Dynamic Decision Trees and Community-based Graph Embeddings : towards Interpretable Machine Learning

Gabriel Damay

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2024IPPAT047

Thèse de doctorat

TELECOM Paris

IP PARIS

Dynamic Decision Trees and Community-based Graph Embeddings: towards Interpretable Machine Learning

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom Paris

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat: Mathématiques et informatique

Thèse présentée et soutenue à Palaiseau, le 19/12/2024, par

GABRIEL DAMAY

Composition du Jury :

Jesse Read Professor at LIX (École Polytechnique)	Président/Examinateur
Matthieu Latapy Research Director (DR) at LIP6 (CNRS)	Rapporteur
Marc Lelarge Research Director (DR) at INRIA and ENS	Rapporteur
Fragkiskos Malliaros Associate Professor at CentraleSupélec	Examinateur
Vincent Labatut Associate Professor at Avignon Université	Examinateur
Marine Le Morvan Junior Researcher (CR) at INRIA	Examinatrice
Mauro Sozio Professor at Télécom Paris (IPP)	Directeur de thèse

Résumé

L'apprentissage automatique est le domaine des sciences informatiques dont le but est de créer des modèles et des solutions à partir de données sans savoir exactement les instructions qui dirigent intrinsèquement ces modèles. Ce domaine a obtenu des résultats impressionnants mais il est le sujet d'inquiétudes en raison notamment de l'impossibilité de comprendre et d'auditer les modèles qu'il produit. L'apprentissage automatique interprétable propose une solution à ces inquiétudes en créant des modèles qui sont interprétables de façon inhérante. Cette thèse contribue à l'apprentissage automatique interprétable de deux façons.

Dans un premier temps, nous étudions les arbres de décision. Il s'agit d'un groupe de méthodes d'apprentissage automatique très connu et qui est interprétable par la façon même dont il est conçu. Cependant, les données réelles sont souvent dynamiques et peu d'algorithmes existent pour maintenir un arbre de décision quand des données peuvent à la fois être ajoutées et supprimées de l'ensemble d'entrainement. Nous proposons un nouvel objectif pour les arbres de décision dynamiques, que nous appelons "ϵ-faisabilité". Ce nouvel objectif relaxe des propriété que nous obtiendrions si l'arbre était recalculé à chaque modification de l'ensemble d'entrainement. En particulier, un recalcul systématique de l'arbre garantirait que chaque séparation de l'ensemble d'entrainement résulte en le meilleur gain possible du score de Gini, mais l'ϵ-faisabilité ne requiert qu'une garantie d'écart maximal entre le gain effectif et le meilleur gain possible. Nous présentons également des propriétés de régularité du score de Gini. Ces propriétés permettent de montrer que, lorsque l'ensemble d'entrainement de l'arbre change peu, les scores de Gini obtenus en séparant cet ensemble d'entrainement changent peu également. Ainsi, il n'est pas nécessaire de reconstruire l'ensemble de l'arbre en repartant de zéro à chaque modification de l'ensemble d'entrainement. Nous nous basons sur ces propriétés pour construire un nouvel algorithms que nous appelons PARFAITE. Celui-ci consiste à conserver sur chaque sommet de l'arbre le nombre de modifications apportées à l'ensemble d'entrainement qui a permis de construire le sommet. Lorsque ce nombre dépasse un seuil, qui dépend de la taille de cet ensemble d'entrainement, un recalcul est déclenché. Pour éviter une situation dans laquelle un sommet est recalculé immédiatement après ses sommets "enfants", nous introduisont également un critère grâce auquel, lorsque la taille de l'ensemble d'entrainement d'un sommet est trop proche de celle de son parent, c'est le parent qui est recalculé.

Dans un second temps, nous étudions l'embedding de graphes. La technique appelée "embedding" est une technique d'apprentissage automatique très commune. Elle consiste à projeter les noeuds d'un graphe sur un espace vectoriel. Ce type de méthodes est cependant non-interprétable en général. Nous proposons un nouvel algorithme d'embedding appelé PARFAITE, qui est basé sur la factorisation de la matrice de PageRank personnalisé. Cet algorithme est

conçu pour que ses résultats soient interprétables. Il consiste tout d'abord à construire une fonction qui approche la multiplication de n'importe quel vecteur par la matrice de PageRank personnalisée, centrée par colonne. Cette fonction peut ensuite être utilisée pour obtenir la décomposition en valeurs singulières de la matrice. Le résultat est la projection sur les sous-espaces singuliers de chaque vecteur de PageRank personnalisé pour chaque sommet du graphe, et symétriquement la projection des colonnes de la matrice. Un clustering est alors réalisé pour détecter les communautés et leur vecteur de PageRank personnalisé, projeté sur le sous-espace singulier. Enfin, une inversion de la projection sur les sous-espaces singuliers est réalisée, afin d'obtenir le vecteur de PageRank personnalisé moyen de chaque communauté, ainsi que la colonne moyenne de la matrice. Ce sont ces vecteurs moyens qui sont utilisés comme embedding. Puisque le vecteur de PageRank personnalisé donne des valeurs plus fortes aux sommets d'une même communauté, ces résultats sont alors interprétables comme des métriques d'importance et de degré d'appartenance de chaque sommet à une communauté.

Nous étudions chacun de ces algorithmes sur un plan à la fois théorique et expérimental. Nous montrons que FuDyADT est au minimum comparable aux algorithmes de l'état de l'art dans les conditions habituelles, tout en étant également capable de fonctionner dans des contextes inhabituels comme dans le cas où des données sont supprimées. Quant à PARFAITE, il produit des dimensions d'embedding qui sont alignées avec les communautés du graphe, et qui sont donc interprétables.

Abstract

Machine learning is the field of computer science that interests in building models and solutions from data without knowing exactly the set of instructions internal to these models and solutions. This field has achieved great results but is now under scrutiny for the inability to understand or audit its models among other concerns. Interpretable Machine Learning addresses these concerns by building models that are inherently interpretable. This thesis contributes to Interpretable Machine Learning in two ways.

First, we study decision trees. This is a very popular group of machine learning methods for classification problems and it is interpretable by design. However, real world data is often dynamic, but few algorithms can maintain a decision tree when data can be both inserted and deleted from the training set. We propose a new algorithm called FuDyADT to solve this problem.

Second, when data is represented as a graph, a very common machine learning technique called "embedding" consists in projecting it onto a vectorial space. This kind of method however is usually not interpretable. We propose a new embedding algorithm called PARFAITE based on the factorization of the Personalized PageRank matrix. This algorithm is designed to provide interpretable results.

We study both algorithms theoretically and experimentally. We show that FuDyADT is at least comparable to state-of-the-art algorithms in the usual setting, while also being able to handle unusual settings such as deletions of data. PARFAITE on the other hand produces embedding dimensions that align with the communities of the graph, making the embedding interpretable.

Acknowledgements

First of all, I must thank Mauro. You trusted me with this PhD, and I also have learned a lot by working with you. I also want to thank Christophe Prieur and Fabien Tarissan. You were involved in the decision to trust me with this PhD, and it has also been very interesting and fulfilling to work with you during these years.

I would also like to thank the reviewers, Marc Lelarge and Matthieu Latapy, who took the time to read my manuscript and review it, and who also trusted me from this manuscript with my ability to defend this PhD. And more generally I want to thank all the members of the jury for your attentive listening of my presentation and your very interesting questions that allowed me to clarify some points of my work and also gave me some new research ideas.

Then, I would like to thank all those who were interested enough in my work to attend the presentation physically or remotely. It really meant a lot to me.

During these 3 years and a bit more, I have been part of the DIG team. I have spent a really great time with you, I felt welcome among you from day one and the lunches with you and the *The Crew* games helped me relax a bit when the workload was high, so thank you very much.

Of course, in 3 years, there are some times when life is a bit harder. I want to thank my friends of the *compagnie*, who have often been sympathetic listeners when I had to vent what was on my heart. I also want to adress special thanks to Diane and Grégoire who have offered to proofread my manuscript. It has really helped me to improve it.

Speaking of proofreading, I want to thank Julie. You have been there before, so you gave me extremely precious advice, both on the manuscript and also on many aspects of the life of a PhD student. I also remember that you, together with Tobias, hosted my when I missed my connection flight coming back from a summer school, and I was trapped in Frankfurt. More generally, I would like to thank my whole family. It's comforting to be with you when times are hard, and great when times are easier.

And last but not least, my deepest thanks go to Zoé. You have helped me so much that I don't know if I could have succeeded without you. You have been my strongest support, you listened to countless of my rehearsals even though my PhD topic is very far from your main interests. You have been an absolutely critical support, both morally and logistically. So you have my greatest gratitude.

Contents

List of Figures

List of Tables

Introduction

Machine learning takes a growing space in our daily lives. From speaking virtual assistants to autonomous cars, from recommender systems to chatbots, or from automatically labelling images to classifying resumes for recruiting, machine learning algorithms are becoming central in our society.

However, these algorithms also raise a lot of concerns, many of which revolve around their obscurity: How can we ensure that the productions or decisions of these algorithms are fair and non-discriminatory? How can we ensure that they respect personal life and confidentiality of data? How can we ensure that the recommender systems do not result in filter bubbles?

The field of interpretable machine learning addresses these concerns by creating algorithms that are inherently interpretable. In other words, when these algorithms are used for solving a problem, not only do they provide a solution, but it is also easy for the human user to understand why they provide this decision.

This thesis addresses two questions related to interpretable machine learning. The first one is: How can we design an algorithm for building and efficiently maintaining decision trees, which are interpretable, in a fully dynamic context? Indeed, although decision trees are widely used to solve classification problems, there are very few solutions to use them in a fully dynamic context. We answer this question by studying the smoothness of functions used to build the tree. We also define a new objective for fully dynamic trees, and we propose an algorithm to build trees that meet this objective. We study the theoretical and practical performances of this new algorithm.

The second question is: Can we make a graph embedding that would be inherently interpretable? Graph embedding is a very useful tool to use classical machine learning algorithms designed for vectors when the input data are graphs. However, if the embedding is not interpretable, the final result of the algorithm will have no chance of being interpretable. To answer that question, we study the PageRank score, and especially its variation called Personalized PageRank score. We use this score, as well as matrix decomposition and clustering, to design an embedding algorithm that is inherently highly interpretable, and we study the interpretability and the efficiency of this algorithm. We also

study the existing metrics of graph embedding interpretability and we design a new one to address some weaknesses of these metrics. While studying the Personalized PageRank, we also discover a property that links it tightly to the well-known spectral graph embedding.

Organization of this thesis

This thesis focuses on two main problems, namely algorithms to build and maintain fully dynamic decision trees, and inherently interpretable graph embeddings.

Chapter 1 provides the general background about these problems. It is organized in two sections, each presenting the background of one of the two problems.

Chapter 2 presents the work about an algorithm for building and maintaining fully dynamic decision trees. It begins with a short review of the literature around dynamic decision tree algorithms. It carries on presenting preliminaries for the algorithms. Notable parts of this preliminaries section are the presentation of a new theorem about the smoothness of Gini index and Gini gain, as well as a new objective for fully dynamic decision trees. The following two sections present the algorithm itself and experiments to assess its running time and efficiency.

Chapter 3 presents the work about an interpretable graph embedding. It starts with several preliminary sections about the various aspects used for the design of this embedding. Notably, it presents new properties of the Personalized PageRank (PPR) matrix that shed a new light on the spectral graph embedding. Then, two sections present the algorithm and experiences to assess its interpretability and efficiency. A last section presents a side work of this doctoral work about the clustering of graph data for analyzing the YouTube website.

Notations

Vectors and matrices In this thesis, matrices and vectors are written with bold uppercase and lowercase letters, respectively. A bold subscript or superscript character is part of the name of the matrix or vector, while regular subscripts or superscripts are operations on the matrix or vector, e.g. index or exponentiation. For example, the notation $\boldsymbol{v_i}$ denotes the i^{th} vector of a sequence of vectors $\{\boldsymbol{v_i}\}_{i\in\mathbb{N}}$. On the opposite, $\boldsymbol{v}_i$ denotes the value at the i^{th} position of the vector $\boldsymbol{v}$.

All non-transposed vectors are column vectors.

The identity matrix is denoted by $\boldsymbol{I}$.

Intervals The common notation is used, a parenthesis denotes that the related side of the interval is open, and a square bracket denotes that the side is closed.

For example:

$$(0, 1] = \{x \in \mathbb{R} : 0 < x \leq 1\}$$

The set on which the interval is defined ($\mathbb{R}$ or $\mathbb{N}$) is derived from the context. When this set is $\mathbb{N}$ and the range is $[1, i]$ for some $i \in \mathbb{N}$, then the set is also denoted $[i]$.

Cardinal number For any set S, its cardinal number is denoted by $|S|$.

Acknowledgements

This work was funded by the French National Agency (ANR) under project APY (ANR-20-CE38-0011).

Chapter 1

Background

1.1 Dynamic decision trees

1.1.1 The classification problem

In machine learning, the class of problems called **supervised learning** problems regroup those for which a set called **training set** of objects called **training examples** is provided, and an expected output value is known for each of these examples. The problem is then to create a model using these examples that would capture the underlying logic of the expected output so that, when applied to new examples for which the expected output is unknown, the model would generalize from the training examples to predict the output as best as possible. Two of the main problems in this class are regression problems and classification problems, and this thesis addresses the latter.

The **classification problem** is a very common problem in machine learning. In this problem, the expected output for each example is a **class**, also called a **label**, i.e. the identifier of a group of examples to which this example belongs. In that case, the prediction of a model is sometimes called the **decision**. For example, we can think of flights, and we want to know if they will arrive early, on time, delayed or if they will be canceled. These 4 possible, mutually-exclusive status are the classes, and each flight is an example. We focus on the case when the examples are points characterized each by several elements called **feature elements** (sometimes shorten to **features**). In the case of flights, features can be the company, the date of flight, the length of the flight and so on. The number d of features is called the **dimension** of the examples. The features are identified by integers from 1 to d. Each feature i can only take its value from a set of admissible values, called the **domain** of the given feature and denoted by $\mathrm{dom}(i)$. For example, the domain of the feature "company" of a flight is the set of existing companies. The list of all features of an example is represented by a vector of dimension d.[1] The Cartesian product of the features

[1] All data that are considered in the part of this thesis related to decision trees can be injected

domains defines the domain of the vector, i.e. the set of possible values for the vector itself. It is called the **feature domain** and is denoted by $\mathcal{X}$. Similarly, the set of labels to which every expected output belongs is called the **label domain** and is denoted by $\mathcal{Y}$. To simplify, the labels are generally projected onto the integers so that the **label domain** ranges from 0 to $|\mathcal{Y}|$.

Definition 1.1 (Classification problem)**.** *Let $\mathcal{Y} = \{0, \dots |\mathcal{Y}|\}$ be a set of classes. Let also $S = \{(\boldsymbol{x_1}, y_1), \dots (\boldsymbol{x_n}, y_n)\}$ be a training set with $(\boldsymbol{x_i}, y_i) \in \mathcal{X} \times \mathcal{Y}$ being an example and its associated label. Let finally $Q = \{(\boldsymbol{x_1^*}, y_1^*), \dots (\boldsymbol{x_m^*}, y_m^*)\}$ be a decision set defined similarly.*

It is assumed that each example $\boldsymbol{x_i^*}$ is correlated with its class y_i^* in the same way as the examples in S are correlated with their classes.

*Then, the **classification problem** consists in building a function called **model** $M_S : \mathcal{X} \to \mathcal{Y}$ using the training set S, so that the prediction $M_S(\boldsymbol{x_i^*})$ of the examples in Q is mostly right.*

The notion of being "mostly right" is vague because several objective or evaluation metrics exist for evaluating the performance of a solution to this problem. This topic will be addressed in more details in Section 1.1.3.

In this thesis, we propose general solutions, but the experiments will mainly be conducted on the binary classification problem. In this problem, the examples belong to one of two classes commonly called "positive" and "negative" classes, i.e. $|\mathcal{Y}| = 2$. We can think for example of disease diagnosis, in which the "positive" class usually means that the patient has the disease. We map the positive class to 1 and the negative class to 0 so that $\mathcal{Y} = \{0, 1\}$.

1.1.2 The dynamic classification problem

The **dynamic classification problem** is a special case of the classification problem. In that case, the training set is updated sequentially by adding or removing examples and the algorithm should update the model considering these insertions and deletions.[2] Insertions are typically the result of new data being collected or revealed, while deletions can be the result of noise removal, removal of personal data for privacy concerns, data becoming obsolete, etc. For this problem the complexity of updating the model is crucial since many applications use real-time streaming data for which the decision needs to be given after a reasonable delay. For example, we can think of a failure prediction

onto the set $\mathbb{R}$ without any loss of information. We therefore consider that we always work on vectors of $\mathbb{R}^d$. However, the ordering of the domain values does not always make sense, and the relative order of two projected features into the $\mathbb{R}$ set should not always be considered as semantically meaningful. This matter is discussed with more details in Section 1.1.5

[2] Some papers in the literature also consider the updating of examples. In this thesis, we consider the updating of an example as the removal of the outdated version and the insertion of the updated one. We do not study the specific optimizations that could be implemented for this specific case.

system that needs to predict the failures as soon as the data of the unlabeled examples are available, and needs to update in order to consider the new labeled examples as they arrive.

Definition 1.2 (Dynamic classification problem)**.** *Let $S_0 = \{(\boldsymbol{x_1}, y_1), \dots (\boldsymbol{x_n}, y_n)\}$ be an initial training set, and $U = \{((\boldsymbol{x_1^U}, y_1^U), u_1), \dots ((\boldsymbol{x_T^U}, y_T^U), u_T)\}$ be a set of updates with $(\boldsymbol{x_t^U}, y_t^U) \in \mathcal{X} \times \mathcal{Y}$ an example/class pair and $u_t \in \{\text{INS}, \text{DEL}\}$ the type of update, indicating that at time t the pair is respectively inserted in the training set or deleted. We denote by S_t the set S_0 affected by the updates $((\boldsymbol{x_1^U}, x_1^U), u_1), \dots ((\boldsymbol{x_t^U}, y_t^U), u_t)$.*

*The **dynamic classification problem** consists in*

1. *Building a model M_{S_0} that addresses the classification problem for S_0,*
2. *For each $t \in [1, T]$, updating the model $M_{S_{t-1}}$ into a new model M_{S_t} which addresses the classification problem for S_t.*

1.1.3 Evaluating a classification model

Many metrics have been proposed over the years for evaluating the model created by a classification algorithm. Although new proposals have been sparse in recent years, there is still some research on the specifics of each metric and the choice of the best one for a given problem (see for example [12]). This subsection provides a quick overview of the most commonly used metrics.

As introduced in the Subsection 1.1.1, a classification algorithm builds a model M_S from a training set S so that this model produces good predictions when applied to the decision set Q.

The first measure that can be used is the so-called **training error**:

Definition 1.3 (Training error (TE))**.** *The **training error** of a model M_S is the number of examples in S that would be attributed to the wrong class by the model.*

$$TE = |\{(\boldsymbol{x_i}, y_i) \in S : M_S(\boldsymbol{x_i}) \neq y_i\}| \tag{1.1}$$

Providing that S does not contain two identical examples associated with different classes, it is always possible to get an optimal training error of 0. Trying to find a model that optimizes the Training Error is called the **Empirical Risk Minimization** (ERM) [52]. However, models with an optimal training error are often not optimal because of the so-called **overfitting**: the model's primary goal is to correctly predict the data of the decision set Q, and excessive fitting to the training set S often reduces its relevance when generalizing to the decision set [27]. An example of a probable overfitting, in the case of a decision tree, is given on Figure 1.2. The reason we can say it is probably overfitting will be given in Section 1.1.6. It has been proven that the set of admissible models can be reduced *a priori* (i.e. before the training phase) in a way that prevents the overfitting and hence makes ERM a good strategy for training and a good indicator of the quality of the model. This strategy is called **induction bias** [52].

Most classification model building algorithms rely on trying to minimize the Training Error with induction bias. It is however difficult to assess the quality of the bias on which the prevention of the model overfitting is based. Methods have therefore been developed to assess the quality of the model using data that have not been used for the training of the model.

The most common of these methods is to split the training set into two subsets. The model is trained only on the first one, that appropriates the name "training set", and the second one is called the "**test set**" and is used as a proxy decision set. We denote by S the training set and Q the test set. This setup allows the following definitions.[3]

Definition 1.4 (True Positive (TP) and True Negative (TN))**.** *In a binary classification problem, the True Positive and True Negative values are the number of examples in the test set that are correctly associated with respectively the positive and the negative class by the model.*

$$\begin{aligned} TP &= |\{(\boldsymbol{x_i}, y_i) \in Q : y_i = 1 \wedge M_S(\boldsymbol{x_i}) = 1\}| \\ TN &= |\{(\boldsymbol{x_i}, y_i) \in Q : y_i = 0 \wedge M_S(\boldsymbol{x_i}) = 0\}| \end{aligned} \tag{1.2}$$

Definition 1.5 (False Positive (FP) and False Negative (FN))**.** *The False Positive and False Negative values are the number of examples in the test set that are incorrectly attributed to the positive and to the negative class, respectively, by the model.*

$$\begin{aligned} FP &= |\{(\boldsymbol{x_i}, y_i) \in Q : y_i = 0 \wedge M_S(\boldsymbol{x_i}) = 1\}| \\ FN &= |\{(\boldsymbol{x_i}, y_i) \in Q : y_i = 1 \wedge M_S(\boldsymbol{x_i}) = 0\}| \end{aligned} \tag{1.3}$$

These measures are raw unnormalized data, hence called "Base measures" in [12]. Three metrics are directly derived from these.

Definition 1.6 (Precision, or confidence (P))**.** *The **precision** of a model is the ratio of the examples associated with the positive class that are indeed in this positive class.*

$$P = \frac{TP}{TP + FP} \tag{1.4}$$

Definition 1.7 (Sensitivity or Recall (TPR), and Specificity (TNR))**.** [4]

The Sensitivity and the Specificity of a model are the fraction of examples belonging to the positive and to the negative class, respectively, for which the model's prediction is correct.

$$\begin{aligned} TPR &= \frac{TP}{TP+FN} \\ TNR &= \frac{TN}{TN+FP} \end{aligned} \tag{1.5}$$

[3] Note that we reuse the name "training set", as well as the notations S and Q. This is because the model is trained solely on the new training set, making it very similar to the old definition, and the model is not trained on the test set but simply outputs guesses about the labels of its examples, making it very similar to the decision set.

[4] TPR and TNR stand respectively for "True Positive Ratio" and "True Negative Ratio".

All these metrics are normalized, but they are complementary and none of them alone is sufficient to draw a complete picture of the model's performance. Therefore, many metrics exist to aggregate them into a single value. In this thesis, we only present one of the best-known, called F1-score.

Definition 1.8 (F1-score)**.** *The F1-score of a model is the harmonic mean of its precision and sensitivity*

$$F_1 = 2\frac{P \cdot TPR}{P + TPR} \tag{1.6}$$

We note that this metric should be used with caution because it gives equal weight to precision and recall, although the relative costs of false positive and false negative predictions vary greatly from one application to another. For example, if the model predicts serious diseases for which the cure is cheap and has few side effects, then a low number of false negative predictions is expected even at the cost of a high number of false positive predictions. This would ensure that as few diseases as possible are missed, while the false positive predictions would result in the useless use of a cheap cure at low risk. The low number of false negative would mean a high recall, and the high number of false positive would mean a low precision.

For this reason, there exist variations of the F1-score, called Fβ-scores, which weight the precision and the sensitivity differently in the harmonic mean. For more details on these scores or other evaluation methods, please refer to [12].

Evaluating dynamic classification models For evaluating a dynamic classification model, we need a special kind of test set. In that test set, each example needs to be associated with a step t of the updating process at which the class should be predicted by the model M_t. One way to build such a test set is to use each example of the update set for which the update type is "INS" both as a test example and as a training example. The "test" step, which consists in predicting its class, is performed using the model as it is just before the "training" step, i.e. the insertion of the example. This method allows training each model on all the examples available at the time at which the model is defined, while never predicting the class of an example using a model that would have been trained on that very example.

It is then possible to use the same metrics as in the static case, e.g. the F1-score.

1.1.4 Decision trees generalities

One well-known group of algorithms for solving the classification problem is the group of decision tree algorithms. These algorithms can be used to solve either the classification problem or another problem called the regression problem. To remove the ambiguity, we sometimes refer to **classification tree** algorithms in the former case and regression trees in the latter. However, since

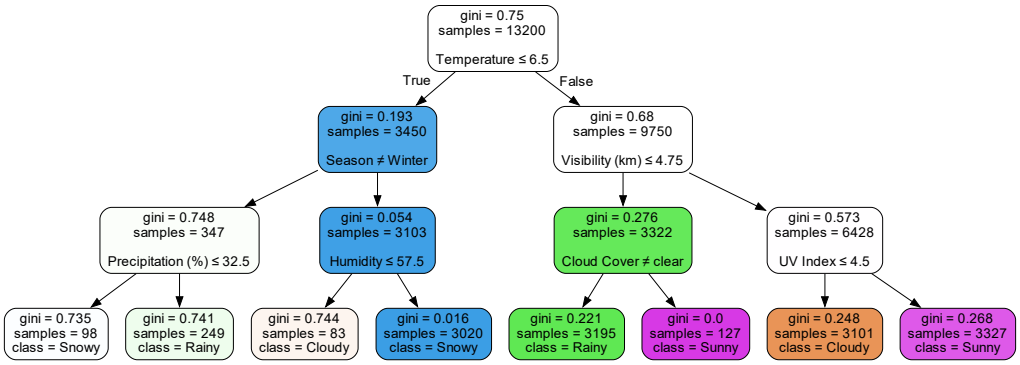


Figure 1.1: Example of a decision tree for the classification of the *Weather* dataset [55]. On each node, the Gini index and the size of the training set are indicated. If the node is internal, then the condition is written. The training set of the left child contains the examples that match the condition. If the node is a leaf, the majority class is indicated. The color hue of each node correspond to its majority class, while the color saturation correspond to the consensus on that class (inversely related to the Gini index).

we focus on classification problems in this thesis, the term "decision tree" will always be used for "classification tree" unless specified otherwise.

Definition 1.9 (Decision tree)**.** *A **decision tree** is a type of classification model built as a tree graph. The nodes of this graph are of two kinds:*

- ***Internal nodes*** *These nodes are characterized by a set of children nodes and a set of rules to deterministically direct any example to one of the children nodes,*

- ***Leaves*** *These nodes are characterized by a class. Any example directed to a leaf will be predicted to belong to the class associated with it.*

The class of an example is predicted by directing it from the root down to a leaf using the rules at each node.

The process of building a decision tree from data is called **tree training** or **tree building**, and we will use both words indifferently in this thesis.

Using trees of rules to organize knowledge is a very intuitive thing to do, and such trees were used long before the automation of their building for machine learning. For example, we can think of the so-called "tree of life" that classify life in a tree and date back at least to 1801 [1]. The automation arrived first in 1963 with the publication of the Automatic Interaction Detection (AID) algorithm for regression trees [43], quickly followed in 1972 by the THeta Automatic Interaction Detection (THAID) algorithm for classification trees [42]. For a more comprehensive history of decision trees in Data Science, see [37].

The decision tree algorithms have since become some of the most popular algorithms for classification. One of their main advantages is the clarity of the model they build, whose decisions can be explained just by listing the rules that direct the examples from the root to the leaf [14, 30].

The usual framework algorithm for building decision trees, sometimes called Hunt's algorithm [49], is described in Algorithm 1.1. This framework algorithm takes as arguments the training set $S \in (\mathcal{X}, \mathcal{Y})^n$. It can also require some context information, denoted by "..." in Algorithm 1.1, such as the depth of the subtree that will be built by this call to the function.

A decision tree is built recursively, starting from the root, with the full set of training examples, by either making the node a leaf or splitting its training set and building children from the subsets. At Line 5, the class of a leaf is usually determined by the majority class in the subset of training examples used for building the leaf. Therefore, the main variation for decision tree building algorithms is the method used at Line 8 to split the set of training examples. The criteria used at Line 4 to decide whether a node should be a leaf are also an important parameter for decision tree building.

Algorithm 1.1 Hunt's recursive framework algorithm for building a decision tree

```
1:  ▷ Note: To build a decision tree, this algorithm is run recursively, starting with
    the root and the full set of training examples. The recursion is at Line 10
2:  procedure HUNTS_FRAMEWORK(S, ...)
3:      v ← new node
4:      if this node should be a leaf then
5:          c ← Determine the class of the leaf
6:          T ← tree of depth 1, with v, which is only leaf, of class c
7:      else
8:          Determine f a function that splits S into l subsets (S_1, ... S_l), i.e. that
            associates each training example of S with one of the l subsets
9:          for S_i ∈ {S_1, ... S_l} do
10:             Build a subtree for this node using this algorithm, with S_i as
                input
11:             Add the subtree root as a child of v
12:         end for
13:         T ← a tree of root v, with the splitting function of v being f
14:     end if
15:     return T
16: end procedure
```

In this thesis, we will focus on binary trees, i.e. decision trees in which internal nodes have exactly 2 children each. In that case, we call one of the children the **right child** and the other the **left child**. The splitting function f can then be seen as a test that outputs a boolean and, by convention, examples

are directed to the right child if the output is `true`, and to the left child if it is `false`. An example of a binary decision tree is given in Figure 1.1. This tree aims at classifying weathers. If this tree predicts that it is snowy, we can easily understand when we see this tree that the reason of this prediction is that the temperature is below 6.5 °C, it is Winter and the humidity is above 57.5%.

Many decision tree building algorithms have a final step that consists in pruning the tree, i.e. reducing its size by turning some internal nodes into leaves and dropping the associated subtree. The methods we propose and study in this thesis do not feature this part, and we therefore only mention it for completeness but will not detail it further. For a survey on decision tree pruning and more generally constraining decision trees, please refer to [44].

1.1.5 Split criteria for decision trees

Before exploring the possible splitting criteria for the decision tree nodes, we need to distinguish between two types of data that the features can be.

Definition 1.10 (Categorical and numerical data)**.** *A feature is said to be **categorical** when its domain is finite and no well-ordering of the values in this domain can be established with sense.*

*A feature is said to be **numerical** when the values in its domain do semantically accept a well-order.*[5]

We note that the definition does not only require the existence of a well-order, but also that this order should make sense. For example, if the domain of a feature is the Airline of a flight, it could accept a well-order as any finite set, but this order would not make sense. Therefore, the feature is categorical. Other examples of categorical data can be a town of origin, a language or a kind of food. Examples of numerical data are heights, temperature or number of people.

In a decision tree, the splitting criteria can rely either on categorical or on numerical data. In the case of categorical data, the splitting criterion is usually in the form of a pair (j, C) with $j \in [1, d]$ and C is a subset of the domain of the feature $\mathrm{dom}(j)$. An example $\boldsymbol{x}$ is directed to the right child if $\boldsymbol{x}_j \in C$. In the case of numerical data, the splitting criterion is usually in the form of a pair $(i, t) \in [1, k] \times \mathbb{R}$, and an example $\boldsymbol{x}$ is directed to the right child if $\boldsymbol{x}_j \leq t$. In that case, t is called the **threshold**.

The solution chosen to define and find efficiently the best splitting criterion is one of the main differences between decision tree algorithms. Let us intro-

[5] It is common in machine learning to consider a third type of data, called *ordinal* data. These data do not map intuitively and directly to numerical values, but do accept a well-ordering. For example, a qualitative evaluation of temperature can take values "cold", "lukewarm" and "hot". In this thesis, we do not consider the specificity of these data and we map them to consecutive integers (e.g. "cold" $\to 0$, "lukewarm" $\to 1$ and "hot" $\to 2$) to treat them as numerical.

duce some of the most commonly used. In these definitions, we denote by $N_{S,y}$ the number of examples of a set S that are associated with the class y.

Definition 1.11 (Gini index and Gini gain)**.** *The **Gini index** of a set $S = \{(\boldsymbol{x_1}, y_1), \dots (\boldsymbol{x_n}, y_n)\}$ of examples $\boldsymbol{x_i}$ associated with classes y_i is the sum of the squared proportion of examples in the set that belong to each class.*

$$g(S) = 1 - \sum_{y \in \mathcal{Y}} \left(\frac{N_{S,y}}{n} \right)^2 \tag{1.7}$$

*The **Gini gain** of a split of a set S which produces two subsets S_1 and S_2 is the difference between the Gini index of S and the weighted mean Gini index of S_1 and S_2:*

$$G(S, S_1, S_2) = g(S) - \left(\frac{|S_1|}{|S|} g(S_1) + \frac{|S_2|}{|S|} g(S_2) \right) \tag{1.8}$$

When the classification problem is binary (i.e. there are only two classes), the Gini index definition can be rewritten to:

$$g(S) = 2 \frac{N_{S,0}}{n} \left(1 - \frac{N_{S,0}}{n} \right) \tag{1.9}$$

The Gini index ranges from 0 to $\frac{|\mathcal{Y}|-1}{|\mathcal{Y}|}$, reaching 0 when the set contains only examples of a single class, and $\frac{|\mathcal{Y}|-1}{|\mathcal{Y}|}$ when it contains the same number of examples of each class. When building a decision tree using this index, the goal will be to minimize the weighted mean Gini index of the subsets of training examples in leaves. Following the Hunt's algorithm, the chosen split of each node will be greedily chosen as the one that maximizes the Gini gain.

Definition 1.12 (Entropy and information gain)**.** *The **entropy** of a set $S = \{(\boldsymbol{x_1}, y_1), \dots (\boldsymbol{x_n}, y_n)\}$ of examples $\boldsymbol{x_i}$ associated with classes $y_i \in \mathcal{Y}$ is defined by*

$$E(S) = \sum_{y \in C : N_{S,y} \neq 0} -\frac{N_{S,y}}{n} \log_2 \left(\frac{N_{S,y}}{n} \right) \tag{1.10}$$

*The **information gain** of a split of a set S which produces two subsets S_1 and S_2 is the difference between the entropy of S and the weighted mean entropy of S_1 and S_2*

$$IG(S, S_1, S_2) = E(S) - \left(\frac{|S_1|}{|S|} E(S_1) + \frac{|S_2|}{|S|} E(S_2) \right) \tag{1.11}$$

The entropy ranges from 0 to $\log_2(|\mathcal{Y}|)$, and these values are reached respectively when the set contains only one class and when each class is equally represented in the set.

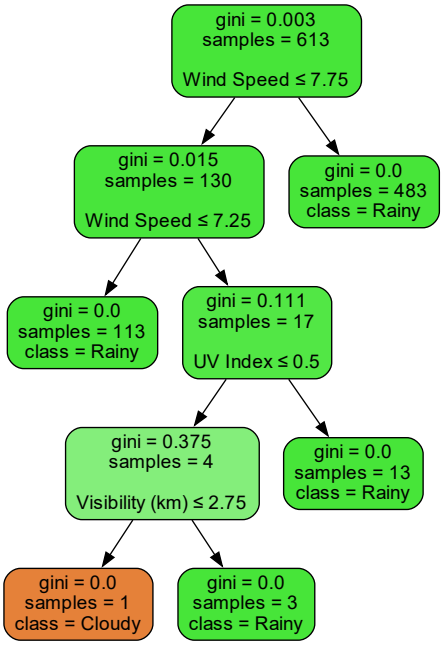


Figure 1.2: Subtree of a decision tree that overfits. The decision tree has been trained on the *Weather* dataset [55]. The root of the subtree represented in this figure is at depth 18 in the full tree. On each node, the Gini index and the size of the training set are indicated. If the node is internal, then the condition is written. The training set of the left child contains the examples that match the condition. If the node is a leaf, the majority class is indicated. The color hue of each node corresponds to its majority class, while the color saturation corresponds to the consensus on that class (inversely related to the Gini index).

1.1.6 Stopping criteria for decision tree building

The criteria used at Line 4 of Algorithm 1.1 to decide if a node should be a leaf are an important part of building decision trees. The most intuitive criterion is to make a node a leaf when the subset of the node is totally homogeneous, i.e. all of its examples are associated with the same class.

The problem with this criterion is that it can result in either very deep trees, overfitting (defined in Section 1.1.3) or both. Very deep trees can be a problem for various reasons, including extended computation times for building and predictions, and reduced interpretability [14, 30].

For example, the tree on Figure 1.1 has been limited to a depth of 3. If we remove this limitation, the tree grows to a depth of 22, and one of its subtrees is represented on Figure 1.2. We can see in that example that the training subset of this whole subtree contains 129 examples labeled "Rainy" and only one example labeled "Cloudy". It is probably that this "Cloudy" example is among the "Rainy" examples because of random variations or incorrect measurements. If an unlabeled example is directed down to this subtree, it seems probable that the target label would be "Rainy". Therefore, the "Cloudy" leaf can probably lead to the misclassification of examples. This hypothesis is strengthened by the fact that, to arrive at the "Cloudy" leaf, the "Wind Speed" feature of the example must be in the range $(7.25, 7.75]$. This feature has a precision of 0.5, so an example that arrives to this subtree needs to have a "Wind Speed" feature value of exactly 7.5 to be classified as "Cloudy".

Three solutions to these problems are widely used, two of which aim to avoid overfitting while the third aims to reduce the depth, although solving one problem generally helps to reduce the other.

The solutions to avoid overfitting have in common that they add conditions for a split to be acceptable. If no acceptable split can be found due to these conditions, no child node is built and the node becomes a leaf.

The first of these solutions is to prevent the split from containing fewer than a given number of training examples. By ensuring that each subset of the tree represents a high enough number of training examples, this solution reduces the possibility that variations due solely to noisy data are considered in the model.

The second solution to avoid overfitting is to make inadmissible the splits that generate less than a given improvement of the objective function, e.g. the splits of which the Gini gain or information gain is below a given threshold. The reasoning behind this solution is that the improvement of the objective function is a proxy for the extent to which the split criterion is relevant to separate the classes. A low improvement would then imply a low relevance, which could even correspond to noise.

The solution to reduce the depth is to simply set as a parameter of the algorithm a maximal depth to which the tree can grow. If a node is to be built at the given maximal depth, it will automatically be a leaf. Not only does this solution prevent deep trees from being built, but by limiting the number of

branching, it also prevents the breaking down of the space of the examples into very small subspaces and thus limits the overfitting to some extent.

1.2 PageRank and graph embedding

1.2.1 Graph generalities

Graphs are a range of mathematical objects used to study the relations between entities. The simplest graphs are defined by two sets V and E.

V is a countable set of objects called **vertices** or **nodes**. These objects can be of any type, but it is common to inject this set into the natural numbers set so that each vertex receives a unique identifier. $E \subset V \times V$ [6] is a set of pairs of vertices, each of these pair being called an **edge**. When graphs are used in practical applications, vertices can correspond to any type of entity, while edges reflect relations or links between a pair of these entities. A graph is called **undirected** if the edges are not directed, i.e. the existence of an edge $(u, v) \in E$ symbolizes a symmetric relation between vertices u and v. In that sense, we can consider that in an undirected graph $(u, v) = (v, u)$. On the opposite, a graph is called **directed** if the edge $(u, v) \in E$ symbolizes a directed relation from u to v and does not imply any similar relation from v to u.

In practical use, and especially when the number of vertices is very large, the graphs are often called **networks**. From a mathematical point of view, however, the words "graph" and "network" are synonyms.

An example of a directed graph is the *routers-rf* network.[7] In this graph, nodes are internet rooters and there is an edge between two rooters if they are connected together. An example of an undirected graph is the *WikipediaFr* graph that we introduce in Section 3.6. In this graph, nodes are Wikipedia pages and there is an edge from a source page to a target page if the source contains an hyperlink to the target.

Definition 1.13 (Order and size of a graph)**.** *The **order** n of a graph is its number of vertices $n = |V|$.*

*The **size** m of a graph is its number of edges $m = |E|$*

If loops, i.e. edges from one vertex to itself, are impossible, then the graph of order n with the maximum size is the graph called the n**-clique**. In such graph, all pairs of vertices are in the edge set. Its order is then $m = n(n-1)$ for directed graphs and $m = \frac{n(n-1)}{2}$ for undirected graphs. The **density** of a graph is the ratio of its size to this maximum size. A graph with low density, i.e. with a number of edges of the same order as the number of vertices, is said to be **sparse**, and most real-world graphs are sparse.

[6] This notation implies that each pair of vertices can be connected by at most one edge. This is often assumed to be true, and it is in this thesis.

[7] This network is available on the website *networkrepository.com* [50].

Definition 1.14 (Weighted graph)**.** *A **weighted graph** is a graph defined with an extra set $W \in \mathbb{R}^m$ that defines a weight for each edge.*

The weight of an edge generally reflects the strength of the link or connection between the vertices.

The simplest way to represent a graph is to give its order and, using $V = [0, n-1]$, listing its edges. However, in many cases it is convenient to represent it in the form of the so-called adjacency matrix.

Definition 1.15 (Adjacency matrix)**.** *The **adjacency matrix** of a graph is the matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ defined by*

$$\boldsymbol{A}_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{else} \end{cases}$$

We note that for an undirected graph, the matrix is symmetric. In the case of weighted graphs, the elements of the matrix are the weights of the edges, instead of 1.

Another important notion is that of neighbors of a vertex.

Definition 1.16 (Neighbors)**.** *In a graph $G = \{V, E\}$, the set $\mathcal{N}(u)$ of **neighbors** of a vertex u is the set of vertices that shares an edge with u.*

$$\mathcal{N}(u) = \{v \in V : (u,v) \in E \vee (v,u) \in E\} \tag{1.12}$$

*In the case of directed graphs, we define the sets $\mathcal{N}^-(u)$ of **out-neighbors** and $\mathcal{N}^+(u)$ of **in-neighbors** of u as the sets of nodes such that there is, respectively, an edge coming from u to them, and going from them to u.*

$$\begin{aligned} \mathcal{N}^-(u) &= \{v \in V : (u,v) \in E\} \\ \mathcal{N}^+(u) &= \{v \in V : (v,u) \in E\} \end{aligned} \tag{1.13}$$

In the case of undirected graphs, the sets of out- and in-neighbors are equivalent to the set of neighbors.

The set of neighbors is sometimes called neighborhood. From this notion, we can define the degree of a vertex and, in the case of directed trees, the in- and out-degree:

Definition 1.17 (Degree)**.** *In a graph $G = \{V, E\}$, the **degree** d_u of a vertex u is the size of its neighborhood:*

$$d_u = |\mathcal{N}(u)| \tag{1.14}$$

*In the case of a directed graph, the **in-degree** d_u^- and **out-degree** d_u^+ of a node are respectively the size of its in- and out-neighborhood:*

$$\begin{aligned} d_u^- &= |\mathcal{N}^-(u)| \\ d_u^+ &= |\mathcal{N}^+(u)| \end{aligned} \tag{1.15}$$

1.2.2 The centrality problem

A common problem on graphs, called the **centrality problem**, is to determine how central each vertex is in the graph. This notion of centrality can be defined in various manners that will affect the metric that will be used to compute it.

The main application of the centrality metrics is to find out which nodes are the most important or influential in a network. For example, the PageRank algorithm [47], which we use extensively in this thesis, was first developed to rank the relative importance of web pages as answers to a Google search.

Many centrality metrics rely on paths and distances in the graph. We first define these notions. A path is a series of adjacent edges leading from one vertex to another.

Definition 1.18 (Path)**.** *In a graph $G = \{V, E\}$, a **path** P from u^* to v^*, with $u^*, v^* \in V$ is a sequence of edges $P = \{(u_i, v_i) \in E : i \in [1, |P|]\}$ so that $u_1 = u^*$, $v_{|P|} = v^*$ and $\forall i \in [1, |P| - 1]$, we have $u_{i+1} = v_i$.*

Definition 1.19 (Distance between vertices)**.** *The **distance** $d(u, v)$ between two vertices $u, v \in V$ is the length of the shortest path from u to v. By convention, the distance between two vertices with no path between them is defined as $+\infty$.*

*Any path from u to v of size equal to the distance between them is called a **geodesic**.*

We present a brief history of the centrality metrics and an introduction to some main ones used in the literature.

One of the first papers to talk about centrality [3] was published in 1950. This paper was very application-oriented as it mainly studied the structures of collaboration to conduct tasks in groups, but these structures were modeled as graphs, which prompted the authors to propose a metric for centrality that is now known as **closeness centrality**.

Definition 1.20 (Closeness centrality)**.** *In a graph $G = \{V, E\}$, the closeness centrality $C_C(u)$ of a vertex $u \in V$ is defined as*

$$C_C(u) = \frac{1}{\sum_{v \in V} d(u, v)} \tag{1.16}$$

Various variations of the numerator have been proposed to normalize the metric. One of the drawbacks of this metric is that it only works for connected graphs, i.e. graphs in which there exists a path between any pair of nodes. This is because the distance between two unconnected points is $+\infty$.

Shortly after, in 1953, the Katz centrality index [32], which is one of the most popular centrality metrics up to this date, was introduced.

Definition 1.21 (Katz centrality index)**.** *In a graph $G = \{V, E\}$, the **Katz Centrality Index** of parameter $\alpha \in [0, 1)$ of a vertex $u \in V$ is defined as:*

$$C_{Katz}(u) = \sum_{k=1}^{+\infty} \sum_{v \in V} \alpha^k (\boldsymbol{A}^k)_{uv} \tag{1.17}$$

For unweighted graphs and for two vertices $u, v \in V$, $(\boldsymbol{A}^k)_{uv}$ is the number of distinct paths of length k from u to v. The sum $\sum_{v \in V} (\boldsymbol{A}^k)_{uv}$ is therefore the number of distinct paths starting at u and of length k that can be found in the graph. The Katz Centrality index is the pondered sum of this number for each value of k, the longer paths having a lower importance in the sum.

We note that α needs to be smaller than the inverse of the maximum Eigenvalue of $\boldsymbol{A}$ to ensure the convergence of the series.

In 1972, Bonacich proposed the Eigenvector centrality [7]:

Definition 1.22 (Eigenvector centrality)**.** *The **Eigenvector centrality** vector $\boldsymbol{x}$ of a graph G is the largest Eigenvector of the adjacency matrix.*

$$\begin{cases} \exists \lambda \in \mathbb{R} : \boldsymbol{A}\boldsymbol{x} = \lambda \boldsymbol{x} \\ \forall \lambda^* \in \mathbb{R}, \boldsymbol{x^*} \in \mathbb{R}^n : \boldsymbol{A}\boldsymbol{x^*} = \lambda^* \boldsymbol{x^*} \Rightarrow \lambda^* \leq \lambda \end{cases} \tag{1.18}$$

The Eigenvector centrality of a given vertex $u \in V$ is the value $\boldsymbol{x}_u$ of the element of the vector $\boldsymbol{x}$ associated with this vertex.

A property of this metric is that $\forall u \in V : \lambda \boldsymbol{x}_u = \sum_{v \in V} \boldsymbol{A}_{uv} \boldsymbol{x}_v$. In other words, the Eigenvector centrality of a vertex is high when the Eigenvector centrality of its neighbors is high.

It has been shown that Eigenvector centrality is related to Katz centrality, the latter approximating the former when $\alpha \to \frac{1}{\lambda}$ [8].

In 1977, another centrality metric that is still widely used today was introduced in [20]. This metric called **Betweenness Centrality** relies on the geodesics of the graph and, to define it, we introduce the notations σ_{uv} which is the number of geodesics from vertex u to vertex v. We also denote by $\sigma_{uv}(w)$ the number of geodesics from u to v that contain the vertex w.

Definition 1.23 (Betweenness Centrality)**.** *In a graph $G = \{V, E\}$, the **Betweenness Centrality** of a vertex w is the sum, for each pair of vertices, of the ratio of geodesics between these vertices that contain w:*

$$C_B(w) = \sum_{u,v \in V \times V : u \neq v \neq w} \frac{\sigma_{uv}(w)}{\sigma_{uv}} \tag{1.19}$$

When the graph is undirected, this value needs to be divided by 2 to account for the fact that each pair of vertices is counted twice.

The value $\frac{\sigma_{uv}(w)}{\sigma_{uv}}$ equals 0 when no shortest path from u to v contains w and 1 when all the shortest paths do (e.g. when there is only one shortest path from u to v). In that sense, the Betweenness Centrality of w is the number of pairs of vertices that the removal of w would spread apart, but it also considers the cases when w is on some geodesics between two vertices but not all. The interpretation of the authors is that the Betweenness Centrality of a vertex is a metric of the control the vertex has on information passing in the graph.

Finally in 1999, the centrality metric that interests us the most in this thesis, PageRank, was introduced in a very famous paper [47].

To define PageRank, we first define the concept of random walk.

Definition 1.24 (Random walk)**.** *Given a graph $G = \{V, E\}$, a **random walk** on the graph is a sequence $\{V_t\}_{t\in\mathbb{N}}$ of random variables. The domain of each variable is the vertices of the graph, and the probability distribution of each variable V_t is defined as a function of the realizations of the previous variables $\{V_t\}_{t\in[0,t-1]}$.*

To clarify the explanations about a random walk, it is convenient to represent it as an entity that we call "**walker**" that goes randomly from one vertex to another following the rules of the random walk. Therefore, in this thesis, we will use phrases like "the walker is on the vertex u at time t" or "the position of the walker at time t is u", meaning that the realization of the random variable V_t is u. We will also use phrases like "the probability of the walker to go from u to v", meaning the conditional probability that $V_{t+1} = v$ given that $V_t = u$.

Some random walks also accept an extra vertex w in the domain of the positions. This extra vertex is not a vertex of the graph. If the walker is on this vertex, it can never move again. Formally, $v_{t^*} = w \Rightarrow \forall t > t^*, v_t = w$. When the walker goes to that extra vertex, we say that the walker **stops walking**, and a realization of the random variable can be defined simply by the time f at which the walker stops walking and the previous positions $\{V_t\}_{t\in[0,f-1]}$.

We can now introduce the random walk on which PageRank is based. In that random walk, when the walker in on a vertex u, there is an equal probability that it goes to any out-neighbors of u.[8]

We denote by $\boldsymbol{D} \in \mathbb{R}^{n\times n}$ the diagonal matrix of out-degrees of the graph, i.e. the matrix in which each diagonal element is the number of edges starting from the related vertex. We can then define $\boldsymbol{M} = \boldsymbol{D}^{-1}\boldsymbol{A}$ the **stochastic matrix** of the random walk, i.e. $\boldsymbol{M}_{uv}$ is the probability that the walker will go to vertex v on the next step given that it is on vertex u at this step.

Finally, we define a random walk with restart of parameter α as a random walk in which, at each step, the walker has a probability α to go onto a random vertex of the graph instead of necessarily going to an out-neighbor. In that case, we say that the walker **restarts**. The probability distribution of the vertex it restarts from is the same as the probability distribution of the initial position V_0.

Definition 1.25 (PageRank score)**.** *In a graph $G = \{V, E\}$, the **PageRank score** $\boldsymbol{\pi}_u$ of a vertex $u \in V$ with parameter α is the asymptotic probability that the position of the walker is the vertex u at a step of the random walk with restart of parameter α*

$$\boldsymbol{\pi}_u = \lim_{t\to+\infty} \mathbb{P}\left(V_t = u\right) = \frac{\alpha}{n}\sum_{i=0}^{+\infty}\sum_{v\in V}(1-\alpha)^i(\boldsymbol{M}^i)_{uv} \tag{1.20}$$

[8] In the case of weighted graphs, another version is to give each vertex a probability proportional to its weight.

Some of these centrality metrics exist in **personalized** version, i.e. metrics for the importance of each vertex in relation to a given subset W of vertices instead of the whole graph. This is particularly true for the Katz and PageRank metrics. For Katz, the personalized version can be obtained by computing the sum over the given subset W of vertices instead of all the vertices in the graph.

$$P_{\text{Katz}}(W, u) = \sum_{k=1}^{+\infty} \sum_{v \in W} \alpha^k (\boldsymbol{A}^k)_{uv} \tag{1.21}$$

For PageRank, the personalized version consists in reducing the start and restart set of vertices of the random walk to the given subset. Mathematically, this has the same impact as summing only over the given subset, only also affecting the normalization.

$$P_{\text{PageRank}}(W, u) = \frac{\alpha}{|W|} \sum_{i=0}^{+\infty} \sum_{v \in W} (1-\alpha)^i (\boldsymbol{M}^i)_{uv} \tag{1.22}$$

When the subset is reduced to a single vertex, the metric is either called a personalized version or the **rooted version** of the metric. In this thesis, unless specified otherwise, we will use the term "**Personalized PageRank**" (PPR) for this rooted version because it is the one that most interests us and it is easy to see that personalized versions with bigger subsets are just linear combinations of the rooted ones. A more in-depth study of PageRank and PPR is presented in Section 3.2.2

1.2.3 Singular Value Decomposition

The **Singular Value Decomposition** (SVD) is a method of matrix factorization.

Definition 1.26. *The **Singular Value Decomposition** of a matrix $\boldsymbol{M} \in \mathbb{R}^{m \times n}$ is a group of a rectangular diagonal matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{m \times n}$ and two unitary matrices $\boldsymbol{U} \in \mathbb{R}^{m \times m}$ and $\boldsymbol{V} \in \mathbb{R}^{n \times n}$ such that*

$$\boldsymbol{M} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top \tag{1.23}$$

A real square matrix $\boldsymbol{U}$ is said to be **unitary** if $\boldsymbol{U}^\top \boldsymbol{U} = \boldsymbol{U}\boldsymbol{U}^\top = \boldsymbol{I}$ the identity matrix.

We can assume that $m \leq n$. This assumption is made without loss of generality, since the transposed matrix can be used when the assumption is false.

The columns $\{\boldsymbol{u_1}, \dots \boldsymbol{u_m}\}$ and $\{\boldsymbol{v_1}, \dots \boldsymbol{v_n}\}$ of the matrices $\boldsymbol{U}$ and $\boldsymbol{V}$ are called the **left and right singular vectors** of the matrix, respectively, and the diagonal elements $\{\sigma_1, \dots, \sigma_m\}$ of $\boldsymbol{\Sigma}$ are called the **singular values**. For a given index $i \in [m]$, the triplet $\{\sigma_i, \boldsymbol{u_i}, \boldsymbol{v_i}\}$ is called a singular triplet. The equation in the SVD decomposition can be rewritten as a combination of singular triplets:

$$\boldsymbol{M} = \sum_{i=1}^{m} \sigma_i \boldsymbol{u_i} \boldsymbol{v_i}^\top \tag{1.24}$$

The SVD is massively used in Data Science for reducing the dimensions of data by removing redundancy and keeping only the most influential features. In that case, the lines of the matrix are usually the samples while the columns are the features. We can see from Equation (1.24) that if the values σ_i are indexed in decreasing order, then the matrix $\boldsymbol{M}$ can be approached by a limited number k of them

$$\boldsymbol{M} \approx \sum_{i=1}^{k} \sigma_i \boldsymbol{u_i} \boldsymbol{v_i}^\top \tag{1.25}$$

This equation is sometimes written in its matrix form

$$\boldsymbol{M} \approx \boldsymbol{U_k} \boldsymbol{\Sigma_k} \boldsymbol{V_k}^\top \tag{1.26}$$

where $\boldsymbol{U_k}$ and $\boldsymbol{V_k}$ are made of the first k columns of $\boldsymbol{U}$ and $\boldsymbol{V}$, and $\boldsymbol{\Sigma_k}$ is $\boldsymbol{\Sigma}$ reduced to its $k \times k$ upper-left submatrix.

This technique is called **truncated SVD**. It has been proven that this approximation is the best possible of order k, meaning that there is no set of k triplets $\{\sigma_i, \boldsymbol{u_i}, \boldsymbol{v_i}\}$ that would result in a better approximation in Equation (1.25). This result is known as the Eckart-Young theorem [17]. When applied to a matrix of data of which the features are centered, it is equivalent to the well-known Principal Components Analysis.

When applied to a matrix of data with each line representing a sample and each column a feature, the right singular vectors associated with the highest singular values represent the "patterns" of features that have the highest importance to represent the data, i.e. linear combination of the features that can be multiplied by constants to approximate the data at best. The left singular vectors multiplied by the singular values are these constants, i.e. the importance of each right singular vector to represent the data.

1.2.4 Graph embedding

The data mining technique of **embedding** consists in simplifying the study of complex objects by representing them into a low-dimensional vectorial space. For example, in Natural Language Processing, the word embedding technique consists in representing each word of a corpus in a vectorial space so that words from a same lexicon are close together.

In graph mining, the objects that are embedded can be either vertices, edges or the entire graph. In this thesis, we focus on the case of vertices. This technique can be applied for solving various problems as node recommendation, link prediction or node classification. For most of these problems there exist algorithms running directly on graphs that can solve them, but it is not always possible to use other external data. In addition, since these algorithms are specifically designed for graphs, they do not benefit from the same attention as similar algorithms for vectorial data, a problem that embedding solves. Finally, the embedding, although often computationally costly, can be computed only once and then be used to address multiple problems.

The name "**graph embedding**" can stand either for the embedding of the whole graph as one vector or for the embedding of the components of the graph, either the vertices, the edges or both, as multiple vectors, one for each component. In this thesis we are interested in the embedding of vertices and unless specified otherwise, we will always use the name "graph embedding" in its "vertices embedding" meaning.

Chapter 2

Dynamic Decision Trees

2.1 Introduction

Decision trees are a cornerstone of machine learning and an essential tool in any machine learning library. Decision-tree-based algorithms, such as random forests [9] or XGBoost [13], are widely used to solve real-world machine learning problems. They also boast the appealing feature of being explainable [30, 14]. As many other machine learning problems, the classification problem has a dynamic version, as explained in Section 1.1.2. In that version, the training dataset evolves over time by inserting and deleting labeled examples. The dynamic classification problem is called **fully dynamic** if insertions and deletions are permitted, and **online** or **incremental** if only the former applies.

In the latter case, there is an additional difficulty called **concept drift**. In that case, the underlying model, which assign data to one ground-truth class or another, changes over time, so that the relevance of the data is maximal at the time of insertion and slowly decays as other data is inserted. Thus, if the model is updated by considering each data equally, its accuracy will be sharply reduced. If the algorithm used to build the model can handle the fully dynamic problem, then it is possible to handle the concept drift by working on a window of data, i.e. setting a constant t and only working on the last t inserted data, deleting the oldest example each time a new one is inserted. There are however algorithms built specifically for the online problem that can handle concept drift.

Many variants of the decision tree method exist to deal with the online problem. However, to the best of our knowledge, the only variant compatible with the fully dynamic problem is BOAT [22], and this method was not specifically designed for this task.

2.1.1 Main contributions

This chapter presents a new algorithm for fully dynamic decision trees building and maintenance, called FUDYADT. To the best of our knowledge, this is the first decision tree algorithm to be specifically designed for the fully dynamic setting. This new algorithm is based on a new theorem about the smoothness of the Gini index and Gini gain, and also on a new objective for dynamic decision trees. Theoretical guarantees and experimental results are also provided to show the performance of our new algorithm.

2.1.2 Organization of the chapter

In this chapter, we first present a brief review of the literature about dynamic decision trees. Then, we formulate the preliminary concepts necessary for understanding the FUDYADT algorithm. Notably, among these preliminaries, we state and prove a theorem on the smoothness of the Gini index and gain, and we define a new objective for dynamic decision trees called "ϵ-feasibility". The two following sections present the FUDYADT algorithm itself and its theoretical properties, and then its experimental performance compared, when relevant, to the state-of-the-art EFDT algorithm, presented in Section 2.2. Finally, we conclude with a summary of the findings and suggestions for future works around this topic.

2.1.3 Acknowledgements

Sections 2.3 and 2.4 are a modified version of the conference paper "Fully-Dynamic Decision Trees" [10] presented in 2023 at the AAAI conference. This paper is reproduced here in a modified version in compliance with the copyright agreement.

My contribution to the article was mainly on the "experiments" part. However, as part of my doctoral work, I extended the theorems and proof to training sets with more than two labels. These extended versions are the ones presented in that chapter. I did not contribute significantly to section 4 of the paper that presents lower bounds on the memory use and computation time of algorithms that build ϵ-feasible decision trees, nor did I expand it. Therefore, I only present it briefly in Section 2.4.5.

2.2 Online decision trees: A brief literature review

There is a rich literature for online decision tree algorithms (see [41] for a survey). We propose a quick overview of this literature.

The BOAT algorithm To the best of our knowledge, the first algorithm to build and maintain decision trees in a dynamic setting is BOAT [22]. This is

also, to the best of our knowledge, the only algorithm before ours to maintain decision trees in a fully dynamic environment, not only an incremental one.

The BOAT algorithm was primarily designed to build a decision tree from huge data, too big to be represented at once in memory. It works in the following way: first, it takes a sample of the data and uses this sample to create a sample decision tree. In this sample decision tree, the splitting features correspond with high probability to the splitting features of the target decision tree, and for each splitting threshold of the target tree, a range is given to which the threshold belongs with high probability. Then, the data are sequentially loaded in memory. The only data kept in memory are those that could contribute in determining a target threshold, i.e. the data are directed to a node, where their value on the splitting feature is within the range of the splitting threshold. Finally, the algorithm goes up through the tree to detect where the best split was possibly missed, and in that case the related subtree is rebuilt.

The dynamic maintenance of the tree is done by considering that the training set at a given time is a sample of the training set after new insertions and deletions. This assumption is reasonable, even in a concept drift setting, as long as the concept drift is slow enough. The last part of the algorithm, which checks if the best split was possibly missed, will trigger the recomputation of some subtrees when the data have changed so much that the best split has changed.

Hoeffding trees The Hoeffding tree algorithms, first introduced in [16] and then improved and expanded in [21, 29, 31, 51], aims to build the tree using incremental insertions of data, while using as little memory as possible. The main idea is that the data are directed to leaves as they arrive, and only statistics about the data are stored in each leaf. These statistics allow for the computation of the gain, whether it is information gain or Gini gain, to split the data. Once the difference between the two best splits exceeds a threshold called Hoeffding bound, the leaf becomes an internal node. This bound ensures that the split is the best with high probability. However, the statistics do not allow directing the examples that contributed to find the split into the new leaves, therefore, the new leaves are created with their statistics at 0.

Extremely Fast Decision Tree (EFDT) Finally, in 2018, the state-of-the-art algorithm named Extremely Fast Decision Tree (EFDT), also known as Hoeffding AnyTime Tree (HATT), was introduced [40]. This algorithm improves the Hoeffding Tree algorithm by allowing to reconsider a split that was performed before. Indeed, when the Hoeffding tree algorithm creates an internal node, it considers this node permanent and drops the statistics that led to the split. EFDT, on the other hand, keeps this statistics and keeps them up to date. This way, if the best split according to the statistics would become much better than the split used before, it will reconsider and change the split. This process of reconsidering the splits allows EFDT to converge faster to better

solutions, and also to be resilient to concept drift. A paper published in 2022 [41] was dedicated to compare EFDT to other algorithms through "the largest and most comprehensive set of testbenches in the online learning literature", and it showed that EFDT is "a superior alternative to Hoeffding Tree in a large number of ensemble settings".

Hoeffding trees and most of the following algorithms for online decision trees focus on minimizing the memory usage. However, when working with fully dynamic decision trees, it is much more difficult to reduce memory usage. The theorems presented in Section 2.4.5 will show that it is impossible to build and maintain fully dynamic decision trees that satisfy some reasonable criteria (ϵ-feasibility, presented in section 2.3.2) with a memory usage lower than $\Omega\big(\frac{n \cdot d}{k \cdot \log n}\big)$. Therefore, the algorithm presented in this chapter focuses on minimizing the amortized computation time per update, rather than the memory usage.

2.3 Preliminaries

2.3.1 Gini index and Gini gain smoothness

The Gini index and Gini gain were introduced in Section 1.1.5. The Gini gain is one of the main metrics used for determining the best splits in decision trees. We want to study to which extent a limited number of modifications to the training set can change the Gini index or the best Gini gain from the set.

We use the following notation with o denoting a boolean operation:

$$\delta_o = \begin{cases} 1 & \text{if } o \text{ is } \texttt{true} \\ 0 & \text{else} \end{cases} \tag{2.1}$$

We then have the following equivalent definition of the Gini index:

Property 2.1. *We denote by $S = \{(\boldsymbol{x_1}, y_1), \dots (\boldsymbol{x_n}, y_n)\}$ a set of labeled examples. Then we have:*

$$g(S) = \frac{2}{n^2} \sum_{i=1}^{n} \sum_{j=i+1}^{n} \delta_{y_i \neq y_j} \tag{2.2}$$

Proof. We have $\sum_{i=1}^{n} \sum_{j=1}^{n} \delta_{y_i = y_j} = \sum_{y \in \mathcal{Y}} N_{S,y}^2$. From Equation (1.7) we have:

$$\begin{aligned} g(S) &= 1 - \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \delta_{y_i = y_j} \\ \Leftrightarrow g(S) &= \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} 1 - \delta_{y_i = y_j} \end{aligned} \tag{2.3}$$

We know that $1 - \delta_{y_i = y_j} = \delta_{y_i \neq y_j}$. We rearrange the terms to find the sum in Equation (2.2) and conclude the proof. □

We call **edit distance** $\triangle(S, S')$ from one set of data S to another S' the minimal number of insertions and deletions needed to build S' from S. We first establish two lemmas on the local smoothness of the Gini index and the Gini gain:

Lemma 2.2. *Let S, S' be two sets of training data of size at least 1. If $\triangle(S, S') \leq 1$. then $|g(S) - g(S')| \leq \frac{2}{\max(|S|,|S'|)}$*

Lemma 2.3. *Let S, S' be two sets of training data of size at least 1. Let S_1 and S_2 the two subsets created by a splitting function applied on S, and S'_1 and S'_2 the subsets created by the same splitting function applied on S'. If $\triangle(S, S') \leq 1$. then*

$$|G(S, S_1, S_2) - G(S', S'_1, S'_2)| \leq \frac{12}{\max(|S|, |S'|)}$$

Proof of Lemma 2.2. The case $\triangle(S, S') = 0$ is trivial, therefore, we assume that $\triangle(S, S') = 1$. Let $n = |S|$. Without loss of generality, we assume that $|S| = |S'| + 1$, and we denote $S = \{(\boldsymbol{x_1}, y_1)\} \cup S' = \{(\boldsymbol{x_1}, y_1), \dots (\boldsymbol{x_n}, y_n)\}$.

Let $D_S = 2\sum_{i=1}^{n}\sum_{j=i+1}^{n} \delta_{y_i \neq y_j}$. We see that $g(S) = \frac{1}{n^2} D_S$. We also see that:

$$D_S - 2(n-1) \leq D_{S'} \leq D_S$$

therefore we have:

$$g(S') - g(S) \geq \left(\frac{1}{(n-1)^2} - \frac{1}{n^2}\right) D_S - \frac{2}{n-1} \tag{2.4}$$

$$g(S') - g(S) \leq \left(\frac{1}{(n-1)^2} - \frac{1}{n^2}\right) D_S \tag{2.5}$$

We can factorize each inequality using $\frac{1}{(n-1)^2} - \frac{1}{n^2} = \frac{2n-1}{n^2(n-1)^2}$

We first study (2.5). We know that $D_S < (n-1)^2$, and so we have:

$$g(S') - g(S) < \frac{2n-1}{n^2} < \frac{2}{n}$$

Now, we study (2.4). First, note that if $D_S = 0$ then $g(S) = g(S') = 0$ and the lemma holds. We therefore focus on the case when $D_S > 0$.

We note that if we consider the class y with the lowest number of examples in S and move one example from that class to another, then the value of D_S is necessarily reduced or equal. We can apply this process recursively on any training set S s.t. $D_S > 0$, to create a new training set S^* in which all examples but one are in the same class. Since $D_{S^*} = 2(n-1)$ and the process only decreases D_S, we deduce that if $D_S > 0$ then $D_S \geq 2(n-1)$.

Then, (2.4) becomes:

$$\begin{aligned} g(S') - g(S) &\geq \frac{2n-1}{n^2(n-1)^2} D_S - \frac{2}{n-1} \\ &\geq 2\frac{2n-1}{n^2(n-1)} - \frac{2}{n-1} \\ &\geq -\frac{2}{n}\left(\frac{n^2-2n+1}{n(n-1)}\right) \\ &\geq -\frac{2}{n}\left(\frac{n-1}{n}\right) \\ &\geq -\frac{2}{n} \end{aligned}$$

□

Proof of Lemma 2.3. The lemma is trivial if $\triangle(S, S') = 0$, therefore, we focus on the case when $\triangle(S, S') = 1$. From the definition of the Gini gain and the triangular inequality, we get:

$$|G(S, S_1, S_2) - G(S', S'_1, S'_2)| \leq |g(S) - g(S')| \tag{2.6}$$
$$+ \left| \frac{|S_1|}{|S|} g(S_1) - \frac{|S'_1|}{|S'|} g(S'_1) \right| \tag{2.7}$$
$$+ \left| \frac{|S_2|}{|S|} g(S_2) - \frac{|S'_2|}{|S'|} g(S'_2) \right| \tag{2.8}$$

From Lemma 2.2, we know an upper bound for the part numbered (2.6). We now search a bound for the two other parts, (2.7) and (2.8). Since $\triangle(S, S') = 1$, we know that either $S_1 = S'_1$ or $S_2 = S'_2$. Without loss of generality, we assure $S_2 = S'_2$

$$\begin{aligned} \tfrac{|S_1|}{|S|} g(S_1) &\leq \tfrac{|S_1|}{|S|} \left(g(S'_1) + \tfrac{2}{\max(|S_1|, |S'_1|)} \right) && \text{from Lemma 2.2} \\ &\leq \tfrac{|S_1|}{|S|} g(S'_1) + \tfrac{2}{|S|} \\ &\leq \tfrac{|S_1|+1}{|S|+1} g(S'_1) + \tfrac{2}{|S|} && \text{because } |S_1| \leq |S| \\ &\leq \tfrac{|S'_1|+2}{|S'|} g(S'_1) + \tfrac{2}{|S|} \\ &< \tfrac{|S'_1|}{|S'|} g(S'_1) + \tfrac{2}{|S'|} + \tfrac{2}{|S|} && \text{because } g(S'_1) < 1 \\ &\leq \tfrac{|S'_1|}{|S'|} g(S'_1) + \tfrac{6}{\max(|S|, |S'|)} && \text{because } \tfrac{1}{2} \leq \tfrac{|S|}{|S'|} \leq 2 \end{aligned} \tag{2.9}$$

Inverting S and S' in this sequence of inequalities leads to:

$$\frac{|S'_1|}{|S'|} g(S'_1) \leq \frac{|S_1|}{|S|} g(S_1) + \frac{6}{\max(|S|, |S'|)} \tag{2.10}$$

Together, these two inequalities lead to:

$$\left|\frac{|S_1|}{|S|}g(S_1) - \frac{|S'_1|}{|S'|}g(S'_1)\right| \leq \frac{6}{\max(|S|,|S'|)} \tag{2.11}$$

The sequence of inequalities can also be used to find a bound to the part numbered (2.8) but, since $S_2 = S'_2$, we can remove $\frac{2}{|S|}$ until the penultimate inequality and therefore the bound is:

$$\left|\frac{|S_2|}{|S|}g(S_2) - \frac{|S'_2|}{|S'|}g(S'_2)\right| \leq \frac{4}{\max(|S|,|S'|)} \tag{2.12}$$

Altogether, we have:

$$\begin{aligned} |G(S,S_1,S_2) - G(S',S'_1,S'_2)| \leq & \frac{2}{\max(|S|,|S'|)} \\ & + \frac{6}{\max(|S|,|S'|)} \\ & + \frac{4}{\max(|S|,|S'|)} \end{aligned} \tag{2.13}$$

□

Noting $\triangle^*(S,S') = \frac{\triangle(S,S')}{\max(|S|,|S'|)}$ the relative edit distance, we can now establish the following theorem:

Theorem 2.4. *Let S, S' be two sets of training data of size at least 1. Let S_1 and S_2 be the two subsets created by a splitting function applied on S, and S'_1 and S'_2 the subsets created by the same splitting function applied on S'. Then:*

1. $|G(S,S_1,S_2) - G(S',S'_1,S'_2)| \leq \left(13 - \frac{1}{|\mathcal{Y}|}\right) \triangle^*(S,S')$
2. $|g(S) - g(S')| \leq \left(3 - \frac{1}{|\mathcal{Y}|}\right) \triangle^*(S,S')$

This theorem proves that small variations in the set of data only causes small variations in the Gini index, and in the Gini gain of any split of this set.

Proof for the Gini gain part. If $\triangle^*(S,S') \geq \frac{|\mathcal{Y}|-1}{13|\mathcal{Y}|-1}$, then

$$(13 - \frac{1}{|\mathcal{Y}|})\triangle^*(S,S') \geq \frac{|\mathcal{Y}|-1}{|\mathcal{Y}|} \geq |G(S,S_1,S_2) - G(S',S'_1,S'_2)|$$

and the theorem holds. Therefore, we focus on the case when $\triangle^*(S,S') < \frac{|\mathcal{Y}|-1}{13|\mathcal{Y}|-1}$.

Let $l = \triangle(S, S')$. Then, there exist a sequence of training sets $\{S^{(0)}, \dots S^{(l)}\}$ s.t. $S^{(0)} = S$, $S^{(l)} = S'$ and $\forall i \in [0, l-1] : \triangle(S^{(i)}, S^{(i+1)}) = 1$. We have:

$$\begin{aligned}
|S^{(i)}| &\geq \max(|S|, |S'|) - \triangle(S, S') \\
&\geq (1 - \triangle^*(S, S')) \max(|S|, |S'|) \\
&\geq (1 - \frac{|\mathcal{Y}| - 1}{13|\mathcal{Y}| - 1}) \max(|S|, |S'|) \qquad \text{because } \triangle^*(S, S') < \frac{|\mathcal{Y}| - 1}{13|\mathcal{Y}| - 1} \\
&\geq \frac{12|\mathcal{Y}|}{13|\mathcal{Y}| - 1} \max(|S|, |S'|) \qquad (2.14)
\end{aligned}$$

We see from the triangular inequality that:

$$|G(S, S_1, S_2) - G(S', S'_1, S'_2)| \leq \sum_{i=0}^{l-1} |G(S^{(i)}, S_1^{(i)}, S_2^{(i)}) - G(S^{(i+1)}, S_1^{(i+1)}, S_2^{(i+1)})|$$

and so we have:

$$\begin{aligned}
|G(S, S_1, S_2) - G(S', S'_1, S'_2)| &\leq \sum_{i=0}^{l-1} \frac{12}{\max(|S^{(i)}|, |S^{(i+1)}|)} && \text{Lemma 2.3} \\
&\leq \sum_{i=0}^{l-1} \frac{12}{\frac{12|\mathcal{Y}|}{13|\mathcal{Y}|-1} \max(|S|, |S'|)} && \text{From Equation (2.14)} \\
&\leq \left(13 - \frac{1}{|\mathcal{Y}|}\right) \triangle^*(S, S') && \text{because } l = \triangle(S, S')
\end{aligned}$$

□

Proof for the Gini index part. This proof is very similar to the one for the Gini gain part, so we do not go into as many details. We first see that if $\triangle^*(S, S') \geq \frac{|\mathcal{Y}|-1}{3|\mathcal{Y}|-1}$ then the theorem holds, therefore, we focus on the case when $\triangle^*(S, S') < \frac{|\mathcal{Y}|-1}{3|\mathcal{Y}|-1}$.

We define a sequence of training sets as in the previous proof. Knowing that $\triangle^*(S, S') < \frac{|\mathcal{Y}|-1}{3|\mathcal{Y}|-1}$ and in the same way that we have shown Equation (2.14), we show that:

$$|S^{(i)}| \geq \frac{2|\mathcal{Y}|}{3|\mathcal{Y}| - 1} \max(|S|, |S'|) \qquad (2.15)$$

Finally, we can use the triangular inequality in the same way and find the following:

$$\begin{aligned}
|g(S) - g(S')| &\leq \sum_{i=0}^{l-1} \frac{2}{\max(|S^{(i)}|, |S^{(i+1)}|)} && \text{Lemma 2.2} \\
&\leq \sum_{i=0}^{l-1} \frac{2}{\frac{2|\mathcal{Y}|}{3|\mathcal{Y}|-1} \max(|S|, |S'|)} && \text{From Equation (2.15)} \qquad (2.16) \\
&\leq \left(3 - \frac{1}{|\mathcal{Y}|}\right) \triangle^*(S, S') && \text{because } l = \triangle(S, S')
\end{aligned}$$

□

2.3.2 ϵ-feasibility: a new performance guarantee for dynamic decision trees

Fully computing a decision tree is computationally expensive. It would therefore not be efficient to recompute the whole decision tree at each insertion or deletion of an example. To avoid this, our goal is not to compute the perfect decision tree at each step. Instead, we introduce a new objective called ϵ-feasibility. Roughly speaking, a decision tree is ϵ-feasible if the Gini gain of each of its splits is not too far from the Gini gain of the optimal split.

To clarify this definition, we denote by $S^+_{v,j,a}$ and $S^-_{v,j,a}$ the left and right subsets of a training set S_v when splitting along the feature j and with threshold or category a. It means that if the feature j is numerical, then:

$$\begin{aligned} S^-_{v,j,a} &= \{(\boldsymbol{x}, y) \in S : \boldsymbol{x}_j \leq a\} \\ S^+_{v,j,a} &= \{(\boldsymbol{x}, y) \in S : \boldsymbol{x}_j > a\} \end{aligned} \tag{2.17}$$

If the feature j is categorical, then:

$$\begin{aligned} S^-_{v,j,a} &= \{(\boldsymbol{x}, y) \in S : \boldsymbol{x}_j = a\} \\ S^+_{v,j,a} &= \{(\boldsymbol{x}, y) \in S : \boldsymbol{x}_j \neq a\} \end{aligned} \tag{2.18}$$

For any node v of a decision tree, we also denote by S_v the subset of the training set that is associated with this node, i.e. the examples from the training set that would be directed to or through that node if evaluated by the tree. We can now define the ϵ-feasibility:

Definition 2.5 (ϵ-feasibility)**.** *Let $k, h \in \mathbb{N}$, and let $\boldsymbol{\epsilon} = (\alpha, \beta)$ where $\alpha, \beta \in (0, 1]$. A decision tree T is ϵ-feasible, with pruning thresholds (k, h), w.r.t. a training set S if for every node v of the tree, the following conditions hold:*

1. *if $|S_v| \leq k$ or $g(S_v) = 0$ or $\mathrm{depth}_T(v) = h$, then v is a leaf, else if $g(S_v) \geq \alpha$, then v is an internal node;*
2. *if v is an internal node and its splitting criterion is (j, a), then we have*

$$G(S_v, S^+_{v,j,a}, S^-_{v,j,a}) \geq G(S_v, S^+_{v,j',a'}, S^-_{v,j',a'}) - \beta$$

   for all $(j', a') \in [d] \times \mathbb{R}$;

3. *if v is a leaf, then the label L_v associated with the leaf is a majority label of S_v;*

For any fixed pruning thresholds k, h, we say that a fully dynamic decision tree building algorithm is ϵ-feasible if, at any point t in time, the tree T_t built by the algorithm is ϵ-feasible with respect to the training set that is up-to-date at that time.

2.4 The Fully Dynamic Decision Tree algorithm

2.4.1 Main ideas

We first present a broad overview of our new algorithm. Our goal is to propose an ϵ-feasible fully dynamic decision tree building algorithm. We also want this algorithm to be as efficient as possible, especially when updating the model to insertions and deletions. More precisely, we focus on the **amortized** cost of updating, i.e. the mean computational cost per update (insertion or deletion).

The main idea is to use the smoothness properties presented in Theorem 2.4. This theorem states that the change in the gain of a splitting criterion after a series of updates is linearly dependent on the number of updates. In other words, to change the gain of a splitting criterion by a constant ε or more, one needs to update the training set at least $\Omega(\varepsilon|S|)$ times. Therefore, it should be enough to recompute any node of the tree every $\Theta(\varepsilon|S_v|)$ updates that affect it to keep the tree consistent with the second condition of the ϵ-feasibility definition. This process only requires keeping track of the number of updates that affect each node. It does not require keeping an estimate of the gain of the current split or any of the alternative splits.

The second idea is that if a node u is recomputed at some point in time and a node v in the path from u to the root is recomputed a few updates later, then u will be replaced in the process, making its recomputing a waste of computational time. In fact, the worst-case scenario would be that a leaf would need to be rebuilt at some time t, then its parent at time $t+1$ and so on all the way up to the root at time $t+h$, with h the height of the tree. In that case, it would be way more efficient to directly rebuilt the whole tree at time t.

If the condition to rebuild the node u is tight to keep the tree ϵ-feasible, it is not possible to delay its rebuilding as it would make our tree not-ϵ-feasible. It is possible however to check the nodes from u to the root to see if they would need to be rebuilt soon, i.e. if the number $|S_v|$ of examples in their training set is close to the one of u and, if so, we can rebuild them instead.

We now proceed to the rigorous presentation of the algorithm. The algorithm is composed of a procedure named BUILD, which defines how a given node of the tree is built or rebuilt, and a procedure named UPDATE, which defines how the algorithm deals with insertions and deletions.

2.4.2 The BUILD procedure

This procedure is shown as Algorithm 2.1. It is a special case of Hunt's framework algorithm presented as Algorithm 1.1 in Section 1.1.4. S_r is the training set. η is the current depth of the tree. If BUILD is called to build the tree from the root, then $\eta = 1$. The procedure is called first at the initialization of the tree, and then by the UPDATE procedure when needed.

The constants k and $\frac{\alpha}{2}$ are respectively the number of examples and the Gini

Algorithm 2.1 FUDYADT.BUILD

```
1: procedure BUILD(S_r, η)
2:     r ← new vertex,  c(r) ← 0,  s(r) ← |S_r|
3:     if |S_r| ≤ k or g(S_r) ≤ α/2 or η = h then
4:         Store S_r in a self-balancing binary tree
5:         T ← decision tree with r as root
6:         L_r ← any majority label in S_r
7:     else
8:         (j, a) ← arg max{G(S, S^+_{r,î,â}, S^-_{r,î,â}) : (î, â) ∈ [d] × ℝ}
9:         T_1 ← BUILD(S^+_{r,j,a}, η + 1)
10:        T_2 ← BUILD(S^-_{r,j,a}, η + 1)
11:        T ← decision tree with root r, T_1, T_2 as left, right subtrees, and split
           (j, a)
12:    end if
13:    return T
14: end procedure
```

score of the training set of a node, below which the node must be a leaf. The constant h is the maximal depth of the tree.

First, a new vertex r of the tree is created. The variable $c(r)$ is the number of updates that the node has received since last built and it is initialized to 0. The variable $s(r)$ is the size of the training set of the node when last built, and it is initialized to the size of S_r. Then, the procedure checks if any of the constants k, $\frac{\alpha}{2}$ or h is reached (Line 3). If so, the node is a leaf and a tree of height 1 is returned. If not, the best splitting criterion (j, a) is determined (Line 8). Finally, two children subtrees are built from the training subsets $S^+_{r,j,a}$ and $S^-_{r,j,a}$ and set as left and right children of r.

Theorem 2.6. *The* BUILD *procedure can be implemented to run in time*

$$\mathcal{O}(hd|S_r|(\log|S_r| + |\mathcal{Y}|))$$

If all the features are categorical, this can is reduced to

$$\mathcal{O}(|S_r|(hd|\mathcal{Y}| + \log|S_r|))$$

Proof. There are five operations in the procedure that cannot be computed in constant time:

- Computing the Gini index at Line 3
- Storing the training set of a leaf at Line 4
- Finding the best splitting criterion at Line 8
- Building the new training subsets $S^+_{r,j,a}$ and $S^-_{r,j,a}$ from the criterion, used in Lines 9-10

- Building the left and right children at Lines 9-10

We suppose that S_r is stored in a doubly linked list, so that it can be constructed or enumerated in time $\mathcal{O}(|S_r|)$. The Gini index computation at Line 3 only requires going through the data once and counting the number of each label, so it can be computed in time $\mathcal{O}(|S_r| + |\mathcal{Y}|)$.

Storing the data of a leaf requires $|S_r|$ insertions in a self-balancing tree of maximal size $|S_r|$, so it is done in time $\mathcal{O}(|S_r| \log |S_r|)$

Algorithm 2.2 Computing the best threshold for a numerical feature

```
1:  procedure BEST_THRESHOLD_NUMERICAL(S_r, j, initial_counters)
2:      S* ← S_r sorted along the feature j
3:      best_gain ← 0
4:      best_threshold ← min(x_j : (x, y) ∈ S*)
5:      counters_below ← {y_0 : 0, ... y_|Y| : 0}
6:      counters_above ← initial_counters
7:      for (x, y) ∈ S* do
8:          counters_above[y] ← counters_above[y] − 1
9:          counters_below[y] ← counters_below[y] + 1
10:         if next example has a different value for the feature j then
11:             gain ← compute_gain(counters_below, counters_above)
12:             if gain > best_gain then
13:                 best_gain ← gain
14:                 best_threshold ← x_j
15:             end if
16:         end if
17:     end for
18:     return best_threshold
19: end procedure
```

The procedure to find the best threshold for a numerical feature j is shown in Algorithm 2.2. The variable "initial_counters" is a dictionary that contains the number of examples for each label. This dictionary is a by-product of the Gini index calculation and can therefore be given as input with no time cost. Line 10 ensures that we only compute the gain for a valid split along the feature, i.e. that we do not split values that have the same value for the feature.

In Algorithm 2.2, Line 2 is computed in time $\mathcal{O}(|S_r| \log(|S_r|))$. Line 4 is computed in constant time, since S^* is sorted. Lines 5-6 are computed in time $\mathcal{O}(|\mathcal{Y}|)$. If the counters are stored in arrays of size $|\mathcal{Y}|$, the gain computation at Line 11 is done in time $\mathcal{O}(|\mathcal{Y}|)$ and all the other lines except for the loop can be computed in constant time. Therefore, the loop can be computed in time $\mathcal{O}(|S_r||\mathcal{Y}|)$ and the whole procedure can be computed in time $\mathcal{O}(|S_r|(\log(|S_r|) + |\mathcal{Y}|))$.

If the feature j is categorical, then the procedure for computing its best threshold goes as follows: for each category of this feature, counters are set

to the number of examples of this category that has each label. To compute this counters requires going through all the examples in S_r, which can be done in time $\mathcal{O}(|S_r|)$ because S_r is stored in a self-balancing tree. Then, for each category of the feature, the counters are enough to compute the Gini gain in time $\mathcal{O}(|\mathcal{Y}|)$. All in all, finding the best category a for a given feature j and the associated gain can be computed in time $\mathcal{O}(|S_r| + b_j|\mathcal{Y}|)$, where b_j is the number of categories for the feature j in S_r, and since $b_j \leq |S_r|$, then the computation can be done in time $\mathcal{O}(|S_r||\mathcal{Y}|)$.

Building the training subset $S^+_{r,j,a}$ requires $|S^+_{r,j,a}|$ insertions in a doubly linked list, computed in time $\mathcal{O}(|S^+_{r,j,a}|)$. The same is true for $S^-_{r,j,a}$ and since $|S^+_{r,j,a}| + |S^-_{r,j,a}| = |S_r|$, the construction of both these children subsets can be computed in $\mathcal{O}(|S_r|)$.

We now see that we can compute the BUILD procedure of an internal node except for the recursion Lines 9-10 in time $\mathcal{O}(d|S_r|(\log(|S_r|) + |\mathcal{Y}|))$, reduced to $\mathcal{O}(d|S_r||\mathcal{Y}|)$ if all the features are numerical. Also recall that $|S^+_{r,j,a}|+|S^-_{r,j,a}| = |S_r|$ and $\forall a, b \in \mathbb{R}^+ : a \log a + b \log b \leq (a + b) \log(a + b)$. Therefore, we can see by induction that the BUILD procedure including its recursion lines can be computed in time $\mathcal{O}(hd|S_r|(\log(|S_r|)+|\mathcal{Y}|))$, reduced to $\mathcal{O}(|S_r|(hd|\mathcal{Y}|+\log|S_r|))$ if all the features are categorical. □

2.4.3 The UPDATE procedure

Algorithm 2.3 FUDYADT.UPDATE

```
1:  procedure UPDATE(T, (x, y), o)
2:      P_x ← v_{κ_1}, ..., v_{κ_ℓ} with v_{κ_1} = r(T), v_{κ_ℓ} = v(x)
3:      update S_{v_{κ_ℓ}} according to (x, y), o
4:      Update L_{v_{κ_ℓ}} if necessary
5:      for i = 1, ..., ℓ do
6:          c(v_{κ_i}) ← c(v_{κ_i}) + 1
7:          if c(v_{κ_i}) ≥ ε · s(v_{κ_i}) then
8:              ŝ ← 2^⌈log_2 s(v_{κ_i})⌉
9:              j ← min{j' ∈ {0, ..., i} : s(v_{κ_j'}) ≤ ŝ}
10:             T' ← BUILD(S_{v_{κ_j}}, j)
11:             return T'
12:         end if
13:     end for
14: end procedure
```

The UPDATE procedure is presented in Algorithm 2.3. T is the decision tree before the update, $(\boldsymbol{x}, y)$ is the labeled example to insert or delete and o is the type of update, either INS for "insertion" or DEL for "deletion". The constant $\varepsilon \in \mathbb{R}^+$ is a parameter that decides how rarely the nodes need to be rebuilt. Theorem 2.7 will give the value of ε for FUDYADT to be $\boldsymbol{\epsilon}$-feasible.

The first step of this algorithm is to direct the new example from the root to the relevant leaf w.r.t. the splitting criteria of the nodes. This gives the path $P_{v_{\kappa_\ell}}$. Then, the training set of each node is updated. This may lead to a change in the label of the leaf v_{κ_ℓ} if the label of the inserted example becomes the majority label, or if the deleted example belongs to the majority label. If so, the label is updated at Line 4.

Then for all vertices in the path from the root to the leaf, its count of updates is incremented. If this count overpasses the threshold $\varepsilon \cdot s(v_{\kappa_i})$ then the node needs to be rebuilt. However, as explained in Section 2.4.1, we do not want to rebuild this node now and its parent in a near future. To avoid this, we set a new threshold $\hat{s} = 2^{\lceil \log_2 s(v_{\kappa_i}) \rceil}$, i.e. the smallest power of 2 greater or equal to s. Instead of rebuilding v_{κ_i}, we recompute the vertex of which size is less or equal to $\hat{s}$, and that is closest to the root.

Theorem 2.7. *If* $\varepsilon = \min\left(\frac{1}{k+1}, \frac{\alpha}{6-\frac{2}{|\mathcal{Y}|}}, \frac{\beta}{13-\frac{1}{|\mathcal{Y}|}}\right)$*, then the tree initialized with* BUILD *and updated with* UPDATE *is* (α, β)*-feasible, with pruning thresholds* (k, h)

The value of ε in this theorem depends on $|\mathcal{Y}|$ to be as tight as possible. However, if $|\mathcal{Y}|$ is unknown or if we want to unify the value of ε, it is possible to use $\varepsilon = \min\left(\frac{1}{k+1}, \frac{\alpha}{6}, \frac{\beta}{13}\right)$ with the same guarantees and only a slight decrease in efficiency.

Proof. First, we see that BUILD builds a $\boldsymbol{\epsilon}$-feasible tree.

Let us suppose that, after calling UPDATE, a vertex v does not match the conditions for the tree to be $\boldsymbol{\epsilon}$-feasible. We denote by S the training set of that vertex at that time, and S_0 the training set of that vertex at the last time BUILD was used on it. We see that $c(v) \geq \triangle(S_0, S)$ and so $\frac{c(v)}{s(v)} \geq \triangle^*(S_0, S)$. We also know from Line 7 of UPDATE that $\frac{c(v)}{s(v)} \leq \varepsilon$ because BUILD would have been called on v or one of its ancestors otherwise. Therefore, we will show that any condition on v that would cause the tree to not be $\boldsymbol{\epsilon}$-feasible would also cause $\triangle^*(S_0, S) > \varepsilon$. From that, we will deduce that such vertex can not exist and that the tree is $\boldsymbol{\epsilon}$-feasible. We go through all the $\boldsymbol{\epsilon}$-feasibility conditions that could be broken by v.

v should be a leaf This condition applies if $|S| \leq k$, $g(S) = 0$ or $\text{depth}_T(v) \geq h$. It is broken if v is an internal node. UPDATE does never change the depth of a vertex or change it from leaf to internal or vice versa without calling BUILD. It is therefore impossible that $\text{depth}_T(v) \geq h$ and that v is an internal node.

If v is an internal node, then it means that $|S_0| > k$ and $g(S_0) > \frac{\alpha}{2}$. If $|S| \leq k$ then $\triangle^*(S_0, S) \geq \frac{1}{k+1} \geq \varepsilon$. Else, we need $g(S) = 0$ for the condition to apply. This means that $g(S_0) - g(S) \geq \frac{\alpha}{2}$ and thanks to Theorem 2.4, this means that $\triangle^*(S_0, S) \geq \frac{\alpha}{6-\frac{2}{|\mathcal{Y}|}} \geq \varepsilon$.

v **should be an internal node** This condition applies if $|S| > k$, $g(S) \geq \alpha$ and $\text{depth}_T(v) < h$. It is broken if v is a leaf. For v to be a leaf, we need either $|S_0| \leq k$, $g(S_0) = 0$ or $\text{depth}_T(v) = h$. Since the depth of v can not have changed since it was last built, we can disregard the option that $\text{depth}_T(v) = h$. With a similar reasoning as the previous condition, we get that if $|S_0| \leq k$ then $\triangle^*(S_0, S) \geq \frac{1}{k+1} \geq \varepsilon$, and if $g(S_0) = 0$ then $\triangle^*(S_0, S) \geq \frac{\alpha}{6-\frac{2}{|\mathcal{Y}|}} \geq \varepsilon$.

The Gini gain should be close to the optimal We denote by (j, t) the current splitting criterion, i.e. the optimal splitting criterion of S_0, and (j^*, t^*) the optimal splitting criterion of S. We also denote by G the gain of the current splitting criterion on S, G^* the gain of (j^*, t^*) on S, G_0 the gain of (j, t) on S_0 and G_0^* the gain of (j^*, t^*) on S_0.

This criterion is broken if $G^* - G > \beta$. Since (i, t) is the best splitting parameter for S_0, we know that $G_0 - G_0^* > 0$. Therefore $G_0 - G + G^* - G_0^* > \beta$ and so $\max(|G_0 - G|, |G^* - G_0^*|) > \beta$. Thanks to Theorem 2.4, this leads to $\triangle^*(S_0, S) > \frac{\beta}{13-\frac{1}{|\mathcal{Y}|}} \geq \varepsilon$.

The label of a leaf should be the majority Line 4 of UPDATE ensures that this condition is never broken. □

2.4.4 Performance of the FUDYADT algorithm

For each vertex v of a tree, we denote by $s^t(v)$ the size of the training set of that vertex when it was last (re-)built using BUILD before or at time t. We also denote by S_v^t the training set of the vertex at time t. We prove the two following lemmas which will be used to study the performance of the algorithm:

Lemma 2.8. *Let T_t be a decision tree built by the* BUILD *procedure and then updated through $t \geq 0$ calls to the* UPDATE *procedure. Then, for each vertex v of the tree, $(1 - \varepsilon) \cdot s^t(v) \leq |S_v^t| \leq (1 + \varepsilon) \cdot s^t(v)$.*

Lemma 2.9. *Let T be a decision tree built on a training set S. If every vertex v uses a split with a gain at least $\gamma > 0$ w.r.t. S_v, then T has height $\mathcal{O}\left(\frac{\log |S|}{\gamma}\right)$.*

Proof of Lemma 2.8. We know that $c^t(v) \geq ||S_v^t| - s^t(v)|$, with $c^t(v)$ the number of updates to the training set of v since it was last built. But we also know from Line 7 of algorithm 2.3 that at the end of any call of UPDATE, $c^t(v) \leq \varepsilon s^t(v)$. Therefore, we have

$$\left||S_v^t| - s^t(v)\right| \leq \varepsilon s^t(v)$$

and this is equivalent to the lemma. □

Proof of Lemma 2.9. We consider an internal node v of the tree, and u and z its children. We know from Theorem 2.4 that:

$$\left(3 - \frac{1}{|\mathcal{Y}|}\right) \triangle^*(S_v, S_z) \geq g(S_v) - g(S_z)$$

But since $\triangle^*(S_v, S_z) = \frac{|S_u|}{|S_v|}$, we have

$$\begin{aligned}\left(4 - \frac{1}{|\mathcal{Y}|}\right)\frac{|S_u|}{|S_v|} &\geq g(S_v) - g(S_z) + \frac{|S_u|}{|S_v|} \\ &\geq g(S_v) - (1 - \frac{|S_u|}{|S_v|})g(S_z) \\ &\geq g(S_v) - \frac{|S_z|}{|S_v|}g(S_z) - \frac{|S_u|}{|S_v|}g(S_u) \geq \gamma\end{aligned} \tag{2.19}$$

Therefore we know that $\left(4 - \frac{1}{|\mathcal{Y}|}\right)|S_u| \geq \gamma|S_v|$ and so $|S_u| \geq \frac{\gamma}{4}|S_v|$. But then $|S_z| \leq \frac{3\gamma}{4}|S_v|$ and by the same reasoning we can find the same result on S_u.

Hence, the size of the training set of any vertex at depth η is in $\mathcal{O}(\gamma^\eta|S|)$. But since the size of the training set of a vertex can not be below 1, then the size of the tree is in $\mathcal{O}(\frac{\log|S|}{\gamma})$. □

We can now prove the performances theorem:

Theorem 2.10. *If T is a tree built from* BUILD *and not updated since. The* UPDATE *procedure can be implemented so that $\mathcal{T}$ invocations of* UPDATE *on T run in time*

$$\mathcal{O}\left(\mathcal{T}hd\frac{\log(n)}{\varepsilon}(\log(n) + |\mathcal{Y}|)\right) = \mathcal{O}\left(\mathcal{T}d\frac{\log^2(n)}{\varepsilon^2}(\log(n) + |\mathcal{Y}|)\right) \tag{2.20}$$

If all the features are categorical, the time is reduced to

$$\mathcal{O}\left(\mathcal{T}hd\frac{\log(n)}{\varepsilon}|\mathcal{Y}|\right) = \mathcal{O}\left(\mathcal{T}d\frac{\log^2(n)}{\varepsilon^2}|\mathcal{Y}|\right) \tag{2.21}$$

Proof. We denote by cost($\mathcal{T}$) the running time of $\mathcal{T}$ invocations of UPDATE.

In Algorithm 2.3, the lines that can not be implemented to run in constant time are Line 2, which can be computed in time $\mathcal{O}(h)$, Line 3, which can be computed in time $\mathcal{O}(\log n)$ because S_r is stored in a self-balancing tree in the leaf, Line 4, which can be computed in time $\mathcal{O}(|\mathcal{Y}|)$ as long as we keep counters in the leaf that can be updated in constant time, and Line 9, which can be computed in time $\mathcal{O}(h)$. There is also the BUILD procedure. Its time complexity has been given in Theorem 2.6. Finally, to call BUILD, one needs to rebuild the training set $S_{v_{\kappa_j}}$. Recall that, although the training sets in the leafs are stored in self-balancing trees, the training sets used for calling BUILD are double-linked lists. Therefore, rebuilding $S_{v_{\kappa_j}}$ can be done is $\mathcal{O}(|S_{v_{\kappa_j}}|)$ by visiting the subtree of v_{κ_j} and aggregating the data at the leaves.

Let the invocations of UPDATE be identified by consecutive integer from 1 to $\mathcal{T}$ and let $B \subseteq [\mathcal{T}]$ be the invocations when BUILD is called. For each $t \in B$, let also $i(t)$ and $b(t)$ be such that $v_{i(t)}$ and $v_{b(t)}$ are respectively the node that

triggers the call to BUILD (Line 7 of Algorithm 2.3) and the node on which BUILD is called (Line 9) at time t. Then:

$$\begin{aligned}\text{cost}(\mathcal{T}) \leq &\sum_{t=1}^{\mathcal{T}} \mathcal{O}(h + \log(n) + |\mathcal{Y}|) \\ &+ \sum_{t \in B} \mathcal{O}\big(hd|S^t_{v_{b(t)}}|(\log(n) + |\mathcal{Y}|)\big)\end{aligned} \tag{2.22}$$

The first term contributes $\mathcal{O}(\mathcal{T}(h + \log(n) + |\mathcal{Y}|))$. We also have

$$\begin{aligned}\sum_{t \in B} |S^t_{v_{b(t)}}| &\leq 2 \sum_{t \in B} s^t(v_{b(t)}) && \text{From Lemma 2.8 and } \varepsilon \leq 1 \\ &\leq 4 \sum_{t \in B} s^t(v_{i(t)}) && \text{Lines 8-9 of UPDATE} \\ &\leq \frac{4}{\varepsilon} \sum_{t \in B} c^t(v_{i(t)}) && \text{Line 7 of UPDATE} \\ &\leq \frac{4}{\varepsilon} \sum_{t \in B} c^t(v_{b(t)}) && b(t) \text{ is an ancestor of } i(t)\end{aligned} \tag{2.23}$$

We now have

$$\begin{aligned}\text{cost}(\mathcal{T}) \leq &\mathcal{O}(\mathcal{T}(h + d + \log(n) + |\mathcal{Y}|)) \\ &+ \sum_{t \in B} \mathcal{O}\left(hd \frac{c^t(v_{b(t)})}{\varepsilon}(\log(n) + |\mathcal{Y}|)\right)\end{aligned} \tag{2.24}$$

We consider an invocation t of UPDATE and $(\boldsymbol{x}, y)$ the labeled example that is inserted or deleted at that time. $P_{\boldsymbol{x}}$ is the path of the example in the tree, i.e. the set of nodes affected by the update. We denote by $C^t = \{v_{k_1}, \dots v_{k_M}\} \subseteq P_{\boldsymbol{x}}$ the set of nodes from $P_{\boldsymbol{x}}$ such that BUILD$(S^{t_i}_{v_{k_i}}, \text{depth}(v_{k_i}))$ will be executed at some time $t_i \geq t$. In other words, the nodes themselves will be rebuilt at some point in time. We suppose that C^t is sorted so that the highest nodes come first. In other words, v_{k_1} is an ancestor of every other node of C^t, v_{k_2} is an ancestor of every other nodes except v_{k_1} and so on.

We see that if $i < j$ and t_i and t_j are the time at which v_{k_i} and v_{k_j} respectively will be recomputed, then $t_i > t_j$ because otherwise v_{k_j} would be deleted when v_{k_i} would be rebuilt and would not exist at time t_j to be rebuilt. We also see that $\forall i \in [M-1]$, we have $\lceil \log_2(s^{t_{i+1}}(v_{k_i})) \rceil > \lceil \log_2(s^{t_{i+1}}(v_{k_{i+1}})) \rceil$, because otherwise v_{k_i} would be rebuilt at time t_{i+1} instead of $v_{k_{i+1}}$, because of Line 8 of Algorithm 2.3. Since $t_i > t_{i+1}$ neither v_{k_i} nor $v_{k_{i+1}}$ is rebuilt between time t and t_i, and so

$$\forall i \in [M-1] : \lceil \log_2(s^t(v_{k_i})) \rceil > \lceil \log_2(s^t(v_{k_{i+1}})) \rceil$$

We deduce that:

$$\forall t \in \mathcal{T} : |C^t| \leq \lceil \log_2(n) \rceil \tag{2.25}$$

We see that $\forall t \in B$, $c^t(v_{b(t)})$ equals the number of times it was in a set C^t since it was last built. Since the sets C^t contain by design only nodes that will be built at some time $t \in B$, we have:

$$\begin{aligned}\sum_{t\in B} c^t(v_{b(t)}) &= \sum_{t=1}^{\mathcal{T}} |C^t| \\ &\leq \sum_{t\in B} |C^t| \\ &\in \mathcal{O}(\mathcal{T}\log(n))\end{aligned} \tag{2.26}$$

We combine this result with Equation (2.24) and we get

$$\text{cost}(\mathcal{T}) \leq \mathcal{O}\left(\mathcal{T}hd\frac{\log(n)}{\varepsilon}(\log(n) + |\mathcal{Y}|)\right)$$

We conclude this proof for numerical features by using Lemma 2.9.

The proof for categorical features is the same, except that we use the specific result for categorical features of Theorem 2.6 in Equation (2.24). □

The amortized cost of UPDATE is obtained by dividing Equations (2.20) and (2.21) by $\mathcal{T}$. This is because, even if the tree has been updated before, it will be rebuilt entirely after at most n updates. Therefore, Theorem 2.10 applies at least to the last $\mathcal{T} - n$ updates and, since the first n updates are computed in finite time, then their computation time has no impact on the amortized cost.

2.4.5 Lower bounds

The paper "Fully-Dynamic Decision Trees" [10] also contains the following theorems:

Theorem 2.11. *Let $k^*, h^* \geq 1$, and let $\boldsymbol{\epsilon} = (\alpha, \beta)$ with $0 \leq \alpha \leq 1$ and $0 \leq \beta < \frac{1}{24}$. Any weakly $(\boldsymbol{\epsilon}, \frac{3}{4})$-feasible fully dynamic algorithm with pruning thresholds k^*, h^* uses space $\Omega\big(\frac{n\cdot d}{k\cdot\log n}\big)$, where d is the number of features and n is the maximum size of the active set at any point in time.*

and

Theorem 2.12. *Let $k, h \geq 1$ and $\alpha, \beta \in [0, \frac{1}{2})$. For arbitrarily large n and d there exist sequences of n INS and DEL operations over $\{0,1\}^d \times \{0,1\}$ such that, in the matrix access model, any weakly $(\boldsymbol{\epsilon}, 2/3)$-feasible fully dynamic algorithm has expected running time $\Omega(nd)$.*

The notion of the weakly $(\boldsymbol{\epsilon}, \frac{3}{4})$-feasibility is a generalization of the $\boldsymbol{\epsilon}$-feasibility, so that every $\boldsymbol{\epsilon}$-feasible algorithm is also weakly $(\boldsymbol{\epsilon}, \frac{3}{4})$-feasible. Therefore, these theorems ensure that $\boldsymbol{\epsilon}$-feasible algorithms can not be much more space- or time-efficient than FUDYADT.

For the reasons explained in the Section 2.1.3, these theorems are not studied in more details in this thesis.

	d	# of examples	1-class
Electricity	8	45 311	UP
Forest Covertype	54	581 011	2
INSECTS v1-v5	33	24 150 – 79 986	*-male
KDDCUP99	41	494 021	smurf.
NOAA Weather	8	18 159	1
Poker	10	829 201	0

Table 2.1: Datasets statistics.

2.5 Experiments

2.5.1 Experimental settings

We compare FUDYADT against the state-of-the-art algorithms for incremental decision tree learning, EFDT [40], using the MOA software [5]. Our goal is to show that FUDYADT performs at least similarly as this competitor in the incremental setting. We also show the time performance of FUDYADT in a fully dynamic setting.

Settings. We implemented FUDYADT in C++.[1] We conducted all experiments on an Ubuntu 20.04.2 LTS server equipped with 144 Intel(R) Xeon(R) Gold 6154 @ 3.00GHz CPUs and 264 GB of RAM. We observe that the algorithms have not been implemented in the same programming language, which limits the relevance of the runtime comparison.

Datasets. Our datasets are shown in Table 2.1. We have chosen them among standard datasets for classification; some of them, such as INSECTS, feature the so-called concept drift. Not all datasets have binary labels. Because the experiments have been performed on a first version of our algorithm, that worked only with binary labels, we adapted the datasets to have only two labels. For the INSECTS datasets, we assigned label 1 to the union of `male` classes. For every other dataset, we assigned label 1 to the majority class. The INSECTS dataset contains 5 sets of examples, each representing a different way of introducing concept drift.

Input models. We consider two input models. Let $(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_T}, y_T)$ be the sequence of examples as given by the dataset at hand (typically in chronological order). The simplest model is one in which data are inserted in the order of the dataset, aka *incremental* model. Formally, a first batch of g data, called **grace period**, is provided to initialize the model. Then at every $t \in [g+1, T]$ the algorithm receives $\text{INS}(\boldsymbol{x_t}, y_t)$. This model is supported by both algorithms

[1] `https://github.com/GDamay/dynamic-tree`

(FUDYADT and EFDT), hence we use it to compare them against each other. Because the data feature concept drift that EFDT claims to handle and we do not, FUDYADT is tested in that setting with a sliding window. This means that every time it receives an example for insertion, the oldest example of its training set is subsequently deleted.

The next model involves deletions and thus is supported only by FUDYADT. It is the *random update* model (RU): for all $t \in [T]$, with probability $1/2$ the algorithm receives $\text{INS}(\boldsymbol{x}, y)$ where $(\boldsymbol{x}, y)$ is a new example chosen uniformly at random from the remaining training set, and with probability $1/2$ it receives $\text{DEL}(\boldsymbol{x}, y)$ where $(\boldsymbol{x}, y)$ is chosen uniformly at random from the active set S^t. This last model will help check the performances of FUDYADT in a fully dynamic setting. Its F1-score performances are not expected to vary with ε.

In all cases, before performing any insertion $\text{INS}(\boldsymbol{x}, y)$, the example $(\boldsymbol{x}, y)$ is assigned a label $\bar{y}$ by the decision tree, which is used for the evaluation of the F1-score.

Metrics. The results of the models are evaluated using the F1-score presented in Section 1.1.3.

Parameters. For FUDYADT, we let $\alpha = 0$, $\beta = 0$ $k = 1$, $h \in \{5, 10\}$, and we manually set $\varepsilon \in [0, 2]$. Note that it breaks the condition of Theorem 2.7. It allows us to test the effect of ε without fine-tuning of the other parameters. The parameters of EFDT are set to the original values specified by the authors; we only vary the grace period in $\{100, 500, 1000\}$ to find the value yielding highest F1-score. For the size of the sliding window, we use $W \in \{100, 1000\}$. Several parameter configurations show similar trends.

2.5.2 Limitations

We acknowledge several limitations in these experiments that limit the extent of the comparison between FUDYADT and its competitors. First, the programming language is not the same, FUDYADT being implemented in C++ while the competitors are implemented in Java. The memory consumption is also not comparable, FUDYADT having memory needs in $\mathcal{O}(dn)$ while EFDT has been specifically designed to limit the memory consumption. Finally, while EFDT is evaluated in a purely incremental setting, FUDYADT is evaluated in a sliding window setting.

Therefore, the comparison between FUDYADT and its competitors will be very limited. We only aim at showing that, although FUDYADT is capable of handling new settings, i.e. fully dynamic settings, it is also at least as good as its competitors in the classical online setting with concept drift.

2.5.3 Results and discussion

FUDYADT versus EFDT. We compare the F1-scores of EFDT and FUDYADT when allowed the same mean time per update. To this end, we tuned the parameter ε of FUDYADT to make its running time very close to (and never exceeding) that of EFDT. The results are shown in Table 2.2; remarkably, FUDYADT outperforms consistently EFDT in terms of F1-score. We can argue from these results that FUDYADT performs at least as good as EFDT. However, we refrain from drawing stronger conclusions from these results, as all the limitations we discussed in Section 2.5.2 may have played a role.

FUDYADT on the incremental (sliding window) setting These experiments aim at finding how changes in ε affect the performance of the algorithm, i.e. its ability to adapt to concept drift, as well as the effect on the mean time per updated. We set $h = 10$, $k = 1$, $\alpha = 0$, and $W = 100$ for Electricity and $W = 1000$ otherwise.

Figure 2.1 shows the F1 score as a function of ε (left column) and the average time per update in milliseconds in logarithmic scale as a function of ε (left column). The smaller ε is, the more often subtrees are recomputed, yielding a higher amortized running time. The more frequent recomputing also allows the tree to capture the concept drift better, yielding higher F1-scores.

This behavior is clear in the Electricity and Poker datasets, where from $\varepsilon = 0$ to $\varepsilon = 1$ the F1 score decreases by roughly 0.1 and the running time decreases by three orders of magnitude. For INSECTS the F1-score is much more stable. A good tradeoff could be $\varepsilon = 0.1$, where the F1-score is close to that of $\varepsilon = 0$ but with an amortized running time per update smaller by orders of magnitude (≈ 0.5ms). However, this tradeoff probably depends on the speed at which the concept drift affects the data. A slower concept drift will make the frequent recomputing less important, and hence will favor higher values of ε.

All other datasets and parameter settings yielded very similar qualitative behaviors.

FUDYADT the Random Updates setting Figure 2.2 shows the average running time for the RU model. We observe a similar trend on this figure as on Figure 2.1. This result confirms that our algorithm performs well even in a context of random insertions and deletions.

FUDYADT versus BOAT At the time the paper was written for the conference, we did not know of the BOAT algorithm, which explains why it is not part of the experiments. However, given that BOAT computes an exact decision tree without trying to leverage the smoothness of the gain function, our algorithm gives much more control on how often the tree need to be rebuilt. Furthermore, the paper that presents BOAT [22] states several times that when it is possible to hold all the data in memory, BOAT is less efficient than an algorithm that do

so. Therefore, we are confident that with the right choice of the parameter ε, our algorithm is more efficient than BOAT.

2.6 Conclusion and future work

In this chapter, we have presented a new algorithm for building and maintaining decision trees in a fully dynamic setting.

This solution is based on two novel theoretical contributions. First, the formulation of a new smoothness theorem for the Gini index and Gini gain. This theorem states that the Gini index of a training set, when this set is updated by insertions and deletions of points, never changes by more than 3 times the relative number of updates, i.e. the number of updates divided by the size of the set. Furthermore, the Gini gain of any split performed on that set never changes by more than 13 times the relative number of updates.

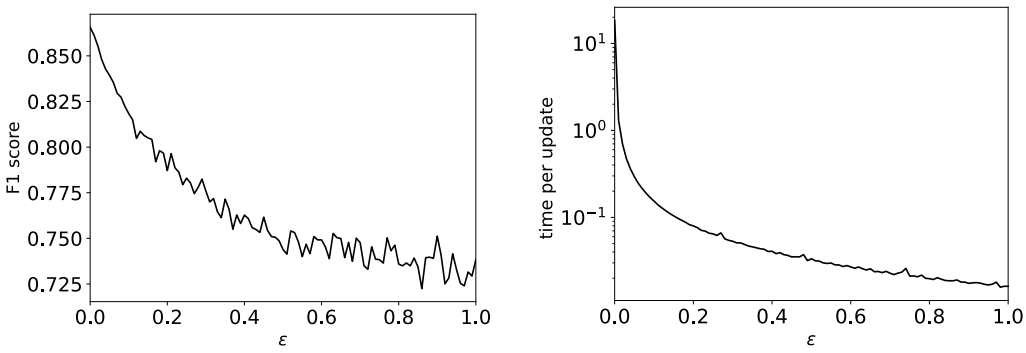
The second theoretical contribution that the solution uses is a new objective for fully dynamic decision trees. We call this new objective "$\boldsymbol{\epsilon}$-feasibility". A decision tree that meet this objective splits its training set almost as best as possible, meaning that the Gini gain of its splits is close to the optimal gain, while also meeting other criteria of tree building as the maximal depth.

Then, this chapter has presented the FUDYADT algorithm that relies on the theorem for maintaining an $\boldsymbol{\epsilon}$-feasible fully dynamic decision tree. With an amortized time per update (insertion or deletion) of $\mathcal{O}\left(d\frac{\log^3(n)}{\varepsilon^2}\right)$ when the number of class $|\mathcal{Y}|$ is small compared to d and $\log(n)$. This algorithm has been compared with the EFDT algorithm and proven to be at least as efficient in an incremental setting, while also operating within the fully dynamic setting.

A very interesting future work would be to study the same kind of fully dynamic algorithm for the information gain criterion. Would it be possible to get similar smoothness properties, leading to similar guarantees for the algorithm?

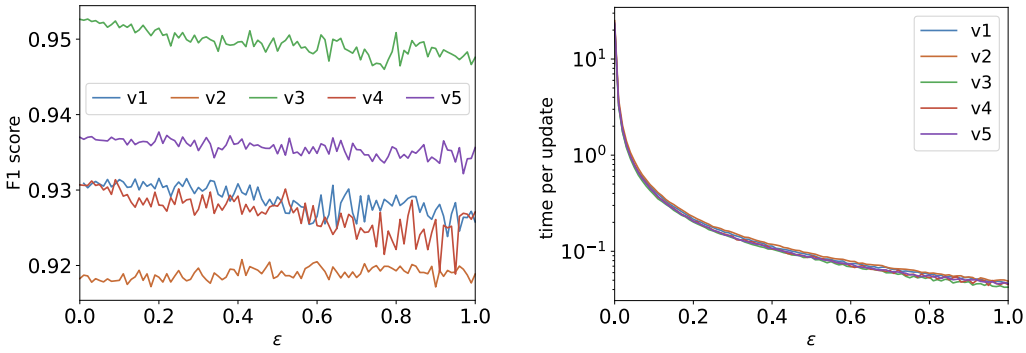
Decision trees are also much used as the basis for the random forest models. It would be very interesting to study how our algorithm can be adapted to create fully dynamic random forests.

A final future work direction could be to try to combine our finding with the ideas of the BOAT algorithm, among which the bootstrapping, to design algorithms that would maintain fully dynamic decision trees with a high probability, and with a very high efficiency.



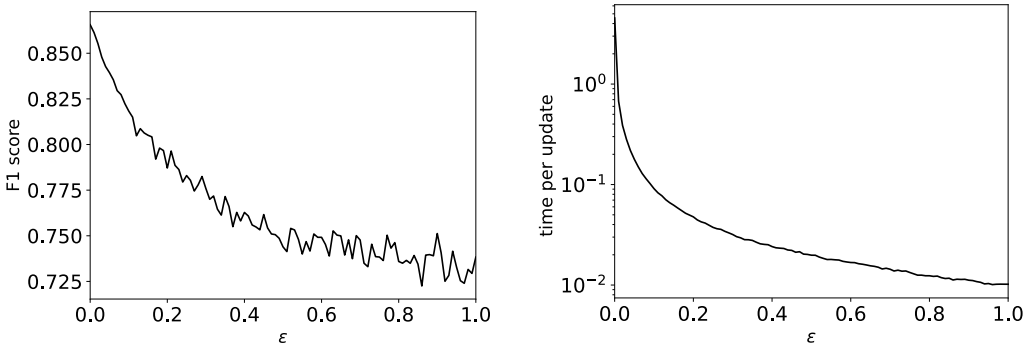
F1-score amortized time/update (ms)

(a) Electricity



F1-score amortized time/update (ms)

(b) INSECTS



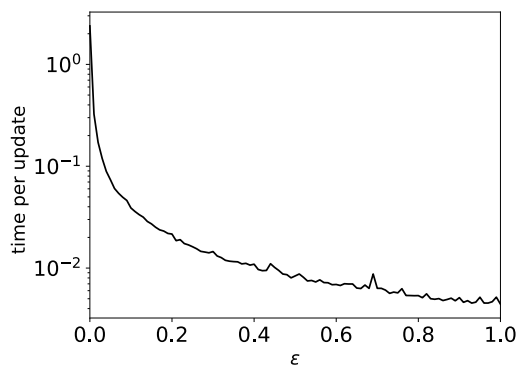
F1-score amortized time/update (ms)

(c) Poker

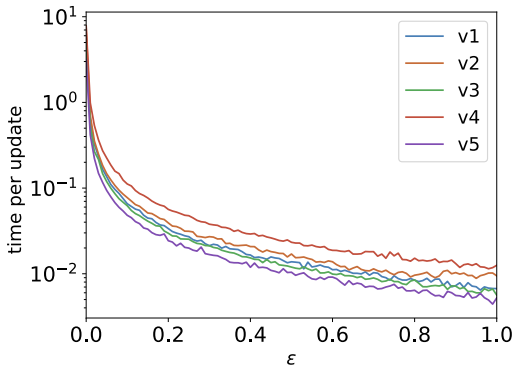
Figure 2.1: Performance of FUDYADT in the incremental model on the Electricity, INSECTS and Poker datasets (top to bottom), in terms of F1-score (left) and amortized milliseconds per update (right) as a function of ε.

	EFDT		FUDYADT		
	RT	F1	RT	F1	ε
Electricity	1.65	72.05	1.53	**83.93**	0.15
Forest Covertype	42.47	83.64	42.37	**90.33**	0.29
INSECTS v1	4.85	88.96	4.51	**92.17**	1.00
INSECTS v2	3.13	87.40	3.09	**92.53**	0.92
INSECTS v3	7.54	92.51	7.43	**94.76**	1.00
INSECTS v4	6.30	91.15	6.02	**91.91**	0.95
INSECTS v5	6.84	89.85	6.77	**93.34**	1.00
KDDCUP99	17.72	97.98	17.43	**99.91**	0.17
NOAA Weather	0.73	80.78	0.73	**81.43**	0.36
Poker	16.26	79.69	16.14	**86.07**	1.03

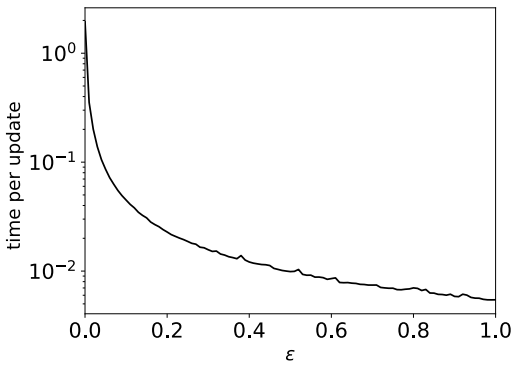
Table 2.2: Running time in seconds (labeled *RT*) and F1-score (labeled *F1*) of EFDT and FUDYADT in the incremental model. The last column shows the value of ε in UPDATE.



(a) Electricity



(b) INSECTS



(c) Poker

Figure 2.2: Amortized running time per update (in milliseconds) of FUDYADT in the RU model on the Electricity, Poker and INSECTS datasets.

Chapter 3

Personalized PageRank for Graph embedding

3.1 Introduction

Graph embeddings are one of the main techniques for graph mining. They allow the use of standard machine learning techniques designed for vectorial data, even when the input data are graphs. For example, graph embedding allows the use of techniques such as k-means or DB-scan for node clustering, or decision trees for node classification.

However, as most embedding techniques, graph embedding does not easily produce interpretable results. This is because the embedding is affected by many orders of relation, from the direct neighborhood to long paths, and these relations are very difficult to sum up in an interpretation that could be understood by a human user.

The lack of interpretable graph embeddings is a problem, as it prevents any subsequent interpretable use of these embeddings. For example, decision trees are interpretable machine learning models, but if the input data are not interpretable, then the interpretation provided by these models will not make sense to any human user. This leads us to propose a method for embedding graphs that is inherently interpretable.

3.1.1 Main contributions

The main contribution in this chapter is a new embedding, called PageRank Factorization-based Interpretable Graph Embedding (PARFAITE) that aims first and foremost at providing an highly interpretable graph embedding. This novel approach is based on multiple ideas. The first one is to consider the Personalized PageRank (PPR) matrix as a data matrix and hence centering it before decomposing using an SVD. The second idea is to use the unaltered singular values in the left and right representations, instead of the rooted

singular values as matrix decomposition embeddings usually do. The final idea is not to use the result of the SVD, i.e. the projection of the matrix on the singular spaces, as the embedding. Instead, we perform a clustering and project back the centers of the clusters onto the original space to get the embedding. This last part ensures a high interpretability through communities.

In Section 3.4.3, we also provide a new metric for interpretability and, in Section 3.6.2, we present a novel dataset constructed from all pages of the French version of Wikipedia, which we release for reproducibility and benchmarking.

Our last contribution is an exploration of the properties of the PPR matrix, presented in Section 3.7. To the best of our knowledge, the properties stated in this section were not known beforehand. Among other things, this exploration gives us a glimpse on the possibility of an embedding using the eigendecomposition of a matrix linked to the PPR. We also discover that this eigendecomposition is closely related to the spectral embedding of graphs, and this sheds a new light on the spectral embedding.

3.1.2 Organization of the chapter

This chapter begins with a presentation of the PageRank vector, the Personalized PageRank matrix and their properties. Then, a short section presents the notion of communities in graphs. A section is then dedicated to graph embeddings, with a presentation of a taxonomy and a brief history of the existing graph embeddings. This section also features a presentation of the problem of interpretability of graph embedding, with a review of the existing metrics and the presentation of a new one. After this, our new embedding, PARFAITE, is presented, along with experiments that show its higher interpretability and its reasonable performance on the task of link prediction. Finally, a section study some new properties of the PPR matrix and their implications to the spectral embedding of graphs, a short section presents the work done on a practical application for the clustering of YouTube channels and users, and the chapter ends with a recap of the findings and suggestions for future works around this topic.

3.1.3 Acknowledgements

This chapter is based on the conference paper "PARFAITE: PageRank-Matrix Factorization for Interpretable Graph Embeddings", which was presented at the 2024 ASONAM conference. The proceedings of this conference are still in press at the time of that writing. Parts of this paper are reproduced in a modified version in this chapter with permission from Springer Nature.

3.2 Preliminaries on PageRank

The PageRank score was briefly introduced in Section 1.2.2. We propose here a study of this metric and its personalized version, and of their computation.

Let us first recall the definition of the PageRank score and give some equivalent definitions.

3.2.1 PageRank

The first definition of PageRank with a parameter $\alpha \in [0, 1)$ uses a random walk with restart in the graph. The walker starts from a vertex drawn uniformly at random, i.e. $\forall v \in V, \mathbb{P}(V_0 = v) = \frac{1}{n}$. Then, at each step, it makes one of the following two moves:

- **Restart** The walker goes to any vertex of the graph at random.
- **Walk** The walker draws uniformly at random one of the neighbors (out-neighbors in the case of directed graphs) of the vertex it is on and goes to that neighbor.

At each step, the probability to restart is α and the probability to walk is $1 - \alpha$. Let us denote by $\boldsymbol{p_i} \in \mathbb{R}^n$ the vector that, for each vertex, contains the probability of being on that vertex at step i. The random walk is defined by the following sequence:

$$\begin{cases} \boldsymbol{p_0} = \frac{1}{n}\mathbb{1} \\ \boldsymbol{p_{i+1}}^\top = \alpha \underbrace{\boldsymbol{p_0}^\top}_{\text{Restart}} + (1-\alpha) \underbrace{\boldsymbol{p_i}^\top \boldsymbol{M}}_{\text{Walk}} \end{cases} \tag{3.1}$$

where $\mathbb{1} \in \mathbb{R}^n$ is the vector of which all elements are 1.

Property 3.1. *The sequence defined in Equation (3.1) converges to the vector $\boldsymbol{\pi}$ defined by*

$$\boldsymbol{\pi}^\top = \alpha \boldsymbol{p_0}^\top + (1-\alpha)\boldsymbol{\pi}^\top \boldsymbol{M} \tag{3.2}$$

The vector $\boldsymbol{\pi}$ is the PageRank vector of the graph.

Proof. Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be the iteration function, $f(\boldsymbol{x})^\top = \alpha \boldsymbol{p_0}^\top + (1-\alpha)\boldsymbol{x}^\top \boldsymbol{M}$. We have

$$\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n : f(\boldsymbol{x})^\top - f(\boldsymbol{y})^\top = (1-\alpha)(\boldsymbol{x}-\boldsymbol{y})^\top \boldsymbol{M}$$

Since $\boldsymbol{M}$ is a stochastic matrix, we have $\forall \boldsymbol{z} \in \mathbb{R}^n : ||\boldsymbol{z}^\top \boldsymbol{M}||_1 \leq ||\boldsymbol{z}||_1$. Therefore, we have

$$\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n : ||f(\boldsymbol{x})^\top - f(\boldsymbol{y})^\top||_1 \leq (1-\alpha)||(\boldsymbol{x}-\boldsymbol{y})||_1$$

This proves that f is a contraction mapping on $\mathbb{R}^n$ and so the Banach fixed-point theorem [15] concludes the proof. □

The PageRank vector of the graph G is the asymptotic limit of the sequence defined in Equation (3.1), i.e. it is the vector $\boldsymbol{\pi}$ defined in Equation (3.2).

Property 3.2. *The PageRank vector $\boldsymbol{\pi}$ defined in Equation (3.2) can be equivalently written as the series*

$$\boldsymbol{\pi}^\top = \alpha \sum_{i=0}^{+\infty} (1-\alpha)^i \boldsymbol{p_0}^\top \boldsymbol{M}^i \tag{3.3}$$

Proof. We can show by induction that the general term of the sequence described in Equation (3.1) is

$$\boldsymbol{p_i} = (1-\alpha)^i \boldsymbol{p_0}^\top \boldsymbol{M}^i + \alpha \sum_{j=0}^{i-1} (1-\alpha)^j \boldsymbol{p_0}^\top \boldsymbol{M}^j$$

It has already been shown that the sequence converges.
Given that $\lim_{i\to+\infty} ||(1-\alpha)^i \boldsymbol{p_0}^\top \boldsymbol{M}^i||_1 = 0$ then

$$\boldsymbol{p_i} \sim_{i\to+\infty} \alpha \sum_{i=0}^{i-1} (1-\alpha)^i \boldsymbol{p_0}^\top \boldsymbol{M}^i$$

□

3.2.2 Personalized PageRank

In a graph $G = \{V, E\}$, the Personalized PageRank (PPR) vector $\boldsymbol{\pi_u}$ of a vertex $u \in V$ with parameter α is the asymptotic vector of the sequence defined by

$$\begin{cases} \boldsymbol{p_0} = \boldsymbol{e_u} \\ \boldsymbol{p_{i+1}}^\top = \alpha \boldsymbol{p_0}^\top + (1-\alpha) \boldsymbol{p_i}^\top \boldsymbol{M} \end{cases} \tag{3.4}$$

with $\boldsymbol{e_u} \in \mathbb{R}^n$ the vector of which the only non-zero element is 1 at the u^{th} position.

This definition is identical to Equation (3.1) except for the first term. It is easy to see that Properties 3.1 and 3.2 also apply.

Definition 3.3 (Personalized PageRank (PPR) matrix)**.** *The **Personalized PageRank matrix** $\boldsymbol{\Pi} \in \mathbb{R}^{n\times n}$ of a graph $G = \{V, E\}$ with parameter α is the matrix of which each line is the PPR vector of the related vector in the graph.*

We know from Properties 3.1 and 3.2 that:

$$\boldsymbol{\pi_u}^\top = \alpha \boldsymbol{e_u}^\top + (1-\alpha) \boldsymbol{p_u}^\top \boldsymbol{M}$$

and

$$\boldsymbol{\pi_u}^\top = \alpha \sum_{i=0}^{+\infty} (1-\alpha)^i \boldsymbol{e_u}^\top \boldsymbol{M}^i$$

We can deduce that:

$$\mathbf{\Pi} = \alpha \boldsymbol{I} + (1-\alpha)\mathbf{\Pi}\boldsymbol{M} \tag{3.5}$$

and

$$\mathbf{\Pi} = \alpha \sum_{i=0}^{+\infty} (1-\alpha)^i \boldsymbol{M}^i \tag{3.6}$$

where $\boldsymbol{I} \in \mathbb{R}^{n \times n}$ is the identity matrix.

Finally, we see from Equation (3.5) that:

$$\mathbf{\Pi} = \alpha(\boldsymbol{I} - (1-\alpha)\boldsymbol{M})^{-1} \tag{3.7}$$

Note that $\boldsymbol{I} - (1-\alpha)\boldsymbol{M}$ is a strictly diagonally dominant matrix, and hence is invertible.

We use the PPR matrix to introduce the reversed PPR vector:

Definition 3.4 (Reversed PPR)**.** *The **Reversed PPR** vector of a vertex u is the column associated with u in the PPR matrix $\mathbf{\Pi}$. In other words, the reversed PPR score from u to v is the probability $\mathbf{\Pi}_{vu}$ that the PageRank random walk starting at the vertex v ends at u.*

3.2.3 Interpretations

A first interpretation of the PageRank and Personalized PageRank vectors has been given in its definition. We propose two other interpretations.

In the first new interpretation, a content (e.g. information, liquid…) flows from the vertices in $\boldsymbol{p_0}$. At each step, a fraction α of the still-flowing content it kept or dissipated on its current vertex and the rest divides equally to keep flowing to the neighbors. The PageRank or PPR vector is the proportion of the content that have been kept or dissipated on each vertex, i.e. the exposure of each vertex to the content.

The second interpretation comes directly from Equation (3.6) and uses another random walk. In this new random walk, the initial position of the walker is selected as in the first definition of PageRank or PPR, but then the actions of the walker are one of these two:

- **Stop** The walker stops walking and the random walk ends
- **Walk** The walker draws uniformly at random one of the neighbors (out-neighbors in the case of directed graphs) of the vertex it is on and goes to that neighbor.

The probability of stopping is α. The value for a vertex v in the PageRank or PPR vector is the probability that the walker is on that vertex when he stops walking. This interpretation helps to understand why it is usually enough to compute only a few steps of the walk to approximate the PageRank or PPR vectors. For example, if $\alpha = 0.1$ then after only 66 steps the probability that the walker is still walking is less than 10^{-3}.

3.2.4 Computing the PPR matrix and vectors

We present the most-used algorithm for computing the PPR matrix and vectors.

We denote by $\pi(\boldsymbol{p_0})$ the PPR vector generated from the PageRank random walk with restart using $\boldsymbol{p_0}$ as the initial vector. This can be the PageRank vector, a rooted PPR vector, or any PPR vector starting from a set of nodes.

The main idea of this algorithm is to build sequences $(\boldsymbol{q_i})_{i\in\mathbb{N}}$ and $(\boldsymbol{r_i})_{i\in\mathbb{N}}$, that maintain the following invariant

$$\forall i \in \mathbb{N}, \left(\pi(\boldsymbol{p_0})\right)^\top = \boldsymbol{q_i}^\top + \boldsymbol{r_i}^\top \boldsymbol{\Pi} \tag{3.8}$$

The pseudocode is shown as Algorithm 3.1. In this algorithm, the parameter α is the parameter of the PPR walk, $\boldsymbol{M}$ is the stochastic matrix of a random walk in the graph, $\boldsymbol{p_0} \in (\mathbb{R}^+)^n$ is the initial random walk vector, max_iter$\in \mathbb{N}$ is the number of iterations and $\varepsilon \in (0, 1)$ is the maximum accepted residual value. $\boldsymbol{p_0}$ is a vector of probability, so it must satisfy $||\boldsymbol{q_0}||_1 = 1$. The vectors p_i are the successive approximations of the PPR vector, while the vectors r_i are the residual values, that are still to be considered in the approximation. This algorithm can be seen as a simulation of the interpretation as content flowing given in Section 3.2.3. With that interpretation, $\boldsymbol{p_i}$ is the content that has already stopped while $\boldsymbol{r_i}$ represents the content that keeps flowing.

The convergence of this algorithm is guaranteed by the fact that $\boldsymbol{M}$ is a stochastic matrix and hence $\forall i \in \mathbb{N} : ||\boldsymbol{r_i M}||_1 \leq ||\boldsymbol{r_i}||_1$. Therefore $\forall i \in \mathbb{N} : ||\boldsymbol{r_{i+1}}||_1 \leq (1-\alpha)||\boldsymbol{r_i}||_1$. $||\boldsymbol{r_i}||_1$ is upper bounded by a geometric sequence of common ratio less than 1, therefore it converges to $\boldsymbol{0}_{\mathbb{R}^n}$ and, thanks to the invariant in Equation (3.8), we deduce that $\boldsymbol{q_i}$ converges to $\pi(\boldsymbol{p_0})$

Algorithm 3.1 PPR computing algorithm using an invariant

procedure APPROXIMATE_PPR(α, $\boldsymbol{M}$, $\boldsymbol{p_0} \in (\mathbb{R}^+)^n$, max_iter$\in \mathbb{N}$, $\varepsilon \in (0, 1)$)
    $i = 0$
    $\boldsymbol{q_i} = \boldsymbol{0}_{\mathbb{R}^n}$
    $\boldsymbol{r_i} = \boldsymbol{p_0}$
    while $i <$ max_iter and $||\boldsymbol{r_i}||_1 < \varepsilon$ **do**
        $i++$
        $\boldsymbol{p_i} = \boldsymbol{p_{i-1}} + \alpha \boldsymbol{r_{i-1}}$
        $\boldsymbol{r_i}^\top = (1-\alpha)\boldsymbol{r_{i-1}}^\top \boldsymbol{M}$
    end while
    return p_i
end procedure

3.3 Preliminaries on communities in graphs

The concept of **communities** in graphs is very close to the concept of clusters, in that they both represent groups of nodes that are densely linked together and

sparsely linked with the rest of the graph. These two concepts are sometimes used interchangeably, e.g. [39, 19].

To the best of our knowledge, there is no rigorous and fully-consensual definition that would separate the two concepts. In this thesis, we name "cluster" the abstract concept of a group of densely-linked nodes. In that sense, the quality of a clustering can be evaluated with measures as the modularity [45]. On the contrary, we name "**community**" a group of nodes that it makes humanly sense to group together, e.g. social communities, genre of film or field of research. With this definition, the evaluation of an automatically-detected community requires comparing it with a ground truth.

The main problem associated with communities in graph mining is the so-called **community detection problem**. It is an unsupervised problem in which we want to extract from the graph groups of nodes that belong to the same community. Some research have also focus on a semi-supervised version of the problem that consists in, given a small set of nodes that belong to the same community, finding the other nodes that belong to this community. See for example [28, 59, 34]. However, in this thesis, we use communities as a target for the interpretability of the embedding. Therefore, we do not address the community detection problem.

A new dataset We release a new dataset with ground-truth communities (Wikipedia fr).[1] This dataset has been constructed from all the pages of the French version of Wikipedia.[2] In such a graph, nodes represent Wikipedia pages while directed edges represent links between the corresponding Wikipedia pages. In the French version of Wikipedia, it is common to add links to so-called "portals" at the end of the page, which serve as reference pages for given topics and can be seen as ground-truth communities. We observe that links to portals are rarer in the English version of Wikipedia, while portals are more semantically related to their corresponding Wikipedia pages than Wikipedia categories. Such novel graph contains 2.52 millions vertices, 102 million edges and 2 700 ground-truth communities.

3.4 Preliminaries on graph embedding

3.4.1 Taxonomy of graph embedding methods

As introduced in Section 1.2.4, the objective of graph embedding is to find a function $\phi : V \to \mathbb{R}^k$, that represents the vertices of the studied graph into a low-dimension vectorial space $\mathbb{R}^k$. The result can be represented as a matrix $\boldsymbol{Y} \in \mathbb{R}^{n \times k}$ in which each line is the embedding of the related vertex. There

[1] https://gitlab.telecom-paris.fr/gabriel.damay/WikipediaFRNetwork

[2] All pages from the main space of Wikipedia, i.e. all the pages usually accessed by the public, excluding discussions, user pages etc.

exists in the literature a consensual taxonomy of the methods to perform such embedding [11, 24]. This taxonomy contains three main classes of techniques, namely the Matrix Factorization, the Random Walk and the Deep Learning techniques.

A Matrix Factorization technique is based on a $n \times n$ matrix $\boldsymbol{X}$ of which each element represents some sense of proximity between the two related vertices. For example, the simplest of these matrices is the adjacency matrix $\boldsymbol{A}$ of the graph. Then a factorization method uses the eigendecomposition or the Singular Value Decomposition and parts or all of the results of this decomposition are used as the embedding results. When a non-symmetric decomposition is performed as the SVD $\boldsymbol{X} \approx \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$, each vertex can receive an embedding both from $\boldsymbol{U}$ and $\boldsymbol{V}$. In that case, the embeddings from $\boldsymbol{U}$ and $\boldsymbol{V}$ are usually called respectively **left embedding** and **right embedding**.

A Random Walk technique consists in sampling random walks in the graph and then embed the paths of these random walks, considering that they reflect the structure of the graph.

Some Matrix Factorization techniques are deeply related to Random Walk techniques because the matrices they are based on represent probabilities to go from one edge to another in a random walk. For example, as defined in Section 3.2.3, the Personalized PageRank matrix is the probability matrix of a random walk with restart and therefore the matrix and an embedding derived from it could be approximated by sampling this random walk.

Finally, as suggested by their name, Deep Learning techniques rely on deep learning algorithms to learn the embedding of the vertices. Two main approaches exist, namely Autoencoder techniques and Graph Convolutional Networks (GCN). Autoencoders are neural networks, that consist in a first part called the **encoder** of which role is to compute the embedding and a second part called the **decoder** of which role is to retrieve the graph or part of it based on the embedding vectors. The two parts are trained together in an unsupervised manner. Graph Convolutional Networks on the other hand first find an embedding based on the direct neighborhood of each vertex, and then iteratively takes into account the embedding of vertices further away.

3.4.2 A brief history

One of the first graph embedding techniques was introduced in 2001 and is called **spectral embedding** [4]. Given an undirected graph, its Laplacian matrix $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}$ is symmetric and positive semidefinite. Therefore, this matrix has an eigendecomposition, i.e. there exist a diagonal matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{n \times n}$ and an unitary matrix $\boldsymbol{U} \in \mathbb{R}^{n \times n}$ so that:

$$\boldsymbol{L} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{U}^\top \tag{3.9}$$

The values in $\boldsymbol{D}$ called **eigenvalues** are in increasing order. The first eigenvalue will always be 0, and there will be as many eigenvalues that are 0 as there

are connected components in the matrix [4]. The embedding vectors, are the columns of $\boldsymbol{U}$, called **eigenvectors**, except for the first one which corresponds to the eigenvalue of 0.

If we denote by $\boldsymbol{Y}$ the embedding matrix, this embedding has been shown in [4] to solve the following problem:

$$Y = \begin{cases} \min\limits_{\boldsymbol{X} \in \mathbb{R}^{n \times k}} \sum\limits_{u=1}^{n} \sum\limits_{v=1}^{n} ||\boldsymbol{X}_{u,\cdot} - \boldsymbol{X}_{v,\cdot}||_2^2 \boldsymbol{A}_{uv} \\ \text{s.t. } \boldsymbol{X}^\top \mathbb{1} = \boldsymbol{0}_{\mathbb{R}^k} \\ \boldsymbol{X}^\top \boldsymbol{X} = \boldsymbol{I} \end{cases} \tag{3.10}$$

with $\boldsymbol{X}_{u,\cdot}$ the u^{th} line of $\boldsymbol{X}$.

The first constraint $\boldsymbol{X}^\top \mathbb{1} = \boldsymbol{0}_{\mathbb{R}^k}$ ensures that the embeddings are centered, and the second constraint $\boldsymbol{X}^\top \boldsymbol{X} = \boldsymbol{I}$ ensures that the embedding dimensions are uncorrelated.

The algorithm HOPE also relies on a matrix factorization technique and was presented in 2016 [46]. This algorithm is the closest to the one we present in Section 3.5. A parameter of this algorithm is the proximity matrix it will decompose. This can be chosen in a range of matrices the algorithm is defined on, including the personalized Katz and PageRank matrices. Then the chosen matrix is decomposed using a generalized version of the SVD. This results in two matrices $\boldsymbol{U}, \boldsymbol{V} \in \mathbb{R}^{n \times k}$, and a diagonal matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{k \times k}$. The left and right resulting embeddings are $\boldsymbol{U}\boldsymbol{\Sigma}^{1/2}$ and $\boldsymbol{V}\boldsymbol{\Sigma}^{1/2}$.

The first Random Walk embedding named DeepWalk was presented in 2014 [48]. This algorithm consists in sampling random walks of fixed length and it considers the sequence of vertices that form each random walk as a sentence of a corpus. The embedding is then computed by using Language Processing tools. Specifically, this method uses the SkipGram words embedding model.

This idea was taken one step further in 2016 with the very famous embedding method called node2vec [25]. In this method, the random walk is biased by two parameters p and q. The parameter p limits the "return" likelihood of the random walk, i.e. a high value of p will make it unlikely that the walker goes back to the node it just left. The parameter q limits the "spreading" behavior of the random walk, i.e. a high value of q will make it unlikely that the walker goes to a neighbor of the current node that would not also be a neighbor of the previous node.

3.4.3 The problem of interpretability

The interest for explainable algorithms is growing recently among the broad research community and in the general public alike. Many data-processing algorithms are fed with embeddings of complex data. If the embedding methods are not interpretable, there is little hope to provide satisfying explanations of the result of the algorithm. As a result, many recent papers focus on developing

embedding methods that are interpretable, such as [36] and [56] for image and video processing respectively. In the field of graph embeddings, most of the efforts have focused on defining measures to assess the interpretability of a graph embedding algorithm, with community-based metrics emerging as one of the most popular metrics. In particular, [23] and [33] develop three different community-based interpretability metrics and evaluate node2vec and HOPE in terms of those metrics. Those works suggest that a satisfactory solution for an interpretable graph embedding is still missing.

Metrics overview

Several metrics have been proposed to evaluate the interpretability of a graph embedding, such as the Interpretability Score (IS) [23], the Betweenness Centrality Importance (BCI) and Closeness Centrality Importance (CCI) [33]. Those metrics all consider the vector representation of a given node interpretable if it encodes somehow whether that node belongs to some real-world communities. To define them, we call "the value of a vertex u in the embedding dimension k" the k^{th} value in the vector $\boldsymbol{v_u}$ that represents u as a result of the embedding. We also denote by $\{C_1, \dots C_G\}$ a set of ground-truth communities as defined in section 3.3.

Interpretability score (IS)

The Interpretability Score (IS) was presented in [23]. Consider the k^{th} dimension of our embedding, and let g be the index of one of the ground-truth communities. To compute the IS for a (k, g) pair, a top score $IS_{top(k,g)}$ and a bottom score $IS_{bottom(k,g)}$ must be computed first. The top score is the recall@$|C_g|$ of the top-value vertices of the embedding, i.e. the ratio of the $|C_g|$ vertices of maximum value in the k^{th} embedding dimension that belongs to C_g. The bottom score is defined similarly on the lowest values. These scores evaluate respectively how well the highest and lowest values of the embedding reflect the belonging to the group.

The article then proposes to aggregate the scores by dimension or ground-truth communities using aggregation functions $f_{g\in[0,G]} : \mathbb{R}^G \longrightarrow \mathbb{R}$ and $h : \mathbb{R}^2 \longrightarrow \mathbb{R}$. These functions can be either the "max" or the "mean" function.

$$IS = \sum_{k=1}^{K} f_{g\in[0,G]}\big(h(IS_{top(k,g)}, IS_{bottom(k,g)})\big) \tag{3.11}$$

Betweenness Centrality Importance (BCI) and Closeness Centrality Importance (CCI)

These scores were introduced in [33]. They are defined based on the well-known Betweenness Centrality and Closeness Centrality scores, introduced in [6], and which have been presented in Section 1.2.2.

Betweenness Centrality Importance (BCI) Let C_g be a community, as defined in Section 3.3. Let $v, u_i, u_j \in C_g$ be three vertices of that community, so that $v \neq u_i \neq u_j$. We call "Local Betweenness Importance" of v between u_i to u_j the ratio:

$$\text{LBI}_{u_i,u_j}(v) = \frac{\sigma_{u_i u_j}(v)}{\sigma_{u_i u_j}} \tag{3.12}$$

where $\sigma_{u_i u_j}$ is the number of distinct shortest paths from u_i to u_j, and $\sigma_{u_i u_j}(v)$ is the number of these shortest paths that goes through v. In other words, $\text{LBI}_{u_i,u_j}(v)$ is the ratio of the shortest paths from u_i to u_j that go through v.[3]

Then the **Betweenness Centrality Importance** of a vertex $v \in C_g$ w.r.t. the community C_g is:

$$\text{BCI}_g(v) = \frac{\sum_{u_i,u_j \in C_g : u_i \neq u_j \neq v} \text{LBI}_{u_i,u_j}(v)}{n_v} \tag{3.13}$$

where n_v is the number of pairs of vertices in C_g, such that at least one of the shortest paths between them goes through v. In other words, the BCI score of a vertex v is the mean of its non-zero Local Betweenness Importance scores. If $n_v = 0$, then the score is set to 0.

Given an embedding dimension d and a community C_g the top-BCI score, denoted by $\text{BCI}^+_{d,g}$ is a metric of how the vertices of top-value in the embedding dimension d match the community. It uses the set $N^+_{d,g}$, which is defined as the set of $|C_g|$ vertices of highest values in the embedding dimension. Then, $\text{BCI}^+_{d,g}$ is defined by:

$$\text{BCI}^+_{d,g} = \text{mean}\{\text{BCI}_g(v) : v \in N^+_{d,g} \cap C_g \wedge \text{BCI}_g(v) \neq 0\} \tag{3.14}$$

The bottom-BCI score $\text{BCI}^-_{d,g}$ is defined similarly using the vertices of lowest values in the embedding dimension, and the final BCI score $\text{BCI}_{d,g}$ for the dimension d and the community C_g is the maximum of $\text{BCI}^+_{d,g}$ and $\text{BCI}^-_{d,g}$.

Finally, the BCI score of the dimension BCI_d is:

$$\text{BCI}_d = \max_{g \in [G]} \text{BCI}_{d,g} \tag{3.15}$$

and the BCI score of the embedding is the mean of the BCI scores of its dimensions.

Closeness Centrality Importance (CCI) Then the **Closeness Centrality Importance** of a vertex $v \in C_g$ w.r.t. the community C_g is:

$$\text{CCI}_g(v) = \frac{|C_g| - 1}{\sum_{u_i \in C_g, u_i \neq v} d(v, u_i)} \tag{3.16}$$

[3] The original article does not define the Local Betweenness Importance and directly introduces the BCI score with equation (3.13). However, we think that this intermediary step helps to understand both the computation and the meaning of this score.

where $d(v, u_i)$ is the length of the shortest path from v to u_i. This is the inverse of the mean distance from v to the other vertices of the community. Note that $\text{CCI}_g(v)$ is not defined when the community in not connected.

Then, the aggregation to a single metric for the whole embedding is very similar to the BCI. First, a top- and a bottom-CCI score is computed as the mean of the non-zero CCI scores of the vertices of highest and lowest values in the embedding dimension that belong to the community:

$$\text{CCI}^+_{d,g} = \text{mean}\{\text{CCI}_g(v) : v \in N^+_{d,g} \cap C_g \wedge \text{CCI}_g(v) \neq 0\} \tag{3.17}$$

Then the CCI scores of a dimension is the maximum of its top and bottom scores over all the communities. Finally, the CCI score of the embedding is the mean of the CCI scores of the dimensions.

Limitations of the BCI and CCI The BCI or CCI score for an embedding dimension and a community is the mean of the non-zero scores for the vertices that belong to $N^+_{d,g} \cap C_g$ (see equations (3.14) and (3.17)). The fact that only the non-zero scores and only the vertices from the community are considered make it very possible that the results of these metric would not really reflect the interpretability of the embedding.

Let us consider for example a community C_g and two embedding dimensions k_1 and k_2 so that the first one contains only one very central vertex of C_g among its top vertices, and the second contains this same vertex plus another slightly less central vertex of C_g. Then, although arguably much more interpretable w.r.t. the g^{th} ground-truth community, the k_2 embedding dimension will have a lower score than the k_1 one.

A new interpretability metric: CISIP

A weak point of the metrics already proposed in the literature is that, although they evaluate the fitness of an embedding dimension to a ground-truth group, they do not evaluate how well the embedding separates the data and the redundancy between the dimensions.

Let us take the IS as an example: if 50% of the $|C_1|$ highest values for the first dimension belong to C_1, then $IS_{top(1,1)} = 0.5$. But then it is possible that 50% of the highest values for the second dimension are made either of the exact same part of C_1, or of the other points of C_1. The former case would denote a strong redundancy for the interpretation of these dimensions, while the latter case would denote that the first group is halved between these dimensions, hence reducing the interpretability of both dimensions. In both cases, $IS_{top(2,1)} = 0.5$.

On top of that all these metrics only consider the nodes that receive top- or bottom-$|C_g|$ scores from the embedding, and all these vertices are considered with equal weight. It seems however natural to consider that the importance should be decreasing before reaching the $|C_g|$ threshold, e.g. the vertex with the highest value should have higher importance than the vertex with the

second-highest value. Similarly, it seems that the importance should not drop to 0 after the $|C_g|^{\text{th}}$ value: if two embedding dimensions have the exact same top-$|C_g|$ vertices, but one has its $(|C_g|+1)^{\text{th}}$ vertex belonging to C_g while the other has not, it seems natural to consider that the first one is more interpretable w.r.t. C_g.

Algorithm 3.2 CISIP

```
1: procedure CISIP(E, {C_1, ... C_G}, f)
2:     l ← min(K, G)
3:     {(k_1, g_1), ... (k_l, g_l)} ← hungarian(E, {C_1, ... C_G})
4:     sum_scores ← 0
5:     for i ∈ [l] do
6:         sum_scores ← sum_scores + Weighted_Kendall_Tau(E_{·,k_i}, f(C_{g_i}))
7:     end for
8:     return sum_scores/l
9: end procedure
```

To tackle these weaknesses, we propose the new Complete Interpretability Score Integrating Priority (CISIP) metric. The procedure to compute this metric is presented in Algorithm 3.2. The input parameters are the embedding matrix $E \in \mathbb{R}^{n \times k}$ in which each line corresponds to a vertex and each column to an embedding dimension, the set of ground-truth communities $\{C_1, \dots C_G\}$, and a smoothing function f that, given a community C_g, creates a vector $f(C_g) \in \mathbb{R}^n$ that scores how much each vertex belong to the community, or how important each vertex is to the community. The simplest of these function, which we call the "identity function", gives a score of 1 to the nodes that belong to the community, and 0 to the other nodes:

$$\forall u \in V, \Big(\text{identity}(C_g)\Big)_u = \begin{cases} 1 & \text{if } u \in C_g \\ 0 & \text{else} \end{cases} \tag{3.18}$$

Then, CISIP assign each embedding dimension to a unique community at Line 3.[4] This is done using the Hungarian algorithm for optimal assignment. Given two sets P and Q and a score $s(p_i, q_j)$ for each pair $(p_i, q_j) \in P \times Q$, this algorithm finds the assignments $\{(p_{i_1}, q_{j_1}), \dots (p_{i_l}, q_{j_l})\}$, with $l = \min(|P|, |Q|)$ that maximizes the sum of scores of the pairs, with the constraint that each element of the smaller set must be uniquely assigned with an element of the bigger set.

In our case, the sets are the set of indices $[K]$ of the embedding dimensions, and $[G]$ of the communities. For each pair (k, g) we would ideally use the weighted Kendall Tau score of $E_{\cdot,k_i}$ and $f(C_{g_i})$ (explained later in this section), as this would make the assignment maximize the final CISIP score. However,

[4] If there are fewer communities than dimension, then each community is assigned to a unique dimension instead.

the weighted Kendall Tau score is computationally expensive, and it would be untractable to compute it for each pair of embedding dimension and community. Therefore, we use a much simpler and much more time-efficient score at this step: the sum of embedding dimension values for the vertices of the community. To account for the fact that vertices belonging to the community can have high negative scores instead of high positive scores, we use the maximum between this sum and its opposite:

$$s(k, g) = \max\Big((-\sum_{u \in C_g} E_{u,k}), (\sum_{u \in C_g} E_{u,k})\Big) \tag{3.19}$$

Note that this is effectively the absolute value of the sum but, crucially, we need to keep track of whether we keep the sum or its opposite, in order to know whether we expect the vertices with the highest scores or those with the lowest scores to match the community.

Once this assignment has been computed, we score each pair (k, g) of the assignment. We first transform the community C_g into a vector using the smoothing function f. This function gives to each vertex a value that reflects its belonging or importance in the community. Then, the **weighted Kendall Tau** scoring function is used to score similarity between the vector and the embedding dimension (Line 6). This function was introduced in [53]. It scores two vectors r and s between -1 and 1. The score 1 is reached when the ranking of the indices in both vectors is the same, i.e. the index of highest value is the same in both vectors, as is this index of second-highest value and so on, and there are no ties, i.e. $\forall i, j : r_i \neq r_j$ and $s_i \neq s_j$. The score -1 is reached when the ranking is opposite between the two vectors, i.e. the index of highest value in r is the index of lowest value in s and so on, and there are no ties. Crucially, this scoring function values more the higher-rank indices.

Finally, the CISIP score is the mean weighted Kendall Tau score of the pairs assigned together.

The smoothing function is a parameter of the CISIP metric. In this thesis, we consider the identity function, which is defined in Equation (3.18), the Personalized PageRank of the community, and the Mean Neighbors Belonging (MNB) function, which is defined as follows:

$$\forall u \in V, \Big(\text{MNB}(C_g)\Big)_u = \frac{1}{d_u^+} \sum_{v \in \mathcal{N}^+(u)} \Big(\text{identity}(C_g)\Big)_v \tag{3.20}$$

Note that, as explained in Definition 1.16, we can define the out-neighborhood of a vertex in an undirected graph as equal to the neighborhood. This way, the definition of MNB applies both to directed and undirected graphs.

One weakness of our method that should be noted is that ties in any of the vectors that would not be in the other vector would reduce the score. This behavior is actually wanted when the tie is in the embedding dimension because an embedding with ties is arguably less interpretable when these

ties are not also in the ground-truth scoring, but it limits drastically the max achievable score when the ground-truth scoring contains many ties (e.g. when the smoothing uses the identity or the Mean Neighbors Belonging function).

3.5 A new interpretable graph embedding

3.5.1 Overview

We introduce the new PageRank Factorization-based Interpretable Graph Embedding (PARFAITE) method. It produces two embeddings, PARFAITE_L (left) and PARFAITE_R (right).

Algorithm 3.3 PARFAITE

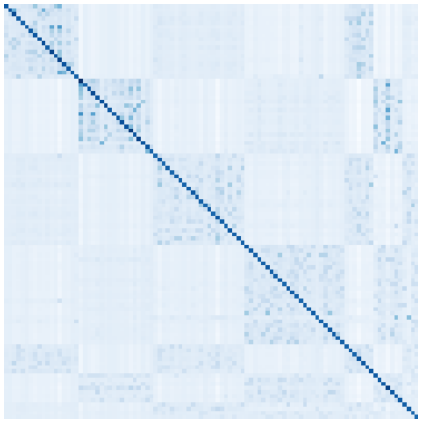
1: **procedure** PARFAITE($m_P : \mathbb{R}^n \to \mathbb{R}^n$)
2: $\boldsymbol{U}, \boldsymbol{\Sigma}, \boldsymbol{V} \leftarrow$ SVD(m_P)
3: clustering $\leftarrow$ kmeans(concat(normalize($\boldsymbol{U\Sigma}$), normalize($\boldsymbol{V\Sigma}$)))
4: $\boldsymbol{C} \leftarrow$ clustering.clusters_centers
5: PARFAITE_L $\leftarrow \boldsymbol{U}\boldsymbol{C}^{\top}_{\cdot,d:}$
6: PARFAITE_R $\leftarrow \boldsymbol{V}\boldsymbol{C}^{\top}_{\cdot,:d}$
7: **return** PARFAITE_L, PARFAITE_R
8: **end procedure**

Figure 3.1 illustrates the algorithm. A pseudocode of our algorithm is given in Algorithm 3.3. The parameter m_P is a function that, for each vector $\boldsymbol{v} \in \mathbb{R}^n$, approximate $\bar{\boldsymbol{\Pi}}\boldsymbol{v}$. Section 3.5.3 gives more details about this function.

Our method consists of three main parts. First, a truncated Singular Values Decomposition is performed on the centered PPR matrix (Line 2 of Algorithm 3.3). To overcome the issues of computing and representing the PPR matrix exactly for very large graphs, we employ a function m so that $\forall v \in \mathbb{R}^n : m(v) \approx \bar{\Pi}(v)$, where $\bar{\Pi}$ is the centered PPR matrix. This approximation function will be defined in section 3.5.3.

This results in two representations of the vertices as $\boldsymbol{U\Sigma}$ and $\boldsymbol{V\Sigma}$, illustrated on Figures 3.1b and 3.1c.

Then, vertices are clustered, with each vertex being represented by a concatenation of its left and right representation (Line 3). This provides the central points of the communities in the space of these representations, stored in the matrix $\boldsymbol{C}$. These central points are represented on Figure 3.1d. Finally, our left and right embeddings, respectively PARFAITE_L and PARFAITE_R are computed by projecting back these central points onto the original spaces (Lines 5-6). The results are represented on Figures 3.1e and 3.1f.



(a) Initial centered matrix $\bar{\mathbf{\Pi}}$

(b) Projection on the left singular subspace

(c) Projection on the right singular subspace

Left Right

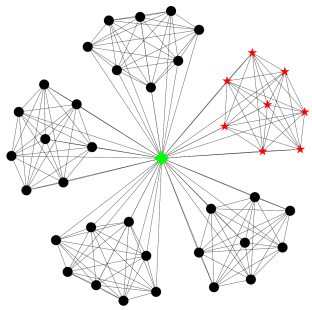
(d) Centers of clusters in the singular subspaces

(e) Left embedding PARFAITE_L

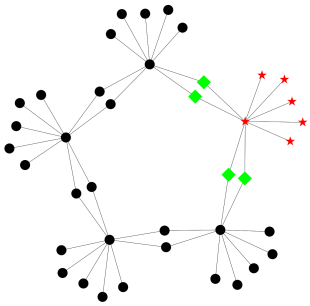
(f) Right embedding PARFAITE_R

(g) Ground-truth community belonging

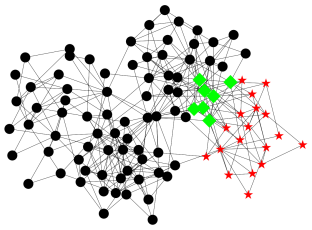
Figure 3.1: Illustration of the PARFAITE embedding algorithm applied on the graph on Figure 3.2c. For layout reasons, the scale of the matrix 3.1a is much smaller than the scale of the other matrices, and the matrices 3.1b, 3.1c, 3.1e and 3.1f are transposed.
An SVD is performed on the matrix 3.1a resulting in the matrices 3.1b and 3.1c. Then a clustering results in the cluster centers on Subfigure 3.1d. Finally, these centers are projected back on the original space, resulting in the matrices 3.1e and 3.1f. Subfigure 3.1g gives the ground-truth community belonging.



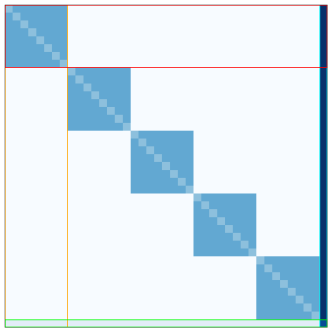
(a) Star of cliques



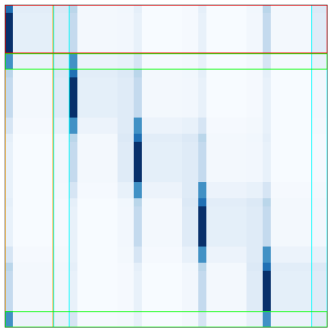
(b) Ring of stars



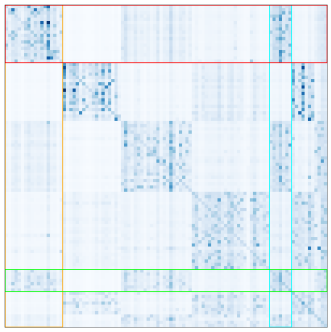
(c) Toy Graph from SBM



(d) $\mathbf{\Pi} - \alpha \boldsymbol{I}$ matrix of the star of cliques



(e) $\mathbf{\Pi} - \alpha \boldsymbol{I}$ matrix of the ring of stars



(f) $\mathbf{\Pi} - \alpha \boldsymbol{I}$ matrix of the toy graph from SBM

Figure 3.2: Toy graphs and their respective $\mathbf{\Pi} - \alpha \boldsymbol{I}$ matrices. On each graph, a community is highlighted, the red stars vertices belonging exclusively to the community, and the green diamonds ones belonging both to the highlighted community and to at least one other. On each matrix, the rows and columns relative to the red star vertices are highlighted through red and orange boxes, and the rows and columns relative to the green diamond vertices are highlighted through green and cyan boxes. We use $\mathbf{\Pi} - \alpha \boldsymbol{I}$ instead of $\mathbf{\Pi}$ for better readability.

3.5.2 Interpretation of the steps

The PARFAITE method relies on the well-established fact that the PPR vector of each vertex often contains large scores at dimensions corresponding to vertices it shares at least one community with, while it contains small scores at other dimensions [28, 34]. When we consider the PPR vector as an importance scoring of the vertices of the graph w.r.t. the source vertex, it means that the vertices of the graph that are the most important to the source vertex are those which share a community with it.

This property strengthens the one stated in [59] that two nodes are likely to share a community not only if the PPR score from one node to the other is high, but especially if their PPR vectors are similar., meaning that they "agree with each other in terms of their personalized views about the network" [59].

We illustrate this property with three toy graphs represented in Figure 3.2. The first two graphs provide ideal cases where we can find communities according to two natural community definitions: a) a clique, where each node is connected to every other node in the community, and b) a star where all nodes but one are connected to a single node (e.g. they are connected to a same influencer in social media). In particular, Figure 3.2a represents a star of cliques, while Figure 3.2b represents a ring of stars. Observe that the communities in the graphs of Figure 3.2 do **overlap**, which means that some nodes belong to several communities at the same time. On each graph represented in Figure 3.2, one community is outlined. Red star vertices are those that belong exclusively to the outlined community, while green diamond vertices belong to the outlined community and at least another one. The graph in Figure 3.2c is built using the Stochastic Block Model (SBM) with 100 vertices, 300 edges, 4 communities, with each pair of vertices of a same community being 100 times more likely to be connected than two vertices from different communities. As a result, in Figure 3.2c, there are 19 vertices belonging to exactly two communities, 7 of which are depicted in green.

Figures 3.2d, 3.2e and 3.2f show the corresponding PPR matrices. Recall that each row represents the PPR vector of some vertex v, i.e. the PPR scores when v is the source vertex of the random walk. Indeed, we can see that PPR vectors contain larger scores to vertices of a same community as the source node, generating "square" patterns in the matrices.

Similarly, we observe that the reversed PPR vectors (columns in the matrix) exhibit the same pattern of large scores inside the community and low scores outside.

However, there might be relatively few vertices that have very large PPR scores even if they do not share communities with other nodes. This is apparent in Figure 3.2e, where we observe that the central nodes in the neighboring cliques have higher scores in the PPR vector of the studied community than other nodes in the same community. The reversed PPR is not affected by this issue.

To avoid this bias, we define the $\bar{\mathbf{\Pi}}$ matrix obtained by centering the columns of $\mathbf{\Pi}$.

We study this centering using the interpretation of a PPR vector as the final probability vector of a random walk that has probability α at each step to stop permanently (see Section 3.2.3). We have the following property

Property 3.5. *Let X_f be the random variable of the final node of the walk, and X_0 the random variable of the initial node of the walk. We have:*

$$\bar{\mathbf{\Pi}}_{i,j} = \frac{1}{n}\mathbb{P}\left(X_f = j\right)\left(\mathbb{P}\left(X_0 = i | X_f = j\right) - \mathbb{P}\left(X_0 = i\right)\right)$$

Proof. We know that $\mathbf{\Pi}_{ij} = \mathbb{P}(X_f = j | X_0 = i)$. Therefore, we see that:

$$\begin{aligned}\bar{\mathbf{\Pi}}_{i,j} &= \mathbb{P}(X_f = j | X_0 = i) - \mathbb{P}(X_f = j) \\ &= \frac{\mathbb{P}(X_0 = i | X_f = j)\,\mathbb{P}(X_f = j)}{\mathbb{P}(X_0 = i)} - \mathbb{P}(X_f = j)\end{aligned}$$

□

If we look at the rows of this new matrix, each entry is then the excess of probability to go to each vertex from the reference vertex, compared to the agnostic probability. If we look at the columns and because $\mathbb{P}(X_f = j)$ is a constant along a column, each entry is proportional to the excess of probability to come from each vertex given that the walk arrived at the reference vertex.

Then a Singular Value Decomposition is performed. One of the main advantages of the SVD, which make it especially attractive for embedding, is that it is known to filter out the noise in the data. More specifically, the SVD removes the small variations between data so that only the main patterns in the matrix are kept in the final result [35].

Therefore, we expect the result of the SVD to exhibit the typical rows and columns for each community. We expect these typical rows and columns to have patterns of the excess of probability to respectively come from and arrive into the community, given that we respectively arrived into and came from the community. We could then interpret these vectors as respectively the belonging of each node to the community and its importance in the community.

Our last problem is that, although the truncated SVD should make the community patterns in the matrix apparent, each dimension of the decomposition usually does not match a community, hindering the interpretation. To tackle this issue, we perform a clustering on the vertices represented by the SVD, and we use the central points to obtain the desired representative vectors for each community. Note that, although the clustering itself is non-overlapping and non-fuzzy, the resulting vectors are fuzzy scoring of belonging and importance of the nodes in each community.

3.5.3 Decomposition of the PPR matrix

The PPR matrix is a dense matrix belonging to $\mathbb{R}^{n \times n}$. In most cases of big graphs, this matrix is too big to be explicitly represented. Most modern SVD algorithms do not require an explicit representation of the matrix $\boldsymbol{M}$ but only a function $m_P(\boldsymbol{v}) = \boldsymbol{M}\boldsymbol{v}$. We use a reduced form of equation (3.6) to approximate the PPR matrix.

$$m_P(\boldsymbol{v}) = \bar{\mathbf{\Pi}}_{\boldsymbol{l}}\boldsymbol{v} = \left(\alpha \sum_{i=0}^{l} (1-\alpha)^i \boldsymbol{P}^i \boldsymbol{v}\right) - (\boldsymbol{\pi} \cdot \boldsymbol{v})\mathbf{1} \tag{3.21}$$

where $\mathbf{1}$ is the vector of which all entries equal 1 and $\boldsymbol{\pi}$ is the (not-personalized) PageRank vector of the matrix.

We know from [34] that a few steps are usually enough to compute an approximation of the PageRank vector that outlines the community of the node. We fix $l = 10$ for the rest of this section.

3.5.4 Finding the communities

SVD provides representations that should, if used to reconstruct the matrix, contain only the main patterns of the matrix which are the communities. There is no reason however to think that the dimensions of the representations correspond themselves to communities. That is why we perform a clustering of the vertices using their SVD representations to find the communities. Since both matrices of the embedding provide relevant and distinct information, there is no reason to exclude one and therefore we use a concatenation of both sides as the vectors for the clustering. Unfortunately, we did not have enough time during this thesis to study which clustering algorithm would perform best for this task, and we simply use the well-known k-means algorithm [38] and its initialization variant k-means++ [2].

As we saw before, the PPR and reversed PPR vectors for vertices of a same community are expected to have similar patterns of high and low entries, which correspond to similar directions of the vectors. However, they are not supposed to have similar norms, especially for the reversed PPR, for which the norm typically follows the importance of the vertex. To make the clustering algorithm work on directions and not on Euclidean distance, both embeddings are normalized before concatenation and then the cosine similarity is used.

3.5.5 Reconstructing the communities rows and columns

The $\boldsymbol{U}\boldsymbol{\Sigma}$ and $\boldsymbol{V}\boldsymbol{\Sigma}$ embeddings are the projection of the columns and rows of the centered PPR matrix onto the singular spaces, e.g. for a vertex of the graph w, we have $(\boldsymbol{U}\boldsymbol{\Sigma})_{w,\cdot} = \bar{\boldsymbol{\pi}}_{\boldsymbol{w}}{}^{\top}\boldsymbol{V}$. Therefore, the clusters centers are the central columns and rows of each community, projected on the singular spaces. To reconstruct the true central columns and rows of the communities, which are the typical centered reversed PPR and centered PPR vectors for the community, we multiply by the transposed of the projection matrices, which are $\boldsymbol{U}$ and $\boldsymbol{V}$. Note that this reconstruction is not perfect because the dimensions of the singular spaces we use are smaller than the dimension of the original space.

3.6 Experiments

3.6.1 Experimental setting

Our main goal is to show that our method provides better interpretability scores than state-of-the-art approaches, while boasting similar results for a popular machine learning task in graph analysis, such as link prediction.

Datasets. We use two datasets that are available on the SNAP website [57]. The first one named *Wikispeedia* contains 4592 vertices, and 120 000 edges. 105 overlapping ground-truth community are given. The second one named *Facebook* contains 10 graphs. We exclude 2 of them, numbered 698 and 3980, because they contain fewer than 128 nodes and we could therefore not compute the SVD for them using the same parameters used for the others. The remaining 8 graphs of *Facebook* contain between 155 and 1035 vertices and between 3312 and 60050 edges. Between 7 and 46 overlapping ground-truth communities are given for each graph.

We also use the WikipediaFr dataset presented in Section 3.3. This graph contains 2.52 millions vertices, 102 million edges and 2 700 ground-truth communities. To keep the dimension of the embeddings manageable, however, we only study the 117 communities that contain more than 10 000 vertices.

Methods. We evaluate our method against the two widely used algorithms for graph embedding HOPE and node2vec described in Section 3.4.2.

For node2vec, we use the most-used python3 implementation [18]. We keep the default parameters, i.e. the number of walks is 200 per vertex, the length of the walks is 30 and the window size is 10.

For HOPE, we use the official implementation in Matlab[5] provided by the authors of [46], which we reimplement in python3 (while using numpy and scipy), to provide a fair comparison with the other approaches. We keep all the constants and parameters as available in this implementation.

Our algorithm is implemented in python3, using mainly the numpy [26] and scipy [54] packages. The restart parameter α for the PPR algorithm is set to $\alpha = 0.1$. The number l of iterations to approximate the PPR matrix is set to $l = 10$ and the dimension D of the intermediate SVD is set to $D = 128$.

For all embeddings, the dimension K of the embedding is set to be the number G of known ground-truth communities.

Metrics. We measure the interpretability of the methods using each of the metrics mentioned in the "metrics" part of Section 3.4.3.

We consider as the Interpretability Scores (IS) of a pair (embedding dimension, community) the maximum between the top- and bottom IS score for that pair. In order to obtain a single result, the IS are then aggregated along the ground-truth groups using the max function, and then along the embedding dimensions using the mean function. The use of the mean allows us to obtain scores between 0 and 1. However, we should keep in mind that, counterintuitively, it makes it possible that the adding of new dimensions to the embedding worsens it score. In contrast with [23], we do not multiply the result by 100, which only changes the results by this factor without any other impact. Formally, the definition of the Interpretability Score we use is:

[5] https://github.com/ZW-ZHANG/HOPE

$$IS = \frac{1}{K}\sum_{k=1}^{K} \max_{g\in[0,G]} \left(\max(IS_{top(k,g)}, IS_{bottom(k,g)})\right) \tag{3.22}$$

The size of the top and bottom sets considered for the Betweenness and Closeness Centrality Indices (BCI and CCI) are set to the size of the studied ground-truth group, as suggested by the original paper. For CISIP, three smoothing functions f are considered. The first one, which we call "identity", is the direct use of the binary ground-truth belonging feature, the second one is the Mean Neighbors Belonging (MNB), i.e. the mean of the belonging features of the neighbors, and the last one is the PPR of the community.

The BCI and CCI are not computed for the WikipediaFr dataset for performances reasons. The BCI and CCI results are not considered reliable because of the limitations due to their normalization, as explained in Section 3.4.3.

3.6.2 Results and discussion

The results for the IS and CISIP scores are given in Tables 3.1 and 3.2. Results for the BCI and CCI scores, for PARFAITE and node2vec, are given in Appendix A.

As we see in these results, our algorithm's interpretability is much higher than node2vec's when evaluated using the Interpretability Score or any of the variants of CISIP. HOPE's interpretability is closer, but still generally below PARFAITE's.

Our left embedding greatly outperforms the right one when using CISIP with the PPR smoothing, and this result was expected, as the left embedding is an approximation of the typical PPR vector of each cluster detected. The results are however equivalent between our two embeddings when compared using either the IS or CISIP with the identity smoothing, which both evaluate directly the matching between the embedding and the belonging features, or with the MNB smoothing. This is consistent with our interpretation that the left embedding represents which communities a vertex belongs to, while the right embedding measures somehow the "importance" of a vertex in a community.

Is the clustering needed ?

To check if the last step of our algorithm of clustering the data and computing the final embeddings is really needed, we compare our results to what we would have without the clustering step. To achieve this, we take the $\boldsymbol{U\Sigma}$ and $\boldsymbol{V\Sigma}$ results from the SVD, and we keep only the G first dimensions to match the dimension of the PARFAITE embeddings so that the dimension of this embedding is identical to the dimension of the PARFAITE embeddings.

The results for IS and for CISIP with the three smoothing functions already used are presented in Table 3.3 and 3.4. The results for the BCI and CCI scores are given in Appendix A. We see that in most cases the results of our

PARFAITE_L and PARFAITE_R outperform those before clustering, sometimes significantly (e.g. more than 0.1 points of difference for the IS). This is especially true for the IS and CISIP with PPR smoothing. We note however that the SVD provides better results in several occurrences when compared to our embeddings using CISIP with the identity or the MNB smoothing.

Overall we conclude that PARFAITE_L and PARFAITE_R do generally perform better than single SVD, i.e. without the clustering and reconstruction steps.

Is our embedding efficient ?

The method we propose mainly focuses on the interpretability of the embedding. However, this interpretability should not come at the cost of an excessive loss of efficiency in the task the embedding helps to solve.

We check the efficiency of PARFAITE against HOPE and node2vec at the task of Link Prediction. We build test graphs by removing 0.1% of the edges of a real-world graph. We store the pairs of vertices of these edges as "positive" pairs, and build a set of "negative" pairs by drawing the same number of pairs of vertices that are not connected by an edge.

The embedding of the test graph is computed, and a score is attributed to each positive or negative pair of vertices using this embedding.

For HOPE, the score is the dot product of the left embedding of the first vertex in the pair and the right embedding of the second vertex. The score for PARFAITE is similar, but the left embedding of the first vertex is normalized to account for the entire use of the singular values on each side of the embedding. For node2vec, following [25], a Logistic Regression is trained on the test graph by representing the pairs with a concatenation of their vertices' embeddings.

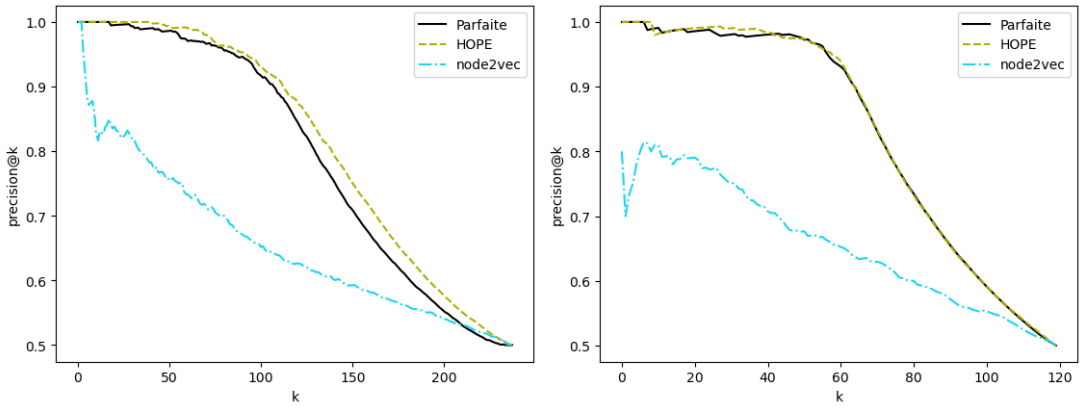
We run this experiment on 10 test graph for both the Wikispeedia dataset and on the 1912 part of the Facebook dataset, as the part with the highest order. The mean results are given in Figure 3.3. As we can see, node2vec is outperformed by both HOPE and PARFAITE. HOPE achieves similar results as PARFAITE on Facebook 1912 and slightly better on Wikispeedia, but overall we can say that the greater interpretability of PARFAITE does not come at the cost of a much lower efficiency on the task of link prediction.

3.7 New Personalized PageRank properties

We present new properties of the PPR matrix that, to the best of our knowledge, were not known beforehand. Theorem 3.8, particularly, opens the door to a new embedding and sheds a new light on the spectral embedding.

Theorem 3.6. *If* $\mathbf{\Pi}$ *is the PPR matrix and* $\boldsymbol{M}$ *the stochastic matrix of the graph, we have*

$$\mathbf{\Pi}\boldsymbol{M} = \boldsymbol{M}\mathbf{\Pi} \tag{3.23}$$



(a) Wikispeedia (b) Facebook 1912

Figure 3.3: Mean Precision@k of the Link Prediction on 20 test graphs built from the Wikispeedia and Facebook 1912 datasets

Proof.

$$\begin{aligned}\boldsymbol{\Pi M} - \boldsymbol{M\Pi} &= \cancel{\alpha \boldsymbol{M}} + (1-\alpha)\boldsymbol{\Pi M}^2 \cancel{- \alpha \boldsymbol{M}} - (1-\alpha)\boldsymbol{M\Pi M} \\ &= (1-\alpha)(\boldsymbol{\Pi M} - \boldsymbol{M\Pi})\boldsymbol{M}\end{aligned}$$

so

$$(\boldsymbol{\Pi M} - \boldsymbol{M\Pi})(\boldsymbol{I} - (1-\alpha)\boldsymbol{M}) = 0_{\mathbb{R}^{n\times n}}$$

with $0_{\mathbb{R}^{n\times n}} \in \mathbb{R}^{n\times n}$ the zero matrix.

Since $(\boldsymbol{I} - (1-\alpha)\boldsymbol{M})$ is invertible, we deduce that

$$(\boldsymbol{\Pi M} - \boldsymbol{M\Pi}) = 0_{\mathbb{R}^{n\times n}}$$

□

Theorem 3.7. *If the graph G is undirected, then the matrices $\boldsymbol{D\Pi}$ and $\boldsymbol{\Pi D}^{-1}$ are symmetric.*

Proof. We first prove that $\boldsymbol{\Pi D}^{-1}$ is symmetric. We note that since G is undirected, $\boldsymbol{A}$ is symmetric. We also recall that $\forall \boldsymbol{B}, \boldsymbol{C}$ invertible matrices, $(\boldsymbol{BC})^{-1} = \boldsymbol{C}^{-1}\boldsymbol{B}^{-1}$.

We use the definition of $\boldsymbol{\Pi}$ given by the Equation (3.7). Then we have

$$\begin{aligned}\boldsymbol{\Pi D}^{-1} &= \alpha(\boldsymbol{I} - (1-\alpha)\boldsymbol{M})^{-1}\boldsymbol{D}^{-1} \\ &= \alpha(\boldsymbol{D} - (1-\alpha)\boldsymbol{D}\underbrace{\boldsymbol{M}}_{=\boldsymbol{D}^{-1}A})^{-1} \\ &= \alpha(\boldsymbol{D} - (1-\alpha)\underbrace{\boldsymbol{A}\boldsymbol{D}^{-1}\boldsymbol{D}}_{=\boldsymbol{M}^\top \boldsymbol{D}})^{-1} \\ &= \alpha\boldsymbol{D}^{-1}(\boldsymbol{I} - (1-\alpha)\boldsymbol{M}^\top)^{-1} \\ &= \boldsymbol{D}^{-1}\boldsymbol{\Pi}^\top\end{aligned} \tag{3.24}$$

Table 3.1: Interpretability Score results for PARFAITE, HOPE and node2vec

Dataset	PARFAITE		HOPE		node2vec
	left	right	left	right	
Wikispeedia	**0.389**	0.375	0.200	0.266	0.139
Facebook 0	0.520	**0.567**	0.529	0.530	0.429
Facebook 107	**0.626**	0.601	0.550	0.550	0.323
Facebook 348	**0.968**	0.928	0.888	0.888	0.895
Facebook 414	0.854	**0.861**	0.683	0.683	0.414
Facebook 686	**0.736**	0.730	0.650	0.650	0.617
Facebook 1684	**0.863**	0.807	0.572	0.572	0.304
Facebook 1912	**0.766**	0.764	0.603	0.603	0.348
Facebook 3437	0.430	**0.759**	0.429	0.429	0.188
WikipediaFr	0.040	**0.083**	0.041	**0.083**	*

* The embedding of this graph with node2vec as been stopped after 36h of computation.

The third inequality derives from the facts that $\boldsymbol{D}^{-1}\boldsymbol{D} = \boldsymbol{I}$ and that $\boldsymbol{A} = \boldsymbol{I}\boldsymbol{A} = \boldsymbol{A}\boldsymbol{I}$

The proof that $\boldsymbol{D}\boldsymbol{\Pi}$ is symmetric follows directly from $\boldsymbol{\Pi}\boldsymbol{D}^{-1} = \boldsymbol{D}^{-1}\boldsymbol{\Pi}^\top$ □

Theorem 3.8. *If the graph G is undirected, then the matrices $\boldsymbol{D}\boldsymbol{\Pi}$ and $\boldsymbol{\Pi}\boldsymbol{D}^{-1}$ are positive-definite*

We note that if a matrix $\boldsymbol{\Gamma}$ is symmetric and positive-definite, then there is an eigendecomposition $\boldsymbol{\Gamma} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{U}^\top$ so that $\boldsymbol{U}$ is a unitary matrix and $\boldsymbol{D}$ is diagonal and strictly positive. Therefore, the inverse matrix $\boldsymbol{\Gamma}^{-1} = \boldsymbol{U}\boldsymbol{\Sigma}^{-1}\boldsymbol{U}^\top$ is also symmetric and positive-definite. We also know from Equation (3.7) that the inverse of $\boldsymbol{D}\boldsymbol{\Pi}$ and $\boldsymbol{\Pi}\boldsymbol{D}^{-1}$ are respectively $\frac{1}{\alpha}\big(\boldsymbol{D}^{-1} - (1-\alpha)\boldsymbol{M}\boldsymbol{D}^{-1}\big)$ and $\frac{1}{\alpha}\big(\boldsymbol{D} - (1-\alpha)\boldsymbol{A}\big)$. The proof will therefore consist in proving that the matrices $\big(\boldsymbol{D}^{-1} - (1-\alpha)\boldsymbol{D}^{-1}\boldsymbol{A}\boldsymbol{D}^{-1}\big)$ and $\big(\boldsymbol{D} - (1-\alpha)\boldsymbol{A}\big)$ are positive-definite, i.e. that the quadratic forms defined by these matrices over $\mathbb{R}^n$ is an inner product.

Proof that $\big(\boldsymbol{D}^{-1} - (1-\alpha)\boldsymbol{M}\boldsymbol{D}^{-1}\big)$ *is positive-definite.* $\forall \boldsymbol{x} \in \mathbb{R}^n$, we have:

$$\begin{aligned}
\boldsymbol{x}^\top\big(\boldsymbol{D}^{-1} - (1-\alpha)\boldsymbol{M}\boldsymbol{D}^{-1}\big)\boldsymbol{x} &= \boldsymbol{x}^\top\big(\boldsymbol{D}^{-1} - (1-\alpha)\boldsymbol{D}^{-1}\boldsymbol{A}\boldsymbol{D}^{-1}\big)\boldsymbol{x} \\
&= \sum_{u=1}^{n}\frac{\boldsymbol{x}_u^2}{d_u} - (1-\alpha)\sum_{u=1}^{n}\sum_{v=1}^{n}\frac{\boldsymbol{x}_u\boldsymbol{x}_v}{d_u d_v}\boldsymbol{A}_{uv} \\
&= \alpha\sum_{u=1}^{n}\frac{\boldsymbol{x}_u^2}{d_u} + (1-\alpha)\sum_{u=1}^{n}\big(\frac{\boldsymbol{x}_u^2}{d_u} - \sum_{v=1}^{n}\frac{\boldsymbol{x}_u\boldsymbol{x}_v}{d_u d_v}\boldsymbol{A}_{uv}\big) \\
&= \alpha\sum_{u=1}^{n}\frac{\boldsymbol{x}_u^2}{d_u} + (1-\alpha)\sum_{u=1}^{n}\sum_{v=1}^{n}\big(\frac{\boldsymbol{x}_u^2}{d_u^2} - \frac{\boldsymbol{x}_u\boldsymbol{x}_v}{d_u d_v}\big)\boldsymbol{A}_{uv} \\
&= \alpha\sum_{u=1}^{n}\frac{\boldsymbol{x}_u^2}{d_u} + (1-\alpha)\sum_{u=1}^{n}\sum_{v=1}^{n}\frac{\boldsymbol{x}_u}{d_u}\big(\frac{\boldsymbol{x}_u}{d_u} - \frac{\boldsymbol{x}_v}{d_v}\big)\boldsymbol{A}_{uv} \\
&= \alpha\sum_{u=1}^{n}\frac{\boldsymbol{x}_u^2}{d_u} + \frac{1-\alpha}{2}\sum_{u=1}^{n}\sum_{v=1}^{n}\big(\frac{\boldsymbol{x}_u}{d_u} - \frac{\boldsymbol{x}_v}{d_v}\big)^2\boldsymbol{A}_{uv}
\end{aligned} \tag{3.25}$$

We see that $\boldsymbol{x}^\top\big(\boldsymbol{D}^{-1} - (1-\alpha)\boldsymbol{M}\boldsymbol{A}\boldsymbol{D}^{-1}\big)\boldsymbol{x} \geq 0$ and that 0 is reached only when $\boldsymbol{x} = \mathbf{0}_{\mathbb{R}^n}$ □

Proof that $\big(\boldsymbol{D} - (1-\alpha)\boldsymbol{A}\big)$ *is positive-definite.* With a similar reasoning as for $\big(\boldsymbol{D}^{-1} - (1-\alpha)\boldsymbol{D}^{-1}\boldsymbol{A}\boldsymbol{D}^{-1}\big)$, we get that $\forall \boldsymbol{x} \in \mathbb{R}^n$, we have

$$\boldsymbol{x}^\top\big(\boldsymbol{D} - (1-\alpha)\boldsymbol{A}\big)\boldsymbol{x} = \alpha\sum_{u=1}^{n} d_u\boldsymbol{x}_u^2 + \frac{1-\alpha}{2}\sum_{u=1}^{n}\sum_{v=1}^{n}(\boldsymbol{x}_u - \boldsymbol{x}_v)^2\boldsymbol{A}_{uv} \tag{3.26}$$

□

This property opens the possibility to make an eigendecomposition of $\boldsymbol{D}\boldsymbol{\Pi}$ or $\boldsymbol{\Pi}\boldsymbol{D}^{-1}$, which could be used as an embedding. We note that, as used in the proof, the inverse of $\boldsymbol{\Pi}\boldsymbol{D}^{-1}$ is $\frac{1}{\alpha}(\boldsymbol{D} - (1-\alpha)\boldsymbol{A})$. Therefore, the eigenvectors of $\boldsymbol{\Pi}\boldsymbol{D}^{-1}$ are the same as the eigenvectors of $\frac{1}{\alpha}(\boldsymbol{D} - (1-\alpha)\boldsymbol{A})$, and the eigenvalues are the inverse.

We see that an embedding based on this eigendecomposition is very close from the spectral embedding, which defined in Section 3.4.2. Indeed, the spectral embedding relies on the eigendecomposition of the $\boldsymbol{D} - \boldsymbol{A}$ matrix. This sheds a new light on the spectral embedding, which is sometimes disregarded as an embedding that only considers the direct neighborhood (e.g. see [58]). On the contrary, this result shows that it is closely related to the PPR matrix, which considers by design high orders of neighborhoods.

3.8 A practical application

We teamed up with a company specialized in YouTube economics and data to extract a bipartite graph of YouTube. A **bipartite graph** is a graph in which the

nodes are split into two parts and a node from one part can only share an edge with nodes of the other part. Each part usually represents a type of real-world entity. For example, here, one part represents the YouTube users while the other part represented YouTube videos. An edge represented the fact that the user had commented the video and it is weighted by the number of comments. However, due to technical limitations, we only had the last 15 000 comments on the videos of a same YouTube channel at the time of the scrapping.

The goal was to separate the graph into **clusters**, i.e. groups of nodes densely connected to one another, and sparsely connected to the rest of the graph. The graph had many small connected components and also some series of users connected to only a few videos, themselves connected to a few users. To remove these less interesting parts that made the clustering more difficult, we kept only the k**-core** of the graph, i.e. the maximal subgraph in which every node have at least k neighbors. The values of k we have considered are 2, 5 and 10.

Then we tried two clustering methods in the graph. The first one computed a spectral embedding and then performed a k-means clustering on the result. The second one used the Louvain algorithm, which is a well-known graph clustering algorithm.

We presented the results as a small local website to help reading and browsing them. Screenshots of this website are presented in Appendix B.

Unfortunately, the manual analysis of the results revealed poor performances. We worked with researchers interested in sociology who had partially clustered the videos and who did not find a convincing matching between their work and the results. After analyzing the data, we think that the limitation to the last 15 000 comments on the videos of a same YouTube channel removed a lot of information. For example, we probably lost a lot of the occurrences when a same user who was very loyal to a channel had commented on most of the videos at the time of their publication, because those comments would have been old and therefore "erased" by new comments.

3.9 Conclusion and future work

In this chapter, we have addressed the lack of inherently interpretable embedding methods for graphs. Indeed, the usual embedding algorithms produce results from complex processes and do not align with easy-to-understand concepts. Our solution is a new embedding called PARFAITE, which relies on a projection of the Personalized PageRank (PPR) matrix into the singular subspaces, then clustering and a projection back from the singular subspaces onto the original space. This procedure makes PARFAITE the first embedding to be interpretable by design. We have shown that the results of PARFAITE are indeed very interpretable, and that this result is achieved without much loss of efficiency at tasks like link prediction. This appeals for many new research questions: Could the k-means clustering be replaced by another, more efficient

method? Could the projection into the singular spaces be replaced by other embedding methods? And if so, what interpretation could be found from the resulting embeddings? What is the best way to label the resulting dimensions to transmit the interpretability information to the user?

We have also studied the metrics that exist to evaluate the interpretability of a graph embedding. However, we found common weaknesses in the available methods, in that none consider the potential redundancy or separation of the communities that constitute the interpretation. Moreover, both consider equally the top or bottom vertices of the embedding dimensions and completely disregard the vertices after a given rank. We have tackled these weaknesses with a new metric called CISIP. This new metric compares the ordering of the vertices in the embedding with the ordering in a smoothed version of the belonging to the communities. The smoothing is a parameter of the metric and so it would be very interesting to study more functions for it, e.g. personalized version of centrality metrics like the BCI, the CCI or the Katz index. It would also be very interesting to study how these functions affect the results and what they say about the interpretability of the embedding.

Finally, we have discovered that, for undirected graphs, the matrix $\mathbf{\Pi}\boldsymbol{D}^{-1}$, which is the PPR matrix normalized by the inverse degrees, can be factorized by an eigendecomposition. This decomposition produces an embedding that is strongly related to the spectral embedding. It would be very interesting to study more in-depth the link between these embeddings, their common aspects and their differences. It would also be interesting to study if this eigendecomposition can be used as the first step of PARFAITE instead of the SVD.

Table 3.2: CISIP results for PARFAITE, HOPE and node2vec

Dataset	**PARFAITE**		**HOPE**		**node2vec**
	left	right	left	right	
No smoothing (identity function)					
Wikispeedia	**0.429**	0.414	0.339	0.318	0.255
Facebook 0	**0.333**	0.311	0.233	0.232	0.248
Facebook 107	0.403	**0.408**	0.316	0.316	0.234
Facebook 348	0.561	**0.566**	0.524	0.524	0.252
Facebook 414	0.545	**0.605**	0.540	0.540	0.283
Facebook 686	**0.527**	0.489	0.384	0.384	0.295
Facebook 1684	0.483	**0.493**	0.369	0.369	0.264
Facebook 1912	0.401	**0.425**	0.352	0.353	0.266
Facebook 3437	0.300	0.310	**0.331**	**0.331**	0.256
WikipediaFr	0.350	**0.376**	0.339	0.353	*
Smoothing with MNB					
Wikispeedia	0.431	**0.508**	0.282	0.369	0.105
Facebook 0	**0.436**	0.434	0.230	0.230	0.028
Facebook 107	**0.545**	0.515	0.384	0.384	0.161
Facebook 348	**0.582**	0.509	0.349	0.349	0.105
Facebook 414	**0.627**	0.584	0.474	0.474	0.084
Facebook 686	**0.475**	0.400	0.234	0.234	0.102
Facebook 1684	**0.539**	0.532	0.389	0.389	0.122
Facebook 1912	0.504	**0.511**	0.269	0.267	0.029
Facebook 3437	0.513	**0.524**	0.373	0.373	0.067
WikipediaFr	0.272	**0.499**	0.400	0.493	*
Smoothing with PPR					
Wikispeedia	0.449	**0.516**	0.189	0.174	0.012
Facebook 0	**0.429**	**0.429**	0.143	0.138	0.074
Facebook 107	**0.585**	0.576	0.304	0.304	0.093
Facebook 348	0.633	**0.682**	0.510	0.510	0.071
Facebook 414	0.589	**0.774**	0.531	0.531	0.283
Facebook 686	0.454	**0.550**	0.285	0.285	0.106
Facebook 1684	0.538	**0.649**	0.393	0.393	0.085
Facebook 1912	0.557	**0.633**	0.278	0.279	0.101
Facebook 3437	0.568	**0.688**	0.402	0.402	0.073
WikipediaFr	**0.123**	-0.067	0.049	-0.015	*

Table 3.3: Interpretability Score results for the SVD of the PPR matrix. The results in bold as those at least as good as the related PARFAITE results

Dataset	**SVD (left)**	**SVD (right)**
Wikispeedia	0.234	0.204
Facebook 0	**0.538**	**0.593**
Facebook 107	0.538	0.516
Facebook 348	0.923	**0.928**
Facebook 414	0.808	0.812
Facebook 686	0.690	0.696
Facebook 1684	0.792	0.768
Facebook 1912	0.589	0.604
Facebook 3437	0.270	0.463
WikipediaFr	**0.043**	0.061

Table 3.4: CISIP results for the SVD of the PPR matrix. The results in bold as those at least as good as the related PARFAITE results

Dataset	SVD (left)	SVD (right)
No smoothing (identity function)		
Wikispeedia	0.363	0.362
Facebook 0	0.333	**0.360**
Facebook 107	**0.493**	**0.464**
Facebook 348	0.520	0.525
Facebook 414	**0.545**	0.549
Facebook 686	0.438	0.440
Facebook 1684	**0.532**	**0.545**
Facebook 1912	0.377	0.392
Facebook 3437	**0.304**	**0.337**
WikipediaFr	**0.350**	**0.381**
Smoothing with MNB		
Wikispeedia	0.280	0.369
Facebook 0	0.330	0.395
Facebook 107	**0.596**	**0.571**
Facebook 348	0.404	0.412
Facebook 414	0.431	0.404
Facebook 686	0.320	0.337
Facebook 1684	0.512	0.493
Facebook 1912	0.299	0.320
Facebook 3437	0.429	0.443
WikipediaFr	0.259	**0.502**
Smoothing with PPR		
Wikispeedia	0.244	0.202
Facebook 0	0.284	0.356
Facebook 107	**0.609**	**0.604**
Facebook 348	0.449	0.485
Facebook 414	0.527	0.541
Facebook 686	0.378	0.407
Facebook 1684	0.511	0.549
Facebook 1912	0.319	0.320
Facebook 3437	0.419	0.421
WikipediaFr	**0.150**	**-0.011**

Conclusion

Decision trees are very popular models in machine learning. They are efficient to train, easy to understand and inherently interpretable. Algorithms such as EFDT exist for building and maintaining decision trees in a dynamic setting. Very few algorithms exist to fulfill this task in a fully dynamic setting, in which training examples can be both inserted and deleted. In this thesis, we have presented a new algorithm for fully dynamic decision trees. To the best of our knowledge, this is the first algorithm to be specifically designed for this task.

This algorithm relies on a new smoothness theorem for the Gini index and Gini gain. The theorem states that given a training set and a series of updates, i.e. insertions and deletions from the training set, the difference of Gini index of the training set before and after the updates is no more than 3 times the ratio of updates by the size of the training set. Moreover, for any split of the training set, the difference of Gini gain of that split between before and after the updates is not more than 13 times the ratio of updates by the size of the training set.

This algorithm also relies on a objective for fully dynamic decision trees that loosen the strict objective of maintaining a tree that would be as good as if it was trained from scratch on the updated training set. Our new objective, called ϵ-feasibility, requires the splits of the trees to have Gini gains close to the optimal Gini gain. We have introduced an algorithm called FuDyADT. This algorithm meets ϵ-feasibility and, provided that the number of labels is small in comparison with the dimension of the examples and the logarithm of the number of data, runs in time $\mathcal{O}\left(d\frac{\log^3(n)}{\varepsilon^2}\right)$. We have shown experimentally that this algorithm performs at least as good as the state-of-the-art EFDT algorithm in contexts when this comparison is relevant, while also operating in fully dynamic contexts.

Another very popular machine learning method is graph embedding. It allows transforming graph data into classical vectorial that can be used in classical machine learning algorithms. However, little work has been done so far to make embedding methods interpretable. This is important because the interpretations of methods that take vectors as input usually relies on the semantics of the values in the vectors. Therefore, a non-interpretable graph

embedding, when used as the input for another machine learning method, makes this method non-interpretable.

Therefore, we have proposed a new algorithm for inherently interpretable graph embedding. This new method relies on properties of the Personalized PageRank (PPR) matrix that make embedding methods based on it outline communities. It also relies on the property that the Singular Value Decomposition (SVD) further outline these communities. Finally, it uses a clustering to make the final embeddings align with the communities. The whole process makes our new embedding strongly interpretable by design, and we have shown that it is also interpretable in practice through experiments against two well-known graph embeddings called node2vec and HOPE.

Studying this new embedding, we found that few metrics exist for assessing the interpretability of graph embedding. Crucially, we found that the existing metrics have common weaknesses on not considering the ordering of the values in the embedding dimension, and not considering redundant interpretations. To address these weaknesses, we proposed a new metric called CISIP. The process to compute this metric is to first automatically assign each embedding dimension to a community, and then to compare the order of the values in the embedding vector with a smoothed version of the belonging of the vertices to the communities.

Finally, we also discovered that a modified version of the PPR matrix can be eigendecomposed and that this eigendecomposition is closely related to the graph spectral embedding. This finding open new embedding perspectives, as well as shedding a new light on the spectral embedding.

Our two new algorithms, as well as our other contributions, expand the range of machine learning problems that can be addressed using interpretable techniques. Moreover, they open new research questions that will expand this range further.

As the general public demands more and more for machine learning models that could be understood, audited and discussed, these new findings and the next to come will surely contribute to make machine learning less of a black box to trust blindly, and more of a real decision assistant that the user can understand and tune to get decisions or solution he is satisfied with.

Bibliography

[1] J. David Archibald. "Edward Hitchcock's Pre-Darwinian (1840) "Tree of Life"". In: *Journal of the History of Biology* 42.3 (Aug. 1, 2009), pp. 561–592. ISSN: 1573-0387. DOI: 10.1007/s10739-008-9163-y. (Visited on 10/26/2024).

[2] David Arthur and Sergei Vassilvitskii. *k-means++: The Advantages of Careful Seeding*. Technical Report 2006-13. Stanford InfoLab, June 2006.

[3] Alex Bavelas. "Communication Patterns in Task-Oriented Groups". In: *The Journal of the Acoustical Society of America* 22.6 (Nov. 1950), pp. 725–730. ISSN: 0001-4966. DOI: 10.1121/1.1906679. (Visited on 09/25/2024).

[4] Mikhail Belkin and Partha Niyogi. "Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering". In: *Advances in Neural Information Processing Systems*. Vol. 14. MIT Press, 2001. (Visited on 10/07/2024).

[5] Albert Bifet et al. "MOA: Massive Online Analysis". In: *J. Mach. Learn. Res.* 11 (2010), pp. 1601–1604.

[6] Francis Bloch, Matthew O Jackson, and Pietro Tebaldi. "Centrality measures in networks". In: *Social Choice and Welfare* 61.2 (2023), pp. 413–453.

[7] Phillip Bonacich. "Factoring and Weighting Approaches to Status Scores and Clique Identification". In: *Journal of Mathematical Sociology* (Jan. 1972). ISSN: 0022-250X. (Visited on 09/26/2024).

[8] Phillip Bonacich. "Some Unique Properties of Eigenvector Centrality". In: *Social Networks* 29.4 (Oct. 2007), pp. 555–564. ISSN: 0378-8733. DOI: 10.1016/j.socnet.2007.04.002. (Visited on 09/26/2024).

[9] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (Oct. 2001), pp. 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. (Visited on 09/24/2024).

[10] Marco Bressan, Gabriel Damay, and Mauro Sozio. "Fully-Dynamic Decision Trees". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 37.6 (6 June 26, 2023), pp. 6842–6849. ISSN: 2374-3468. DOI: 10.1609/aaai.v37i6.25838. (Visited on 10/23/2024).

[11] HongYun Cai, Vincent W. Zheng, and Kevin Chen-Chuan Chang. "A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications". In: *IEEE Transactions on Knowledge and Data Engineering* 30.9 (Sept. 2018), pp. 1616–1637. ISSN: 1558-2191. DOI: 10.1109/TKDE.2018.2807452. (Visited on 10/07/2024).

[12] Gürol Canbek et al. "Binary classification performance measures/metrics: A comprehensive visualized roadmap to gain new insights". In: *2017 International Conference on Computer Science and Engineering (UBMK)*. 2017, pp. 821–826. DOI: 10.1109/UBMK.2017.8093539.

[13] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: ACM, 2016, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785. URL: http://doi.acm.org/10.1145/2939672.2939785.

[14] Vinícius G. Costa and Carlos E. Pedreira. "Recent Advances in Decision Trees: An Updated Survey". In: *Artificial Intelligence Review* 56.5 (May 1, 2023), pp. 4765–4800. ISSN: 1573-7462. DOI: 10.1007/s10462-022-10275-5. (Visited on 10/25/2024).

[15] Constantinos Daskalakis, Christos Tzamos, and Manolis Zampetakis. "A converse to Banach's fixed point theorem and its CLS-completeness". In: *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2018. New York, NY, USA: Association for Computing Machinery, 2018. DOI: 10.1145/3188745.3188968.

[16] Pedro Domingos and Geoff Hulten. "Mining High-Speed Data Streams". In: *Proc. of ACM KDD*. Boston, Massachusetts, USA, 2000, pp. 71–80. DOI: 10.1145/347090.347107.

[17] Carl Eckart and Gale Young. "The Approximation of One Matrix by Another of Lower Rank". In: *Psychometrika* 1.3 (Sept. 1, 1936), pp. 211–218. ISSN: 1860-0980. DOI: 10.1007/BF02288367. (Visited on 10/23/2024).

[18] Elior Cohen. *node2vec 0.4.6*. URL: https://pypi.org/project/node2vec/.

[19] Santo Fortunato. "Community Detection in Graphs". In: *Physics Reports* 486.3 (Feb. 1, 2010), pp. 75–174. ISSN: 0370-1573. DOI: 10.1016/j.physrep.2009.11.002. (Visited on 10/24/2024).

[20] Linton C. Freeman. "A Set of Measures of Centrality Based on Betweenness". In: *Sociometry* 40.1 (1977), pp. 35–41. ISSN: 0038-0431. DOI: 10.2307/3033543. JSTOR: 3033543. (Visited on 09/26/2024).

[21] João Gama, Ricardo Rocha, and Pedro Medas. "Accurate Decision Trees for Mining High-Speed Data Streams". In: *Proc. of ACM KDD*. 2003, pp. 523–528. DOI: 10.1145/956750.956813.

[22] Johannes Gehrke et al. “BOAT—Optimistic Decision Tree Construction”. In: *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’99. New York, NY, USA: Association for Computing Machinery, June 1, 1999, pp. 169–180. ISBN: 978-1-58113-084-3. DOI: 10.1145/304182.304197. (Visited on 10/02/2024).

[23] Antonia Gogoglou, C. Bayan Bruss, and Keegan E. Hines. “On the Interpretability and Evaluation of Graph Representation Learning”. In: *NeurIPS workshop on Graph Representation Learning*. 2019.

[24] Palash Goyal and Emilio Ferrara. “Graph Embedding Techniques, Applications, and Performance: A Survey”. In: *Knowledge-Based Systems* 151 (July 1, 2018), pp. 78–94. ISSN: 0950-7051. DOI: 10.1016/j.knosys.2018.03.022. (Visited on 10/07/2024).

[25] Aditya Grover and Jure Leskovec. “Node2vec: Scalable Feature Learning for Networks”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. New York, NY, USA: Association for Computing Machinery, Aug. 13, 2016, pp. 855–864. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939754. (Visited on 03/15/2024).

[26] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[27] Douglas M. Hawkins. “The Problem of Overfitting”. In: *Journal of Chemical Information and Computer Sciences* 44.1 (Jan. 1, 2004), pp. 1–12. ISSN: 0095-2338. DOI: 10.1021/ci0342472. (Visited on 10/24/2024).

[28] Alexandre Hollocou, Thomas Bonald, and Marc Lelarge. “Multiple Local Community Detection”. In: *SIGMETRICS Perform. Eval. Rev.* 45.3 (2018), pp. 76–83. ISSN: 0163-5999. DOI: 10.1145/3199524.3199537.

[29] Geoff Hulten, Laurie Spencer, and Pedro Domingos. “Mining Time-Changing Data Streams”. In: *Proc. of ACM KDD*. 2001, pp. 97–106. DOI: 10.1145/502512.502529.

[30] Sangheum Hwang, Hyeon Gyu Yeo, and Jung-Sik Hong. “A New Splitting Criterion for Better Interpretable Trees”. In: *IEEE Access* 8 (2020), pp. 62762–62774. DOI: 10.1109/ACCESS.2020.2985255.

[31] Ruoming Jin and Gagan Agrawal. “Efficient Decision Tree Construction on Streaming Data”. In: *Proc. of ACM KDD*. 2003, pp. 571–576. DOI: 10.1145/956750.956821.

[32] Leo Katz. “A New Status Index Derived from Sociometric Analysis”. In: *Psychometrika* 18.1 (Mar. 1953), pp. 39–43. ISSN: 1860-0980. DOI: 10.1007/BF02289026. (Visited on 09/25/2024).

[33] Shima Khoshraftar, Sedigheh Mahdavi, and Aijun An. "Centrality-based Interpretability Measures for Graph Embeddings". In: *2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA)*. 2021, pp. 1–10. DOI: 10.1109/DSAA53316.2021.9564221. (Visited on 03/14/2024).

[34] Isabel M. Kloumann and Jon M. Kleinberg. "Community Membership Identification from Small Seed Sets". In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2014, pp. 1366–1375. ISBN: 9781450329569. DOI: 10.1145/2623330.2623621.

[35] K. Konstantinides, B. Natarajan, and G.S. Yovanof. "Noise estimation and filtering using block-based singular value decomposition". In: *IEEE Transactions on Image Processing* 6.3 (1997), pp. 479–483. DOI: 10.1109/83.557359.

[36] Seunghyun Lee and Byung Cheol Song. "Interpretable Embedding Procedure Knowledge Transfer via Stacked Principal Component Analysis and Graph Neural Network". en. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.9 (2021), pp. 8297–8305. ISSN: 2374-3468. DOI: 10.1609/aaai.v35i9.17009. (Visited on 03/19/2024).

[37] Wei-Yin Loh. "Fifty Years of Classification and Regression Trees". In: *International Statistical Review* 82.3 (2014), pp. 329–348. DOI: https://doi.org/10.1111/insr.12016.

[38] J MacQueen. "Some methods for classification and analysis of multivariate observations". In: *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability/University of California Press*. 1967.

[39] Fragkiskos D. Malliaros and Michalis Vazirgiannis. "Clustering and Community Detection in Directed Networks: A Survey". In: *Physics Reports*. Clustering and Community Detection in Directed Networks: A Survey 533.4 (Dec. 30, 2013), pp. 95–142. ISSN: 0370-1573. DOI: 10.1016/j.physrep.2013.08.002. (Visited on 10/24/2024).

[40] Chaitanya Manapragada, Geoffrey I. Webb, and Mahsa Salehi. "Extremely Fast Decision Tree". In: *Proc. of ACM KDD*. 2018, pp. 1953–1962. DOI: 10.1145/3219819.3220005.

[41] Chaitanya Manapragada et al. "An eager splitting strategy for online decision trees in ensembles". In: *Data Mining and Knowledge Discovery* 36.2 (2022), pp. 566–619. DOI: 10.1007/s10618-021-00816-x.

[42] Robert Messenger and Lewis Mandell. "A Modal Search Technique for Predictive Nominal Scale Multivariate Analysis". In: *Journal of the American Statistical Association* 67.340 (1972), pp. 768–772. DOI: 10.1080/01621459.1972.10481290.

[43] James N. Morgan and John A. Sonquist. "Problems in the Analysis of Survey Data, and a Proposal". In: *Journal of the American Statistical Association* 58.302 (1963), pp. 415–434. DOI: 10.1080/01621459.1963.10500855.

[44] Geraldin Nanfack, Paul Temple, and Benoit Frenay. "Constraint Enforcement on Decision Trees: A Survey". In: *ACM Comput. Surv.* 54.10s (Sept. 2022). DOI: 10.1145/3506734.

[45] M. E. J. Newman and M. Girvan. "Finding and evaluating community structure in networks". In: *Phys. Rev. E* 69 (2 Feb. 2004), p. 026113. DOI: 10.1103/PhysRevE.69.026113. URL: https://link.aps.org/doi/10.1103/PhysRevE.69.026113.

[46] Mingdong Ou et al. "Asymmetric Transitivity Preserving Graph Embedding". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 1105–1114. ISBN: 9781450342322. DOI: 10.1145/2939672.2939751.

[47] Lawrence Page et al. *The pagerank citation ranking: Bringing order to the web.* Tech. rep. Stanford InfoLab, 1999.

[48] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. "DeepWalk: Online Learning of Social Representations". In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '14. New York, NY, USA: Association for Computing Machinery, Aug. 24, 2014, pp. 701–710. ISBN: 978-1-4503-2956-9. DOI: 10.1145/2623330.2623732. (Visited on 10/08/2024).

[49] Priyanka and Dharmender Kumar. "Decision tree classifier: a detailed survey". In: *International Journal of Information and Decision Sciences* 12.3 (2020), pp. 246–269. DOI: 10.1504/IJIDS.2020.108141.

[50] Ryan A. Rossi and Nesreen K. Ahmed. "The Network Data Repository with Interactive Graph Analytics and Visualization". In: *AAAI*. 2015. URL: https://networkrepository.com.

[51] Leszek Rutkowski et al. "Decision Trees for Mining Data Streams Based on the McDiarmid's Bound". In: *IEEE Transactions on Knowledge and Data Engineering* 25.6 (2013), pp. 1272–1279. DOI: 10.1109/TKDE.2012.66.

[52] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. USA: Cambridge University Press, May 2014. ISBN: 978-1-107-05713-5.

[53] Sebastiano Vigna. "A Weighted Correlation Index for Rankings with Ties". In: *Proceedings of the 24th International Conference on World Wide Web*. 2015, pp. 1166–1176. ISBN: 978-1-4503-3469-3. DOI: 10.1145/2736277.2741088. (Visited on 03/08/2024).

[54] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[55] *Weather Type Classification*. URL: https://www.kaggle.com/datasets/nikhil7280/weather-type-classification (visited on 10/24/2024).

[56] Jiaxin Wu and Chong-Wah Ngo. "Interpretable Embedding for Ad-Hoc Video Search". In: *Proceedings of the 28th ACM International Conference on Multimedia*. 2020, pp. 3357–3366. ISBN: 978-1-4503-7988-5. DOI: 10.1145/3394171.3413916. (Visited on 03/05/2024).

[57] Jaewon Yang and Jure Leskovec. "Defining and Evaluating Network Communities Based on Ground-Truth". In: *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*. 2012. ISBN: 9781450315463. DOI: 10.1145/2350190.2350193.

[58] Renchi Yang et al. "Homogeneous Network Embedding for Massive Graphs via Reweighted Personalized PageRank". In: *Proceedings of the VLDB Endowment* 13.5 (Jan. 1, 2020), pp. 670–683. ISSN: 2150-8097. DOI: 10.14778/3377369.3377376. (Visited on 04/11/2024).

[59] Yinglong Zhang et al. "Robust Hierarchical Overlapping Community Detection With Personalized PageRank". In: *IEEE Access* 8 (2020), pp. 102867–102882. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2998860. (Visited on 04/14/2024).

Appendix A

Additional results for PARFAITE experiments

This appendix contains the results of the experiments on PARFAITE, HOPE and node2vec, measured with the Betweenness and Closeness Centrality Indices.

Table A.1: Betweenness Centrality Index results for PARFAITE and node2vec

Dataset	**PARFAITE_L**	**PARFAITE_R**	**node2vec**
Wikispeedia	0.585	0.566	**0.650**
Facebook 0	**0.635**	0.456	0.630
Facebook 107	0.118	0.127	**0.299**
Facebook 348	0.405	0.400	**0.450**
Facebook 414	0.361	**0.484**	0.429
Facebook 686	0.527	**0.686**	0.444
Facebook 1684	**0.415**	0.292	0.329
Facebook 1912	0.604	**0.701**	0.458
Facebook 3437	0.400	**0.435**	0.381

Table A.2: Closeness Centrality Index results for PARFAITE and node2vec

Dataset	**PARFAITE_L**	**PARFAITE_R**	**node2vec**
Wikispeedia	0.711	0.746	**0.767**
Facebook 0	**0.418**	0.413	0.341
Facebook 107	0.345	0.314	**0.404**
Facebook 348	0.430	**0.459**	**0.458**
Facebook 414	0.461	**0.500**	0.463
Facebook 686	0.350	**0.359**	0.356
Facebook 1684	0.344	0.350	**0.463**
Facebook 1912	0.465	**0.496**	0.464
Facebook 3437	0.414	**0.462**	0.400

Table A.3: BCI results for the SVD of the PPR matrix. The results in bold as those at least as good as the related PARFAITE results

Dataset	**SVD (left)**	**SVD (right)**
Wikispeedia	**0.645**	**0.684**
Facebook 0	**0.686**	**0.564**
Facebook 107	**0.197**	**0.213**
Facebook 348	**0.445**	**0.585**
Facebook 414	**0.435**	0.480
Facebook 686	**0.619**	0.607
Facebook 1684	0.384	**0.396**
Facebook 1912	0.501	0.532
Facebook 3437	0.351	**0.460**

Table A.4: CCI results for the SVD of the PPR matrix. The results in bold as those at least as good as the related PARFAITE results

Dataset	**SVD (left)**	**SVD (right)**
Wikispeedia	**0.757**	**0.757**
Facebook 0	0.342	0.385
Facebook 107	0.337	**0.331**
Facebook 348	**0.447**	**0.469**
Facebook 414	**0.481**	0.489
Facebook 686	0.328	0.350
Facebook 1684	**0.384**	**0.395**
Facebook 1912	0.415	0.423
Facebook 3437	0.309	0.384

Appendix B

Screenshots of the website for clusters presentation

This appendix shows screenshots of the website presented in section 3.8, and made to present and help to navigate the results of the clustering.

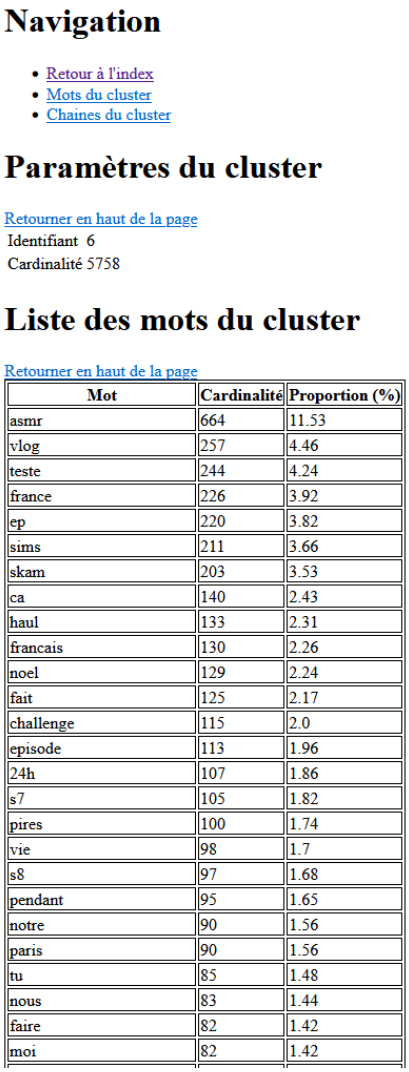
Paramètres de clustering

Algorithme	KMeans
Nombre de clusters	100
Auto-commentaires supprimés ?	Oui

Liste des clusters

ID_cluster	Nombre de videos	Mots les plus présents	Chaines les plus présentes
0	11934	fortnite, ft	GameMixTreize, bySankah Officiel
1	13853	fortnite, minecraft	L'étoile Noire, Kikle
2	17694	pokemon, legends	FitzAdri, Brice Gaming Z
3	19399	warzone, best	John Joodan, Alderiate VOD
4	5195	minecraft, roblox	MISS QUEEN, PRIME FURIOUS - RaVeNeiD GaMiNG
5	8760	one, critique	Psyhodelik, La chaîne du geek
6	5758	asmr, vlog	francetv slash, Duo
7	6221	fifa, fut	SerialShaj, Les Twins Fifa
8	3973	pokemon, fortnite	OcariKnights, Marcus27500
9	7886	ps5, gameplay	Jeux Vidéo Magazine, Julien Chièze
10	5025	canal, bitcoin	Brut, Le Trone - Analyse Bitcoin Crypto FR
11	4852	test, pro	Monsieur GRrr [FR], Avis Express - Testeur High-Tech
12	3879	vlog, haul	MarionCameleon, Pikiti bouquine
13	4014	bitcoin, afp	Europe 1, Investigations et Enquêtes
14	3517	test, review	Virtual Devil, Philippe Demerliac
15	2534	exclu, booba	nan, Catch Club
16	5040	covid, contre	RT France, Europe 1
17	1008	givdt, vs	sagiiovanna, Bastos
18	2835	rap, prod	InstruRap, Jeux Vidéo Magazine
19	2069	resume, psg	RMC Sport, beIN SPORTS France
20	540	yo, jovenel	UNISSONS NOUS, AZ 237
21	2871	psg, om	RMC Sport, PSG - Paris Saint-Germain
22	1686	urbex, moto	GLB, Marty Japan
23	2817	vlog, haul	LOdoesmakeup, JuJuBaby
24	2284	maeva, lpdla8	sam zirah, Snapchat RED
25	2755	contre, sanitaire	RT France, Europe 1
26	1990	vlog, haul	Marion Botanical, MYF Move Your Fit
27	1735	rap, by	InstruRap, Nakryum
28	986	psg, messi	Top Mercato, Foot Mercato
29	612	dragon, ball	Saikyo Devin, FlorianOnAir
30	686	arrivage, nana	Rim Channel, Mes secrets de nana
31	741	live, vlog	GOTTA-11, GLB
32	1891	arrivage, action	Mes secrets de nana, Mama Jumeauxx

Figure B.1: Screenshot of the main page of the website.



Navigation

- Retour à l'index
- Mots du cluster
- Chaines du cluster

Paramètres du cluster

Retourner en haut de la page
Identifiant 6
Cardinalité 5758

Liste des mots du cluster

Retourner en haut de la page

Mot	Cardinalité	Proportion (%)
asmr	664	11.53
vlog	257	4.46
teste	244	4.24
france	226	3.92
ep	220	3.82
sims	211	3.66
skam	203	3.53
ca	140	2.43
haul	133	2.31
francais	130	2.26
noel	129	2.24
fait	125	2.17
challenge	115	2.0
episode	113	1.96
24h	107	1.86
s7	105	1.82
pires	100	1.74
vie	98	1.7
s8	97	1.68
pendant	95	1.65
notre	90	1.56
paris	90	1.56
tu	85	1.48
nous	83	1.44
faire	82	1.42
moi	82	1.42

Figure B.2: Screenshot of the page of the website that presents main information about a cluster.

Liste des chaines du cluster

Retourner en haut de la page

Nom	Identifiant	# Total de vidéos	# de vidéos dans le cluster	% Vidéos du cluster	% Vidéos de la chaine
francetv slash	UCguxKRCUwOq97ouGaBMtrQQ	249	206	3.58	82.73
Duo	UCTe0xGTK0guuVzjQ5YGEo8Q	434	162	2.81	37.33
Reve Compulsif	UCb0Da06kxBS8KEZdQcy1xzg	160	155	2.69	96.88
Tina ASMR	UCfe-YBaja0apgXcuozSuPTw	170	130	2.26	76.47
Moguiz	UCJcpFTLj9Rlez6R8sgx30Ng	215	129	2.24	60.0
LoryLyn	UCwYTXi52JfJd3XZzOtmJSWA	130	110	1.91	84.62
Mamsmyrcus	UC0k1_SilIbDQMt5CTrFqchg	124	106	1.84	85.48
Fabian CR	UCdN2J5p8r4hHeqC7AirRzcw	103	99	1.72	96.12
Zoey Vidéos	UC7IBETC7KeyWaJEFTUKhw_w	109	93	1.62	85.32
Wonder Hook	UCDsqF-kL2i47_OQIZ8Pu1Dw	127	88	1.53	69.29
TibouTwo	UCT26afx7_Az_mgYz5RT_8nQ	284	76	1.32	26.76
Tiboudouboudou	UC2TndVNTUc15LFYtWU5Cgbg	166	75	1.3	45.18
Les Mots de l'Imaginaire	UC4cbdC-gY8zPaBJFbtpxYTg	112	73	1.27	65.18
Rendez-vous ASMR	UCBd1I80D5VkUhsnhLCFV0ug	83	71	1.23	85.54

Figure B.3: Screenshot of the page of the website that presents the channels in a cluster.

Videos de la chaine francetv slash (UCguxKRCUwOq97ouGaBMtrQQ)

Retourner en haut de la page Retourner à la liste des chaines

Nom	Identifiant	Date d'ajout
GIRLSQUAD - Épisode 2 - C'est[...]	Znt3XI1Xerw	2021-07-30 18:00:12
SKAM FRANCE EP.10 S8 : Samed[...]	6hna8Tzrs3s	2021-07-10 00:49:02
SKAM FRANCE EP.10 S8 : Vendre[...]	gPNzSBGU-5Q	2021-07-09 22:46:01
SKAM FRANCE EP.10 S8 : Jeudi [...]	vezLkwL7zaE	2021-07-08 22:45:01
SKAM FRANCE EP.10 S8 : Mercre[...]	VemDRtyuScg	2021-07-07 23:34:02
SKAM FRANCE EP.10 S8 : Mercre[...]	vuqv2vPi-Dg	2021-07-07 20:32:03
SKAM FRANCE EP.10 S8 : Mardi [...]	SAhf3x0khL8	2021-07-06 10:14:07
SKAM FRANCE EP.10 S8 : Lundi [...]	duf5dI8h9Zg	2021-07-05 18:45:02
SKAM FRANCE EP.9 S8 : Vendred[...]	OMCAIyJJqBY	2021-07-02 17:28:06
SKAM FRANCE EP.9 S8 : Vendred[...]	3fSaCyF7WOU	2021-07-02 16:02:06
SKAM FRANCE EP.9 S8 : Jeudi 1[...]	27KE-_5nIHk	2021-07-01 18:47:05
SKAM FRANCE EP.9 S8 : Mercred[...]	i9hrz0yR4bk	2021-06-30 19:14:03
SKAM FRANCE EP.9 S8 : Mardi 1[...]	aezkMtTsjoc	2021-06-29 18:07:07
SKAM FRANCE EP.9 S8 : Lundi 1[...]	XG7hTyqaDK8	2021-06-28 19:57:45
SKAM FRANCE EP.9 S8 : Lundi 1[...]	5sXmkOrHkXk	2021-06-28 10:49:19
SKAM FRANCE EP.9 S8 : Samedi [...]	mViMAT7CKWc	2021-06-26 10:20:20

Figure B.4: Screenshot of the page of the website that presents the videos of a channel in a cluster.

ECOLE DOCTORALE

Titre: Arbres de décisions dynamiques et embedding de graphes basés sur les communautés: contributions à l'apprentissage automatique interprétable

Mots clés: Exploration de graphes, détection de communautés, embedding de graph, Arbre de décision, Apprentissage automatique dynamique, IA explicable

Résumé: L'apprentissage automatique est le domaine des sciences informatiques dont le but est de créer des modèles et des solutions à partir de données sans savoir exactement les instructions qui dirigent intrinsèquement ces modèles. Ce domaine a obtenu des résultats impressionnants mais il est le sujet d'inquiétudes en raison notamment de l'impossibilité de comprendre et d'auditer les modèles qu'il produit. L'apprentissage automatique interprétable propose une solution à ces inquiétudes en créant des modèles qui sont interprétables de façon inhérante. Cette thèse contribue à l'apprentissage automatique interprétable de deux façons.
Dans un premier temps, nous étudions les arbres de décision. Il s'agit d'un groupe de méthodes d'apprentissage automatique très connu et qui est interprétable par la façon même dont il est conçu. Cependant, les données réelles sont souvent dynamiques et peu d'algorithmes existent pour maintenir un arbre de décision quand des données peuvent à la fois être ajoutées et supprimées de l'ensemble d'entrainement. Nous proposons un nouvel algorithme nommé FuDyADT pour résoudre ce problème.
Dans un second temps, nous étudions l'embedding de graphes. La technique appelée "embedding" est une technique d'apprentissage automatique très commune. Elle consiste à projeter les noeuds d'un graphe sur un espace vectoriel. Ce type de méthodes est cependant non-interprétable en général. Nous proposons un nouvel algorithme d'embedding appelé PARFAITE, qui est basé sur la factorisation de la matrice de PageRank personnalisé. Cet algorithme est conçu pour que ses résultats soient interprétables.
Nous étudions chacun de ces algorithmes sur un plan à la fois théorique et expérimental. Nous montrons que FuDyADT est au minimum comparable aux algorithmes de l'état de l'art dans les conditions habituelles, tout en étant également capable de fonctionner dans des contextes inhabituels comme dans le cas où des données sont supprimées. Quant à PARFAITE, il produit des dimensions d'embedding qui sont alignées avec les communautés du graphe, et qui sont donc interprétables.

Title: Dynamic Decision Trees and Community-based Graph Embeddings: towards Interpretable Machine Learning

Keywords: Graph mining, Community detection, Graph embedding, Decision Tree, Dynamic machine learning, Explainable AI

Abstract: Machine learning is the field of computer science that interests in building models and solutions from data without knowing exactly the set of instructions internal to these models and solutions. This field has achieved great results but is now under scrutiny for the inability to understand or audit its models among other concerns. Interpretable Machine Learning addresses these concerns by building models that are inherently interpretable. This thesis contributes to Interpretable Machine Learning in two ways.
First, we study decision trees. This is a very popular group of machine learning methods for classification problems and it is interpretable by design. However, real world data is often dynamic, but few algorithms can maintain a decision tree when data can be both inserted and deleted from the training set. We propose a new algorithm called FuDyADT to solve this problem.
Second, when data is represented as a graph, a very common machine learning technique called "embedding" consists in projecting it onto a vectorial space. This kind of method however is usually not interpretable. We propose a new embedding algorithm called PARFAITE based on the factorization of the Personalized PageRank matrix. This algorithm is designed to provide interpretable results.
We study both algorithms theoretically and experimentally. We show that FuDyADT is at least comparable to state-of-the-art algorithms in the usual setting, while also being able to handle unusual settings such as deletions of data. PARFAITE on the other hand produces embedding dimensions that align with the communities of the graph, making the embedding interpretable.

Institut Polytechnique de Paris
91120 Palaiseau, France