimized very fast decision tree with balanced classification accuracy and compact tree size. In: *The 3rd International Conference on Data Mining and Intelligent Information Technology Applications*. [S.l.: s.n.], 2011. p. 57–64.

[18] YANG, H.; FONG, S. Incremental optimization mechanism for constructing a decision tree in data stream mining. *Mathematical Problems in Engineering*, Hindawi Limited, v. 2013, p. 1–14, 2013. Disponível em: <https://doi.org/10.1155/2013/580397>.

[19] BREIMAN, L. Random Forests. *Machine learning*, v. 45.1, p. 5–32, 2001. ISSN 1098-6596.

[20] FRANK, E. *Pruning decision trees and lists*. Tese (Doutorado) — University of Waikato, 1 2000.

[21] BISHOP, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN 0387310738.

[22] FLACH, P. *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. New York, NY, USA: Cambridge University Press, 2012. ISBN 1107422221, 9781107422223.

[23] GAMA, J. et al. *Inteligência artificial: uma abordagem de aprendizado de máquina*. Grupo Gen - LTC, 2011. ISBN 9788521618805. Disponível em: <https://books.google.com.br/books?id=4DwelAEACAAJ>.

[24] MITCHELL, T. *Machine Learning*. McGraw-Hill Education, 1997. (McGraw-Hill international editions - computer science series). ISBN 9780070428072. Disponível em: <https://books.google.com.br/books?id=xOGAngEACAAJ>.

[25] BREIMAN, L. et al. *Classification and Regression Trees*. Taylor & Francis, 1984. (The Wadsworth and Brooks-Cole statistics-probability series). ISBN 9780412048418. Disponível em: <https://books.google.com.br/books?id=JwQx-WOmSyQC>.

[26] QUINLAN, J. R. Induction of decision trees. *Mach. Learn.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 1, n. 1, p. 81–106, mar. 1986. ISSN 0885-6125. Disponível em: <http://dx.doi.org/10.1023/A:1022643204877>.

[27] QUINLAN, J. R. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN 1-55860-238-0.

[28] RAILEANU, L. E.; STOFFEL, K. Theoretical comparison between the gini index and information gain criteria. *Annals of Mathematics and Artificial Intelligence*, v. 41, n. 1, p. 77–93, May 2004. ISSN 1573-7470. Disponível em: <https://doi.org/10.1023/B:AMAI.0000018580.96245.c6>.

[29] SRIVASTAVA, N. et al. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, v. 15, p. 1929–1958, 2014. Disponível em: <http://jmlr.org/papers/v15/srivastava14a.html>.

[30] LEWIS, D. D. Naive (bayes) at forty: The independence assumption in information retrieval. In: NÉDELLEC, C.; ROUVEIROL, C. (Ed.). *Machine Learning: ECML-98*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. p. 4–15. ISBN 978-3-540-69781-7.

[31] PATIL, T. R. Performance Analysis of Naive Bayes and J48 Classification Algorithm for Data Classification. *International Journal Of Computer Science And Applications, ISSN: 0974-1011*, v. 6, n. 2, p. 256–261, 2013.

[32] KHAMASSI, I. et al. Discussion and review on evolving data streams and concept drift adapting. *Evolving Systems*, v. 9, n. 1, p. 1–23, Mar 2018. ISSN 1868-6486. Disponível em: <https://doi.org/10.1007/s12530-016-9168-2>.

[33] WEBB, G. I. et al. Characterizing concept drift. *Data Mining and Knowledge Discovery*, v. 30, n. 4, p. 964–994, Jul 2016. ISSN 1573-756X. Disponível em: <https://doi.org/10.1007/s10618-015-0448-4>.

[34] MINKU, L. L.; WHITE, A. P.; YAO, X. The impact of diversity on online ensemble learning in the presence of concept drift. *IEEE Transactions on Knowledge and Data Engineering*, v. 22, n. 5, p. 730–742, May 2010. ISSN 1041-4347.

[35] DONGRE, P. B.; MALIK, L. G. A review on real time data stream classification and adapting to various concept drift scenarios. In: *2014 IEEE International Advance Computing Conference (IACC)*. [S.l.: s.n.], 2014. p. 533–537.

[36] OZA, N. C. Online bagging and boosting. In: *2005 IEEE International Conference on Systems, Man and Cybernetics*. [S.l.: s.n.], 2005. v. 3, p. 2340–2345 Vol. 3. ISSN 1062-922X.

[37] PFAHRINGER, B.; HOLMES, G.; KIRKBY, R. New options for hoeffding trees. In: ORGUN, M. A.; THORNTON, J. (Ed.). *AI 2007: Advances in Artificial Intelligence*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 90–99. ISBN 978-3-540-76928-6.

[38] BRZEZINSKI, D.; STEFANOWSKI, J. Combining block-based and online methods in learning ensembles from concept drifting data streams. *Information Sciences*, v. 265, p. 50 – 67, 2014. ISSN 0020-0255. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0020025513008554>.

[39] BIFET, A.; HOLMES, G.; PFAHRINGER, B. Leveraging bagging for evolving data streams. In: *Proceedings of the 2010 European Conference on Machine Learning and Knowledge Discovery in Databases: Part I*. Berlin, Heidelberg: Springer-Verlag, 2010. (ECML PKDD'10), p. 135–150. ISBN 3-642-15879-X, 978-3-642-15879-7. Disponível em: <http://dl.acm.org/citation.cfm?id=1888258.1888275>.

[40] GOMES, H. M. et al. Adaptive random forests for evolving data stream classification. *Machine Learning*, v. 106, n. 9, p. 1469–1495, Oct 2017. ISSN 1573-0565. Disponível em: <https://doi.org/10.1007/s10994-017-5642-8>.

[41] HOEFFDING, W. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, v. 58, n. 301, p. 13–30, March 1963. Disponível em: <http://www.jstor.org/stable/2282952?>

[42] MARON, O.; MOORE, A. W. Hoeffding races: Accelerating model selection search for classification and function approximation. In: COWAN, J. D.; TESAURO, G.; ALSPECTOR, J. (Ed.). *NIPS*. Morgan Kaufmann, 1993. p. 59–66. ISBN 1-55860-322-0. Disponível em: <http://dblp.uni-trier.de/db/conf/nips/nips1993.html#MaronM93>.

[43] JAWORSKI, M.; DUDA, P.; RUTKOWSKI, L. New splitting criteria for decision trees in stationary data streams. *IEEE Transactions on Neural Networks and Learning Systems*, v. 29, n. 6, p. 2516–2529, June 2018.

[44] GAMA, J. a.; ROCHA, R.; MEDAS, P. Accurate decision trees for mining high-speed data streams. In: *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: ACM, 2003. (KDD '03), p. 523–528. ISBN 1-58113-737-0. Disponível em: <http://doi.acm.org/10.1145/956750.956813>.

[45] WELFORD, B. P. Note on a method for calculating corrected sums of squares and products. *Technometrics*, Taylor & Francis, v. 4, n. 3, p. 419–420, 1962. Disponível em: <http://amstat.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022>.

[46] LING, R. F. Comparison of several algorithms for computing sample means and variances. *Journal of the American Statistical Association*, [American Statistical Association, Taylor & Francis, Ltd.], v. 69, n. 348, p. 859–866, 1974. ISSN 01621459. Disponível em: <http://www.jstor.org/stable/2286154>.

[47] WELFORD, B. P. Note on a method for calculating corrected sums of squares and products. *Technometrics*, [Taylor & Francis, Ltd., American Statistical Association, American Society for Quality], v. 4, n. 3, p. 419–420, 1962. ISSN 00401706. Disponível em: <http://www.jstor.org/stable/1266577>.

[48] KNUTH, D. E. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-89683-4.

[49] LATIF, R. et al. EVFDT : An Enhanced Very Fast Decision Tree Algorithm for Detecting Distributed Denial of Service Attack in Cloud- Assisted Wireless Body Area Network. *Mobile Information Systems*, v. 2015, 2015. ISSN 1574017X.

[50] LOSING, V.; WERSING, H.; HAMMER, B. Enhancing very fast decision trees with local split-time predictions. In: *2018 IEEE International Conference on Data Mining (ICDM)*. [S.l.: s.n.], 2018. p. 287–296. ISSN 2374-8486.

[51] HULTEN, G.; SPENCER, L.; DOMINGOS, P. Mining time-changing data streams. *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '01*, p. 97–106, 2001. ISSN 10224653.

[52] AGRAWAL, R.; SWAMI, A.; IMIELINSKI, T. Database Mining: A Performance Perspective. *IEEE Transactions on Knowledge and Data Engineering*, v. 5, n. 6, p. 914–925, 1993. ISSN 10414347.

[53] BIFET, A. et al. MOA: massive online analysis. *Journal of Machine Learning Research*, v. 11, p. 1601–1604, 2010.

[54] GARCíA, S. et al. An empirical comparison of botnet detection methods. *Computers & Security*, v. 45, n. Supplement C, p. 100 – 123, 2014. ISSN 0167-4048.

[55] HALL, M. et al. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, ACM, New York, NY, USA, v. 11, n. 1, p. 10–18, nov. 2009. ISSN 1931-0145.

[56] STREET, W. N.; KIM, Y. A streaming ensemble algorithm (SEA) for large-scale classification. *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '01*, v. 4, p. 377–382, 2001.

[57] KATAKIS, I.; TSOUMAKAS, G.; VLAHAVAS, I. Tracking recurring contexts using ensemble classifiers: An application to email filtering. *Knowledge and Information Systems*, v. 22, n. 3, p. 371–391, 2010. ISSN 02191377.

[58] BIFET, A. et al. Efficient online evaluation of big data stream classifiers. In: *Proc. of the XXI ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: ACM, 2015. (KDD '15), p. 59–68. ISBN 978-1-4503-3664-2.

[59] ROSSUM, G. *Python Reference Manual*. Amsterdam, The Netherlands, The Netherlands, 1995.

[60] FRIEDMAN, M. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, [American Statistical Association, Taylor & Francis, Ltd.], v. 32, n. 200, p. 675–701, 1937. ISSN 01621459. Disponível em: <http://www.jstor.org/stable/2279372>.

[61] NEMENYI, P. *Distribution-free Multiple Comparisons*. Tese (Doutorado) — Princeton University, 1963.

[62] DEMŠAR, J. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research*, JMLR. org, v. 7, p. 1–30, 2006.

Appendix

APPENDIX A – WORST AND BEST POSSIBLE SCENARIOS FOR THE SVFDTS

Table 9 – Performance metrics of all algorithms by dataset for $\tau = 0.01$, $GP = 800$, entropy and IG (worst case).

Dataset	Algorithm	Accuracy	Kappa M	Memory (MB)	Splits	Time (s)	Rel. Accuracy	Rel. Memory	Rel. Time
	VFDT	0.9487	0.8437	0.278	15	68.60 ± 0.15	-	-	-
agrawal	SVFDT-I	0.9484	0.8427	0.175	9	66.81 ± 0.15	0.9997	0.6296	0.9740
	SVFDT-II	0.9484	0.8427	0.175	9	66.94 ± 0.08	0.9997	0.6296	0.9758
	VFDT	0.6528	0.2205	1.939	14	96.33 ± 0.30	-	-	-
airlines	SVFDT-I	0.6534	0.2218	1.679	12	88.11 ± 0.30	1.0009	0.8659	0.9147
	SVFDT-II	0.6535	0.2222	1.940	14	116.62 ± 0.40	1.0011	1.0007	1.2106
	VFDT	0.6774	0.3704	2.092	29	1141.53 ± 9.28	-	-	-
covType	SVFDT-I	0.6951	0.4050	2.720	38	1111.49 ± 6.63	1.0262	1.3003	0.9737
	SVFDT-II	0.7104	0.4349	3.556	50	1139.50 ± 7.80	1.0488	1.6999	0.9982
	VFDT	0.9975	0.8296	0.739	35	180.56 ± 0.26	-	-	-
ctu_1	SVFDT-I	0.9892	0.2550	0.095	4	143.39 ± 0.19	0.9916	0.1288	0.7941
	SVFDT-II	0.9975	0.8265	0.829	29	177.26 ± 0.40	1.0000	1.1219	0.9817
	VFDT	0.9944	0.5148	0.872	27	113.73 ± 0.58	-	-	-
ctu_2	SVFDT-I	0.9886	0.0188	0.076	3	114.42 ± 5.62	0.9942	0.0875	1.0060
	SVFDT-II	0.9935	0.4411	0.646	19	113.69 ± 0.65	0.9991	0.7408	0.9996
	VFDT	0.9991	0.8432	0.863	31	297.63 ± 1.39	-	-	-
ctu_3	SVFDT-I	0.9992	0.8512	0.574	15	300.29 ± 13.47	1.0000	0.6650	1.0089
	SVFDT-II	0.9991	0.8421	0.792	27	303.69 ± 17.23	1.0000	0.9174	1.0204
	VFDT	0.9986	0.4015	0.422	7	69.97 ± 3.96	-	-	-
ctu_4	SVFDT-I	0.9983	0.2492	0.370	4	67.94 ± 2.83	0.9996	0.8753	0.9710
	SVFDT-II	0.9986	0.4015	0.424	7	69.31 ± 4.08	1.0000	1.0031	0.9906
	VFDT	0.9958	0.3929	0.094	4	8.01 ± 0.43	-	-	-
ctu_5	SVFDT-I	0.9957	0.3807	0.076	3	7.85 ± 0.34	0.9999	0.8152	0.9804
	SVFDT-II	0.9961	0.4317	0.110	5	8.00 ± 0.44	1.0003	1.1815	0.9991
	VFDT	0.9990	0.8775	0.443	8	32.87 ± 1.52	-	-	-
ctu_6	SVFDT-I	0.9945	0.3315	0.094	4	33.33 ± 1.92	0.9955	0.2114	1.0139
	SVFDT-II	0.9990	0.8838	0.427	7	33.60 ± 1.91	1.0001	0.9621	1.0221
	VFDT	0.9994	-0.0476	0.037	1	6.41 ± 0.36	-	-	-
ctu_7	SVFDT-I	0.9994	-0.0476	0.038	1	6.33 ± 0.26	1.0000	1.0359	0.9872
	SVFDT-II	0.9994	-0.0476	0.038	1	6.42 ± 0.35	1.0000	1.0359	1.0020
	VFDT	0.9997	0.8670	0.940	20	183.00 ± 1.10	-	-	-
ctu_8	SVFDT-I	0.9985	0.2815	0.219	11	171.69 ± 0.91	0.9988	0.2331	0.9382
	SVFDT-II	0.9997	0.8436	0.643	19	183.00 ± 1.33	1.0000	0.6844	1.0000
	VFDT	0.9838	0.8171	1.794	61	134.39 ± 7.44	-	-	-
ctu_9	SVFDT-I	0.9567	0.5112	0.076	3	121.88 ± 5.19	0.9724	0.0426	0.9069
	SVFDT-II	0.9833	0.8114	1.250	36	132.39 ± 9.24	0.9995	0.6970	0.9851
	VFDT	0.9977	0.9722	0.164	8	92.76 ± 0.68	-	-	-
ctu_10	SVFDT-I	0.9925	0.9071	0.408	6	94.09 ± 0.44	0.9947	2.4946	1.0143
	SVFDT-II	0.9977	0.9721	0.147	7	91.51 ± 0.67	1.0000	0.8975	0.9864
	VFDT	0.9918	0.8923	0.055	2	6.70 ± 0.38	-	-	-
ctu_11	SVFDT-I	0.9919	0.8940	0.056	2	6.76 ± 0.49	1.0001	1.0242	1.0084
	SVFDT-II	0.9918	0.8923	0.056	2	6.63 ± 0.28	1.0000	1.0241	0.9891
	VFDT	0.9941	0.1208	0.164	8	20.22 ± 0.35	-	-	-
ctu_12	SVFDT-I	0.9935	0.0240	0.058	2	17.30 ± 0.50	0.9994	0.3519	0.8557
	SVFDT-II	0.9942	0.1232	0.114	5	19.71 ± 0.10	1.0000	0.6942	0.9749
	VFDT	0.9923	0.6304	3.133	105	141.49 ± 0.57	-	-	-
ctu_13	SVFDT-I	0.9792	-0.0003	0.058	2	84.44 ± 4.79	0.9868	0.0184	0.5968
	SVFDT-II	0.9896	0.4994	1.634	71	125.20 ± 7.16	0.9973	0.5215	0.8848
	VFDT	0.7897	0.5047	0.329	11	1.91 ± 0.01	-	-	-
elec	SVFDT-I	0.7906	0.5068	0.211	9	1.81 ± 0.01	1.0011	0.6429	0.9481
	SVFDT-II	0.7944	0.5156	0.319	10	2.00 ± 0.01	1.0058	0.9698	1.0501
	VFDT	0.9120	0.8240	0.054	3	9.05 ± 0.03	-	-	-
hyper	SVFDT-I	0.9123	0.8245	0.055	3	9.03 ± 0.04	1.0003	1.0245	0.9975
	SVFDT-II	0.9120	0.8240	0.055	3	9.08 ± 0.04	1.0000	1.0244	1.0028
	VFDT	0.7281	0.6978	0.335	10	1461.11 ± 2.28	-	-	-
led_10	SVFDT-I	0.7243	0.6936	0.123	3	1507.33 ± 1.89	0.9948	0.3679	1.0316
	SVFDT-II	0.7246	0.6938	0.306	9	1465.43 ± 1.72	0.9951	0.9132	1.0030
	VFDT	0.4977	0.4416	0.244	7	1451.54 ± 6.07	-	-	-
led_20	SVFDT-I	0.4964	0.4402	0.123	3	1501.45 ± 3.51	0.9974	0.5054	1.0344
	SVFDT-II	0.4953	0.4390	0.154	4	1478.41 ± 9.45	0.9952	0.6305	1.0185
	VFDT	0.6655	0.3296	1.134	44	299.07 ± 2.94	-	-	-
poker	SVFDT-I	0.6626	0.3237	0.942	36	306.93 ± 2.39	0.9956	0.8308	1.0263
	SVFDT-II	0.6633	0.3252	0.990	38	302.90 ± 2.22	0.9967	0.8734	1.0128
	VFDT	0.7952	0.5578	0.375	27	19.91 ± 1.74	-	-	-
rbf_500k	SVFDT-I	0.7686	0.5002	0.138	9	19.12 ± 1.38	0.9665	0.3680	0.9603
	SVFDT-II	0.7942	0.5555	0.323	23	19.02 ± 1.36	0.9987	0.8611	0.9554
	VFDT	0.8405	0.6560	0.657	48	34.12 ± 0.11	-	-	-
rbf_1kk	SVFDT-I	0.7997	0.5682	0.179	12	36.13 ± 0.15	0.9515	0.2724	1.0591
	SVFDT-II	0.8372	0.6489	0.565	41	34.13 ± 0.14	0.9961	0.8597	1.0004
	VFDT	0.9591	0.9173	0.671	12	27.79 ± 0.85	-	-	-
rbf_250k(50)	SVFDT-I	0.9551	0.9093	0.363	6	26.66 ± 0.70	0.9959	0.5406	0.9594
	SVFDT-II	0.9609	0.9211	0.863	16	27.11 ± 0.59	1.0020	1.2853	0.9755
	VFDT	0.8450	0.5845	0.027	3	1.41 ± 0.14	-	-	-
sea	SVFDT-I	0.8448	0.5840	0.015	1	1.38 ± 0.11	0.9998	0.5560	0.9771
	SVFDT-II	0.8448	0.5840	0.022	2	1.41 ± 0.15	0.9998	0.8025	0.9976
	VFDT	0.5363	0.0645	0.622	0	56.16 ± 0.31	-	-	-
usenet	SVFDT-I	0.5363	0.0645	0.623	0	56.25 ± 0.26	1.0000	1.0021	1.0014
	SVFDT-II	0.5363	0.0645	0.623	0	56.10 ± 0.45	1.0000	1.0021	0.9989

Table 10 – Performance metrics of all algorithms by dataset for $\tau = 0.20$, $GP = 100$, entropy and IG (best case).

Dataset	Algorithm	Accuracy	Kappa M	Memory (MB)	Splits	Time (s)	Rel. Accuracy	Rel. Memory	Rel. Time
	VFDT	0.9448	0.8317	44.408	1261	86.47 ± 5.17	-	-	-
agrawal	SVFDT-I	0.9453	0.8331	0.125	6	82.18 ± 4.17	1.0005	0.0028	0.9504
	SVFDT-II	0.9453	0.8332	22.778	356	85.52 ± 3.87	1.0005	0.5129	0.9891
	VFDT	0.6421	0.1965	445.654	3456	62.25 ± 3.55	-	-	-
airlines	SVFDT-I	0.6541	0.2234	53.971	417	158.27 ± 12.44	1.0187	0.1211	2.5426
	SVFDT-II	0.6425	0.1975	133.043	1029	104.14 ± 6.32	1.0007	0.2985	1.6730
	VFDT	0.7955	0.6008	43.960	631	1153.90 ± 12.27	-	-	-
covType	SVFDT-I	0.7896	0.5895	30.755	441	1155.10 ± 14.87	0.9927	0.6996	1.0010
	SVFDT-II	0.7896	0.5893	38.332	550	1167.40 ± 12.65	0.9926	0.8720	1.0117
	VFDT	0.9988	0.9202	9.044	442	212.61 ± 2.61	-	-	-
ctu_1	SVFDT-I	0.9948	0.6414	0.095	4	190.80 ± 2.48	0.9960	0.0105	0.8974
	SVFDT-II	0.9988	0.9156	2.808	103	216.75 ± 2.18	0.9999	0.3105	1.0195
	VFDT	0.9974	0.7771	9.508	409	138.26 ± 0.54	-	-	-
ctu_2	SVFDT-I	0.9885	0.0090	0.076	3	113.61 ± 1.16	0.9911	0.0080	0.8217
	SVFDT-II	0.9970	0.7370	5.259	171	135.10 ± 1.10	0.9995	0.5531	0.9771
	VFDT	0.9999	0.9836	3.762	162	347.14 ± 2.07	-	-	-
ctu_3	SVFDT-I	0.9998	0.9636	1.677	56	356.62 ± 5.66	0.9999	0.4459	1.0273
	SVFDT-II	0.9998	0.9723	2.180	99	357.64 ± 2.08	0.9999	0.5796	1.0302
	VFDT	0.9997	0.8868	2.243	87	77.02 ± 4.54	-	-	-
ctu_4	SVFDT-I	0.9991	0.6186	0.717	23	84.25 ± 0.69	0.9994	0.3198	1.0939
	SVFDT-II	0.9997	0.8907	1.128	41	83.23 ± 1.46	1.0000	0.5031	1.0807
	VFDT	0.9981	0.7314	0.708	18	8.65 ± 0.43	-	-	-
ctu_5	SVFDT-I	0.9955	0.3540	0.076	3	9.04 ± 0.46	0.9974	0.1077	1.0458
	SVFDT-II	0.9980	0.7103	0.256	13	8.90 ± 0.50	0.9999	0.3612	1.0292
	VFDT	0.9993	0.9207	2.736	99	41.47 ± 1.49	-	-	-
ctu_6	SVFDT-I	0.9990	0.8832	1.213	35	38.05 ± 1.53	0.9997	0.4434	0.9175
	SVFDT-II	0.9990	0.8847	1.503	51	39.16 ± 2.42	0.9997	0.5493	0.9441
	VFDT	0.9996	0.3333	0.109	5	7.52 ± 0.41	-	-	-
ctu_7	SVFDT-I	0.9994	-0.0318	0.074	3	7.49 ± 0.34	0.9998	0.6792	0.9955
	SVFDT-II	0.9995	0.0952	0.092	4	7.52 ± 0.41	0.9999	0.8462	1.0002
	VFDT	0.9998	0.9239	4.493	164	215.62 ± 8.95	-	-	-
ctu_8	SVFDT-I	0.9998	0.8804	1.442	63	198.79 ± 8.29	0.9999	0.3210	0.9220
	SVFDT-II	0.9998	0.9246	1.893	84	201.21 ± 11.48	1.0000	0.4215	0.9332
	VFDT	0.9940	0.9318	22.855	854	145.97 ± 7.69	-	-	-
ctu_9	SVFDT-I	0.9700	0.6610	0.189	9	141.00 ± 7.54	0.9759	0.0083	0.9660
	SVFDT-II	0.9923	0.9130	12.814	437	146.79 ± 6.07	0.9983	0.5607	1.0056
	VFDT	0.9997	0.9967	2.270	58	87.72 ± 4.49	-	-	-
ctu_10	SVFDT-I	0.9990	0.9878	0.626	18	87.75 ± 4.67	0.9993	0.2759	1.0004
	SVFDT-II	0.9997	0.9965	1.016	35	87.10 ± 3.34	1.0000	0.4476	0.9930
	VFDT	0.9994	0.9925	0.671	16	7.24 ± 0.43	-	-	-
ctu_11	SVFDT-I	0.9965	0.9537	0.408	6	7.11 ± 0.31	0.9970	0.6084	0.9810
	SVFDT-II	0.9984	0.9793	0.165	8	7.21 ± 0.31	0.9990	0.2459	0.9952
	VFDT	0.9972	0.5756	2.469	104	22.26 ± 1.32	-	-	-
ctu_12	SVFDT-I	0.9933	-0.0037	0.058	2	26.06 ± 1.34	0.9961	0.0233	1.1707
	SVFDT-II	0.9965	0.4792	0.962	52	23.77 ± 1.40	0.9994	0.3895	1.0679
	VFDT	0.9977	0.8887	18.490	608	135.61 ± 6.30	-	-	-
ctu_13	SVFDT-I	0.9792	-0.0002	0.058	2	98.63 ± 4.87	0.9815	0.0031	0.7273
	SVFDT-II	0.9970	0.8533	11.044	242	140.86 ± 8.40	0.9993	0.5973	1.0387
	VFDT	0.8195	0.5749	3.216	271	3.04 ± 0.24	-	-	-
elec	SVFDT-I	0.8052	0.5411	1.090	87	3.01 ± 0.18	0.9825	0.3390	0.9886
	SVFDT-II	0.8064	0.5439	1.499	123	3.11 ± 0.24	0.9839	0.4660	1.0210
	VFDT	0.8476	0.6951	20.425	1550	24.59 ± 0.69	-	-	-
hyper	SVFDT-I	0.8903	0.7806	1.102	81	25.86 ± 0.08	1.0504	0.0540	1.0519
	SVFDT-II	0.8632	0.7263	5.754	436	27.39 ± 2.31	1.0184	0.2817	1.1142
	VFDT	0.7355	0.7059	18.287	609	1323.01 ± 15.51	-	-	-
led_10	SVFDT-I	0.7336	0.7038	1.707	55	1283.69 ± 22.00	0.9975	0.0934	0.9703
	SVFDT-II	0.7352	0.7057	8.697	289	1326.79 ± 16.33	0.9997	0.4756	1.0029
	VFDT	0.5087	0.4538	19.421	647	1143.01 ± 14.43	-	-	-
led_20	SVFDT-I	0.5077	0.4527	3.452	114	1180.24 ± 6.73	0.9980	0.1777	1.0326
	SVFDT-II	0.5086	0.4538	10.403	346	1130.95 ± 4.93	0.9999	0.5356	0.9895
	VFDT	0.8188	0.6368	17.719	626	316.51 ± 3.31	-	-	-
poker	SVFDT-I	0.8645	0.7284	11.060	374	314.27 ± 3.06	1.0558	0.6242	0.9929
	SVFDT-II	0.8065	0.6122	14.299	521	313.57 ± 2.73	0.9850	0.8070	0.9907
	VFDT	0.9163	0.8193	27.115	2055	39.17 ± 2.34	-	-	-
rbf_500k	SVFDT-I	0.9018	0.7879	1.262	93	36.00 ± 0.13	0.9841	0.0465	0.9191
	SVFDT-II	0.9098	0.8053	8.486	643	51.30 ± 0.17	0.9929	0.3130	1.3096
	VFDT	0.9241	0.8364	48.444	3670	71.66 ± 4.59	-	-	-
rbf_1kk	SVFDT-I	0.9084	0.8026	1.289	95	60.32 ± 4.98	0.9830	0.0266	0.8417
	SVFDT-II	0.9169	0.8209	15.293	1138	96.20 ± 8.15	0.9922	0.3157	1.3424
	VFDT	0.9922	0.9842	16.286	328	35.31 ± 1.14	-	-	-
rbf_250k(50)	SVFDT-I	0.9894	0.9786	1.500	28	90.73 ± 0.13	0.9972	0.0921	2.5696
	SVFDT-II	0.9928	0.9855	11.734	235	39.51 ± 0.09	1.0006	0.7205	1.1191
	VFDT	0.8465	0.5884	2.430	363	2.30 ± 0.20	-	-	-
sea	SVFDT-I	0.8513	0.6014	0.616	91	2.41 ± 0.17	1.0057	0.2533	1.0477
	SVFDT-II	0.8518	0.6029	1.077	160	2.50 ± 0.22	1.0064	0.4433	1.0837
	VFDT	0.5169	0.0253	30.416	48	43.51 ± 0.33	-	-	-
usenet	SVFDT-I	0.5342	0.0604	6.209	9	48.85 ± 0.55	1.0336	0.2041	1.1229
	SVFDT-II	0.5268	0.0454	8.692	13	50.12 ± 0.64	1.0192	0.2858	1.1521

Table 11 – Performance metrics of the algorithms by dataset for $\tau = 0.01$, $GP = 100$ and GI (worst case).

Dataset	Algorithm	Accuracy	Kappa M	Memory (MB)	Splits	Time (s)	Rel. Accuracy	Rel. Memory	Rel. Time
agrawal	VFDT	0.9356	0.8038	0.713	7	83.59 ± 0.41	-	-	-
	SVFDT-I	0.9347	0.8010	0.404	6	91.57 ± 0.48	0.9990	0.5673	1.0956
	SVFDT-II	0.9356	0.8038	0.714	7	83.77 ± 0.47	1.0000	1.0019	1.0022
airlines	VFDT	0.6464	0.2061	1.427	10	685.32 ± 3.14	-	-	-
	SVFDT-I	0.6256	0.1595	0.396	2	802.34 ± 74.89	0.9679	0.2774	1.1708
	SVFDT-II	0.6455	0.2042	1.948	14	835.82 ± 1.94	0.9987	1.3647	1.2196
covType	VFDT	0.7256	0.4645	8.698	124	1094.42 ± 6.47	-	-	-
	SVFDT-I	0.7142	0.4422	4.107	58	1130.04 ± 9.27	0.9842	0.4722	1.0325
	SVFDT-II	0.7439	0.5002	9.742	139	1092.03 ± 6.94	1.0252	1.1200	0.9978
ctu_1	VFDT	0.9976	0.8325	0.666	31	226.52 ± 5.15	-	-	-
	SVFDT-I	0.9951	0.6595	0.133	6	272.38 ± 13.33	0.9975	0.1989	1.2024
	SVFDT-II	0.9972	0.8099	0.613	28	211.96 ± 16.51	0.9997	0.9202	0.9357
ctu_2	VFDT	0.9942	0.4977	0.473	25	137.75 ± 3.87	-	-	-
	SVFDT-I	0.9927	0.3722	0.095	4	107.24 ± 8.43	0.9985	0.2013	0.7785
	SVFDT-II	0.9947	0.5424	0.311	16	121.31 ± 8.37	1.0005	0.6576	0.8807
ctu_3	VFDT	0.9999	0.9852	0.936	35	321.14 ± 15.76	-	-	-
	SVFDT-I	0.9999	0.9863	0.683	21	326.37 ± 21.54	1.0000	0.7298	1.0163
	SVFDT-II	0.9999	0.9858	0.901	33	363.72 ± 10.04	1.0000	0.9626	1.1326
ctu_4	VFDT	0.9986	0.3895	0.110	5	85.19 ± 6.12	-	-	-
	SVFDT-I	0.9986	0.3922	0.093	4	79.71 ± 5.52	1.0000	0.8463	0.9356
	SVFDT-II	0.9986	0.3899	0.111	5	87.02 ± 5.45	1.0000	1.0120	1.0214
ctu_5	VFDT	0.9952	0.3052	0.055	2	13.30 ± 0.98	-	-	-
	SVFDT-I	0.9952	0.3041	0.056	2	13.30 ± 1.01	1.0000	1.0241	0.9995
	SVFDT-II	0.9952	0.3052	0.056	2	13.07 ± 0.76	1.0000	1.0240	0.9822
ctu_6	VFDT	0.9993	0.9095	0.426	7	49.99 ± 3.22	-	-	-
	SVFDT-I	0.9992	0.9030	0.111	5	51.02 ± 2.66	0.9999	0.2599	1.0207
	SVFDT-II	0.9992	0.9093	0.427	7	44.74 ± 2.48	1.0000	1.0031	0.8949
ctu_7	VFDT	0.9994	-0.0159	0.037	1	8.95 ± 0.65	-	-	-
	SVFDT-I	0.9994	-0.0159	0.038	1	8.95 ± 0.65	1.0000	1.0358	0.9997
	SVFDT-II	0.9994	-0.0159	0.038	1	8.79 ± 0.48	1.0000	1.0358	0.9816
ctu_8	VFDT	0.9997	0.8474	0.364	19	245.28 ± 1.91	-	-	-
	SVFDT-I	0.9980	0.0344	0.148	7	260.03 ± 2.10	0.9983	0.4055	1.0602
	SVFDT-II	0.9997	0.8469	0.664	20	255.49 ± 1.30	1.0000	1.8238	1.0416
ctu_9	VFDT	0.9849	0.8291	2.115	83	161.63 ± 0.69	-	-	-
	SVFDT-I	0.9672	0.6303	0.076	3	167.61 ± 1.75	0.9821	0.0361	1.0370
	SVFDT-II	0.9787	0.7594	1.382	54	158.71 ± 1.30	0.9937	0.6535	0.9819
ctu_10	VFDT	0.9970	0.9628	0.164	8	97.23 ± 1.65	-	-	-
	SVFDT-I	0.9961	0.9519	0.110	5	98.41 ± 0.35	0.9991	0.6748	1.0121
	SVFDT-II	0.9964	0.9558	0.147	7	96.87 ± 0.40	0.9994	0.8975	0.9963
ctu_11	VFDT	0.9921	0.8966	0.055	2	7.52 ± 0.50	-	-	-
	SVFDT-I	0.9908	0.8796	0.056	2	7.49 ± 0.48	0.9987	1.0241	0.9951
	SVFDT-II	0.9921	0.8966	0.056	2	7.43 ± 0.38	1.0000	1.0240	0.9880
ctu_12	VFDT	0.9933	0.0005	0.128	6	32.49 ± 0.22	-	-	-
	SVFDT-I	0.9933	-0.0018	0.056	2	34.11 ± 0.13	1.0000	0.4406	1.0500
	SVFDT-II	0.9933	0.0005	0.129	6	33.60 ± 0.13	1.0000	1.0103	1.0343
ctu_13	VFDT	0.9913	0.5793	2.113	83	170.31 ± 2.10	-	-	-
	SVFDT-I	0.9814	0.1070	0.075	3	119.49 ± 0.95	0.9901	0.0357	0.7016
	SVFDT-II	0.9913	0.5810	1.071	57	162.19 ± 1.78	1.0000	0.5066	0.9523
elec	VFDT	0.7746	0.4691	0.196	12	3.12 ± 0.01	-	-	-
	SVFDT-I	0.7733	0.4661	0.201	12	3.07 ± 0.01	0.9983	1.0292	0.9835
	SVFDT-II	0.7719	0.4626	0.178	10	3.01 ± 0.01	0.9964	0.9117	0.9641
hyper	VFDT	0.9132	0.8264	0.054	3	20.69 ± 0.11	-	-	-
	SVFDT-I	0.9065	0.8131	0.082	5	19.13 ± 1.08	0.9927	1.5225	0.9247
	SVFDT-II	0.9123	0.8245	0.055	3	19.57 ± 1.08	0.9989	1.0245	0.9461
led_10	VFDT	0.7259	0.6953	1.151	38	1021.06 ± 7.03	-	-	-
	SVFDT-I	0.7216	0.6905	0.709	23	1055.21 ± 7.76	0.9942	0.6164	1.0334
	SVFDT-II	0.7254	0.6947	1.093	36	1031.48 ± 1.16	0.9993	0.9500	1.0102
led_20	VFDT	0.4888	0.4317	1.037	33	1035.03 ± 5.36	-	-	-
	SVFDT-I	0.4775	0.4192	0.459	14	1058.55 ± 7.90	0.9768	0.4428	1.0227
	SVFDT-II	0.4884	0.4312	0.977	31	1036.63 ± 8.13	0.9991	0.9426	1.0016
poker	VFDT	0.7635	0.5259	3.850	153	287.23 ± 6.66	-	-	-
	SVFDT-I	0.7149	0.4286	1.391	54	288.18 ± 7.26	0.9364	0.3612	1.0033
	SVFDT-II	0.7417	0.4823	3.709	149	290.89 ± 7.89	0.9715	0.9633	1.0127
rbf_500k	VFDT	0.8033	0.5753	0.430	31	46.40 ± 4.50	-	-	-
	SVFDT-I	0.7726	0.5090	0.163	11	44.88 ± 2.91	0.9618	0.3780	0.9672
	SVFDT-II	0.8033	0.5752	0.377	27	48.53 ± 0.35	0.9999	0.8763	1.0458
rbf_1kk	VFDT	0.8444	0.6646	0.724	53	90.07 ± 8.36	-	-	-
	SVFDT-I	0.8146	0.6003	0.323	23	71.08 ± 6.07	0.9647	0.4467	0.7891
	SVFDT-II	0.8431	0.6618	0.659	48	79.67 ± 6.89	0.9985	0.9095	0.8845
rbf_250k(50)	VFDT	0.9566	0.9123	0.723	13	54.89 ± 2.82	-	-	-
	SVFDT-I	0.9570	0.9131	0.466	8	34.06 ± 1.84	1.0004	0.6451	0.6206
	SVFDT-II	0.9565	0.9122	0.673	12	57.08 ± 3.86	1.0000	0.9305	1.0399
sea	VFDT	0.8455	0.5859	0.034	4	2.23 ± 0.22	-	-	-
	SVFDT-I	0.8450	0.5847	0.022	2	2.22 ± 0.25	0.9994	0.6424	0.9956
	SVFDT-II	0.8455	0.5859	0.035	4	2.24 ± 0.24	1.0000	1.0390	1.0047
usenet	VFDT	0.5514	0.0951	0.622	0	54.58 ± 0.69	-	-	-
	SVFDT-I	0.5514	0.0951	0.623	0	54.67 ± 0.66	1.0000	1.0021	1.0016
	SVFDT-II	0.5514	0.0951	0.623	0	54.53 ± 0.50	1.0000	1.0021	0.9992

Table 12 – Performance metrics of all algorithms by dataset for $\tau = 0.20$, $GP = 200$ and GI (best case).

Dataset	Algorithm	Accuracy	Kappa M	Memory (MB)	Splits	Time (s)	Rel. Accuracy	Rel. Memory	Rel. Time
agrawal	VFDT	0.9437	0.8283	43.392	1167	77.00 ± 4.00	-	-	-
	SVFDT-I	0.9473	0.8392	3.443	80	81.25 ± 4.63	1.0038	0.0793	1.0551
	SVFDT-II	0.9448	0.8316	30.994	646	92.77 ± 1.87	1.0012	0.7143	1.2048
airlines	VFDT	0.6496	0.2133	275.043	2132	69.38 ± 1.19	-	-	-
	SVFDT-I	0.6533	0.2217	24.792	191	126.28 ± 11.82	1.0057	0.0901	1.8199
	SVFDT-II	0.6507	0.2159	166.004	1285	87.76 ± 0.67	1.0018	0.6036	1.2648
covType	VFDT	0.8482	0.7037	142.503	2108	955.20 ± 7.78	-	-	-
	SVFDT-I	0.7452	0.5028	12.737	182	1030.77 ± 6.38	0.8786	0.0894	1.0791
	SVFDT-II	0.8254	0.6592	92.607	1331	958.46 ± 2.87	0.9731	0.6499	1.0034
ctu_1	VFDT	0.9989	0.9220	7.580	362	185.72 ± 0.99	-	-	-
	SVFDT-I	0.9951	0.6595	0.133	6	190.20 ± 0.97	0.9962	0.0175	1.0241
	SVFDT-II	0.9986	0.9063	2.090	85	186.92 ± 1.22	0.9998	0.2757	1.0064
ctu_2	VFDT	0.9975	0.7866	7.383	308	118 ± 0.79	-	-	-
	SVFDT-I	0.9927	0.3666	0.095	4	101.73 ± 6.27	0.9951	0.0129	0.8621
	SVFDT-II	0.9969	0.7325	2.347	97	119.16 ± 5.50	0.9994	0.3179	1.0099
ctu_3	VFDT	0.9999	0.9826	4.004	160	311.25 ± 14.35	-	-	-
	SVFDT-I	0.9999	0.9854	1.210	46	311.46 ± 0.42	1.0000	0.3021	1.0007
	SVFDT-II	0.9999	0.9787	1.458	44	326.79 ± 0.40	1.0000	0.3641	1.0499
ctu_4	VFDT	0.9998	0.9209	1.742	75	72.63 ± 3.37	-	-	-
	SVFDT-I	0.9990	0.5671	0.391	5	75.70 ± 0.61	0.9992	0.2245	1.0423
	SVFDT-II	0.9997	0.8744	0.979	33	72.23 ± 0.48	0.9999	0.5620	0.9946
ctu_5	VFDT	0.9984	0.7736	0.254	13	8.44 ± 0.52	-	-	-
	SVFDT-I	0.9960	0.4251	0.076	3	8.38 ± 0.37	0.9976	0.3000	0.9924
	SVFDT-II	0.9982	0.7358	0.220	11	8.84 ± 0.56	0.9997	0.8628	1.0473
ctu_6	VFDT	0.9996	0.9471	2.306	80	36.19 ± 0.53	-	-	-
	SVFDT-I	0.9942	0.3056	0.074	3	38.49 ± 0.27	0.9947	0.0323	1.0637
	SVFDT-II	0.9995	0.9363	0.879	32	36.96 ± 0.25	0.9999	0.3813	1.0215
ctu_7	VFDT	0.9997	0.3968	0.389	5	7.38 ± 0.44	-	-	-
	SVFDT-I	0.9994	-0.0476	0.056	2	7.28 ± 0.35	0.9998	0.1442	0.9871
	SVFDT-II	0.9997	0.3968	0.390	5	7.37 ± 0.44	1.0000	1.0034	0.9996
ctu_8	VFDT	0.9999	0.9427	3.477	132	190.75 ± 1.26	-	-	-
	SVFDT-I	0.9991	0.5606	0.383	20	203.47 ± 1.36	0.9992	0.1103	1.0667
	SVFDT-II	0.9998	0.8934	1.075	43	208.21 ± 12.52	0.9999	0.3090	1.0915
ctu_9	VFDT	0.9935	0.9268	14.255	583	134.36 ± 0.41	-	-	-
	SVFDT-I	0.9780	0.7519	0.189	9	139.21 ± 1.22	0.9844	0.0132	1.0361
	SVFDT-II	0.9911	0.8992	7.417	219	142.21 ± 4.70	0.9975	0.5203	1.0585
ctu_10	VFDT	0.9998	0.9971	1.198	45	82.64 ± 3.87	-	-	-
	SVFDT-I	0.9984	0.9807	0.554	14	82.16 ± 0.58	0.9987	0.4625	0.9943
	SVFDT-II	0.9994	0.9930	0.853	26	81.49 ± 0.53	0.9997	0.7122	0.9861
ctu_11	VFDT	0.9996	0.9944	0.561	10	6.78 ± 0.44	-	-	-
	SVFDT-I	0.9933	0.9118	0.074	3	7 ± 0.39	0.9937	0.1321	1.0322
	SVFDT-II	0.9988	0.9844	0.354	3	7.09 ± 0.42	0.9992	0.6306	1.0456
ctu_12	VFDT	0.9964	0.4599	2.617	97	21.23 ± 0.03	-	-	-
	SVFDT-I	0.9933	-0.0069	0.058	2	15.92 ± 0.04	0.9969	0.0221	0.7497
	SVFDT-II	0.9954	0.3123	0.944	51	21.70 ± 0.05	0.9990	0.3607	1.0220
ctu_13	VFDT	0.9978	0.8963	14.338	415	128.09 ± 0.76	-	-	-
	SVFDT-I	0.9792	-0.0002	0.058	2	89.94 ± 0.65	0.9813	0.0040	0.7021
	SVFDT-II	0.9968	0.8471	5.054	165	129.36 ± 0.74	0.9990	0.3525	1.0099
elec	VFDT	0.8132	0.5599	2.212	152	2.33 ± 0.20	-	-	-
	SVFDT-I	0.7926	0.5115	0.534	29	2.36 ± 0.15	0.9747	0.2413	1.0107
	SVFDT-II	0.8031	0.5363	1.565	103	2.40 ± 0.16	0.9877	0.7078	1.0306
hyper	VFDT	0.8583	0.7165	12.499	946	14.38 ± 0.08	-	-	-
	SVFDT-I	0.8950	0.7899	0.538	39	15.71 ± 0.03	1.0427	0.0431	1.0925
	SVFDT-II	0.8632	0.7264	9.683	734	14.96 ± 0.07	1.0057	0.7747	1.0400
led_10	VFDT	0.7314	0.7014	115.169	3779	879.66 ± 1.77	-	-	-
	SVFDT-I	0.7338	0.7041	1.342	43	977.25 ± 1.44	1.0033	0.0117	1.1109
	SVFDT-II	0.7330	0.7032	79.692	2616	917 ± 4.51	1.0022	0.6920	1.0424
led_20	VFDT	0.5053	0.4500	110.075	3616	900.61 ± 2.22	-	-	-
	SVFDT-I	0.5083	0.4534	2.075	67	991.84 ± 2.75	1.0061	0.0188	1.1013
	SVFDT-II	0.5074	0.4524	72.245	2371	961.45 ± 5.20	1.0043	0.6563	1.0676
poker	VFDT	0.9530	0.9058	59.027	1878	254.24 ± 1.13	-	-	-
	SVFDT-I	0.8408	0.6810	4.745	176	280.22 ± 2.74	0.8823	0.0804	1.1022
	SVFDT-II	0.9190	0.8375	20.331	741	267.49 ± 2.65	0.9642	0.3444	1.0521
rbf_500k	VFDT	0.9110	0.8078	20.734	1574	29.22 ± 0.10	-	-	-
	SVFDT-I	0.8780	0.7364	0.548	39	25.70 ± 0.06	0.9637	0.0264	0.8795
	SVFDT-II	0.9150	0.8165	10.257	778	29.09 ± 0.10	1.0044	0.4947	0.9954
rbf_1kk	VFDT	0.9203	0.8282	36.719	2787	58.78 ± 0.14	-	-	-
	SVFDT-I	0.8821	0.7458	0.548	39	48.58 ± 0.11	0.9585	0.0149	0.8266
	SVFDT-II	0.9226	0.8332	14.884	1128	59.94 ± 0.21	1.0026	0.4054	1.0198
rbf_250k(50)	VFDT	0.9907	0.9813	16.385	330	32.33 ± 0.12	-	-	-
	SVFDT-I	0.9758	0.9512	0.363	6	37.86 ± 0.19	0.9849	0.0222	1.1712
	SVFDT-II	0.9892	0.9783	12.035	242	40.91 ± 0.12	0.9985	0.7345	1.2653
sea	VFDT	0.8491	0.5956	1.536	228	1.62 ± 0.05	-	-	-
	SVFDT-I	0.8473	0.5907	0.249	36	1.65 ± 0.01	0.9979	0.1621	1.0154
	SVFDT-II	0.8477	0.5918	1.183	176	1.65 ± 0.01	0.9983	0.7707	1.0158
usenet	VFDT	0.5272	0.0461	15.518	24	45.62 ± 0.44	-	-	-
	SVFDT-I	0.5444	0.0808	4.347	6	50.83 ± 0.45	1.0326	0.2801	1.1140
	SVFDT-II	0.5501	0.0924	4.347	6	51.52 ± 0.42	1.0435	0.2801	1.1291

APPENDIX B – CAN THE SVFDTS PRODUCE TREES SMALLER THAN THE SMALLEST VFDT?

Table 13 – Performance metrics of all algorithms by dataset for $\tau = 0$, $GP = 100$, entropy and IG.

Dataset	Algorithm	Accuracy	Kappa M	Memory (MB)	Splits	Time (s)	Rel. Accuracy	Rel. Memory	Rel. Time
	VFDT	0.9329	0.7953	2.327	50	47.51 ± 6.39	-	-	-
agrawal	SVFDT-I	0.9290	0.7836	0.376	4	46.33 ± 0.61	0.9958	0.1615	0.8023
	SVFDT-II	0.9332	0.7962	2.002	52	50.68 ± 0.22	1.0003	0.8600	0.8455
	VFDT	0.6373	0.1857	0.778	5	213.36 ± 6.37	-	-	-
airlines	SVFDT-I	0.6291	0.1673	1.708	12	593.92 ± 6.12	0.9871	2.1941	2.8598
	SVFDT-II	0.6400	0.1918	1.039	7	158.82 ± 1.63	1.0042	1.3352	0.7555
	VFDT	0.7296	0.4723	5.924	84	215.47 ± 1.77	-	-	-
covType	SVFDT-I	0.7173	0.4483	3.353	47	216.15 ± 4.50	0.9831	0.5661	0.9964
	SVFDT-II	0.7182	0.4501	5.026	71	217.06 ± 8.79	0.9844	0.8486	1.0433
	VFDT	0.9979	0.8550	1.191	34	130.30 ± 17.28	-	-	-
ctu_1	SVFDT-I	0.9930	0.5186	0.077	3	111.02 ± 0.37	0.9951	0.0644	0.7496
	SVFDT-II	0.9977	0.8438	0.950	36	129.10 ± 0.59	0.9998	0.7970	0.8652
	VFDT	0.9955	0.6091	1.157	47	80.27 ± 0.39	-	-	-
ctu_2	SVFDT-I	0.9918	0.2955	0.077	3	70.79 ± 0.29	0.9963	0.0663	0.8778
	SVFDT-II	0.9958	0.6339	0.822	44	80.18 ± 0.39	1.0003	0.7102	0.9969
	VFDT	0.9997	0.9491	0.402	21	375.75 ± 1.00	-	-	-
ctu_3	SVFDT-I	0.9996	0.9247	0.095	4	374.80 ± 1.53	0.9999	0.2372	0.9974
	SVFDT-II	0.9997	0.9405	0.330	17	378.40 ± 28.66	1.0000	0.8220	1.1306
	VFDT	0.9978	0.0512	0.372	4	64.50 ± 7.19	-	-	-
ctu_4	SVFDT-I	0.9978	0.0535	0.058	2	64.27 ± 9.09	1.0000	0.1555	0.9680
	SVFDT-II	0.9978	0.0512	0.373	4	64.20 ± 0.43	1.0000	1.0035	0.8136
	VFDT	0.9976	0.6526	0.167	8	6.898 ± 0.19	-	-	-
ctu_5	SVFDT-I	0.9951	0.2986	0.058	2	6.70 ± 0.14	0.9975	0.3464	0.9745
	SVFDT-II	0.9974	0.6282	0.113	5	6.69 ± 0.10	0.9998	0.6745	0.9714
	VFDT	0.9990	0.8819	0.390	5	41.19 ± 0.66	-	-	-
ctu_6	SVFDT-I	0.9989	0.8683	0.093	4	41.38 ± 0.59	0.9999	0.2382	1.0059
	SVFDT-II	0.9990	0.8814	0.373	4	41.95 ± 5.50	1.0000	0.9567	1.1339
	VFDT	0.9994	-0.0159	0.018	0	8.37 ± 1.05	-	-	-
ctu_7	SVFDT-I	0.9994	-0.0159	0.020	0	8.40 ± 0.14	1.0000	1.0707	0.9331
	SVFDT-II	0.9994	-0.0159	0.020	0	8.33 ± 0.16	1.0000	1.0707	0.9375
	VFDT	0.9985	0.2877	0.147	7	171.52 ± 2.11	-	-	-
ctu_8	SVFDT-I	0.9985	0.2876	0.058	2	171.41 ± 2.19	1.0000	0.3950	0.9955
	SVFDT-II	0.9985	0.2904	0.077	3	171.81 ± 3.49	1.0000	0.5226	0.9976
	VFDT	0.9869	0.8526	3.695	151	117.76 ± 1.57	-	-	-
ctu_9	SVFDT-I	0.9681	0.6400	0.058	2	116.02 ± 1.57	0.9810	0.0157	0.9842
	SVFDT-II	0.9883	0.8679	2.821	136	118.32 ± 1.64	1.0014	0.7635	0.9911
	VFDT	0.9966	0.9585	0.092	4	101.74 ± 13.75	-	-	-
ctu_10	SVFDT-I	0.9967	0.9595	0.075	3	101.58 ± 5.36	1.0001	0.8141	0.9025
	SVFDT-II	0.9965	0.9575	0.093	4	100.34 ± 0.38	0.9999	1.0144	0.8668
	VFDT	0.9887	0.8515	0.037	1	6.22 ± 0.09	-	-	-
ctu_11	SVFDT-I	0.9887	0.8515	0.039	1	6.14 ± 0.27	1.0000	1.0352	1.0828
	SVFDT-II	0.9887	0.8515	0.039	1	6.15 ± 0.62	1.0000	1.0352	1.1386
	VFDT	0.9945	0.1697	0.432	7	22.00 ± 0.96	-	-	-
ctu_12	SVFDT-I	0.9942	0.1287	0.075	3	22.13 ± 2.64	0.9997	0.1746	0.9786
	SVFDT-II	0.9944	0.1568	0.442	8	17.44 ± 1.50	0.9999	1.0233	0.6932
	VFDT	0.9952	0.7689	3.403	139	94.76 ± 2.39	-	-	-
ctu_13	SVFDT-I	0.9793	0.0027	0.058	2	66.48 ± 1.68	0.9840	0.0170	0.7010
	SVFDT-II	0.996	0.8061	3.459	142	94.26 ± 2.04	1.0008	1.0165	0.9862
	VFDT	0.7936	0.5138	0.599	39	2.48 ± 0.27	-	-	-
elec	SVFDT-I	0.7895	0.5042	0.429	23	2.35 ± 0.05	0.9948	0.7158	0.9157
	SVFDT-II	0.7909	0.5075	0.558	31	2.43 ± 0.06	0.9966	0.9312	0.9174
	VFDT	0.9385	0.8770	0.014	0	17.33 ± 1.92	-	-	-
hyper	SVFDT-I	0.9385	0.8770	0.015	0	17.73 ± 1.48	1.0000	1.0955	1.0849
	SVFDT-II	0.9385	0.8770	0.015	0	17.12 ± 1.87	1.0000	1.0955	1.0843
	VFDT	0.7381	0.7089	0.031	0	241.49 ± 1.72	-	-	-
led_10	SVFDT-I	0.7381	0.7089	0.032	0	242.48 ± 5.97	1.0000	1.0425	1.0096
	SVFDT-II	0.7381	0.7089	0.032	0	242.00 ± 15.01	1.0000	1.0425	1.0861
	VFDT	0.5108	0.4561	0.031	0	242.47 ± 12.52	-	-	-
led_20	SVFDT-I	0.5108	0.4561	0.032	0	241.71 ± 30.57	1.0000	1.0425	1.0570
	SVFDT-II	0.5108	0.4561	0.032	0	243.47 ± 5.09	1.0000	1.0425	1.0023
	VFDT	0.6935	0.3857	2.278	90	116.43 ± 9.49	-	-	-
poker	SVFDT-I	0.6850	0.3686	1.734	64	112.76 ± 10.73	0.9877	0.7610	0.9952
	SVFDT-II	0.6913	0.3812	1.839	72	112.43 ± 8.24	0.9968	0.8071	1.0082
	VFDT	0.7224	0.4014	0.176	6	31.28 ± 6.83	-	-	-
rbf_500k	SVFDT-I	0.7034	0.3604	0.068	2	31.25 ± 4.42	0.9928	0.4603	0.8682
	SVFDT-II	0.7221	0.4008	0.177	3	30.78 ± 6.94	0.9997	0.6076	0.9816
	VFDT	0.6960	0.3435	0.093	12	66.42 ± 4.11	-	-	-
rbf_1kk	SVFDT-I	0.6910	0.3328	0.043	4	60.98 ± 0.78	0.9737	0.3863	0.8010
	SVFDT-II	0.6958	0.3430	0.056	12	65.74 ± 0.44	0.9996	1.0075	0.8244
	VFDT	0.9090	0.8162	0.052	0	36.21 ± 3.71	-	-	-
rbf_250k(50)	SVFDT-I	0.9090	0.8162	0.053	0	36.41 ± 0.39	1.0000	1.0251	0.9374
	SVFDT-II	0.9090	0.8162	0.053	0	36.05 ± 0.17	1.0000	1.0251	0.9387
	VFDT	0.8448	0.5840	0.007	0	2.05 ± 0.26	-	-	-
sea	SVFDT-I	0.8448	0.5840	0.008	0	2.12 ± 0.21	1.0000	1.1885	0.9783
	SVFDT-II	0.8448	0.5840	0.008	0	2.07 ± 0.24	1.0000	1.1885	1.1520
	VFDT	0.5514	0.0951	0.622	0	8.61 ± 1.11	-	-	-
usenet	SVFDT-I	0.5514	0.0951	0.623	0	8.71 ± 0.36	1.0000	1.0021	0.9455
	SVFDT-II	0.5514	0.0951	0.623	0	8.77 ± 0.41	1.0000	1.0021	0.9260

Table 14 – Performance metrics of all algorithms by dataset for $\tau = 0$, $GP = 100$ and GI.

Dataset	Algorithm	Accuracy	Kappa M	Memory (MB)	Splits	Time (s)	Rel. Accuracy	Rel. Memory	Rel. Time
	VFDT	0.9357	0.8038	0.3690	4	66.87 ± 1.52	-	-	-
agrawal	SVFDT-I	0.9347	0.801	0.4050	6	56.08 ± 0.63	0.9989	1.0978	0.8386
	SVFDT-II	0.9357	0.8038	0.3703	4	66.47 ± 0.58	1.0000	1.0036	0.9941
	VFDT	0.6266	0.1617	0.5262	3	479.04 ± 6.94	-	-	-
airlines	SVFDT-I	0.6256	0.1594	0.6591	4	536.15 ± 13.82	0.9984	1.2526	1.1192
	SVFDT-II	0.6266	0.1617	0.5275	3	498.47 ± 18.78	1.0000	1.0025	1.0406
	VFDT	0.7231	0.4597	8.7786	125	217.26 ± 14.11	-	-	-
covType	SVFDT-I	0.7073	0.4287	4.6692	66	227.92 ± 11.05	0.9781	0.5319	1.0491
	SVFDT-II	0.7413	0.4951	8.0840	115	241.98 ± 17.36	1.0252	0.9209	1.1138
	VFDT	0.9935	0.5527	0.4631	9	206.87 ± 14.20	-	-	-
ctu_1	SVFDT-I	0.9909	0.3753	0.3733	4	190.74 ± 14.01	0.9974	0.8061	0.9220
	SVFDT-II	0.9927	0.4987	0.4827	10	199.78 ± 18.57	0.9992	1.0422	0.9657
	VFDT	0.9956	0.6206	0.4513	8	132.18 ± 8.19	-	-	-
ctu_2	SVFDT-I	0.9951	0.5753	0.3907	5	84.36 ± 2.52	0.9995	0.8655	0.6382
	SVFDT-II	0.9956	0.618	0.3972	5	135.56 ± 14.75	1.0000	0.8801	1.0255
	VFDT	0.9963	0.3474	0.4085	6	366.89 ± 25.08	-	-	-
ctu_3	SVFDT-I	0.9963	0.3477	0.0564	2	355.70 ± 17.60	1.0000	0.1380	0.9695
	SVFDT-II	0.9963	0.3476	0.3734	4	367.61 ± 17.47	1.0000	0.9140	1.0020
	VFDT	0.9977	-0.0004	0.0185	0	97.12 ± 7.69	-	-	-
ctu_4	SVFDT-I	0.9977	-0.0004	0.0198	0	90.67 ± 8.62	1.0000	1.0707	0.9335
	SVFDT-II	0.9977	-0.0004	0.0198	0	92.27 ± 4.69	1.0000	1.0707	0.9500
	VFDT	0.9931	-0.0011	0.0185	0	9.85 ± 0.53	-	-	-
ctu_5	SVFDT-I	0.9931	-0.0011	0.0198	0	9.60 ± 0.43	1.0000	1.0707	0.9740
	SVFDT-II	0.9931	-0.0011	0.0198	0	10.56 ± 1.17	1.0000	1.0707	1.0714
	VFDT	0.9984	0.8110	0.0373	1	47.06 ± 4.56	-	-	-
ctu_6	SVFDT-I	0.9984	0.8110	0.0386	1	46.08 ± 4.00	1.0000	1.0353	0.9792
	SVFDT-II	0.9984	0.8110	0.0386	1	46.19 ± 2.93	1.0000	1.0353	0.9815
	VFDT	0.9994	-0.0159	0.0185	0	8.07 ± 0.73	-	-	-
ctu_7	SVFDT-I	0.9994	-0.0159	0.0198	0	8.00 ± 0.25	1.0000	1.0707	0.9909
	SVFDT-II	0.9994	-0.0159	0.0198	0	9.68 ± 1.09	1.0000	1.0707	1.1992
	VFDT	0.9979	-0.0002	0.0190	0	229.00 ± 29.05	-	-	-
ctu_8	SVFDT-I	0.9979	-0.0002	0.0203	0	210.02 ± 15.49	1.0000	1.0688	0.9171
	SVFDT-II	0.9979	-0.0002	0.0203	0	200.74 ± 5.13	1.0000	1.0688	0.8766
	VFDT	0.9847	0.8276	0.9720	37	120.44 ± 1.72	-	-	-
ctu_9	SVFDT-I	0.9695	0.6559	0.0767	3	115.93 ± 2.56	0.9846	0.0789	0.9625
	SVFDT-II	0.9755	0.7241	0.3844	20	119.38 ± 4.22	0.9907	0.3955	0.9912
	VFDT	0.9937	0.9226	0.0733	3	98.02 ± 2.59	-	-	-
ctu_10	SVFDT-I	0.9915	0.8949	0.0564	2	97.05 ± 3.70	0.9978	0.7693	0.9902
	SVFDT-II	0.9935	0.9203	0.0746	3	91.278 ± 0.80	0.9998	1.0180	0.9312
	VFDT	0.9886	0.8506	0.0373	1	6.33 ± 0.16	-	-	-
ctu_11	SVFDT-I	0.9886	0.8506	0.0387	1	6.62 ± 0.33	1.0000	1.0353	1.0454
	SVFDT-II	0.9886	0.8506	0.0387	1	7.27 ± 0.56	1.0000	1.0353	1.1492
	VFDT	0.9933	-0.0005	0.0185	0	27.41 ± 3.05	-	-	-
ctu_12	SVFDT-I	0.9933	-0.0005	0.0198	0	24.66 ± 1.73	1.0000	1.0707	0.8993
	SVFDT-II	0.9933	-0.0005	0.0198	0	24.82 ± 1.97	1.0000	1.0707	0.9052
	VFDT	0.9893	0.4846	0.6024	31	134.44 ± 12.93	-	-	-
ctu_13	SVFDT-I	0.9795	0.0138	0.0580	2	64.38 ± 1.00	0.9901	0.0962	0.4789
	SVFDT-II	0.9885	0.4468	0.4902	25	124.28 ± 1.10	0.9992	0.8137	0.9244
	VFDT	0.7746	0.4691	0.1968	12	2.40 ± 0.14	-	-	-
elec	SVFDT-I	0.7733	0.4661	0.2025	12	2.54 ± 0.17	0.9983	1.0290	1.0572
	SVFDT-II	0.7719	0.4626	0.1794	12	2.51 ± 0.13	0.9965	0.9113	1.0420
	VFDT	0.9385	0.8770	0.0137	0	19.61 ± 3.13	-	-	-
hyper	SVFDT-I	0.9385	0.8770	0.0150	0	17.01 ± 1.18	1.0000	1.0955	0.8674
	SVFDT-II	0.9385	0.8770	0.0150	0	18.64 ± 0.75	1.0000	1.0955	0.9505
	VFDT	0.7381	0.7089	0.0307	0	265.20 ± 9.51	-	-	-
led_10	SVFDT-I	0.7381	0.7089	0.0320	0	244.24 ± 23.42	1.0000	1.0425	0.9210
	SVFDT-II	0.7381	0.7089	0.0320	0	233.28 ± 4.20	1.0000	1.0425	0.8796
	VFDT	0.5108	0.4561	0.0307	0	242.35 ± 19.22	-	-	-
led_20	SVFDT-I	0.5108	0.4561	0.0320	0	232.79 ± 6.46	1.0000	1.0425	0.9605
	SVFDT-II	0.5108	0.4561	0.0320	0	234.74 ± 6.05	1.0000	1.0425	0.9686
	VFDT	0.7595	0.518	3.3301	133	109.87 ± 9.34	-	-	-
poker	SVFDT-I	0.7149	0.4285	1.3467	52	98.03 ± 0.91	0.9413	0.4044	0.8922
	SVFDT-II	0.7388	0.4764	3.5514	142	103.44 ± 2.18	0.9727	1.0665	0.9414
	VFDT	0.6916	0.3352	0.0678	0	60.38 ± 4.42	-	-	-
rbf_500k	SVFDT-I	0.6917	0.3354	0.0428	0	59.03 ± 3.12	1.0001	0.6314	0.9775
	SVFDT-II	0.6916	0.3351	0.0555	0	58.86 ± 2.88	1.0000	0.8191	0.9748
	VFDT	0.6885	0.3273	0.0137	4	36.50 ± 3.58	-	-	-
rbf_1kk	SVFDT-I	0.6885	0.3273	0.0150	2	28.98 ± 1.09	1.0000	1.0955	0.7938
	SVFDT-II	0.6885	0.3273	0.0150	3	30.45 ± 1.79	1.0000	1.0955	0.8342
	VFDT	0.9090	0.8162	0.0519	0	35.31 ± 1.80	-	-	-
rbf_250k(50)	SVFDT-I	0.9090	0.8162	0.0532	0	36.90 ± 2.34	1.0000	1.0251	1.0452
	SVFDT-II	0.9090	0.8162	0.0532	0	40.26 ± 5.66	1.0000	1.0251	1.1402
	VFDT	0.8448	0.584	0.0069	0	2.60 ± 0.42	-	-	-
sea	SVFDT-I	0.8448	0.584	0.0082	0	1.93 ± 0.07	1.0000	1.1886	0.7421
	SVFDT-II	0.8448	0.584	0.0082	0	2.03 ± 0.16	1.0000	1.1886	0.7791
	VFDT	0.5514	0.0951	0.6221	0	8.58 ± 0.32	-	-	-
usenet	SVFDT-I	0.5514	0.0951	0.6234	0	8.18 ± 0.19	1.0000	1.0021	0.9539
	SVFDT-II	0.5514	0.0951	0.6234	0	8.15 ± 0.33	1.0000	1.0021	0.9502

WORKS PUBLISHED BY THE AUTHOR

1. ⋆ V. G. T. da COSTA, E. J. Santana and S. Barbon - "Evaluating the Four-way Performance Trade-off for Stream Classification", Proceedings of the 14th International Conference on Green, Pervasive and Cloud Computing (GPC) (in press - 2019) (Qualis CC 2016, B2)

2. V. G. T. da COSTA, S. Barbon, R. S. MIANI, J. J. P. C. Rodrigues and B. B. Zarpelão - "Mobile Botnets Detection based on Machine Learning over System Calls", International Journal of Security and Networks (in press - 2019) (Qualis CC 2016, B1)

3. S. Barbon, V. G. T. da Costa, S.-H. Chen and R. C. Guido - "U-healthcare system for pre-diagnosis of Parkinson's disease from voice signal", Proceedings of the International Symposium on Multimedia, 2018. (Short Paper) (Qualis CC 2016, B1)

4. ⋆ V. G. T. da Costa, A. C. P. de L. F. de Carvalho and S. Barbon - "Strict Very Fast Decision Tree: a memory conservative algorithm for data stream mining", Pattern Recognition Letters, 2018. (Qualis CC 2016, A1)

5. V. H. Bezerra, V. G. T. da Costa, S. Barbon, R. S. Miani, B. B. Zarpelão - "One-class Classification to Detect Botnets in IoT devices", Proceedings of the Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg), Natal, Rio Grande do Norte, 2018. (Qualis CC 2016, B3)

6. V. H. Bezerra, V. G. T. da Costa, R. A. Martins, S. Barbon, R. S. Miani, B. B. Zarpelão - "Providing IoT host-based datasets for intrusion detection research", Proceedings of the Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg), Natal, Rio Grande do Norte, 2018. (Qualis CC 2016, B3)

7. ⋆ V. G. T. da Costa, S. M. Mastelini, A. C. P. de L. F. de Carvalho, S. Barbon - "Making Data Stream Classification Tree-based Ensembles Lighter". Proceedings of the Brazilian Conference on Intelligent Systems (BRACIS), São Paulo, São Paulo, 2018. (Qualis CC 2016, B2)

8. S. M. Mastelini, E. J. Santana, V. G. T. da Costa, S. Barbon - "Benchmarking multi-target regression", Proceedings of the Brazilian Conference on Intelligent Systems (BRACIS), São Paulo, São Paulo, 2018. (Qualis CC 2016, B2)

9. ⋆ G. M. Tavares, V. G. T. da Costa, V. E. Martins, P. Ceravolo, S. Barbon - "Anomaly detection in business process based on data stream mining", Proceedings

of the Brazilian Symposium on Information Systems (SBSI), Caxias do Sul, Brazil, 2018. (Qualis CC 2016, B2)

10. S. M. Mastelini, V. G. T. da Costa, E. J. Santana, F. K. Nakano, R. C. Guido, S. Barbon - "Multi-Output Tree Chaining: An Interpretative Modelling and Lightweight Multi-Target Approach", Journal of Signal Processing Systems, 2018, May, 05. (Qualis CC 2016, B1)

11. ⋆ V. G. T. da Costa, B. B. Zarpelão, R. S. Miani, S. Barbon - "Online detection of Botnets on Network Flows using Stream Mining", Proceedings of the XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), Campos do Jordão, São Paulo, 2018. (Qualis CC 2016, B2)

12. ⋆ S. Barbon, G. M. Tavares, V. G. T. da Costa, P. Ceravolo, and E. Damiani - "A framework for human-in-the-loop monitoring of concept-drift detection in event log stream", The 2018 Web Conference Companion (WWW), Lyon, France, 2018. (Qualis CC 2016, A1)

13. V. G. T. da Costa, S. Barbon, R. S. Miani, J. J. P. C. Rodrigues and B. B. Zarpelão - "Detecting mobile botnets through machine learning and system calls analysis", IEEE International Conference on Communications (ICC), Paris, France, 2017, pp. 1-6 (Qualis CC 2016, A1).

⋆ Related to data streams.