UNIVERSITÀ DEGLI STUDI DI MILANO

Learning on graphs: algorithms for classification and sequential decisions

Dipartimento di Matematica "Federigo Enriques"

Dottorato in Matematica e Statistica per le Scienze Computazionali

Autore: Giovanni Zappella

Relatore: Prof. Nicolò Cesa-Bianchi

Coordinatore MaSSC: Prof. Giovanni Naldi

a Laura

ai miei genitori

Ringraziamenti

Questa è la terza tesi che consegno alle segreterie di unimi (cosa non si fa pur di organizzare qualche festa!) ed ormai so per esperienza che i ringraziamenti sono sempre una sezione difficile da scrivere.
Questa volta però sento veramente il peso dato dalla fine di un percorso durato tanti anni: lunedì non tornerò in via Comelico 39, non vedrò Salvatore consegnarmi la chiave mentre mi dice "Uee", niente secondo piano, niente saluti, insomma avete capito.

Sono grato a molte persone per tutto quello che mi hanno dato in questi anni, sia sul piano umano che sul piano scientifico. Non voglio ringraziare tutti, quindi spero di non dimenticare nessuno di quelli che voglio ringraziare davvero.

Innanzitutto vorrei ringraziare Nicolò, Claudio e Fabio. Con loro ho condiviso il lavoro degli ultimi anni, giorno dopo giorno ed a loro sono grato per tutto quello che mi hanno insegnato. Spero sinceramente di poter lavorare ancora con voi uno di questi giorni. Un ulteriore grazie va a Nicolò per avermi introdotto a tutto questo, supportandomi da sin quando ero uno studente con tante idee ... tutte molto confuse.

Un grazie particolare va a Laura, che con pazienza ed amore ha letteralmente sopportato la mia attività di ricerca, mi ha incoraggiato quando le cose non funzionavano e mi è stata vicina. Ci sarebbero altri mille motivi, ma non posso davvero elencare tutto. Questo traguardo è anche tuo.

Un grazie alla mia famiglia, per tutto quello che hanno fatto per me.

Non posso certo dimenticare tutti i "giovani colleghi" che nel corso del tempo sono diventati "meno giovani" amici: chi con me ha condiviso l'ufficio come Francesco (grazie anche per tutti i consigli ... o meglio, per quelli che non riguardavano la musica), Giuseppe e Rocco. Chi ha occupato parte dell'ufficio dove lavoravo (scusa Luigi, ma questa ci stava troppo bene!). Chi con me ha condiviso le gioie ed i dolori dello studio ad unimi sin dal secondo giorno, Matteo dove sei stato il primo giorno? sarebbe suonato molto meglio!
... e tutti coloro che tra un caffè ed una pizza hanno condiviso con me la quotidianità, spesso fornendo fantastici argomenti di discussione.

Grazie a tutti.

Contents

List of Figures

List of Tables

Part I
Introduction

You must not fight too often with one enemy,
or you will teach him all your art of war.
– Napoleon Bonaparte

Chapter 1
Introduction

Machine Learning was born as a branch of AI, and has now evolved as an autonomous discipline focused on the study and creation of new algorithms that can learn from data. As explained by Mohri et al in [69]: “Machine learning can be broadly defined as computational methods using experience to improve performance or to make accurate predictions”. The basic idea is that the learner wants to generalize the knowledge extracted from its experience in order to provide good predictions for the future.

Historically, machine learning algorithms have been studied and developed to deal with examples represented by points in the feature space. In the simplest case, each of these data points is a vector of features obtained by physical measures. For example, a vegetable can be described by its weight, color, size, etc. In the supervised learning case, this feature vector carries a label such as “tomato” or “pumpkin”. A simple classification task is: given a vector of features regarding a vegetable, predict if the original vegetable is a tomato or not. Roughly speaking, a machine learning algorithm for classification uses a set of labeled points (training set) as “experiences” in order to create a model to predict the unseen and unlabeled points that will be presented to it in the future (test set).

Due to the nature of many industrial applications and their scale, nowadays in many contexts, even marginal increments of the performances can lead to significant gain, depending on point of view. This is a big incentive for researchers to create algorithms that exploit every single bit of information that can be extracted from the data. Although many practical problems can be cast in the framework described above, many others have a richer structure and the aforementioned classical methods are not appropriate for dealing with this rich set of inputs. Real-world scenarios often present complex heterogeneous objects interacting with each other and the information provided by the interactions usually play a crucial role in the structure of the problem. Developing methods that exploit the structure of the problem may also provide better intuition for domain experts, besides improving predictive performance.

In the last few decades, we witnessed the enormous growth of this "networked" data that have now become ubiquitous in our life:

- The World Wide Web is probably the biggest collection of connected heterogeneous data sources, which provides countless large scale inference problems with tremendous importance for industrial applications. Just to cite a few relevant tasks: classification of web-pages, images, video and audio files; ranking of documents and item recommendation (i.e., see [63, 19, 2]). Some of these tasks are extremely relevant for online advertisement, nowadays a multi-billion dollar industry.

- Another significant application, which is probably known to a lesser extent by the general public, that can have a huge impact on our life is *bioinformatics*: for example, a huge amount of data is coming from proteins interacting with each other and which need to be classified. Clearly it is impossible for humans to handle data at this scale manually, so machine learning plays a crucial role in this field of research.

- A major phenomenon of the last few years has been the tremendous diffusion of social networks: just to cite a piece of information, Facebook reached 1 billion of monthly active users in October 2012. Some of these social networks such as Facebook, Twitter, Google+, Linkedin are websites with the only goal of creating, growing and maintaining a network on their platforms. Other social networks are also available in thematic online services, for example: bookmarks (i.e., Delicious), music (i.e., LastFM), books (i.e., GoodReads), movies (i.e., Netflix, Flixster) and personal communications (i.e., Whatsapp) or can be inferred from news (i.e., see [4]), resources sharing (i.e., DropBox, Google Drive) and GPS and geo-located services (i.e., see [32]).
  The impact on politics, social behavior and economics is not well understood yet, but their importance has already been made evident in many contexts (i.e., see [78, 56]).
  There are a number of significant industrial applications raising interesting problems for scientists in opinion mining, community detection, graph compression, experience personalization, content diffusion and, again, item recommendation (i.e., see [40, 45, 84, 13, 70]).

All this applicative problems clearly highlight the limitation of representing data as simple points in the feature space. Moreover, many other significant problems can be easily found also in computer vision and natural language processing.

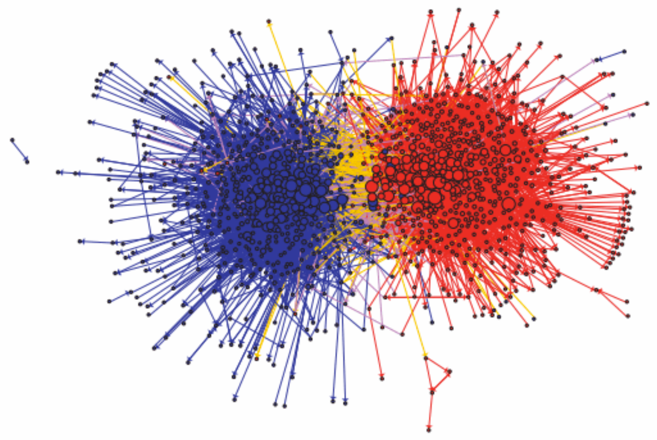


Fig. 1.1: This image contains a plot of the US political blogsphere (2004). Nodes are blogs, and they are colored according to the political “area”: blue nodes are liberal, while reds are conservative. Arcs are links between blogs, orange arcs go from liberal to conservative, and purple ones from conservative to liberal. The structure and the colors of the network make self-evident the homophily phenomenon explained below. The image is taken from [3].

In the following part, we will give a brief outline of the problems addressed in this thesis.

Node Classification. We investigate a method for labeling the nodes of a graph. Given an undirected graph and the labels of some nodes, we want to infer the labels of unlabeled nodes. In order to infer this information, we assume that well-connected nodes “are similar”, so they have the same label. This property is widespread in nature and called *homophily* (see Figure 1.1 for a visual intuition).
Let us assume that we want to classify users in a social network in order to infer the outcome of the next elections. Some users publicly declare that they will vote for party A, and some others declare that they will vote for the party B. We can use a node classification algorithm to estimate which party the remaining users are going to vote and subsequently, then the final result of the elections. The algorithm will consider well connected nodes as similar, so belonging to the same political party. In this way, if a group of friends is very densely connected, and there are just few connections between this group and the rest of the network, and one of them votes for the party

B, the algorithm will probably infer that all the people in the group will vote for the party B.

Link Classification. Since the Fifties, a lot of researchers in graph theory, sociology and political sciences have been studying signed networks. A signed network is a graph whose edges carry a binary label in order to recognize the nature (positive or negative) of interactions among two nodes. A lot of theories have been developed on the structure, and the nature of these interactions among humans. Roughly speaking, we can summarize the social theory underlying our work with the popular motto "the enemy of my enemy is my friend".
Given an undirected graph and some labels on the edges, a link classification algorithm will try to infer the labels for the unlabeled edges. Distinguishing between positive and negative interactions is very useful in many practical applications related to social networks, e-commerce and in general applications involving trust and distrust among peers.

Networked bandits. The last part of this thesis is devoted to sequential decisions in the bandit setting: at each time step the algorithm get a request for one of the nodes in the network and it has to pick one of the options among those available at the time, trying to maximize its payoff. Then it will exploit the networks structure in order to improve its model also for the other users. This a common problem: for example, at a certain time a user in a social network requests a webpage, and the algorithm has to choose among the banners currently available which one to display on the webpage for that specific user. Then the algorithm will observe if the user clicks (or not) on the banner an it will update its model. Moreover, the algorithm will exploit the same homophily property we saw for node classification and it will also update the models of the other users exploiting the topology of the network.

1.1 Outline of the thesis

The thesis is organized as follows.

In Chapter 2, we summarize the motivations and context for each of the problems we address. Moreover, we provide references and a brief explanation of the state of the art.

In Chapter 3, we present an online learning algorithm for nodes classification on trees called Shazoo and its analysis with mistake bounds.
The content of this chapter is based on the paper "See the Tree Through the

Lines: The Shazoo Algorithm", published at the 25th Conference on Neural Information Processing Systems (NIPS 2011)[79].

In Chapter 4, we present a fast algorithm for finding a Nash Equilibrium of the GTG game introduced in [42], for the special case in which the graph is a tree. The content of this chapter is based on the paper "A Scalable Multiclass Algorithm for Node Classification" presented at the MLG Workshop at the 29th International Conference on Machine Learning (ICML 2012) [83].

In Chapter 5 we present a batch experimental comparison on 7 datasets between the algorithms presented in the previous two chapters and some well-known algorithms for nodes classification.

In Chapter 6 we present a suite of active learning algorithms for binary link classification. In the last part of this chapter, we also provide an experimental comparison between the most practical of this algorithms and some spectral heuristics. The content of this chapter is based on the paper "A Linear Time Active Learning Algorithm for Link Classification", published at the 26th Conference on Neural Information Processing Systems (NIPS 2012) [24] and in a pre-print with the same title at the MLG Workshop at the 29th International Conference on Machine Learning (ICML 2012).

In Chapter 7, we provide an algorithm for sequential decisions that exploits the side-observation given by the network structure of the problem. Experiments on both synthetic and real-world datasets are presented in the last part of this chapter. The content of this chapter is based on the paper "A gang of Bandits", recently accepted at the 27th Conference on Neural Information Processing Systems (NIPS 2013) [28].

Chapter 2
Background

In this chapter we will provide a brief discussion of the problems mentioned in the previous chapter and useful references to the state of the art.
We start by introducing some terminology about the settings in order to provide a better overview for inexperienced readers.

Batch vs. Online classification. These settings differ by the way labels of the points are revealed to the learner. In the batch setting, data are split into two sets:

1. *Training Set*: the set of the labeled data then used by the algorithm to build the classifier
2. *Test Set*: the set of the unlabeled data which are typically used to evaluate the performances of the classifier

In the online setting, examples are provided one by one: at each time step the algorithm receives an example to label, and makes a prediction (estimated label). It then receives feedback from the environment providing information about the correct label, which is usually used to update the classifier. In online classification, we do not have a distinction between training set and test set.

Bandit vs. Full-information feedback. These settings differ in the information provided by the environment to the classifier regarding the correct choice. In the classical literature on multi-armed bandits, the bandit problem is usually introduced with an example: a gambler wants to maximize his payoff playing on different slot-machines (also called one-armed bandits). When played, each machine provides a reward randomly drawn from its internal unknown distribution. The purpose of the gambler is to understand which is the machine with the highest expected reward. In this way, he can maximize his return by always pulling the best arm. This maximization process is not trivial and involves finding a good trade-off between exploration, for instance when the gambler tries new slot machines, and exploitation, when he just

exploit its knowledge using its estimations about the slot machines.
In the bandit setting, the outcomes of the non-chosen slot machines are unknown. Since we assume that the gambler can play only one coin at each time-step, after he pulls an arm he only knows the outcome of the single machine he has chosen. In the full information setting, after the gambler pulls the arm he chose, he can observe the outcome of all the available slot machines, but of course he will receive only the reward of the selected slot machine. Clearly, in the full information context, it will be easier for him to understand which machine is the best one.

Active vs. Passive learning. The difference between active and passive learning deals with the choice of the labels revealed to the learner. In the active learning protocol, the training labels are obtained by querying a desired subset of examples. In our case, we will query a subset of edges selected with specific graph-theoretic methods, in this way we can improve the quality and the speed of the learning process. In the passive setting the set of examples used for the training phase are provided by the environment (i.e., selected at random from those available or by an adversary).

All these descriptions are just intended to provide a high-level distinction between the aforementioned settings, since each of them can be declined in many different ways. More details about the specific protocols are provided in the technical chapters.

2.1 Node Classification

A well known problem in the field of machine learning on graphs is the node classification problem: given a graph, the learner has to classify the nodes of the graphs using the few known labels and the topology. The common assumption is that well-connected nodes are somehow similar. In our case, similar nodes tend to have the same label.
All the algorithms presented in this part of the thesis work in the so called "transductive setting". This means that the graph structure is given in advance and does not change during the learning process. Moreover, in the online setting, predictions and feedback are provided over time but the topology of the graph does not change.
The first algorithm proposed in the node classification section (SHAZOO) works in the online setting, so it can easily be applied to the more standard train/test (or "batch") transductive setting. In this way we can also compare the performances of both the algorithms in the same experimental setting.

The second algorithm presented in this part (MUCCA) is based on a strategic game where each player is a node of the graph and interactions among the players are regulated by the topology.

2.1.1 Related work

Standard transductive classification methods, such as label propagation [9, 10, 86], work by optimizing a cost function defined on the graph, which includes the training information as labels assigned to training nodes. Although these methods perform well in practice, they are often computationally expensive, and have performance guarantees that require statistical assumptions on the selection of the training nodes.

A general approach to sidestep the above computational issues is to sparsify the graph to the largest possible extent, while retaining much of its spectral properties —see, e.g., [22, 25, 64, 75]. Inspired by [22, 25], we reduce the problem of node classification from graphs to trees by extracting suitable *spanning trees* of the graph, which can be done quickly in many cases. The advantage of performing this reduction is that node prediction is much easier on trees than on graphs. This fact has recently led to the design of very scalable algorithms with nearly optimal performance guarantees in the online transductive model, which comes with no statistical assumptions. Yet, the current results in node classification on trees are not satisfactory. The TREEOPT strategy of [22] is optimal up to constant factors, but only on *unweighted* trees. No equivalent optimality results are available for general weighted trees. To the best of our knowledge, the only other comparable result is WTA by [25], which is optimal (within log factors) only on weighted lines. In fact, WTA can still be applied to weighted trees by exploiting an idea contained in [54]. In practice, WTAtransforms a weighted tree in a weighted line with a simple depth-first visit of the tree, and after removing the duplicates, predicts using nearest neighbor on the resulting weighted line. Since linearization loses most of the structural information of the tree, this approach yields suboptimal mistake bounds. This theoretical drawback is also confirmed by empirical performance: throwing away the tree structure negatively affects the practical behavior of the algorithm on real-world weighted graphs. In Chapter 3, we present SHAZOO, an algorithm that can be viewed as a common nontrivial generalization of both TREEOPT and WTA.

In Chapter 4 we introduce another algorithm (MUCCA) based on the game-theoretic framework presented in [42]. In the same way we saw above, MUCCA works on a sparsification of the original graph: a spanning tree. In this way, it can take advantage of the special nature of the problem and it can find a Nash Equilibrium even on very large graphs. Realistic scenarios like so-

cial networks, where each entity in the networks decides only for itself, and there is little or no coordination between the entities are modeled in the idea underlying these algorithms. Some experiments for friends recommendation with an inherent idea were presented in [84].
A similar algorithm with completely different motivations has been independently presented in [85].

2.2 Link Classification

A rapidly emerging theme in the analysis of networked data is the study of signed networks. From a mathematical point of view, signed networks are graphs whose edges carry a sign representing the positive or negative nature of the relationship between the incident nodes. For example, in a protein network two proteins may interact in an excitatory or inhibitory fashion. The domain of social networks and e-commerce offers several examples of signed connections: Slashdot users can tag other users as friends or foes, Epinions users can rate other users positively or negatively, Ebay users develop trust and distrust towards sellers in the network. More generally, two individuals that are related because they rate similar products in a recommendation website may agree or disagree in their ratings.

2.2.1 Related work

The availability of signed networks has stimulated the design of link classification algorithms, primarily in the domain of social networks. Early studies of signed social networks are from the Fifties (i.e., see [51] and [18]) model dislike and distrust relationships among individuals as (signed) weighted edges in a graph. The conceptual underpinning is provided by the theory of *social balance*, formulated as a way to understand the structure of conflicts in a network of individuals whose mutual relationships can be classified as friendship or hostility [52]. The advent of online social networks has revamped the interest in these theories, and spurred a significant amount of recent work —see, e.g., [49, 60, 61, 31, 44, 26], and references therein.

Many heuristics for link classification in social networks are based on a form of social balance summarized by the motto “the enemy of my enemy is my friend”. This is equivalent to saying that the signs on the edges of a social graph tend to be consistent with some two-clustering of the nodes. By consistency we mean the following: the nodes of the graph can be partitioned into two sets (the two clusters) in such a way that edges connecting nodes from the same set are positive, and edges connecting nodes from different sets are negative. Although two-clustering heuristics do not require strict consistency to work, this is admittedly a rather strong inductive bias. Despite that, social network theorists and practitioners found this to be a reasonable bias in many social contexts, and recent experiments with online social networks reported a good predictive power for algorithms based on the two-clustering assumption [60, 62, 61, 31]. Finally, this assumption is also fairly convenient from the viewpoint of algorithmic design.

In the case of undirected signed graphs $G = (V, E)$, the best performing heuristics exploiting the two-clustering bias are based on spectral decompositions of the signed adjacency matrix. Noticeably, these heuristics run in time $\Omega\big(|V|^2\big)$, and often require a similar amount of memory storage even on sparse networks, which makes them impractical on large graphs.

In order to obtain scalable algorithms with formal performance guarantees, we focus on the active learning protocol, where training labels are obtained by querying a desired subset of edges. Since the allocation of queries can match the graph topology, a broad range of graph-theoretic techniques can be applied to the analysis of active learning algorithms. Differently from [26], labels are generated by a simple stochastic model since in many practical applications the adversarial labeling may have an overkilling effect. At the end of the section we report on the results of our experiments on medium-sized synthetic and real-world datasets, where a simple and fast algorithm suggested by our theoretical findings is compared against the best performing spectral heuristics based on the same inductive bias.

2.3 Networks of Bandits

The ability of a website to present personalized content recommendations is playing an increasingly critical role in achieving user satisfaction. Because of the appearance of new content, and due to the ever-changing nature of content popularity, modern approaches to content recommendation are strongly adaptive, and attempt to match as closely as possible users' interests by learning good mappings between available content and users. These mappings are based on "contexts", a set of feature vectors that, typically, are extracted from both contents and users. The need to focus on content that raises the user interest and, simultaneously, the need of exploring new content in order to globally improve the user experience creates an exploration-exploitation dilemma, which is commonly formalized as a multi-armed bandit problem. Indeed, contextual bandits have become a reference model for the study of adaptive techniques in recommender systems (e.g, [12, 17, 63] and references therein).

In many cases, however, the users targeted by a recommender system form a social network. The network structure provides an important additional source of information, revealing potential affinities between pairs of users. Being able to exploit such affinities could lead to a dramatic increase in the quality of the recommendations. This is because the knowledge gathered about the interests of a given user may be exploited to improve the recommendation to the user's friends.

In Chapter 7, we propose an algorithmic approach to networked contextual bandits which is provably able to leverage user similarities represented as a graph. Our approach consists in running an instance of a contextual bandit algorithm at each network node. These instances are allowed to communicate during the learning process, sharing contexts and user feedbacks. Under the modeling assumption that user similarities are correctly reflected by the network structure, interactions allow to effectively speed up the learning process that takes place at each node. This mechanism is implemented by running instances of a linear contextual bandit algorithm in a specific reproducing kernel Hilbert space (RKHS). The underlying kernel, previously used for solving online multitask classification problems (e.g., [20]), is defined in terms of the Laplacian matrix of the graph.

The Laplacian matrix provides the information we rely upon to share user feedbacks from one node to the others, according to the network structure. Since the Laplacian kernel is linear, the implementation in kernel space is straightforward. Furthermore, the existing performance guarantees for the specific bandit algorithm we use, can be directly lifted to the RKHS, and expressed in terms of spectral properties of the user network.

The principled approach described above may have some possible drawbacks for the practical usage: scalability and noise in the network. For this reason, we present two variants of the proposed algorithm where it is combined with nodes clustering. In the last part of Chapter 7, we compare these algorithms with the state of the art running experiments on a number synthetic datasets and two real-world datasets: one extracted from the social bookmarking web service Delicious, and the other one from the music streaming platform Last.fm.

2.3.1 Related work

The benefit of using social relationships in order to improve the quality of recommendations is recognized in the literature of content recommender systems —see e.g., [12, 50, 73] and the survey [6]. Linear models for contextual bandits were introduced in [7]. Their application to personalized content recommendation was pioneered in [63], where the LinUCB algorithm was introduced. An analysis of a two-stages variant of LinUCB was given in the subsequent work [33].

To the best of our knowledge, this is the first work that combines contextual bandits with the social graph information. After a pre-print of this work has been made available [29], two papers about multi-armed bandits with social connections have been published:

- the Mixing Bandits algorithm presented in [16] is a heuristic heavy tailored on the cold-start problem for recommendation systems. The authors do not provide any sort of theoretical guarantee on the performances but, on the other side, a limited feedback diffusion allows the algorithm to scale on large scale problems.
- in [76] an extension of UCB1[8] to the networked case is presented and the authors provide regret bounds for the presented algorithm.

Please note that both the aforementioned algorithms are non-contextual multi-armed bandits: they do not provide the possibility to integrate information about the actions (recommended items) in the model. Another non-contextual model of bandit algorithms running on the nodes of a graph was studied in [58]. In that work, only one node reveals its payoffs, and the statistical information acquired by this node over time is spread across the entire network following the graphical structure. The main result shows that the information flow rate is sufficient to control regret at each node of the network. Other works, such as [5, 74], consider contextual bandits assuming metric or probabilistic dependencies on the product space of contexts and actions. A different viewpoint, where each action reveals information about other actions' payoffs, is studied in [17, 65], although without the context provided by feature vectors. More recently, a new model of distributed non-contextual bandit algorithm has been presented in [77], where the number of communications among the nodes is limited, and all the nodes in the network have the same best action.

Part II
Node Classification

A friend to all is a friend to none.
– Aristotle

Chapter 3
Online Node Classification: SHAZOO

In this chapter, we consider a nontrivial extension of TreeOpt and wta to the case of weighted trees. The weights on the edges are usually very helpful for predictions, and they allow to embody side information about the nodes. Moreover, the tree structure is much richer of the line structure and this gives us an advantage that our algorithm exploits to improve the quality of the predictions.
The content of this chapter is a joint work with Nicolò Cesa-Bianchi, Claudio Gentile and Fabio Vitale.

3.1 Problem setup

Let $T = (V, E, W)$ be an undirected and weighted tree with $|V| = n$ nodes, positive edge weights $W_{i,j} > 0$ for $(i, j) \in E$, and $W_{i,j} = 0$ for $(i, j) \notin E$. A binary labeling of T is any assignment $\boldsymbol{y} = (y_1, \dots, y_n) \in \{-1, +1\}^n$ of binary labels to its nodes. We use $(T, \boldsymbol{y})$ to denote the resulting labeled weighted tree. The online learning protocol for predicting $(T, \boldsymbol{y})$ is defined as follows. The learner is given T while $\boldsymbol{y}$ is kept hidden. The nodes of T are presented to the learner one by one, according to an unknown and arbitrary permutation $i_1, \dots, i_n$ of V. At each time step $t = 1, \dots, n$ node i_t is presented and the learner must issue a prediction $\widehat{y}_{i_t} \in \{-1, +1\}$ for the label y_{i_t}. Then y_{i_t} is revealed and the learner knows whether a mistake occurred. The learner's goal is to minimize the total number of prediction mistakes.

Following previous works [55, 54, 22, 25, 27], we measure the regularity of a labeling $\boldsymbol{y}$ of T in terms of ϕ-edges, where a ϕ-edge for $(T, \boldsymbol{y})$ is any $(i, j) \in E$ such that $y_i \neq y_j$. The overall amount of irregularity in a labeled tree $(T, \boldsymbol{y})$ is the **weighted cutsize** $\Phi^W = \sum_{(i,j) \in E^\phi} W_{i,j}$, where $E^\phi \subseteq E$ is the subset of ϕ-edges in the tree. We use the weighted cutsize as our learning bias, that is, we want to design algorithms whose predictive performance scales with Φ^W. Unlike the ϕ-edge count $\Phi = |E^\phi|$, which is a good measure of regularity for unweighted graphs, the weighted cutsize takes the edge weight $W_{i,j}$ into account[1] when measuring the irregularity of a ϕ-edge (i, j). In the sequel, when we measure the distance between any pair of nodes i and j on the input tree T we always use the resistance distance metric d, that is, $d(i, j) = \sum_{(r,s)\in\pi(i,j)} \frac{1}{W_{r,s}}$, where $\pi(i, j)$ is the unique path connecting i to j.

3.2 A lower bound for weighted trees

In this section we show that the weighted cutsize can be used as a lower bound on the number of online mistakes made by any algorithm on any tree. In order to do so (and unlike previous papers on this specific subject —see, e.g., [25]), we need to introduce a more refined notion of adversarial "budget". Given $T = (V, E, W)$, let $\xi(M)$ be the maximum number of edges of T such that the sum of their weights does not exceed M, $\xi(M) = \max\left\{|E'| \,:\, E' \subseteq E,\ \sum_{(i,j)\in E'} w_{i,j} \le M\right\}$. We have the following simple lower bound

Theorem 3.1. *For any weighted tree $T = (V, E, W)$ there exists a randomized label assignment to V such that any algorithm can be forced to make at least $\xi(M)/2$ online mistakes in expectation, while $\Phi^W \le M$.*

Proof. The proof of this theorem is given in Appendix A.1

Specializing [25, Theorem 1] to trees gives the lower bound $K/2$ under the constraint $\Phi \le K \le |V|$. The main difference between the two bounds is the measure of label regularity being used: Whereas Theorem 3.1 uses Φ^W, which depends on the weights, [25, Theorem 1] uses the weight-independent quantity Φ. This dependence of the lower bound on the edge weights is consistent with our learning bias, stating that a heavy ϕ-edge violates the bias more than a light one. Since ξ is nondecreasing, the lower bound implies a number of mistakes of at least $\xi(\Phi^W)/2$. Note that $\xi(\Phi^W) \ge \Phi$ for any labeled tree $(T, \boldsymbol{y})$. Hence, whereas a constraint K on Φ implies forcing at least $K/2$ mistakes, a constraint M on Φ^W allows the adversary to force a potentially larger number of mistakes.

In the next section we describe an algorithm whose mistake bound nearly matches the above lower bound on any weighted tree when using $\xi(\Phi^W)$ as the measure of label regularity.

[1] The weight value $W_{i,j}$ typically encodes the strength of the connection (i, j). In fact, when the nodes of a graph host more information than just binary labels, e.g., a vector of feature values, then a reasonable choice is to set $W_{i,j}$ to be some (decreasing) function of the distance between the feature vectors sitting at the two nodes i and j . See also Remark 3.3.

3.3 The Shazoo algorithm

In this section we introduce the SHAZOO algorithm, and relate it to previously proposed methods for online prediction on unweighted trees (TREEOPT from [22]) and weighted line graphs (WTA from [25]). In fact, SHAZOO is optimal on any weighted tree, and reduces to TREEOPT on unweighted trees and to WTA on weighted line graphs. Since TREEOPT and WTA are optimal on *any* unweighted tree and *any* weighted line graph, respectively, SHAZOO necessarily contains elements of both of these algorithms.

In order to understand our algorithm, we now define some relevant structures of the input tree T. See Figure 3.1 (left) for an example. These structures evolve over time according to the set of observed labels. First, we call **revealed** a node whose label has already been observed by the online learner; otherwise, a node is **unrevealed**. A **fork** is any unrevealed node connected to at least three different revealed nodes by edge-disjoint paths. A **hinge node** is either a revealed node or a fork. A **hinge tree** is any component of the forest obtained by removing from T all *edges* incident to hinge nodes; hence any fork or labeled node forms a 1-node hinge tree. When a hinge tree H contains only one hinge node, a **connection node** for H is the node contained in H. In all other cases, we call a connection node for H any node outside H which is adjacent to a node in H. A **connection fork** is a connection node which is also a fork. Finally, a **hinge line** is any path connecting two hinge nodes such that no internal node is a hinge node.

Given an unrevealed node i and a label value $y \in \{-1, +1\}$, the **cut function** $\text{cut}(i, y)$ is the value of the minimum weighted cutsize of T over all labellings $\boldsymbol{y} \in \{-1, +1\}^n$ consistent with the labels seen so far and such that $y_i = y$. Define $\Delta(i) = \text{cut}(i, -1) - \text{cut}(i, +1)$ if i is unrevealed, and $\Delta(i) = y_i$, otherwise. The algorithm's pseudocode is given in Algorithm 1. At time t, in order to predict the label y_{i_t} of node i_t, SHAZOO calculates $\Delta(i)$ for all connection nodes i of $H(i_t)$, where $H(i_t)$ is the hinge tree containing i_t. Then the algorithm predicts y_{i_t} using the label of the connection node i of $H(i_t)$ which is closest to i_t and such that $\Delta(i) \neq 0$ (recall from Section 3.1 that all distances/lengths are measured using the resistance metric). Ties are broken arbitrarily. If $\Delta(i) = 0$ for all connection nodes i in $H(i_t)$ then SHAZOO predicts a default value (-1 in the pseudocode).

If i_t is a fork (which is also a hinge node), then $H(i_t) = \{i_t\}$. In this case, i_t is a connection node of $H(i_t)$, and obviously the one closest to itself. Hence, in this case SHAZOO predicts y_t simply by $\widehat{y}_{i_t} = \text{sgn}\big(\Delta(i_t)\big)$. See Figure 3.1 (middle) for an example. On unweighted trees, computing $\Delta(i)$ for a connection node i reduces to the Fork Label Estimation Procedure in [22, Lemma 13]. On the other hand, predicting with the label of the connection node closest to i_t in resistance distance is reminiscent of the nearest-neighbor prediction of WTA on weighted line graphs [25]. In fact, as in WTA, this enables to take advantage of labellings whose ϕ-edges are light weighted. An important limitation of WTA is that this algorithm linearizes the input tree. On the one

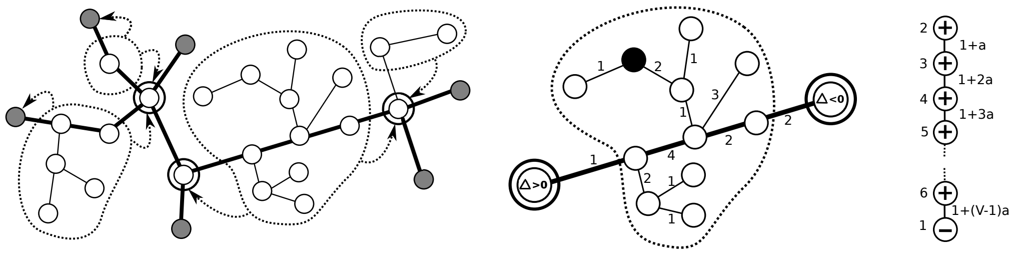


Fig. 3.1: **Left:** An input tree. Revealed nodes are dark gray, forks are doubly circled, and hinge lines have thick black edges. The hinge trees not containing hinge nodes (i.e., the ones that are not singletons) are enclosed by dotted lines. The dotted arrows point to the connection node(s) of such hinge trees. **Middle:** The predictions of SHAZOO on the nodes of a hinge tree. The numbers on the edges denote edge weights. At a given time t, SHAZOO uses the value of Δ on the two hinge nodes (the doubly circled ones, which are also forks in this case), and is required to issue a prediction on node i_t (the black node in this figure). Since i_t is between a positive Δ hinge node and a negative Δ hinge node, SHAZOO goes with the one which is closer in resistance distance, hence predicting $\widehat{y}_{i_t} = -1$. **Right:** A simple example where the mincut prediction strategy does not work well in the weighted case. In this example, mincut mispredicts all labels, yet $\Phi = 1$, and the ratio of Φ^W to the total weight of all edges is about $1/|V|$. The labels to be predicted are presented according to the numbers on the left of each node. Edge weights are also displayed, where a is a very small constant.

```
for $t = 1 \dots n$
    Let $C\big(H(i_t)\big)$ be the set of the connection nodes $i$ of $H(i_t)$ for which $\Delta(i) \neq 0$
    if $C\big(H(i_t)\big) \not\equiv \emptyset$
        Let $j$ be the node of $C\big(H(i_t)\big)$ closest to $i_t$
        Set $\widehat{y}_{i_t} = \mathrm{sgn}\big(\Delta(j)\big)$
    else
        Set $\widehat{y}_{i_t} = -1$ (default value)
```

Algorithm 1: SHAZOO

hand, this greatly simplifies the analysis of nearest-neighbor prediction; on the other hand, this prevents exploiting the structure of T, thereby causing logarithmic slacks in the upper bound of WTA. The TREEOPT algorithm, instead, performs better when the unweighted input tree is very different from a line graph (more precisely, when the input tree cannot be decomposed into long edge-disjoint paths, e.g., a star graph). Indeed, TREEOPT's upper bound does not suffer from logarithmic slacks, and is tight up to constant factors on any unweighted tree. Similar to TREEOPT, SHAZOO does not linearize the input tree and extends to the weighted case TreeOpt's superior performance, also confirmed by the experimental comparison reported in Section 7.3.

In Figure 3.1 (right) we show an example that highlights the importance of using the Δ function to compute the fork labels. Since Δ predicts a fork i_t with the label that minimizes the weighted cutsize of T consistent with the revealed labels, one may wonder whether computing Δ through mincut based on the number of ϕ-edges (rather than their weighted sum) could be an effective prediction strategy. Figure 3.1 (right) illustrates an example of a simple tree where such a Δ mispredicts the labels of all nodes, when both Φ^W and Φ are small.

Remark 3.2. We would like to stress that shazoo can also be used to predict the nodes of an arbitrary *graph* by first drawing a random spanning tree T of the graph, and then predicting optimally on T —see, e.g., [22, 25]. The resulting mistake bound is simply the expected value of shazoo's mistake bound over the random draw of T. By using a fast spanning tree sampler [82], the involved computational overhead amounts to constant amortized time per node prediction on "most" graphs.

Remark 3.3. In certain real-world input graphs, the presence of an edge linking two nodes may also carry information about the extent to which the two nodes are *dissimilar*, rather than similar. This information can be encoded by the sign of the weight, and the resulting network is called a *signed graph*. The regularity measure is naturally extended to signed graphs by counting the weight of *frustrated edges* (e.g.,[57]), where (i, j) is frustrated if $y_i y_j \neq \operatorname{sgn}(w_{i,j})$. Many of the existing algorithms for node classification [86, 54, 55, 22, 53, 25] can in principle be run on signed graphs. However, the computational cost may not always be preserved. For example, mincut [11] is in general NP-hard when the graph is signed [67]. Since our algorithm sparsifies the graph using trees, it can be run efficiently even in the signed case. We just need to re-define the Δ function as $\Delta(i) = \operatorname{fcut}(i, -1) - \operatorname{fcut}(i, +1)$, where fcut is the minimum total weight of frustrated edges consistent with the labels seen so far. The argument contained in Section 3.4 for the positive edge weights (see, e.g., Eq. (3.1) therein) allows us to show that also this version of Δ can be computed efficiently. The prediction rule has to be re-defined as well: We count the parity of the number z of negative-weighted edges along the path connecting i_t to the closest node $j \in C\big(H(i_t)\big)$, i.e., $\widehat{y}_{i_t} = (-1)^z \operatorname{sgn}\big(\Delta(j)\big)$.

Remark 3.4. In [22] the authors note that TreeOpt approximates a version space (Halving) algorithm on the set of tree labellings. Interestingly, shazoo is also an approximation to a more general Halving algorithm for weighted trees. This generalized Halving gives a weight to each labeling consistent with the labels seen so far and with the sign of $\Delta(f)$ for each fork f. These weighted labellings, which depend on the weights of the ϕ-edges generated by each labeling, are used for computing the predictions. One can show (details omitted due to space limitations) that this generalized Halving algorithm has a mistake bound within a constant factor of SHAZOO's.

3.4 Mistake bound analysis and implementation

We now show that SHAZOO is nearly optimal on every weighted tree T. We obtain an upper bound in terms of Φ^W and the structure of T, nearly matching the lower bound of Theorem 3.1. We now give some auxiliary notation that is strictly needed for stating the mistake bound.

Given a labeled tree $(T, \boldsymbol{y})$, a **cluster** is any maximal subtree whose nodes have the same label. An **in-cluster line graph** is any line graph that is entirely contained in a single cluster. Finally, given a line graph L, we set $R_L^W = \sum_{(i,j)\in L} \frac{1}{W_{i,j}}$, i.e., the (resistance) distance between its terminal nodes.

Theorem 3.5. *For any labeled and weighted tree $(T, \boldsymbol{y})$, there exists a set $\mathcal{L}_T$ of $\mathcal{O}\big(\xi(\Phi^W)\big)$ edge-disjoint in-cluster line graphs such that the number of mistakes made by* SHAZOO *is at most of the order of*

$$\sum_{L\in\mathcal{L}_T} \min\Big\{|L|, 1 + \big\lfloor \log\big(1 + \Phi^W R_L^W\big)\big\rfloor\Big\} .$$

Proof. The proof of this theorem is given in Appendix A.1

The above mistake bound depends on the tree structure through $\mathcal{L}_T$. The sum contains $\mathcal{O}\big(\xi(\Phi^W)\big)$ terms, each one being at most logarithmic in the scale-free products $\Phi^W R_L^W$. The bound is governed by the same key quantity $\xi\big(\Phi^W\big)$ occurring in the lower bound of Theorem 3.1. However, Theorem 3.5 also shows that SHAZOO can take advantage of trees that cannot be covered by long line graphs. For example, if the input tree T is a weighted line graph, then it is likely to contain long in-cluster lines. Hence, the factor multiplying $\xi\big(\Phi^W\big)$ may be of the order of $\log\big(1 + \Phi^W R_L^W\big)$. If, instead, T has constant diameter (e.g., a star graph), then the in-cluster lines can only contain a constant number of nodes, and the number of mistakes can never exceed $\mathcal{O}\big(\xi(\Phi^W)\big)$. This is a log factor improvement over WTA which, by its very nature, cannot exploit the structure of the tree it operates on.[2]

As for the implementation, we start by describing a method for calculating $\text{cut}(v, y)$ for any unlabeled node v and label value y. Let T^v be the maximal

[2] One might wonder whether an arbitrarily large gap between upper (Theorem 3.5) and lower (Theorem 3.1) bounds exists due to the extra factors depending on $\Phi^W R_L^W$. One way to get around this is to follow the analysis of WTA in [25]. Specifically, we can adapt here the more general analysis from that paper (see Lemma 2 therein) that allows us to drop, for any integer K, the resistance contribution of K arbitrary non-ϕ edges of the line graphs in $\mathcal{L}_T$ (thereby reducing R_L^W for any L containing any of these edges) at the cost of increasing the mistake bound by K.

subtree of T rooted at v, such that no internal node is revealed. For any node i of T^v, let T_i^v be the subtree of T^v rooted at i. Let $\Phi_i^v(y)$ be the minimum weighted cutsize of T_i^v consistent with the revealed nodes and such that $y_i = y$. Since $\Delta(v) = \mathrm{cut}(v,-1) - \mathrm{cut}(v,+1) = \Phi_v^v(-1) - \Phi_v^v(+1)$, our goal is to compute $\Phi_v^v(y)$. It is easy to see by induction that the quantity $\Phi_i^v(y)$ can be recursively defined as follows, where C_i^v is the set of all children of i in T^v, and $Y_j \equiv \{y_j\}$ if y_j is revealed, and $Y_j \equiv \{-1,+1\}$, otherwise:[3]

$$\Phi_i^v(y) = \begin{cases} \displaystyle\sum_{j \in C_i^v} \min_{y' \in Y_j} \Big(\Phi_j^v(y') + \mathbb{I}\{y' \neq y\}\, w_{i,j} \Big) & \text{if } i \text{ is an internal node of } T^v \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

Now, $\Phi_v^v(y)$ can be computed through a simple depth-first visit of T^v. In all backtracking steps of this visit the algorithm uses (3.1) to compute $\Phi_i^v(y)$ for each node i, the values $\Phi_j^v(y)$ for all children j of i being calculated during the previous backtracking steps. The total running time is therefore linear in the number of nodes of T^v.

Next, we describe the basic implementation of SHAZOO for the on-line setting. A batch learning implementation will be given at the end of this section. The online implementation is made up of three steps.

1. Find the hinge nodes of subtree T^{i_t}. Recall that a hinge-node is either a fork or a revealed node. Observe that a fork is incident to at least three nodes lying on different hinge lines. Hence, in this step we perform a depth-first visit of T^{i_t}, marking each node lying on a hinge line. In order to accomplish this task, it suffices to single out all forks marking each labeled node and, recursively, each parent of a marked node of T^{i_t}. At the end of this process we are able to single out the forks by counting the number of edges (i,j) of each marked node i such that j has been marked, too. The remaining hinge nodes are the leaves of T^{i_t} whose labels have currently been revealed.

2. Compute $\mathrm{sgn}(\Delta(i))$ for all connection forks of $H(i_t)$. From the previous step we can easily find the connection node(s) of $H(i_t)$. Then, we simply exploit the above-described technique for computing the cut function, obtaining $\mathrm{sgn}(\Delta(i))$ for all connection forks i of $H(i_t)$.

3. Propagate the labels of the nodes of $C(H(i_t))$ (only if i_t is not a fork). We perform a visit of $H(i_t)$ starting from every node $r \in C(H(i_t))$. During these visits, we mark each node j of $H(i_t)$ with the label of r computed in the previous step, together with the length of $\pi(r,j)$, which is what we need for predicting any label of $H(i_t)$ at the current time step.

The overall running time is dominated by the first step and the calculation of $\Delta(i)$. Hence the worst case running time is proportional to $\sum_{t \le |V|} |V(T^{i_t})|$. This quantity can be quadratic in $|V|$, though this is rarely encountered in practice if the node presentation order is not adversarial. For example, it is

[3] The recursive computations contained in this section are reminiscent of the sum-product algorithm [59].

easy to show that in a line graph, if the node presentation order is random, then the total time is of the order of $|V| \log |V|$. For a star graph the total time complexity is always linear in $|V|$, even on adversarial orders.

In many real-world scenarios, one is interested in the more standard problem of predicting the labels of a given subset of *test* nodes based on the available labels of another subset of *training* nodes. Building on the above on-line implementation, we now derive an implementation of SHAZOO for this train/test (or "batch learning") setting. We first show that computing $|\Phi_i^i(+1)|$ and $|\Phi_i^i(-1)|$ for all unlabeled nodes i in T takes $\mathcal{O}(|V|)$ time. This allows us to compute $\operatorname{sgn}(\Delta(v))$ for all forks v in $\mathcal{O}(|V|)$ time, and then use the first and the third steps of the on-line implementation. Overall, we show that predicting *all* labels in the test set takes $\mathcal{O}(|V|)$ time.

Consider tree T^i as rooted at i. Given any unlabeled node i, we perform a visit of T^i starting at i. During the backtracking steps of this visit we use (3.1) to calculate $\Phi_j^i(y)$ for each node j in T^i and label $y \in \{-1, +1\}$. Observe now that for any pair i, j of adjacent unlabeled nodes and any label $y \in \{-1, +1\}$, once we have obtained $\Phi_i^i(y)$, $\Phi_j^i(+1)$ and $\Phi_j^i(-1)$, we can compute $\Phi_i^j(y)$ in constant time, as $\Phi_i^j(y) = \Phi_i^i(y) - \min_{y' \in \{-1,+1\}} \big(\Phi_j^i(y') + \mathbb{I}\{y' \neq y\}\, w_{i,j}\big)$. In fact, all children of j in T^i are descendants of i, while the children of i in T^i (but j) are descendants of j in T^j. SHAZOO computes $\Phi_i^i(y)$, we can compute in constant time $\Phi_i^j(y)$ for all child nodes j of i in T^i, and use this value for computing $\Phi_j^j(y)$. Generalizing this argument, it is easy to see that in the next phase we can compute $\Phi_k^k(y)$ in constant time for all nodes k of T^i such that for all ancestors u of k and all $y \in \{-1, +1\}$, the values of $\Phi_u^u(y)$ have previously been computed.

The time for computing $\Phi_s^s(y)$ for all nodes s of T^i and any label y is therefore linear in the time of performing a breadth-first (or depth-first) visit of T^i, i.e., linear in the number of nodes of T^i. Since each labeled node with degree d is part of at most d trees T^i for some i, we have that the total number of nodes of all distinct (edge-disjoint) trees T^i across $i \in V$ is linear in $|V|$.

Finally, we need to propagate the connection node labels of each hinge tree as in the third step of the online implementation. Since also this last step takes linear time, we conclude that the total time for predicting all labels is linear in $|V|$.

3.5 Multiclass implementation

Our experiments later in this part of the thesis will be run on multiclass datasets. The extension of SHAZOO to the multiclass case is pretty simple and involves only the labelling of the fork nodes: instead of computing the mincut with only two classes we should compute the cut for all classes and

then choose the label with the minimum cut instead of the sign of $\varDelta$.
The multiclass version of SHAZOO has a running time that also depends on the number of classes, but for our purposes this can be considered a constant factor.

Note that the multiclass problem is harder than the binary problem. It is easy to show that the generic lower bound on the number of mistakes in the node classification problem with C classes is $\frac{(C-1)}{C}\,\xi(M)$.

Chapter 4
Batch Node Classification: MUCCA

In this chapter, we present an algorithm that is intrinsically multiclass and based on a completely different approach with respect to SHAZOO. The method presented in the following pages has been created for a special case of the strategic game presented in [42]. This "framework" is based on the idea that networks are made of entities making decisions only for themselves without coordination, and their only goal is to maximize their own reward. This may be a good model for many practical applications related to social networks. The presented algorithm is motivated by the need of scalability that can not be addressed by the approach introduced in [42], and provides a simple, but effective, method for finding a Nash Equilibrium of the game.

4.1 Basic Framework

Given a weighted graph $G = (V, E, W)$, a multiclass labelling of G is an assignment $\boldsymbol{y} = (y_1, .., y_n) \in \{0, 1, ..., c\}^n$ where $n = |V|$.
We expect our graph to respect a notion of regularity where adjacent nodes often have the same label: the classic notion of *homophily*. The learner is given the graph G, but just a subset of $\boldsymbol{y}$, that we call training set. The learner's goal is to predict the remaining labels minimizing while the number of mistakes. In [27], the authors introduce an irregularity measure of the graph G, for the labelling $\boldsymbol{y}$, defined as the ratio between the sum of the weights of the edges between nodes with different labels and the sum of all the weights. Intuitively, we can view the weight of an edge as a similarity measure between two nodes. We expect highly similar nodes to have the same label, and edges between nodes with different labels being "light". Based on this intuition, we assign unknown labels to nodes in a way that minimizes the weighted cut induced by this new labelling of the graph.

After these assumptions on the structure of the graph and the labelling, it is easy to see that we can formalize the binary classification problem as

a standard mincut problem (i.e., see [11]). It is sufficient to add to virtual nodes, that we will call respectively source and target. The first connected via new virtual edges to all the nodes with positive labels, and the second connected to all the nodes with negative labels. The weights of these new edges will be $+\infty$. In this way we have a standard mincut problem: we want to isolate the source and the target in two different connected components removing the smallest set of edges.
Generalizing this approach to the multiclass case, naturally leads us to the *Multiway Cut* (or Multiterminal Cut — see [34]) problem. We can use the same technique used for the binary case and connect all the nodes with the same label to the same terminal (a new virtual node), avoiding to connect to the same terminal nodes with different labels. Given a graph and the list of terminal nodes, the multiway cut problem consists in finding a set of edges such that, once removed, each terminal belongs to a different component. The goal is to minimize the sum of the weights of the removed edges.
Unfortunately, multiway cut is proven to be APX-hard when the number of terminals is bigger than two, and this definitely can not be considered a viable way (see [36] for more information about the multiway cut problem).

Furthermore, we would like to point our attention on real-world node classification problem, where it is often not realistic to think of the network as a big coordinated entity. Usually, entities in the network are dependent on each other, and each of them makes decisions only for itself, looking only at its own utility. For example, this is the case when people in a social network should choose an expensive service, or to adopt a particular behavior.

4.2 The Graph Transduction Game

In this section we describe the strategic interaction game introduced in [42] that, in a particular sense, aims at distributing among the nodes the cost of approximating the multiway cut. This is obtained through a non-cooperative game, and the final label assignment is a Nash Equilibrium of the game. Notice that, because this game is non-cooperative, each player maximizes its own payoff disregarding what it can do to maximize the sum of utilities of all the players. In non-cooperative game theory, the maximum social welfare is total utility that a centralized rational entity controlling all the players can achieve. You will notice that by the construction of this game, the maximum social welfare is twice the the sum of the weights of the edges minus twice the value of the multiway cut. Unfortunately, in the general case finding a Nash Equilibrium does not provide any guarantee about the collective result.
In the Graph Transduction Game (later called GTG), the graph topology is known in advance and each node of the graph is a player of the aforementioned game, and each possible label of the nodes is a pure strategy of the

players. Because we are working in a batch setting, we will have a train/test split that induces two kinds of players: **determined players**(I_D) those are nodes with a known label (provided by the training set), so in our game they will be players with a fixed strategy (they are not allowed to change their strategy because we can not change the labels given as training set) and **undetermined players**(I_U) those that do not have a fixed strategy and can choose whatever strategy they prefer (we have to predict their labels).

The game is defined as $\Gamma = (I, S, \pi)$, where $I = \{1, 2, ..., n\}$ is the set of players, $S = \times_{i \in I} S_i$ is the joint strategy space (the Cartesian product of all strategy sets $S_i \subseteq \{1, 2, ...c\}$), and $\pi : S \to \mathbb{R}^n$ is the combined payoff function which assigns a real valued payoff $\pi_i(s) \in \mathbb{R}$ to each pure strategy profile $s \in S$ and player $i \in I$.

A mixed strategy of player $i \in I$ is a probability distribution x over the set of the pure strategies of i. Each pure strategy k corresponds to a mixed strategy where all the strategies but the k-th one have probability equals to zero.

We define the utility function of the player i as

$$u_i(s) = \sum_{s \in S} x(s) \pi_i(s)$$

where $x(s)$ is the probability of s.

We assume the payoff associated to each player is additively separable (this will be clear in the following lines). This makes GTG a member of a subclass of the multi-player games called poly-matrix games. For a pure strategy profile $s = (s_1, s_2, ...s_n) \in S$, the payoff function of every player $i \in I$ is:

$$\pi_i(s) = \sum_{j \sim i} w_{ij} \mathbb{I}_{\{s_i = s_j\}}$$

where $i \sim j$ means that i and j are neighbours, this can be written in matrix form as

$$\pi_i(s) = \sum_{j \sim i} A_{ij}(s_i, s_j)$$

where $A_{ij} \in \mathbb{R}^{c \times c}$ is the partial payoff matrix between i and j, defined as $A_{ij} = I_c \times w_{ij}$, where I_c is the identity matrix of size c and $A_{ij}(x, y)$ represents the element of A_{ij} at row x and column $\boldsymbol{y}$. The utility function of each player $i \in I_U$ can be re-written as follows:

$$\begin{aligned} u_i(s) &= \textstyle\sum_{i \sim j} x_i^T A_{ij} x_j \\ &= \textstyle\sum_{i \sim j} w_{ij} x_i^T x_j \\ &= \textstyle\sum_{i \sim j} w_{ij} \sum_{k=1}^{c} x_{i_k} x_{j_k} \end{aligned}$$

where k is an action selected from the player's set and in case i is a determined node with training label k, x's components will be always zeros except the k-th corresponding to the pure strategy k. Because the utility function of each player is linear, it is easy to see that players can achieve their maximum payoff using pure strategies.

In a non-cooperative game, a vector of strategies S_{NE} is said to be a (pure strategies) Nash Equilibrium, if $\forall i \in I$, $\forall s'_i \in S_i : s'_i \neq s_i \in S_{NE}$, we have that

$$u_i(s_i, S_{NE}^{-i}) \geq u_i(s'_i, S_{NE}^{-i})$$

where $u_i(s_i, S^{-i})$ is the strategy configuration S except the i-th one, replaced by s_i. In practice, no player i can change its strategy s_i to an alternative strategy while improving its payoff.
There are no guarantees that the Nash Equilibrium exists in pure strategies, but *any game with a finite set of players and finite set of strategies has a Nash Equilibrium in mixed strategies* ([71], also see [72]). In this case each player does not have to choose a strategy but it mixes its choices over its strategies. Instead of maximizing its payoff, it will maximize its expected payoff.

With a slight abuse of terminology, we hereafter refer to labels or pure strategies with the same meaning.

4.2.1 The Evolutionary Stable Strategies approach

In this section we briefly present the approach employed in [42] to find a Nash Equilibrium. The goal is achieved using the Evolutionary Stable Strategies (ESS), a well-known approach in the game-theoretic literature (see [80]).
The only aim of this section is to provide the method previously used for the general case of GTG, and make the comparison in Chapter 5 easier to understand for the readers.

The ESS approach considers a game played repeatedly; each repetition of the game is seen as a generation, where an imaginary population evolves through a selection mechanism that, at each step, gives to the best "choices" a growing portion of the total population.
The algorithm (later called GTG-ESS), at each generation, updates the probability associated to every action h of every player i as

$$x_{ih}(t+1) = x_{ih}(t)\frac{u_i(e_h)}{u_i(x(t))}$$

The previous formula is just the discrete version of the so-called multi-population replicator dynamic:

$$\dot{x}_{ih} = x_{ih}(u_i(e_h, x_{-i}) - u_i(x))$$

where the dot notation represents the derivative with respect to time, e_h is a vector of zeros except the h-th component that is one, and x_{ih} is the h-th strategy of player i. The fixed points of the previous equations are Nash Equilibria, and the discrete version has the same properties — for further details see [42].
The computational cost of finding the Nash Equilibrium is $O(k|V|^2)$ where k is the number of iterations, and to the best of our knowledge does not exist an upper bound on the number of iterations. Anyway, the authors of [42] experimentally found that the number of iterations grows linearly with the number of nodes, so they consider the empirical running time close to $O(|V|^3)$.

4.3 A scalable method for the undirected trees

In many practical sequences, GTG-ESS can not be considered a feasible solution, even if the time complexity of the algorithm were to be shown in the order of $\Theta(|V|^3)$.
A possible alternative is to apply some known results about regret minimization, such as those described in [30], to converge to a weaker notion of equilibrium, for example, the Correlated Equilibrium. Unfortunately, the results of our preliminary experiments with the Correlated Equilibrium were not satisfactory.
In this section we present MUCCA: a **Mu**lti**c**lass **C**lassification **A**lgorithm. The algorithm consists in finding a Nash Equilibrium of the Graph Transduction Game on a special graph: an undirected tree. We will show that in this way we can achieve both good accuracy and scalability. The rest of this section will assume that the graph G is a tree.
In the following we will use most of the notions introduced in the previous chapter: hinge tree, fork node, and so forth. The only three new notions introduced in this section are:

- **Native hinge tree**: component of the forest created by removing from G all the edges incident to revealed nodes. Revealed nodes are intended to be part of the tree.
- **ϵ-edge**: given $\mathcal{P}_{ij}$, the path between i and j, an ϵ-edge is $\epsilon_{ij} \in \arg\min_{e \in \mathcal{P}_{ij}} w_e$, where w_e is the weight of the edge e.
- **Grafted tree**: a tree without hinge nodes connected to just one node on a hinge line

Endowed of these new notions, we then proceed to describe the simplest implementation of MUCCA. We split the algorithmic activity into four phases to make it even simpler to understand:

1. Mark all the paths between revealed nodes and find all the fork nodes
2. Estimate the label of each fork node

3. Assign a label to all the nodes on the hinge lines using a min-cut technique
4. Assign a label to all the remaining nodes

The implementation of MUCCA presented in this paper runs in $O(f|V|)$ where f is the number of forks, but it is possible to obtain a better implementation using some strategies explained in the previous chapter. In the following we will describe in detail all the phases of the algorithm:

1. Starting from each revealed node, MUCCA does a breadth-first search until another revealed node or a leaf is found. Then if a revealed node was found, during the backtracking process, MUCCA marks the edges on the path to the starting node with a special flag since they are on an hinge line. After that, each node with more that has more than two disjoint paths on the hinge lines to reach a revealed nodes, is a fork.
2. Given a native hinge tree $\mathcal{H}$ that contains the fork $\mathcal{F}$, we can categorize its revealed nodes into c categories using their labels. For each path between $\mathcal{F}$ and each connection node of $\mathcal{H}$, we have an ϵ-edge as defined before. The label assigned to $\mathcal{F}$ is the same as the category (of the connection nodes) that has the maximum sum of weights over the distinct ϵ-edges on the path between $\mathcal{F}$ and the nodes of that category.
3. On every hinge line, we label the nodes using min-cut: in case the hinge nodes at the beginning and at the end of the line has the same label, all the nodes on the hinge line will be labelled with that label. Otherwise, all the nodes before the ϵ-edge are assigned with the label of the node at the beginning of the line, and the others with the label of the node at the end of the line. In case we have more than one edge with the same weight of the ϵ-edge (for example, all the edges have the same weight), we use nearest neighbour to find the closest revealed node to complete the labelling of nodes in the line.
4. All the remaining nodes are in grafted trees. In this case, we assign the label of the node on the hinge line (connected to the tree) to all the nodes in that grafted tree.

Now we have a complete knowledge of all the operations performed by the algorithm and we can prove that MUCCA finds a Nash Equilibrium for this special case of the GTG.

Theorem 4.1. *The labelling found by* MUCCA *is a Nash Equilibrium of the Graph Transduction Game when the graph is an undirected tree.*

Proof. As we explained in Section 3, a profile of strategies S_{NE} is a Nash Equilibrium if no one has incentive to deviate from its strategy. This means that $\forall i \in I$, $u_i(s_i, S_{NE}^{-i}) \geq u_i(s_i', S_{NE}^{-i})$. For the purpose of contradiction suppose there is a node j such that it can improve its payoff by changing its strategy.
j can not be contained within a grafted tree (those labelled in phase 4) because all the nodes contained in those trees have the same labels, so, whatever

the label, each of them gets a payoff of $\sum_{i \sim j} w_{ij}$, the maximum possible payoff.
j can not be on a hinge line because they are labelled using min-cut, so in the best case the payoff of each node is already the maximum payoff; in the worst case the payoff is the maximum minus the weight of the ϵ-edge. Because the ϵ-edge has the minimum weight, there is no chance to improve the payoff.
j can not be a revealed node (obviously).
j can not be a fork. Because we use min-cut to label the hinge lines if the ϵ-edge of a hinge line is not incident to the fork, the node adjoining the fork on that hinge line will have the same label of the fork. In this way the fork will get the part of payoff given by the edge between it and the adjoining node. Even if the ϵ-edges are incident to the fork, the label prediction can not achieve a payoff better than the one achieved by the majority vote.
Because j can not be a revealed node, nor a fork, nor a node on a hinge line, nor a node on a tree with just a connection node, it can not be in G, and this concludes the proof.

Note that the labels of the unlabelled nodes of every native hinge tree can be predicted using only information about that singular native hinge tree. In this way, once the tree is split into native hinge trees, the predictions for the labels contained in every sub tree are independent from the other sub trees. Predictions can be computed using various threads, processes or even machines and we just need to retrieve a list node ids and labels. This optional parallelization is in addition to the already high scalability of MUCCA.

In the next chapter we report the results of an experimental comparison on synthetic and real world datasets.

Chapter 5

Node Classification: Experiments

In this chapter, we report the results of an experimental comparison between the algorithms described in the previous chapters and the state of the art. We tested the algorithms on four real-world weighted graphs and three artificial graphs with different levels of noise generated from the well-known USPS dataset. The tables with all the numerical results are available in Appendix A.

5.1 Algorithms

In this section we briefly describe some of the algorithms used in the following experiments. Some of them have already been introduced in Chapter 3 and Chapter 4.

- **Evolutionary Stable Strategies for GTG** (GTG-ESS).
  GTG-ESS [42] is a batch algorithm that works in framework described in Chapter 4. It is a generalization of MUCCA that works on the original graph. Its time complexity is not guaranteed to be polynomial, but in all our experiments its running time is comparable with LABPROP's.

- **Label Propagation** (LABPROP).
  LABPROP [86, 9, 10] is a batch transductive learning method computed by solving a system of linear equations which requires total time of the order of $|E| \times |V|$. This relatively high computational cost should be taken into account when comparing LABPROP to faster online algorithms.

- **Weighted Tree Algorithm** (WTA).
  As explained before, WTA can be viewed as a special case of SHAZOO. When the input graph is not a line, WTA turns it into a line by first extracting a spanning tree of the graph, and then linearizing it. The implementation described in [25] runs in constant amortized time per prediction, whenever

the spanning tree sampler runs in time $\Theta(|V|)$.

- **Graph Perceptron algorithm** (GPA).
  GPA [55] is one of the first algorithms for online learning on graphs that we include in our comparison as baseline. The algorithm is a kernelized Perceptron where the kernel is given by the inverse Laplacian matrix[1] of the graph (or tree for the fast implementation).

In our experiments, we combined GPA, MUCCA, SHAZOO and WTA with spanning trees generated in different ways and following [55, 25]. We also ran MUCCA, SHAZOO and WTA using committees of spanning trees, and then aggregating predictions via a majority vote. The resulting algorithms are denoted by k*ALGORITHMNAME, where k is the number of spanning trees in the aggregation. We used $k = 3, 7, 11$ since from our experiments on these datasets $k >> 11$ gives a very limited advantage, but it will probably help in the case of bigger datasets.
Unfortunately we could not run committees of GPA predictors due to our limited computational resources.

5.2 Spanning trees

In this section we briefly explain how we generated the spanning trees used for the experiments described later in this chapter:

- **Random Spanning Tree** (RST). Following Chapter 4 of [64], we draw a weighted spanning tree with probability proportional to the product of its edge weights.

- **Minimum Spanning Tree** (MST). This is the spanning tree that minimizes the sum of the resistors on its edges. This tree best approximates the original graph in terms of the trace norm distance of the corresponding Laplacian matrices.

- **Breadth First Spanning Tree** (BFST). This is a classical spanning tree obtained with a breadth first visit on the graph started from a randomly selected root.

[1] The Laplacian matrix L is defined as $L = \left[L_{i,j}\right]_{i,j=1}^{n}$, where $L_{i,i}$ is the degree of node i (i.e., the number of edges or the sum of the weights of the edges incident to that node) and, for $i \neq j$, $L_{i,j}$ equals $-w_{ij}$ if $(i,j) \in E$, and 0 otherwise

5.3 Datasets

For our experiments, we used four real-world datasets: CORA, COAUTHORS, IMDB, PUBMED.[2].

- **CORA**[3] is a directed graph with 2,708 nodes and 5,429 arcs. Each node represents a scientific publications classified into one of seven classes: Case Based, Genetic Algorithms, Neural Networks, Probabilistic Methods, Reinforcement Learning, Rule Learning and Theory. Arcs are given by citations between publications.

- **COAUTHORS**[4] is an undirected graph with 1,711 nodes and 7,507 edges. Each node of the graph represents an author from a subset DBLP, and each edge represents a collaboration between two authors, weights on the edges are given by the number of co-authored papers. Authors are picked in 4 different areas (classes): Machine Learning, Data Mining, Information Retrieval and Databases. Classes are quite balanced since each class contains about 400 authors.

- **IMDB**[5] is an undirected graph with 17,046 nodes and 993,528 edges. Each node of the graph represents a movie and each node represents a co-authorship. Movies are divide in four genres (classes): “Romance”, “Action”, “Animation” and “Thriller”.

- **PUBMED**[6] is a graph with 19,717 nodes and 44,338 arcs. Each node of the graph represents a scientific publication pertaining to diabetes classified into three different classes.

Most of the presented algorithms work exclusively on undirected graphs, so we created a symmetrized version of the adjacency matrix A as $A_{sym} \leftarrow A + A^{\top}$. In this way connections are stronger when, in the original graph, there are reciprocal arcs.

Moreover, we compared the same algorithms on three synthetic datasets with different levels of noise created from the well-known USPS dataset. All the following datasets have 9,298 nodes and the first version (without “noisy” edges) has 68,818 edges. Given two data points in USPS called $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$, weights on the edges are set as $w_{i,j} = \exp\big(-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2/\sigma^2_{i,j}\big)$, if j is one of the k nearest neighbors of i, and 0 otherwise. To set $\sigma^2_{i,j}$, we first computed the average square distance between i and its k nearest neighbors (call it σ^2_i),

[2] These datasets have been made available by the authors of [47]

[3] http://www.cs.umd.edu/sen/lbc-proj/data/cora.tgz

[4] This dataset is extracted from the DBLP database and previously used in [48]

[5] http://www.imdb.com/

[6] http://www.cs.umd.edu/projects/linqs/projects/lbc/Pubmed-Diabetes.tgz

Datasets	Number of Classes	Number of Nodes	Number of Edges
CORA	7	2708	5069
COAUTHORS	4	1711	7507
IMDB	4	17046	993528
PUBMED	3	19717	44325

Table 5.1: Statistics about the real-world datasets used in the experimental comparison. The number of edges calculated after the (eventual) symmetrization.

then we computed σ_j^2 in the same way, and finally set $\sigma_{i,j}^2 = \left(\sigma_i^2 + \sigma_j^2\right)/2$.

Datasets are the following:

- **USPS-0**: is a graph created connecting the 10 nearest neighbours of each data point in the USPS dataset and then symmetrized. The resulting graph is excellently clustered and it can be considered an easy prediction task for our algorithms.
- **USPS-10**: is the USPS-0 dataset with 10,000 more edges created uniformly at random and with random weights in $(0, 1)$ on them.
- **USPS-25**: is the USPS-10 dataset with 15,000 more edges created uniformly at random and with random weights in $(0, 1)$ on them.

The noisy versions of the datasets are intended to be used to explore situations in which the assumptions made about the labels are not respected. In this way we can observe how the performances degrade when the noise increase.

5.4 Results

In the below figures, we show the averaged classification error rates (percentages) achieved by the various algorithms on the first four datasets mentioned above. For each dataset we trained ten times over a random subset of 2.5%, 5%, 10%, 25%, 33,33% and 50% of the total number of nodes and tested on the remaining ones. We used GTG-ESS and LABPROP as “yardsticks” for our comparison. Standard deviations are reported in parenthesis.

Our empirical results can be briefly summarized as follows:

- GTG-ESS is always the best classifier and its advantage over LABPROP becomes bigger and bigger when the training set is small or the dataset is noisy. Moreover, the “noisy” edges we added affect only marginally its

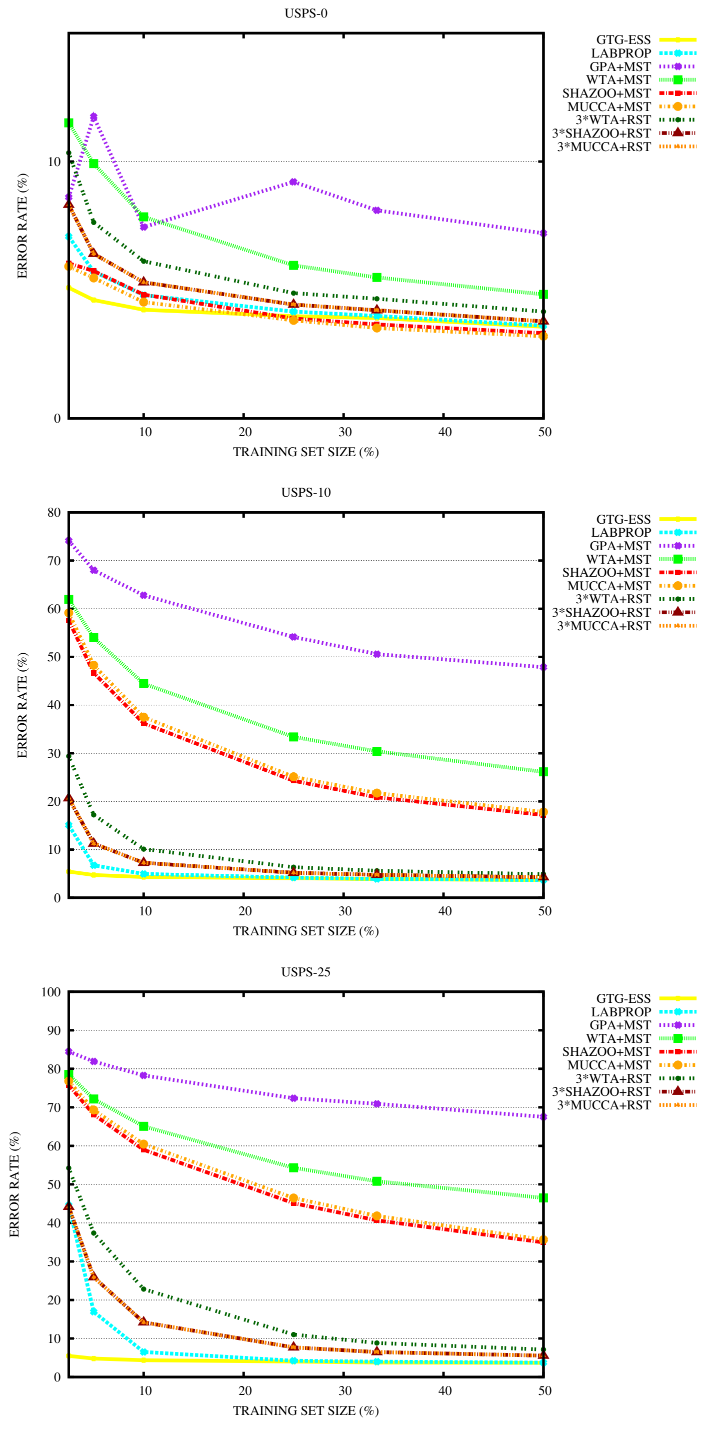
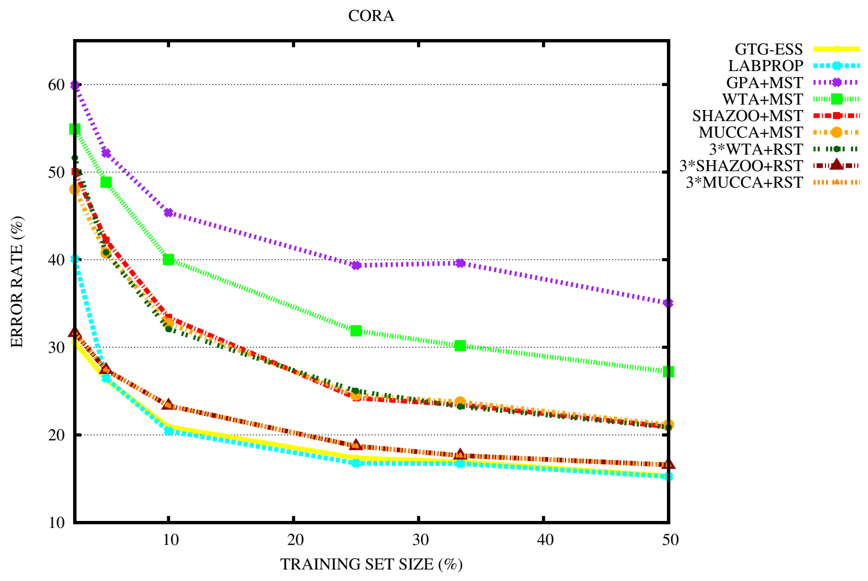


Fig. 5.1: Plots of the most interesting results on the USPSs datasets.



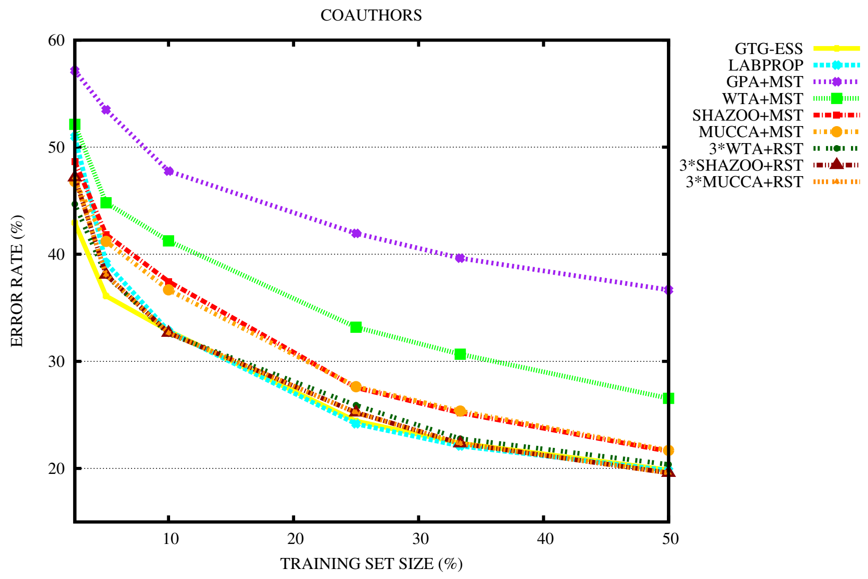
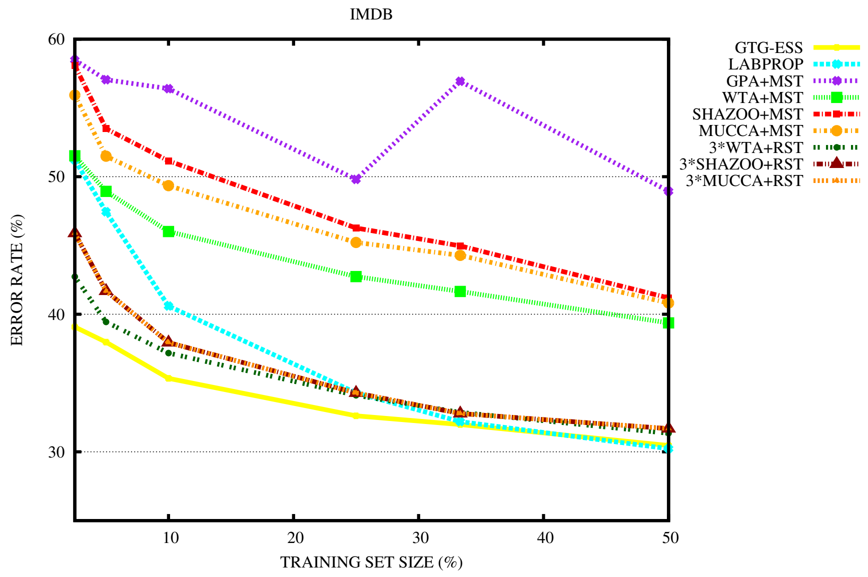


Fig. 5.2: Plots of the most interesting results on the CORA and COAUTHORS datasets: the smallest real-world datasets in this comparison.



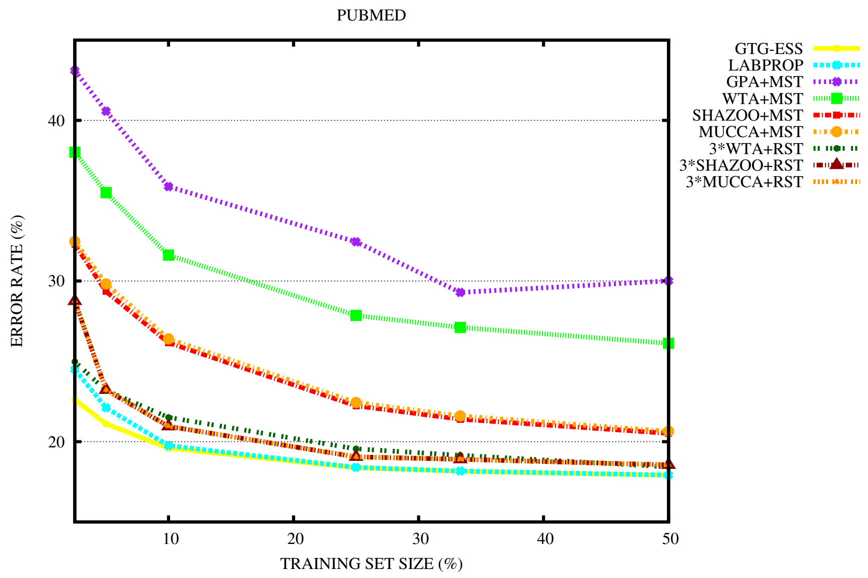


Fig. 5.3: Plots of the most interesting results on the IMDB and PUBMED datasets, two medium-size real-world networks.

performances.

- When the clusters of the graph are well separated (like in USPS-0) the tree-predictors perform best using the MST. When the topology of the graph contains some noise, the tree-based predictors can not rely on a single tree but they need a committee of spanning trees. In the latter case the committees of spanning trees provide a great improvement over the performances of a single tree.

- Committees of predictors on trees are often in the same ballpark of GTG-ESS and LABPROP. Moreover, GTG-ESS, 11*SHAZOO, 11*MUCCA and LABPROP's performances come often closer as the training set grows.

- The variance is often too high to determine which is the best predictor between SHAZOO and MUCCA, but it seems that: MUCCA performs better on the real datasets, while SHAZOO performs better on noisy synthetic datasets. Anyway, usually both of them perform better than WTA.

- The wall-clock time is similar for MUCCAand SHAZOO, as expected. Committees of MUCCAmay be up to 15x faster than GTG-ESS.

- GPA, as shown in literature, is not a very good option in practice.

Part III
Link Classification

All the world is made of faith, and trust, and pixie dust.
– James Matthew Barrie

Chapter 6
Link Classification

In this chapter, we consider the problem of link classification on signed networks with stochastic noise. Signed networks are an important emerging topic in the machine learning community since they are often employed to formalize many practical scenarios, mostly related with trust and distrust among different users. From a mathematical point of view, these networks are graphs whose edges are endowed with a sign representing the positive or negative nature of the relationship between the incident nodes. In the following, we assume that the underlying structure of the graph is composed of two clusters where inter-cluster edges are negative and intra-cluster edges are positive. This particular bias is supported and explained by the social balance theory.

The content of this chapter is a joint work with Nicolò Cesa-Bianchi, Claudio Gentile and Fabio Vitale.

6.1 Problem Setup

We consider undirected and connected graphs $G = (V, E)$ with unknown edge labeling $Y_{i,j} \in \{-1, +1\}$ for each $(i, j) \in E$. Edge labels can collectively be represented by the associated *signed* adjacency matrix Y, where $Y_{i,j} = 0$ whenever $(i, j) \notin E$. In the sequel, the edge-labeled graph G will be denoted by (G, Y).

We define a simple stochastic model for assigning binary labels Y to the edges of G. This is used as a basis and motivation for the design of our link classification strategies. We assume that the noise in the labeling is stochastic, in particular that edge's labels are obtained by perturbing an underlying labeling which is initially consistent with an arbitrary (and unknown) two-clustering. More formally, given an undirected and connected graph $G = (V, E)$, the labels $Y_{i,j} \in \{-1, +1\}$, for $(i, j) \in E$, are assigned as follows. First, the nodes in V are arbitrarily partitioned into two sets, and labels $Y_{i,j}$ are initially assigned consistently with this partition (within-cluster edges are positive and between-cluster edges are negative). Note that the consistency is equivalent to the following *multiplicative rule*: For any $(i, j) \in E$, the label $Y_{i,j}$ is equal to the product of signs on the edges of *any* path connecting i to j in G. This is in turn equivalent to say that any simple cycle within the graph contains an *even* number of negative edges. Then, given a nonnegative constant $p < \frac{1}{2}$, labels are randomly flipped in such a way that $\mathbb{P}\big(Y_{i,j} \text{ is flipped}\big) \le p$ for each $(i, j) \in E$. We call this a p-stochastic assignment. Note that this model allows for correlations between flipped labels.

A learning algorithm in the link classification setting receives a training set of signed edges and, out of this information, builds a prediction model for the labels of the remaining edges. It is quite easy to prove a lower bound on the number of mistakes that any learning algorithm makes in this model.

Theorem 6.1. *For any undirected graph $G = (V, E)$, any training set $E_0 \subset E$ of edges, and any learning algorithm that is given the labels of the edges in E_0, the number M of mistakes made by A on the remaining $E \setminus E_0$ edges satisfies $\mathbb{E}\, M \ge p\big|E \setminus E_0\big|$, where the expectation is with respect to a p-stochastic assignment of the labels Y.*

Proof. Let Y be the following randomized labeling: first, edge labels are set consistently with an arbitrary two-clustering of V. Then, a set of $2p|E|$ edges is selected uniformly at random and the labels of these edges are set randomly (i.e., flipped or not flipped with equal probability). Clearly, $\mathbb{P}(Y_{i,j} \text{ is flipped}) = p$ for each $(i, j) \in E$. Hence this is a p-stochastic assignment of the labels. Moreover, $E \setminus E_0$ contains in expectation $2p\big|E \setminus E_0\big|$ randomly labeled edges, on which A makes $p\big|E \setminus E_0\big|$ mistakes in expectation.

In this paper we focus on active learning algorithms. An active learner for link classification first constructs a query set E_0 of edges, and then receives the labels of all edges in the query set. Based on this training information, the learner builds a prediction model for the labels of the remaining edges $E \setminus E_0$. We assume that the only labels ever revealed to the learner are those in the query set. In particular, no labels are revealed during the prediction phase. It is clear from Theorem 6.1 that any active learning algorithm that queries the labels of at most a constant fraction of the total number of edges will make on average $\Omega(p|E|)$ mistakes.

We often write V_G and E_G to denote, respectively, the node set and the edge set of some underlying graph G. For any two nodes $i, j \in V_G$, $\mathrm{P}(i, j)$ is any path in G having i and j as terminals, and $|\mathrm{P}(i, j)|$ is its length (number of edges). The diameter D_G of a graph G is the maximum over pairs $i, j \in V_G$ of the shortest path between i and j. Given a tree $T = (V_T, E_T)$ in G, and two nodes $i, j \in V_T$, we denote by $d_T(i, j)$ the distance of i and j within T, i.e., the length of the (unique) path $\mathrm{P}_T(i, j)$ connecting the two nodes in T. Moreover, $\pi_T(i, j)$ denotes the *parity* of this path, i.e., the product of edge signs along it. When T is a rooted tree, we denote by $\mathrm{Children}_T(i)$ the set of children of i in T. Finally, given two disjoint subtrees $T', T'' \subseteq G$ such that $V_{T'} \cap V_{T''} \equiv \emptyset$, we let $E_G(T', T'') \equiv \big\{(i,j) \in E_G \,:\, i \in V_{T'},\, j \in V_{T''}\big\}$.

6.2 Algorithms and their analysis

In this section, we introduce and analyze a family of active learning algorithms for link classification. The analysis is carried out under the p-stochastic assumption. As a warm up, we start off recalling the connection to the theory of low-stretch spanning trees (e.g., [41]), which turns out to be useful in the important special case when the active learner is afforded to query only $|V| - 1$ labels.

Let $E_{\text{flip}} \subset E$ denote the (random) subset of edges whose labels have been flipped in a p-stochastic assignment, and consider the following class of active learning algorithms parameterized by an arbitrary spanning tree $T = (V_T, E_T)$ of G. The algorithms in this class use $E_0 = E_T$ as query set. The label of any test edge $e' = (i,j) \notin E_T$ is predicted as the parity $\pi_T(e')$. Clearly enough, if a test edge e' is predicted wrongly, then either $e' \in E_{\text{flip}}$ or $\mathrm{P}_T(e')$ contains at least one flipped edge. Hence, the number of mistakes M_T made by our active learner on the set of test edges $E \setminus E_T$ can be deterministically bounded by

$$M_T \le |E_{\text{flip}}| + \sum_{e' \in E \setminus E_T} \sum_{e \in E} \mathbb{I}\left\{e \in \mathrm{P}_T(e')\right\} \mathbb{I}\left\{e \in E_{\text{flip}}\right\} \tag{6.1}$$

where $\mathbb{I}\{\cdot\}$ denotes the indicator of the Boolean predicate at argument. A quantity which can be related to M_T is the *average stretch* of a spanning tree T which, for our purposes, reduces to

$$\tfrac{1}{|E|}\left[|V| - 1 + \textstyle\sum_{e' \in E \setminus E_T} \big|\mathrm{P}_T(e')\big|\right] .$$

A stunning result of [41] shows that every connected, undirected and unweighted graph has a spanning tree with an average stretch of just $\mathcal{O}\big(\log^2 |V| \log\log |V|\big)$. If our active learner uses a spanning tree with the same low stretch, then the following result holds.

Theorem 6.2 ([26]). *Let $(G, Y) = ((V, E), Y)$ be a labeled graph with p-stochastic assigned labels Y. If the active learner queries the edges of a spanning tree $T = (V_T, E_T)$ with average stretch $\mathcal{O}\big(\log^2 |V| \log\log |V|\big)$, then $\mathbb{E}\, M_T \le p|E| \times \mathcal{O}\big(\log^2 |V| \log\log |V|\big)$.*

We call the quantity multiplying $p\,|E|$ in the upper bound the *optimality factor* of the algorithm. Recall that Theorem 6.1 implies that this factor cannot be smaller than a constant when the query set size is a constant fraction of $|E|$.

Although low-stretch trees can be constructed in time $\mathcal{O}\big(|E|\ln|V|\big)$, the algorithms are fairly complicated (we are not aware of available implementations), and the constants hidden in the asymptotics can be high. Another disadvantage is that we are forced to use a query set of small and fixed size $|V|-1$. In what follows we introduce algorithms that overcome both limitations.

A key aspect in the analysis of prediction performance is the ability to select a query set so that each test edge creates a short circuit with a training path. This is quantified by $\sum_{e\in E}\mathbb{I}\left\{e\in \mathrm{P}_T(e')\right\}$ in (6.1). We make this explicit as follows. Given a test edge (i,j) and a path $\mathrm{P}(i,j)$ whose edges are queried edges, we say that we are predicting label $Y_{i,j}$ *using path* $\mathrm{P}(i,j)$ Since (i,j) closes $\mathrm{P}(i,j)$ into a circuit, in this case we also say that (i,j) is predicted using the circuit.

Theorem 6.3. *Let $(G,Y)=((V,E),Y)$ be a labeled graph with p-stochastic assigned labels Y. Given query set $E_0\subseteq E$, the number M of mistakes made when predicting test edges $(i,j)\in E\setminus E_0$ using training paths $\mathrm{P}(i,j)$ whose length is uniformly bounded by ℓ satisfies $\mathbb{E}M\le \ell\,p\,|E\setminus E_0|$.*

Proof. We have the chain of inequalities

$$\begin{aligned}
\mathbb{E}M &\le \sum_{(i,j)\in E\setminus E_0}\big(1-(1-p)^{|\mathrm{P}(i,j)|}\big)\\
&\le \sum_{(i,j)\in E\setminus E_0}\big(1-(1-p)^{\ell}\big)\\
&\le \sum_{(i,j)\in E\setminus E_0}\ell\,p\\
&\le \ell\,p\,|E\setminus E_0|\ .
\end{aligned}$$

For instance, if the input graph $G=(V,E)$ has diameter D_G and the queried edges are those of a breadth-first spanning tree, which can be generated in $O(|E|)$ time, then the above fact holds with $|E_0|=|V|-1$, and $\ell=2\,D_G$. Comparing to Theorem 6.1 shows that this simple breadth-first strategy is optimal up to constants factors whenever G has a constant diameter. This simple observation is especially relevant in the light of the typical graph topologies encountered in practice, whose diameters are often small. This argument is at the basis of our experimental comparison.

Yet, this mistake bound can be vacuous on graph having a larger diameter. Hence, one may think of adding to the training spanning tree new edges so as to reduce the length of the circuits used for prediction, at the cost of increasing the size of the query set. A similar technique based on short circuits has been used in [26], the goal there being to solve the link classification problem in a harder adversarial environment. The precise tradeoff between prediction accuracy (as measured by the expected number of mistakes) and fraction of queried edges is the main theoretical concern of this paper.

We now introduce an intermediate (and simpler) algorithm, called TREECUTTER, which improves on the optimality factor when the diameter D_G is not small. In particular, we demonstrate that TREECUTTER achieves a good upper bound on the number of mistakes on any graph such that $|E| \geq 3|V| + \sqrt{|V|}$. This algorithm is especially effective when the input graph is dense, with an optimality factor between $\mathcal{O}(1)$ and $\mathcal{O}(\sqrt{|V|})$. Moreover, the total time for predicting the test edges scales linearly with the number of such edges, i.e., TREECUTTER predicts edges in *constant amortized* time. Also, the space is linear in the size of the input graph.

The algorithm (pseudocode given in Figure 6.1) is parametrized by a positive integer k ranging from 2 to $|V|$. The actual setting of k depends on the graph topology and the desired fraction of query set edges, and plays a crucial role in determining the prediction performance. Setting $k \leq D_G$ makes TREECUTTER reduce to querying only the edges of a breadth-first spanning tree of G, otherwise it operates in a more involved way by splitting G into smaller node-disjoint subtrees.

In a preliminary step (Line 1 in Figure 6.1), TREECUTTER draws an arbitrary breadth-first spanning tree $T = (V_T, E_T)$. Then subroutine EXTRACTTREELET(T, k) is used in a do-while loop to split T into vertex-disjoint subtrees T' whose height is k (one of them might have a smaller height). EXTRACTTREELET(T, k) is a very simple procedure that performs a depth-first visit of the tree T at argument. During this visit, each internal node may be visited several times (during backtracking steps). We assign each node i a tag $h_T(i)$ representing the height of the subtree of T rooted at i. $h_T(i)$ can be recursively computed during the visit. After this assignment, if we have $h_T(i) = k$ (or i is the root of T) we return the subtree T_i of T rooted at i. Then TREECUTTER removes (Line 6) T_i from T along with all edges of E_T which are incident to nodes of T_i, and then iterates until V_T gets empty. By construction, the diameter of the generated subtrees will not be larger than $2k$. Let $\mathcal{T}$ denote the set of these subtrees. For each $T' \in \mathcal{T}$, the algorithm queries all the labels of $E_{T'}$, each edge $(i,j) \in E_G \setminus E_{T'}$ such that $i, j \in V_{T'}$ is set to be a test edge, and label $Y_{i,j}$ is predicted using $\mathrm{P}_{T'}(i,j)$ (note that this coincides with $\mathrm{P}_{T'}(i,j)$, since $T' \subseteq T$), that is, $\hat{Y}_{i,j} = \pi_T(i,j)$. Finally, for each pair of distinct subtrees $T', T'' \in \mathcal{T}$ such that there exists a node of $V_{T'}$ adjacent to a node of $V_{T''}$, i.e., such that $E_G(T', T'')$ is not empty, we query the label of an arbitrarily selected edge $(i', i'') \in E_G(T', T'')$ (Lines 8 and 9 in Figure 6.1). Each edge $(u,v) \in E_G(T', T'')$ whose label has not been previously queried is then part of the test set, and its label will be predicted as $\hat{Y}_{u,v} \leftarrow \pi_T(u, i') \cdot Y_{i',i''} \cdot \pi_T(i'', v)$ (Line 11). That is, using the path obtained by concatenating $\mathrm{P}_{T'}(u, i')$ to edge (i', i'') to $\mathrm{P}_{T'}(i'', v)$.

The following theorem[1] quantifies the number of mistakes made by TREECUTTER. The requirement on the graph density in the statement, i.e., $|V| - 1 + \frac{|V|^2}{2k^2} + \frac{|V|}{2k} \leq \frac{|E|}{2}$ implies that the test set is not larger than the

[1] Due to space limitations long proofs are presented in the supplementary material.

```
treeCutter($k$)        Parameter: $k \geq 2$
Initialization: $\mathcal{T} \leftarrow \emptyset$.
1.  Draw an arbitrary breadth-first spanning tree $T$ of $G$
2.  Do
3.      $T' \leftarrow$ extractTreelet($T, k$), and query all labels in $E_{T'}$
4.      $\mathcal{T} \leftarrow \mathcal{T} \cup \{T'\}$
5.      For each $i, j \in V_{T'}$, set predict $\hat{Y}_{i,j} \leftarrow \pi_T(i, j)$
6.      $T \leftarrow T \setminus T'$
7.  While ($V_T \not\equiv \emptyset$)
8.  For each $T', T'' \in \mathcal{T} : T' \not\equiv T''$
9.      If $E_G(T', T'') \not\equiv \emptyset$ query the label of an arbitrary edge $(i', i'') \in E_G(T', T'')$
10.     For each $(u, v) \in E_G(T', T'') \setminus \{(i', i'')\}$, with $i', u \in V_{T'}$ and $v, i'' \in V_{T''}$
11.         predict $\hat{Y}_{u,v} \leftarrow \pi_{T'}(u, i') \cdot Y_{i',i''} \cdot \pi_{T''}(i'', v)$
```

Fig. 6.1: treeCutter pseudocode.

```
extractTreelet($T, k$)        Parameters: tree $T$, $k \geq 2$.
1.  Perform a depth-first visit of $T$ starting from the root.
2.  During the visit
3.      For each $i \in V_T$ visited for the $|1 + \text{Children}_T(i)|$-th time (i.e., the last visit of $i$)
4.          If $i$ is a leaf set $h_T(i) \leftarrow 0$
5.          Else set $h_T(i) \leftarrow 1 + \max\{h_T(j) : j \in \text{Children}_T(i)\}$
6.          If $h_T(i) = k$ or $i \equiv T$'s root return subtree rooted at $i$
```

Fig. 6.2: extractTreelet pseudocode.

query set. This is a plausible assumption in active learning scenarios, and a way of adding meaning to the bounds.

Theorem 6.4. *For any integer $k \geq 2$, the number M of mistakes made by* treeCutter *on any graph $G(V, E)$ with $|E| \geq 2|V| - 2 + \frac{|V|^2}{k^2} + \frac{|V|}{k}$ satisfies $\mathbb{E}M \leq \min\{4k + 1, 2D_G\}p|E|$, while the query set size is bounded by $|V| - 1 + \frac{|V|^2}{2k^2} + \frac{|V|}{2k} \leq \frac{|E|}{2}$.*

Proof. By Theorem 6.3, it suffices to show that the length of each path used for predicting the test edges is bounded by $4k + 1$. For each $T' \in \mathcal{T}$, we have $D_{T'} \leq 2k$, since the height of each subree is not bigger than k. Hence, any test edge incident to vertices of the same subtree $T' \in \mathcal{T}$ is predicted (Line 5 in Figure 1) using a path whose length is bounded by $2k < 4k + 1$. Any test edge (u, v) incident to vertices belonging to two different subtrees $T', T'' \in \mathcal{T}$ is predicted (Line 11 in Figure 1) using a path whose length is bounded by $D_{T'} + D_{T''} + 1 \leq 2k + 2k + 1 = 4k + 1$, where the extra $+1$ is due to the query edge (i', i'') connecting T' to T'' (Line 9 in Figure 1).

In order to prove that $|V| - 1 + \frac{|V|^2}{2k^2} + \frac{|V|}{2k}$ is an upper bound on the query set size, observe that each query edge either belongs to T or connects a pair of distinct subtrees contained in $\mathcal{T}$. The number of edges in T is $|V| - 1$, and the number of the remaining query edges is bounded by the number of distinct pairs of subtrees contained in $|\mathcal{T}|$, which can be calculated as follows. First of all, note that only the last subtree returned by EXTRACTTREELET may have a height smaller than k, all the others must have height k. Note also that each subtree of height k must contain at least $k+1$ vertices of V_T, while the subtree of $\mathcal{T}$ having height smaller than k (if present) must contain at least one vertex. Hence, the number of distinct pairs of subtrees contained in $\mathcal{T}$ can be upper bounded by

$$\frac{|\mathcal{T}|(|\mathcal{T}|-1)}{2} \le \frac{1}{2}\Big(\frac{|V|-1}{k+1}+1\Big)\Big(\frac{|V|-1}{k+1}\Big) \le \frac{|V|^2}{k^2}+\frac{|V|}{k} \ .$$

This shows that the query set size cannot be larger than $|V|-1+\frac{|V|^2}{2k^2}+\frac{|V|}{2k}$.

Finally, observe that $D_T \le 2D_G$ because of the breadth-first visit generating T. If $D_T \le k$, the subroutine EXTRACTTREELET is invoked only once, and the algorithm does not ask for any additional label of $E_G \setminus E_T$ (the query set size equals $|V|-1$). In this case $\mathbb{E}M$ is clearly upper bounded by $2D_G\, p|E|$.

6.2.1 Refinements

We now refine the simple argument leading to TREECUTTER, and present our active link classifier. The pseudocode of our refined algorithm, called STARMAKER, follows that of Figure 6.1 with the following differences: Line 1 is dropped (i.e., STARMAKER does not draw an initial spanning tree), and the call to EXTRACTTREELET in Line 3 is replaced by a call to EXTRACTSTAR. This new subroutine just selects the star T' centered on the node of G having largest degree, and queries all labels of the edges in $E_{T'}$. The next result shows that this algorithm gets a *constant* optimality factor while using a query set of size $\mathcal{O}(|V|^{3/2})$.

Theorem 6.5. *The number M of mistakes made by* STARMAKER *on any given graph $G(V,E)$ with $|E| \ge 2|V|-2+2|V|^{\frac{3}{2}}$ satisfies $\mathbb{E}M \le 5\,p|E|$, while the query set size is upper bounded by $|V|-1+|V|^{\frac{3}{2}} \le \frac{|E|}{2}$.*

Proof. In order to prove the claimed mistake bound, it suffices to show that each test edge is predicted with a path whose length is at most 5. This is easily seen by the fact that summing the diameter of two stars plus the query edge (i',i'') that connects them is equal to $2+2+1=5$, which is therefore the diameter of the tree made up by two stars connected by the additional query edge.

We continue by bounding from the above the query set size. Let S_j be the j-th star returned by the j-th call to EXTRACTSTAR. The overall number of query edges can be bounded by $|V|-1+z$, where $|V|-1$ serves as an upper bound on the number of edges forming all the stars output by EXTRACTSTAR, and z is the sum over $j = 1, 2, \ldots$ of the number of stars $S_{j'}$ with $j' > j$ (i.e., j' is created later than j) connected to S_j by at least one edge.

Now, for any given j, the number of stars $S_{j'}$ with $j' > j$ connected to S_j by at least one edge cannot be larger that $\min\{|V|, |V_{S_j}|^2\}$. To see this, note that if there were a leaf q of S_j connected to more than $|V_{S_j}| - 1$ vertices not previously included in any star, then EXTRACTSTAR would have returned a star centered in q instead. The repeated execution of EXTRACTSTAR can indeed be seen as partitioning V. Let $\mathcal{P}$ be the set of all partitions of V. With this notation in hand, we can bound z as follows:

$$z \le \max_{P \in \mathcal{P}} \sum_{j=1}^{|P|} \min\left\{z_j^2(P), |V|\right\} \tag{6.2}$$

where $z_j(P)$ is the number of nodes contained in the the j-th element of the partition P, corresponding to the number of nodes in S_j. Since $\sum_{j=1}^{|P|} z_j(P) = |V|$ for any $P \in \mathcal{P}$, it is easy to see that the partition P^* maximizing the above expression is such that $z_j(P^*) = \sqrt{|V|}$ for all j, implying $|P^*| = \sqrt{|V|}$. We conclude that the query set size is bounded by $|V| - 1 + |V|^{\frac{3}{2}}$, as claimed.

Finally, we combine STARMAKER with TREECUTTER so as to obtain an algorithm, called TREELETSTAR, that can work with query sets smaller than $|V| - 1 + |V|^{\frac{3}{2}}$ labels. TREELETSTAR is parameterized by an integer k and follows Lines 1–6 of Figure 6.1 creating a set $\mathcal{T}$ of trees through repeated calls to EXTRACTTREELET. Lines 7–11 are instead replaced by the following procedure: a graph $G' = (V_{G'}, E_{G'})$ is created such that: (1) each node in $V_{G'}$ corresponds to a tree in $\mathcal{T}$, (2) there exists an edge in $E_{G'}$ if and only if the two corresponding trees of $\mathcal{T}$ are connected by at least one edge of E_G. Then, EXTRACTSTAR is used to generate a set $\mathcal{S}$ of stars of vertices of G', i.e., stars of trees of $\mathcal{T}$. Finally, for each pair of distinct stars $S', S'' \in \mathcal{S}$ connected by at least one edge in E_G, the label of an arbitrary edge in $E_G(S', S'')$ is queried. The remaining edges are all predicted.

Theorem 6.6. *For any integer $k \ge 2$ and for any graph $G = (V, E)$ with $|E| \ge 2|V| - 2 + 2\left(\frac{|V|-1}{k} + 1\right)^{\frac{3}{2}}$, the number M of mistakes made by* TREELETSTAR(k) *on G satisfies*

$$v\mathbb{E}M = \mathcal{O}(\min\{k, D_G\})\, p|E|$$

while the query set size is bounded by $|V| - 1 + \left(\frac{|V|-1}{k} + 1\right)^{\frac{3}{2}} \le \frac{|E|}{2}$.

Proof. If the height of T is not larger than k, then EXTRACTTREELET is invoked only once and $\mathcal{T}$ contains the single tree T. The statement then trivially follows from the fact that the length of the longest path in T cannot be larger than twice the diameter of G. Observe that in this case $|V_{G'}| = 1$.

We continue with the case when the height of T is larger than k. We have that the length of each path used in the prediction phase is bounded by 1 plus the sum of the diameters of two trees of $\mathcal{T}$. Since these two trees are not higher than k, the mistake bound follows from Theorem 6.3.

Finally, we combine the upper bound on the query set size in the statement of Theorem 3 with the fact that each vertex of $V_{G'}$ corresponds to a tree of $\mathcal{T}$ containing at least $k+1$ vertices of G. This implies $|V_{G'}| \le \frac{|V|}{k+1}$, and the claim on the query set size of treeletStar follows.

Hence, even if D_G is large, setting $k = |V|^{1/3}$ yields a $\mathcal{O}(|V|^{1/3})$ optimality factor just by querying $\mathcal{O}(|V|)$ edges. On the other hand, a truly constant optimality factor is obtained by querying as few as $\mathcal{O}(|V|^{3/2})$ edges (provided the graph has sufficiently many edges). As a direct consequence (and surprisingly enough), on graphs which are only moderately dense we need not observe too many edges in order to achieve a constant optimality factor. It is instructive to compare the bounds obtained by treeletStar to the ones we can achieve by using the cccc algorithm of [26], or the low-stretch spanning trees given in Theorem 6.2.

Because cccc operates within a harder adversarial setting, it is easy to show that Theorem 9 in [26] extends to the p-stochastic assignment model by replacing $\Delta_2(Y)$ with $p|E|$ therein.[2] The resulting optimality factor is of order $\left(\frac{1-\alpha}{\alpha}\right)^{\frac{3}{2}}\sqrt{|V|}$, where $\alpha \in (0,1]$ is the fraction of queried edges out of the total number of edges. A quick comparison to Theorem 6.6 reveals that treeletStar achieves a sharper mistake bound for any value of α. For instance, in order to obtain an optimality factor which is lower than $\sqrt{|V|}$, cccc has to query in the worst case a fraction of edges that goes to one as $|V| \to \infty$. On top of this, our algorithms are faster and easier to implement —see Section 6.2.2.

Next, we compare to query sets produced by low-stretch spanning trees. A low-stretch spanning tree achieves a polylogarithmic optimality factor by querying $|V|-1$ edge labels. The results in [41] show that we cannot hope to get a better optimality factor using a single low-stretch spanning tree combined by the analysis in (6.1). For a comparable amount $\Theta(|V|)$ of queried labels, Theorem 6.6 offers the larger optimality factor $|V|^{1/3}$. However, we can get a *constant* optimality factor by increasing the query set size to $\mathcal{O}(|V|^{3/2})$. It is not clear how multiple low-stretch trees could be combined to get a similar scaling.

[2] This theoretical comparison is admittedly unfair, as cccc has been designed to work in a harder setting than p-stochastic. Unfortunately, we are not aware of any other general active learning scheme for link classification to compare with.

6.2.2 Complexity analysis and implementation

We now compute bounds on time and space requirements for our three algorithms. Recall the different lower bound conditions on the graph density that must hold to ensure that the query set size is not larger than the test set size. These were $|E| \geq 2|V| - 2 + \frac{|V|^2}{k^2} + \frac{|V|}{k}$ for $\textsc{treeCutter}(k)$ in Theorem 6.4, $|E| \geq 2|V| - 2 + 2|V|^{\frac{3}{2}}$ for $\textsc{starMaker}$ in Theorem 6.5, and $|E| \geq 2|V| - 2 + 2\Big(\frac{|V|-1}{k} + 1\Big)^{\frac{3}{2}}$ for $\textsc{treeletStar}(k)$ in Theorem 6.6.

Theorem 6.7. *For any input graph $G = (V, E)$ which is dense enough to ensure that the query set size is no larger than the test set size, the total time needed for predicting all test labels is:*

$$\begin{aligned} &\mathcal{O}(|E|) && \text{for } \textsc{treeCutter}(k) \text{ and for all } k \\ &\mathcal{O}\big(|E| + |V|\log|V|\big) && \text{for } \textsc{starMaker} \\ &\mathcal{O}\left(|E| + \frac{|V|}{k}\log\frac{|V|}{k}\right) && \text{for } \textsc{treeletStar}(k) \text{ and for all } k. \end{aligned}$$

In particular, whenever $k|E| = \Omega(|V|\log|V|)$ we have that $\textsc{treeletStar}(k)$ works in constant amortized time. For all three algorithms, the space required is always linear in the input graph size $|E|$.

Proof. The proof is reported in Appendix B.

6.3 Experiments

In this set of experiments we tested the predictive performance of $\textsc{treeCutter}(|V|)$. This corresponds to querying only the edges of the initial spanning tree T and predicting all remaining edges (i, j) via the parity of $\mathrm{P}_T(i, j)$. The spanning tree T used by $\textsc{treeCutter}$ is a shortest-path spanning tree generated by a breadth-first visit of the graph (assuming all edges have unit length). As the choice of the starting node in the visit is arbitrary, we picked the highest degree node in the graph. Finally, we run through the adiacency list of each node in random order, which we empirically observed to improve performance.

6.3.1 Algorithms

Our baseline is the heuristic ASymExp from [60] which, among the many spectral heuristics proposed there, turned out to perform best on all our datasets. With integer parameter z, ASymExp(z) predicts using a spectral transformation of the training sign matrix Y_{train}, whose only non-zero entries are the signs of the training edges. The label of edge (i, j) is predicted using $\bigl(\exp(Y_{\text{train}}(z))\bigr)_{i,j}$. Here $\exp\bigl(Y_{\text{train}}(z)\bigr) = U_z \exp(D_z) U_z^\top$, where $U_z D_z U_z^\top$ is the spectral decomposition of Y_{train} containing only the z largest eigenvalues and their corresponding eigenvectors. Following [60], we ran ASymExp(z) with the values $z = 1, 5, 10, 15$. This heuristic uses the two-clustering bias as follows : expand $\exp(Y_{\text{train}})$ in a series of powers Y^n_{train}. Then each $\bigl(Y^n_{\text{train}}\bigr)_{i,j}$ is a sum of values of paths of length n between i and j. Each path has value 0 if it contains at least one test edge, otherwise its value equals the product of queried labels on the path edges. Hence, the sign of $\exp(Y_{\text{train}})$ is the sign of a linear combination of path values, each corresponding to a prediction consistent with the two-clustering bias —compare this to the multiplicative rule used by treeCutter. Note that ASymExp and the other spectral heuristics from [60] have all running times of order $\Omega\bigl(|V|^2\bigr)$.

6.3.2 Datasets

We performed a first set of experiments on synthetic signed graphs created from a subset of the USPS digit recognition dataset. We randomly selected 500 examples labeled "1" and 500 examples labeled "7" (these two classes are not straightforward to tell apart). Then, we created a graph using a k-NN rule with $k = 100$. The edges were labeled as follows: all edges incident to nodes with the same USPS label were labeled $+1$; all edges incident to nodes with different USPS labels were labeled -1. Finally, we randomly pruned the positive edges so to achieve an unbalance of about 20% between the two classes.[3] Starting from this edge label assignment, which is consistent with the two-clustering associated with the USPS labels, we generated a p-stochastic label assignment by flipping the labels of a random subset of the edges. Specifically, we used the three following synthetic datasets:

DELTA0: No flippings ($p = 0$), 1,000 nodes and 9,138 edges;

DELTA100: 100 randomly chosen labels of DELTA0 are flipped;

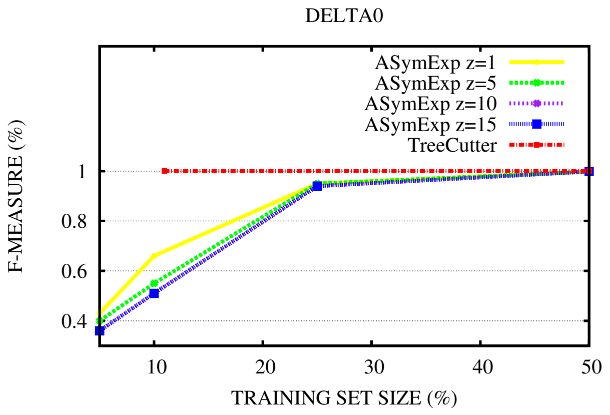
DELTA250: 250 randomly chosen labels of DELTA0 are flipped.

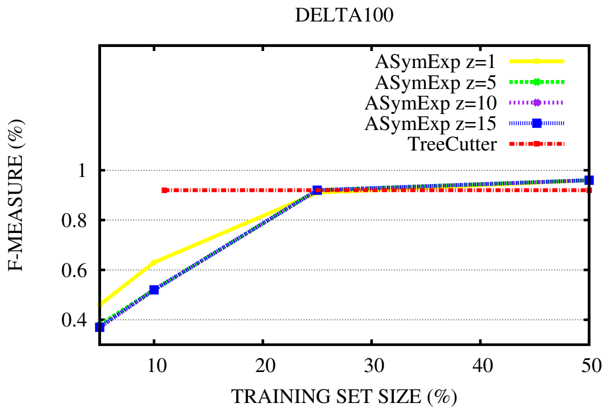
We also used three real-world datasets:

MOVIELENS: A signed graph we created using Movielens ratings.[4] We

[3] This is similar to the class unbalance of real-world signed networks —see below.

[4] `www.grouplens.org/system/files/ml-1m.zip`.





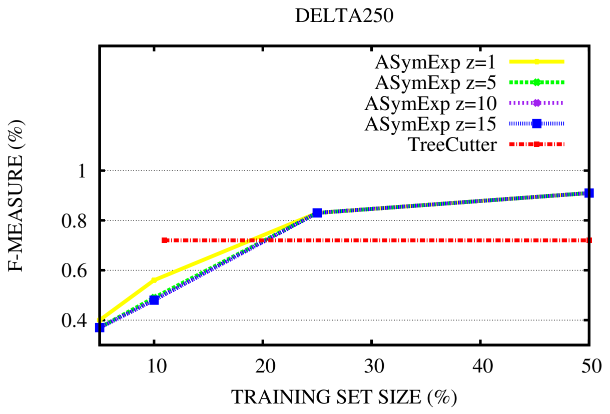
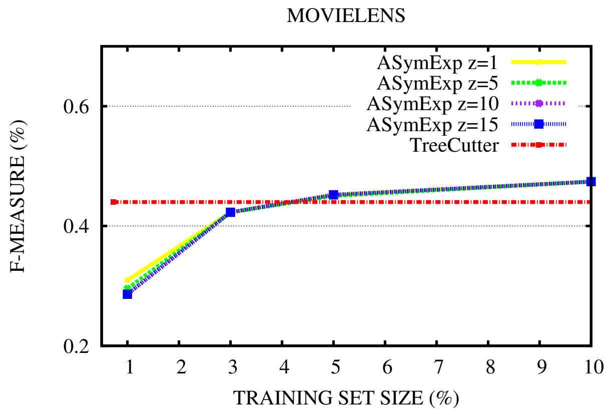
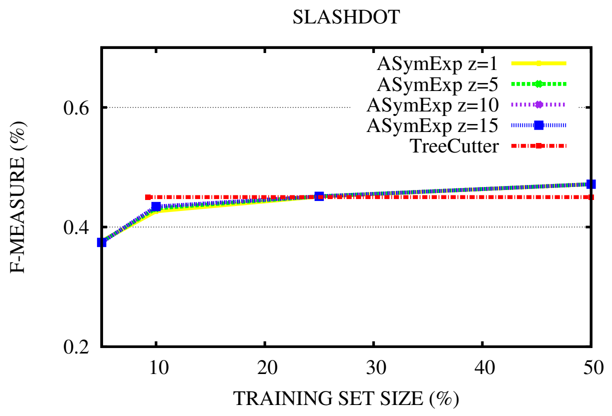


Fig. 6.3: F-measure against training set size for TREECUTTER($|V|$) and ASymExp(z) with different values of z on both synthetic and real-world datasets. By construction, TREECUTTER never makes a mistake when the labeling is consistent with a two-clustering. So on DELTA0 TREECUTTER does not make mistakes whenever the training set contains at least one spanning tree. With the exception of EPINIONS, TREECUTTER outperforms ASymExp using a much smaller training set. We conjecture that ASymExp responds to the bias not as well as TREECUTTER, which on the other hand is less robust than ASymExp to bias violations (supposedly, the labeling of EPINIONS).





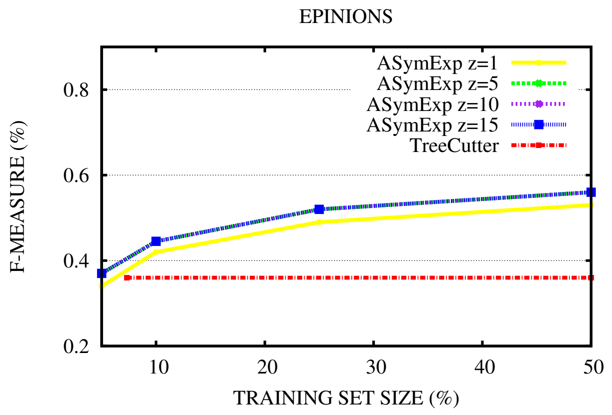


Fig. 6.4: F-measure against training set size for TREECUTTER($|V|$) and ASymExp(z) with different values of z on both synthetic and real-world datasets. By construction, TREECUTTER never makes a mistake when the labeling is consistent with a two-clustering. So on DELTA0 TREECUTTER does not make mistakes whenever the training set contains at least one spanning tree. With the exception of EPINIONS, TREECUTTER outperforms ASymExp using a much smaller training set. We conjecture that ASymExp responds to the bias not as well as TREECUTTER, which on the other hand is less robust than ASymExp to bias violations (supposedly, the labeling of EPINIONS).

first normalized the ratings by subtracting from each user rating the average rating of that user. Then, we created a user-user matrix of cosine distance similarities. This matrix was sparsified by zeroing each entry smaller than 0.1 and removing all self-loops. Finally, we took the sign of each non-zero entry. The resulting graph has 6,040 nodes and 824,818 edges (12.6% of which are negative).

SLASHDOT: The biggest strongly connected component of a snapshot of the Slashdot social network,[5] similar to the one used in [60]. This graph has 26,996 nodes and 290,509 edges (24.7% of which are negative).

EPINIONS: The biggest strongly connected component of a snapshot of the Epinions signed network,[6] similar to the one used in [62, 66]. This graph has 41,441 nodes and 565,900 edges (26.2% of which are negative).

Slashdot and Epinions are originally directed graphs. We removed the reciprocal edges with mismatching labels (which turned out to be only a few), and considered the remaining edges as undirected.

The following table summarizes the key statistics of each dataset: Neg. is the fraction of negative edges, $|V|/|E|$ is the fraction of edges queried by TREECUTTER($|V|$), and Avgdeg is the average degree of the nodes of the network.

Dataset	$\|V\|$	$\|E\|$	Neg.	$\|V\|/\|E\|$	Avgdeg
DELTA0	1000	9138	21.9%	10.9%	18.2
DELTA100	1000	9138	22.7%	10.9%	18.2
DELTA250	1000	9138	23.5%	10.9%	18.2
SLASHDOT	26996	290509	24.7%	9.2%	21.6
EPINIONS	41441	565900	26.2%	7.3%	27.4
MOVIELENS	6040	824818	12.6%	0.7%	273.2

6.3.3 Results

Our results are summarized in Figure 6.3, where we plot F-measure (preferable to accuracy due to the class unbalance) against the fraction of training (or query) set size. On all datasets, but MOVIELENS, the training set size for ASymExp ranges across the values 5%, 10%, 25%, and 50%. Since MOVIELENS has a higher density, we decided to reduce those fractions to 1%, 3%, 5% and 10%. TREECUTTER($|V|$) uses a single spanning tree, and thus we only have a single query set size value. All results are averaged over ten runs of the algorithms. The randomness in ASymExp is due to the random draw

[5] `snap.stanford.edu/data/soc-sign-Slashdot081106.html`.

[6] `snap.stanford.edu/data/soc-sign-epinions.html`.

of the training set. The randomness in treeCutter($|V|$) is caused by the randomized breadth-first visit.

Part IV
Networks of bandits

Only a fool learns from his own mistakes.
The wise man learns from the mistakes of others.
– Otto von Bismarck

Chapter 7
Network of bandits

In this chapter, we consider an extension of the Contextual Bandit problem where there are multiple learners connected by a network, and we assume that strongly connected nodes are similar. The learners share the feedback signal with each other, cooperating in this way to improve their future predictions.
We provide a regret analysis for the presented algorithm and an extensive set of experiments on synthetic and real-world datasets, comparing the original algorithm and two its scalable variants with the state of the art in the field.

The content of this chapter is a joint work with Nicolò Cesa-Bianchi and Claudio Gentile.

7.1 Problem setup

We assume the social relationships over users are encoded as a *known* undirected and connected graph $G = (V, E)$, where $V = \{1, \dots, n\}$ represents a set of n users, and the edges in E represent the social links over pairs of users. Recall that a graph G can be equivalently defined in terms of its Laplacian matrix[1]. Learning proceeds in a sequential fashion: At each time step $t = 1, 2, \dots,$ the learner receives a user index $i_t \in V$ together with a set of context vectors $C_{i_t} = \{\boldsymbol{x}_{t,1}, \boldsymbol{x}_{t,2}, \dots, \boldsymbol{x}_{t,c_t}\} \subseteq \mathbb{R}^d$. The learner then selects some $k_t \in C_{i_t}$ to recommend to user i_t and observes some payoff $a_t \in [-1, 1]$, a function of i_t and $\bar{\boldsymbol{x}}_t = \boldsymbol{x}_{t,k_t}$. No assumptions whatsoever are made on the

[1] The Laplacian matrix L is defined as $L = \big[L_{i,j}\big]_{i,j=1}^{n}$, where $L_{i,i}$ is the degree of node i (i.e., the number of edges or the sum of the weights of the edges incident to that node) and, for $i \neq j$, $L_{i,j}$ equals $-w_{ij}$ if $(i, j) \in E$, and 0 otherwise

way index i_t and set C_{i_t} are generated, in that they can arbitrarily depend on past choices made by the algorithm.[2]

A standard modeling assumption for bandit problems with contextual information (one that is also adopted here) is to assume rewards are generated by noisy versions of unknown linear functions of the context vectors. That is, we assume each node $i \in V$ hosts an unknown parameter vector $\boldsymbol{u}_i \in \mathbb{R}^d$, and that the reward value $a_i(\boldsymbol{x})$ associated with node i and context vector $\boldsymbol{x} \in \mathbb{R}^d$ is given by the random variable $a_i(\boldsymbol{x}) = \boldsymbol{u}_i^\top \boldsymbol{x} + \epsilon_i(\boldsymbol{x})$, where $\epsilon_i(\boldsymbol{x})$ is a conditionally zero-mean and bounded variance noise term. Specifically, denoting by $\mathbb{E}_t[\,\cdot\,]$ the conditional expectation $\mathbb{E}\big[\,\cdot \,\big|\, (i_1, C_{i_1}, a_1), \ldots, (i_{t-1}, C_{i_{t-1}}, a_{t-1})\,\big]$, we take the general approach of [1], and assume that for any fixed $i \in V$ and $\boldsymbol{x} \in \mathbb{R}^d$, the variable $\epsilon_i(\boldsymbol{x})$ is conditionally sub-Gaussian with variance parameter $\sigma^2 > 0$, namely, $\mathbb{E}_t\big[\exp(\gamma\,\epsilon_i(\boldsymbol{x}))\big] \le \exp\big(\sigma^2\,\gamma^2/2\big)$ for all $\gamma \in \mathbb{R}$ and all $\boldsymbol{x}, i$. This implies $\mathbb{E}_t[\epsilon_i(\boldsymbol{x})] = 0$ and $\mathbb{V}_t\big[\epsilon_i(\boldsymbol{x})\big] \le \sigma^2$, where $\mathbb{V}_t[\cdot]$ is a shorthand for the conditional variance $\mathbb{V}\big[\,\cdot \,\big|\, (i_1, C_{i_1}, a_1), \ldots, (i_{t-1}, C_{i_{t-1}}, a_{t-1})\,\big]$ of the variable at argument. So we clearly have $\mathbb{E}_t[a_i(\boldsymbol{x})] = \boldsymbol{u}_i^\top \boldsymbol{x}$ and $\mathbb{V}_t\big[a_i(\boldsymbol{x})\big] \le \sigma^2$. Therefore, $\boldsymbol{u}_i^\top \boldsymbol{x}$ is the expected reward observed at node i for context vector $\boldsymbol{x}$. In the special case when the noise $\epsilon_i(\boldsymbol{x})$ is a bounded random variable taking values in the range $[-1, 1]$, this implies $\mathbb{V}_t[a_i(\boldsymbol{x})] \le 4$.

The regret r_t of the learner at time t is the amount by which the average reward of the best choice in hindsight at node i_t exceeds the average reward of the algorithm's choice, i.e.,

$$r_t = \left(\max_{\boldsymbol{x} \in C_{i_t}} \ \boldsymbol{u}_{i_t}^\top \boldsymbol{x}\right) - \boldsymbol{u}_{i_t}^\top \bar{\boldsymbol{x}}_t \ .$$

The goal of the algorithm is to bound with high probability (over the noise variables ϵ_{i_t}) the cumulative regret $\sum_{t=1}^T r_t$ for the given sequence of nodes $i_1, \ldots, i_T$ and observed context vector sets $C_{i_1}, \ldots, C_{i_T}$.

We model the similarity among users in V by making the assumption that nearby users hold similar underlying vectors $\boldsymbol{u}_i$, so that reward signals received at a given node i_t at time t are also, to some extent, informative to learn the behavior of other users j connected to i_t within G. We make this more precise by taking the perspective of known multitask learning settings (e.g., [20]), and assume that

$$\sum_{(i,j) \in E} \|\boldsymbol{u}_i - \boldsymbol{u}_j\|^2 \tag{7.1}$$

is small compared to $\sum_{i \in V} \|\boldsymbol{u}_i\|^2$, where $\|\cdot\|$ denotes the standard Euclidean norm of vectors. That is, although (7.1) may possibly contain a quadratic number of terms, the closeness of vectors lying on adjacent nodes in G makes this sum comparatively smaller than the actual length of such vec-

[2] Formally, i_t and C_{i_t} can be arbitrary (measurable) functions of past rewards $a_1, \ldots, a_{t-1}$, indices $i_1, \ldots, i_{t-1}$, and sets $C_{i_1}, \ldots, C_{i_{t-1}}$.

tors. This will be our working assumption throughout, one that motivates the Laplacian-regularized algorithm presented in Section 7.2, and empirically tested in Section 7.3.

7.2 Algorithm and regret analysis

Our bandit algorithm maintains at time t an estimate $\boldsymbol{w}_{i,t}$ for vector $\boldsymbol{u}_i$. Vectors $\boldsymbol{w}_{i,t}$ are updated based on the reward signals as in a standard linear bandit algorithm (e.g., [33]) operating on the context vectors contained in C_{i_t}. Every node i of G hosts a linear bandit algorithm like the one described in Figure 7.1.

```
Input: Parameter $\delta$;
Init: $\boldsymbol{b}_0 = \boldsymbol{0} \in \mathbb{R}^d$ and $M_0 = I \in \mathbb{R}^{d \times d}$;
for $t = 1, 2, \ldots, T$ do
    Set $\boldsymbol{w}_{t-1} = M_{t-1}^{-1} \boldsymbol{b}_{t-1}$;
    Get context $C_t = \{\boldsymbol{x}_{t,1}, \ldots, \boldsymbol{x}_{t,c_t}\}$;
    Set
        $k_t = \operatorname{argmax}_{k=1,\ldots,c_t} \left(\boldsymbol{w}_{t-1}^\top \boldsymbol{x}_{t,k} + \mathrm{CB}(\boldsymbol{x}_{t,k})\right)$
    where
        $\mathrm{CB}(\boldsymbol{x}_{t,k_t}) = \sqrt{\boldsymbol{x}_{t,k_t}^\top M_{t-1}^{-1} \boldsymbol{x}_{t,k_t}} \left(\sigma \sqrt{\ln \frac{|M_t|}{\delta}} + \|U\|\right)$
    Set $\bar{\boldsymbol{x}}_t = \boldsymbol{x}_{t,k_t}$;
    Observe reward $a_t \in [-1, 1]$;
    Update
•       $M_t = M_{t-1} + \bar{\boldsymbol{x}}_t \bar{\boldsymbol{x}}_t^\top$,
•       $\boldsymbol{b}_t = \boldsymbol{b}_{t-1} + a_t \bar{\boldsymbol{x}}_t$ .
end for
```

Fig. 7.1: Pseudocode of the linear bandit algorithm sitting at each node i.

The algorithm in Figure 7.1 operating at a given node of G receives in input an exploration parameter $\alpha > 0$, setting the exploration-vs-exploitation trade-off at that node. The algorithm maintains at time t a prototype vector $\boldsymbol{w}_t$ which is the result of a standard linear least-squares approximation to the unknown paremeter vector $\boldsymbol{u}$ associated with the node under consideration. In particular, $\boldsymbol{w}_{t-1}$ is obtained by multiplying the inverse correlation matrix M_{t-1} and the bias vector $\boldsymbol{b}_{t-1}$. At each time $t = 1, 2, \ldots,$ the algorithm receives context vectors $\boldsymbol{x}_{t,1}, \ldots, \boldsymbol{x}_{t,c_t}$ contained in C_t, and must select one among them. The linear bandit algorithm selects $\bar{\boldsymbol{x}}_t = \boldsymbol{x}_{t,k_t}$ as the vector in C_t that maximizes an upper-confidence-corrected estimation of the expected reward achieved over context vectors $\boldsymbol{x}_{t,k}$. The estimation is based on the

```
**Input**: Parameter $\delta$;
**Init**: $\boldsymbol{b}_0 = \mathbf{0} \in \mathbb{R}^{dn}$ and $M_0 = I \in \mathbb{R}^{dn \times dn}$;
**for** $t = 1, 2, \ldots, T$ **do**
    Set $\boldsymbol{w}_{t-1} = M_{t-1}^{-1} \boldsymbol{b}_{t-1}$;
    Get $i_t \in V$, context $C_{i_t} = \{\boldsymbol{x}_{t,1}, \ldots, \boldsymbol{x}_{t,c_t}\}$;
    Construct vectors $\boldsymbol{\phi}_{i_t}(\boldsymbol{x}_{t,1}), \ldots, \boldsymbol{\phi}_{i_t}(\boldsymbol{x}_{t,c_t})$, and *modified* vectors $\widetilde{\boldsymbol{\phi}}_{t,1}, \ldots, \widetilde{\boldsymbol{\phi}}_{t,c_t}$,
    where
        $$\widetilde{\boldsymbol{\phi}}_{t,k} = A_{\otimes}^{-1/2} \boldsymbol{\phi}_{i_t}(\boldsymbol{x}_{t,k}), \quad k = 1, \ldots, c_t;$$
    Set $k_t = \operatorname*{argmax}_{k=1,\ldots,c_t} \left( \boldsymbol{w}_{t-1}^\top \widetilde{\boldsymbol{\phi}}_{t,k} + \textsc{cb}(\widetilde{\boldsymbol{\phi}}_{t,k}) \right)$ where
        $$\textsc{cb}(\widetilde{\boldsymbol{\phi}}_{t,k}) = \sqrt{\widetilde{\boldsymbol{\phi}}_{t,k}^\top M_{t-1}^{-1} \widetilde{\boldsymbol{\phi}}_{t,k}} \left( \sigma \sqrt{\ln \frac{|M_t|}{\delta}} + \|\widetilde{U}\| \right)$$
    Observe reward $a_t \in [-1, 1]$ at node $i_t$;
    Update
•       $M_t = M_{t-1} + \widetilde{\boldsymbol{\phi}}_{t,k_t} \widetilde{\boldsymbol{\phi}}_{t,k_t}^\top$,
•       $\boldsymbol{b}_t = \boldsymbol{b}_{t-1} + a_t \widetilde{\boldsymbol{\phi}}_{t,k}$ .
**end for**
```

Fig. 7.2: Pseudocode of the GOB.Lin algorithm.

current $\boldsymbol{w}_{t-1}$, while the upper confidence level $\textsc{cb}(\boldsymbol{x})$ is suggested by the standard analysis of linear bandit algorithms —see, e.g., [1, 33, 35, 37]. Once the actual reward a_t associated with $\bar{\boldsymbol{x}}_t$ is observed, the algorithm uses $\bar{\boldsymbol{x}}_t$ for updating M_{t-1} to M_t via a rank-one adjustment, and $\boldsymbol{b}_{t-1}$ to $\boldsymbol{b}_t$ via an additive update whose learning rate is precisely a_t. This algorithm can be seen as a version of LinUCB [33], a linear bandit algorithm derived from LinRel [7].

We now turn to describing our **GOB.Lin** (Gang Of Bandits, Linear version) algorithm. GOB.Lin lets the algorithm in Figure 7.1 operate on each node i of G (we should then add subscript i throughout, replacing $\boldsymbol{w}_t$ by $\boldsymbol{w}_{i,t}$, M_t by $M_{i,t}$, and so forth). The updates $M_{i,t-1} \to M_{i,t}$ and $\boldsymbol{b}_{i,t-1} \to \boldsymbol{b}_{i,t}$ are performed at node i through vector $\bar{\boldsymbol{x}}_t$ both when $i = i_t$ (i.e., when node i is the one which the context vectors in C_{i_t} refer to) and to a lesser extent when $i \neq i_t$ (i.e., when node i is not the one which the vectors in C_{i_t} refer to). This is because, as we said, the payoff a_t received for node i_t is somehow informative also for all other nodes $i \neq i_t$. In other words, because we are assuming the underlying parameter vectors $\boldsymbol{u}_i$ are close to each other, we should let the corresponding prototype vectors $\boldsymbol{w}_{i,t}$ undergo similar updates, so as to also keep the $\boldsymbol{w}_{i,t}$ close to each other over time.

With this in mind, we now describe GOB.Lin in more detail. It is convenient to first introduce some extra matrix notation. Let $A = I_n + L$, where L is the Laplacian matrix associated with G, and I_n is the $n \times n$ identity matrix.

Set $A_\otimes = A \otimes I_d$, the Kronecker product[3] of matrices A and I_d. Moreover, the "compound" descriptor for the pairing $(i, \boldsymbol{x})$ is given by the long (and sparse) vector $\boldsymbol{\phi}_i(\boldsymbol{x}) \in \mathbb{R}^{dn}$ defined as

$$\boldsymbol{\phi}_i(\boldsymbol{x})^\top = \Big(\underbrace{0, \ldots, 0}_{(i-1)d \text{ times}}, \boldsymbol{x}^\top, \underbrace{0, \ldots, 0}_{(n-i)d \text{ times}} \Big) .$$

With the above notation handy, a compact description of GOB.Lin is presented in Figure 7.2, where we deliberately tried to mimic the pseudocode of Figure 7.1.

We now explain how the modified long vectors $\widetilde{\boldsymbol{\phi}}_{t,k} = A_\otimes^{-1/2} \boldsymbol{\phi}_{i_t}(\boldsymbol{x}_{t,k})$ act in the update of matrix M_t and vector $\boldsymbol{b}_t$. First, observe that if $A_\otimes$ were the identity matrix then, according to how the long vectors $\boldsymbol{\phi}_{i_t}(\boldsymbol{x}_{t,k})$ are defined, M_t would be a block-diagonal matrix $M_t = \text{diag}(D_1, \ldots, D_n)$, whose i-th block D_i is the $d \times d$ matrix $D_i = I_d + \sum_{t\,:\,k_t = i} \boldsymbol{x}_t \boldsymbol{x}_t^\top$. Similarly, $\boldsymbol{b}_t$ would be the dn-long vector whose i-th d-dimensional block contains $\sum_{t\,:\,k_t=i} a_t \boldsymbol{x}_t$. This would be equivalent to running n independent linear bandit algorithms (Figure 7.1), one per node of G. Now, because $A_\otimes$ is not the identity, but contains graph G represented through its Laplacian matrix, the selected vector $\boldsymbol{x}_{t,k_t} \in C_{i_t}$ for node i_t gets spread via $A_\otimes^{-1/2}$ from the i_t-th block over all other blocks, thereby making the contextual information contained in $\boldsymbol{x}_{t,k_t}$ available to update the internal status of all other nodes. Yet, the only reward signal observed at time t is the one available at node i_t. A theoretical analysis of GOB.Lin relying on the learning model of Section 7.1 is sketched in Section 7.2.1.
Intuitively, our algorithm is multitask linear least squares where each node is a task. Similarity among the tasks (closeness of $\boldsymbol{u}$ vectors) correspond to the weights on the graph's edges. Heavy edges mean similar tasks and vice versa. Moreover, if G is the a clique, then the vectors $\boldsymbol{\phi}_{t,k}$ (built from $\boldsymbol{x}_{t,k}$) are sparse block vectors, but vectors $\widetilde{\boldsymbol{\phi}}_{t,k}$ are dense vectors. In the latter case, the i_t-th block in the vector gets multiplied by some fraction of unity and, in all other blocks, the vector $\boldsymbol{x}_{t,k}$ still occurs, but multiplied by a smaller fraction of unity. These different multipliers are given by the similarity provided by the graph and encoden in the matrix $A_\otimes$. This forces an update on all nodes of G, based on the payoff observed at node i_t, but weighting them differently. In this specific example, the update at node i_t is worth roughly $\sqrt{n}$ times the update made at any other node.
GOB.Lin's running time is mainly affected by the inversion of the $dn \times dn$ matrix M_t, which can be performed in $\mathcal{O}\big((dn)^2\big)$ time per round by using well-known formulas for incremental matrix inversions. The same quadratic dependence holds for memory requirements. In our experiments, we observed that projecting the contexts on the principal components improved perfor-

[3] The Kronecker product between two matrices $M \in \mathbb{R}^{m \times n}$ and $N \in \mathbb{R}^{q \times r}$ is the block matrix $M \otimes N$ of dimension $mq \times nr$ whose block on row i and column j is the $q \times r$ matrix $M_{i,j}N$.

mance. Hence, the quadratic dependence on the context vector dimension d is not really hurting us in practice. On the other hand, the quadratic dependence on the number of nodes n may be a significant limitation to GOB.Lin's practical deployment. In the next section, we show that simple graph compression schemes (like node clustering) can conveniently be applied to both reduce edge noise and bring the algorithm to reasonable scaling behaviors.

7.2.1 Regret Analysis

We now provide a regret analysis for GOB.Lin that relies on the high probability analysis contained in [1] (Theorem 2 therein). The analysis can be seen as a combination of the multitask kernel contained in, e.g., [20, 68, 43] and a version of the linear bandit algorithm described and analyzed in [1].

Theorem 7.1. *Let the* GOB.Lin *algorithm of Figure 7.2 be run on graph* $G = (V, E)$, $V = \{1, \ldots, n\}$, *hosting at each node* $i \in V$ *vector* $\boldsymbol{u}_i \in \mathbb{R}^d$. *Define*

$$L(\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n) = \sum_{i \in V} \|\boldsymbol{u}_i\|^2 + \sum_{(i,j) \in E} \|\boldsymbol{u}_i - \boldsymbol{u}_j\|^2 \ .$$

Being $|\cdot|$ *the determinant of the matrix at argument. Let also the sequence of context vectors* $\boldsymbol{x}_{t,k}$ *be such that* $\|\boldsymbol{x}_{t,k}\| \le B$, *for all* $k = 1, \ldots, c_t$, *and* $t = 1, \ldots, T$. *Then the cumulative regret satisfies*

$$\sum_{t=1}^{T} r_t \le 2 \sqrt{T \left(2\sigma^2 \ln \frac{|M_T|}{\delta} + 2L(\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n)\right) (1 + B^2) \ln |M_T|}$$

with probability at least $1 - \delta$.

Proof. The proof is given in Appendix C.

Compared to running n independent bandit algorithms (which corresponds to $A_\otimes$ being the identity matrix), the bound in the above theorem has an extra term $\sum_{(i,j) \in E} \|\boldsymbol{u}_i - \boldsymbol{u}_j\|^2$, which we assume small according to our working assumption. However, the bound has also a significantly smaller log determinant $\ln |M_T|$ on the resulting matrix M_T, due to the construction of $\widetilde{\phi}_{t,k}$ via $A_\otimes^{-1/2}$. In order to make this advantage more clear, we report two extreme cases (similar arguments can be found in [20]):

1. When G has no edges then trace $\ln |M_t| \le d\, n \, \log(1 + \frac{tB^2}{dn})$.
2. When G is the complete graph $\ln |M_t| \le d\, n \, \log(1 + \frac{2tB^2}{d\, n\, (n+1)})$.

The first case represent the independent bandits case, where there are n instances of LinUCB that do not share the feedback (later called LinUCB-IND). In the second case we have GOB.Lin with a fully connected graph where feedback is shared among all the nodes and all the nodes have the same $\boldsymbol{u}$. In this case, the determinant is about a factor n smaller than the corresponding term for the n independent bandit case.

7.3 Experiments

In this section, we present an empirical comparison of GOB.Lin (and its variants) to linear bandit algorithms which do not exploit the relational information provided by the graph. For the purpose of our experiments we approximate $\textsc{cb}(\boldsymbol{x})$ with $\textsc{acb}(\boldsymbol{x})$ defined as

$$\textsc{acb}(\boldsymbol{x}) = \alpha \sqrt{\boldsymbol{x}^\top M_{t-1}^{-1} \boldsymbol{x} \, \log(t+1)}$$

where the factor α is used as tuning parameter. We did not use CB in our experiments because the it depends on unknown quantities and since it comes from an upper bound analysis, it seems to be too conservative. Moreover, we would like to stress that ACB and CB have a very similar dependence on time, and our preliminary experiments show that this approximation do not affect the predictive performances of the algorithms (up to a re-tuning of α), while it speeds up the computation.

7.3.1 Datasets

We tested our algorithm and its competitors on a synthetic dataset and two freely available real-world datasets extracted from the social bookmarking web service Delicious and from the music streaming service Last.fm. These datasets are structured as follows:

- **4Cliques.** This is an artificial dataset whose graph contains four cliques of 25 nodes each to which we added graph noise. This noise consists in picking a random pair of nodes and deleting or creating an edge between them. More precisely, we created a $n \times n$ symmetric noise matrix of random numbers in $[0, 1]$, and we selected a threshold value such that the expected number of matrix elements above this value is exactly some chosen noise rate parameter. Then we set to 1 all the entries whose content is above the threshold, and to zero the remaining ones. Finally, we XORed the noise matrix with the graph adjacency matrix, thus obtaining a noisy version of the original graph.

- **Last.fm.** This is a social network containing 1,892 nodes and 12,717 edges. There are 17,632 items (artists), described by 11,946 tags. The dataset contains information about the listened artists, and we used this information in order to create the payoffs: if a user listened to an artist at least once the payoff is 1, otherwise the payoff is 0.

- **Delicious.** This is a network with 1,861 nodes and 7,668 edges. There are 69,226 items (URLs) described by 53,388 tags. The payoffs were created using the information about the bookmarked URLs for each user: the payoff is 1 if the user bookmarked the URL, otherwise the payoff is 0.

Last.fm and Delicious were created by the Information Retrieval group at Universidad Autonoma de Madrid for the HetRec 2011 Workshop [15] with the goal of investigating the usage of heterogeneous information in recommendation systems.[4] These two networks are structurally different: on Delicious, payoffs depend on users more strongly than on Last.fm. In other words, there are more popular artists, whom everybody listens to, than popular websites, which everybody bookmarks —see Figure 7.3. This makes a huge difference in practice, and the choice of these two datasets allow us to make a more realistic comparison of recommendation techniques. Since we did not remove any items from these datasets (neither the most frequent nor the least frequent), these differences do influence the behavior of all algorithms —see below.

Some statistics about Last.fm and Delicious are reported in Table 7.1. In Figure 7.3 we plotted the distribution of the number of preferences per item in order to make clear and visible the differences explained in the previous paragraphs.[5]

We preprocessed datasets by breaking down the tags into smaller tags made up of single words. In fact, many users tend to create tags like "web-design_tutorial_css". This tag has been splitted into three smaller tags corresponding to the three words therein. More generally, we splitted all compound tags containing underscores, hyphens and apexes. This makes sense because users create tags independently, and we may have both "rock_and_roll" and "rock_n_roll". Because of this splitting operation, the number of unique tags decreased from 11,946 to 6,036 on Last.fm and from 53,388 to 9,949 on Delicious. On Delicious, we also removed all tags occurring less than ten times.[6]

[4] Datasets and their full descriptions are available at `www.grouplens.org/node/462`.

[5] In the context of recommender systems, these two datasets may be seen as representatives of two "markets" whose products have significantly different market shares (the well-known dichotomy of hit vs. niche products). Niche product markets give rise to power laws in user preference statistics (as in the blue plot of Figure 7.3).

[6] We did not repeat the same operation on Last.fm because this dataset was already extremely sparse.

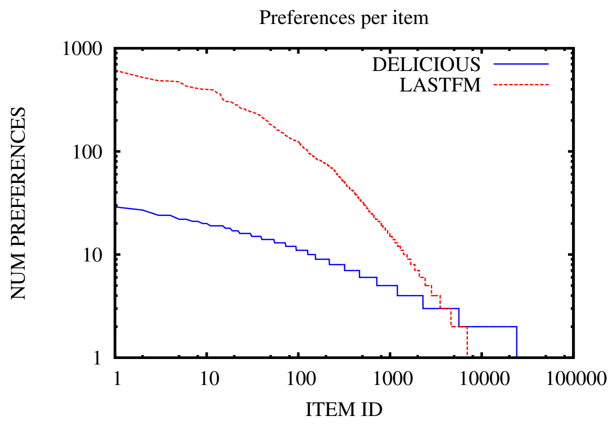


Fig. 7.3: Plot of the number of preferences per item (users who bookmarked the URL or listened to an artist). Both axes have logarithmic scale.

	Last.fm	Delicious
Nodes	1892	1867
Edges	12717	7668
Avg. degree	13.443	8.21
Items	17632	69226
Nonzero payoffs	92834	104799
Tags	11946	53388

Table 7.1: Main statistics for Last.fm and Delicious. Items counts the overall number of items, across all users, from which C_t is selected. Nonzero payoffs is the number of pairs (user, item) for which we have a nonzero payoff. Tags is the number of distinct tags that were used to describe the items.

The algorithms we tested do not use any prior information about which user provided a specific tag. We used all tags associated with a single item to create a TF-IDF context vector that uniquely represents that item, independent of which user the item is proposed to. In both datasets, we only retained the first 25 principal components of context vectors, so that $\boldsymbol{x}_{t,k} \in \mathbb{R}^{25}$ for all t and k.

We generated random context sets C_{i_t} of size 25 for Last.fm and Delicious, and of size 10 for 4Cliques. In practical scenarios, these numbers would be varying over time, but we kept them fixed so as to simplify the experimental setting. In 4Cliques we assigned the same unit norm random vector $\boldsymbol{u}_i$ to every node in the same clique i of the original graph (before adding graph noise). Payoffs were then generated according to the following stochastic model: $a_i(\boldsymbol{x}) = \boldsymbol{u}_i^\top \boldsymbol{x} + \epsilon$, where ϵ (the payoff noise) is uniformly distributed in a bounded interval centered around zero. For Delicious and Last.fm, we created a set of context vectors for every round t as follows: we

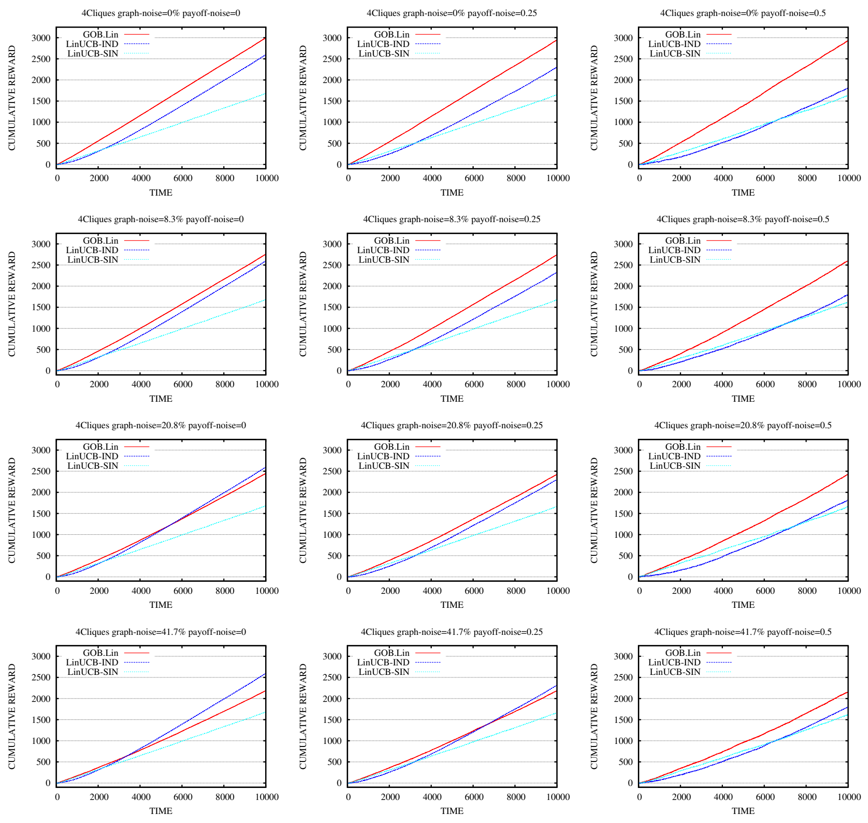


Fig. 7.4: Normalized cumulated reward for different levels of graph noise (expected fraction of perturbed edges) and payoff noise (largest absolute value of noise term ϵ) on the 4Cliques dataset. Graph noise increases from top to bottom, payoff noise increases from left to right. GOB.Lin is clearly more robust to payoff noise than its competitors. On the other hand, GOB.Lin is sensitive to high levels of graph noise. In the last row, graph noise is 41.7%, i.e., the number of perturbed edges is 500 out of 1200 edges of the original graph.

first picked i_t uniformly at random in $\{1, \ldots, n\}$. Then, we generated context vectors $\boldsymbol{x}_{t,1}, \ldots, \boldsymbol{x}_{t,25}$ in C_{i_t} by picking 24 vectors at random from the dataset and one among those vectors with nonzero payoff for user i_t. This is necessary in order to avoid a meaningless comparison: with high probability, a purely random selection would result in payoffs equal to zero for all the context vectors in C_{i_t}.

7.3.2 Algorithms

In our experimental comparison, we tested GOB.Lin and its variants against two baselines: a baseline LinUCB-IND that runs an independent instance of the algorithm in Figure 7.1 at each node (this is equivalent to running GOB.Lin in Figure 7.2 with $A_{\otimes} = I_{dn}$) and a baseline LinUCB-SIN, which runs a single instance of the algorithm in Figure 7.1 shared by all the nodes. LinUCB-IND turns to be a reasonable comparator when, as in the Delicious dataset, there are many moderately popular items. On the other hand, LinUCB-SIN is a competitive baseline when, as in the Last.fm dataset, there are few very popular items. The two scalable variants of GOB.Lin which we empirically analyzed are based on node clustering,[7] and are defined as follows:

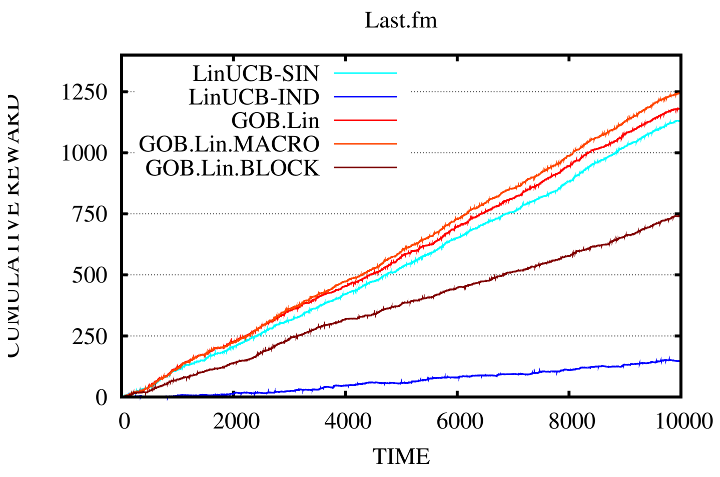
- **GOB.Lin.MACRO:** GOB.Lin is run on a *weighted* graph whose nodes are the clusters of the original graph. The edges are weighted by the number of inter-cluster edges in the original graph. When all nodes are clustered together, then GOB.Lin.MACRO recovers the baseline LinUCB-SIN as a special case. In order to strike a good trade-off between the speed of the algorithms and the loss of information resulting from clustering, we tested three different cluster sizes: 50, 100, and 200. Our plots refer to the best performing choice.

- **GOB.Lin.BLOCK:** GOB.Lin is run on a disconnected graph whose connected components are the clusters. This makes $A_{\otimes}$ and M_t (Figure 7.2) block-diagonal matrices. When each node is clustered individually, then GOB.Lin.BLOCK recovers the baseline LinUCB-IND as a special case. Similar to GOB.Lin.MACRO, in order to trade-off running time and cluster sizes, we tested three different cluster sizes (5, 10, and 20), and report only on the best performing choice.

As the running time of GOB.Lin scales quadratically with the number of nodes, the computational savings provided by the clustering are also quadratic. Moreover, as we will see in the experiments, the clustering acts as a regularizer, limiting the influence of noise.

In all cases, the parameter α in Figures 7.1 and 7.2 was selected based on the scale of instance vectors $\bar{x}_t$ and $\widetilde{\phi}_{t,k_t}$, respectively, and tuned across appropriate ranges.

[7] We used the freely available Graclus graph clustering tool, whose interns are described, e.g., in [39]. We used Graclus with normalized cut, zero local search steps, and no spectral clustering options.



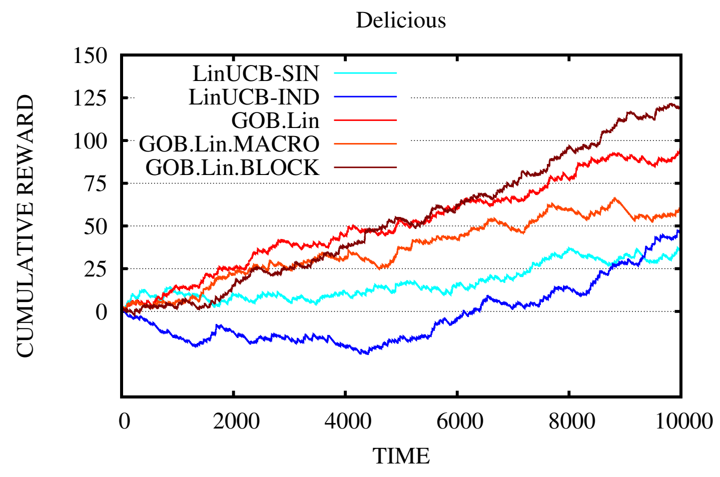


Fig. 7.5: Cumulative reward for all the bandit algorithms introduced in this section.

7.3.3 Results

Figure 7.4 and Figure 7.5 show the cumulative reward for each algorithm, as compared ("normalized") to that of the random predictor, that is $\sum_t (a_t - \bar{a}_t)$, where a_t is the payoff obtained by the algorithm and $\bar{a}_t$ is the payoff obtained by the random predictor, i.e., the average payoff over the context vectors available at time t.

Figure 7.4 (synthetic datasets) shows that GOB.Lin and LinUCB-SIN are more robust to payoff noise than LinUCB-IND. Clearly, LinUCB-SIN is also unaffected by graph noise, but it never outperforms GOB.Lin. When the payoff noise is low and the graph noise grows GOB.Lin's performance tends to degrade.

Figure 7.5 reports the results on the two real-world datasets. Notice that GOB.Lin and its variants always outperform the baselines (not relying on graphical information) on both datasets. As expected, GOB.Lin.MACRO works best on Last.fm, where many users gave positive payoffs to the same few items. Hence, macro nodes apparently help GOB.Lin.MACRO to perform better than its corresponding baseline LinUCB-SIN. In fact, GOB.Lin.MACRO also outperforms GOB.Lin, thus showing the regularization effect of using macro nodes. On Delicious, where we have many moderately popular items, GOB.Lin.BLOCK tends to perform best, GOB.Lin being the runner-up. As expected, LinUCB-IND works better than LinUCB-SIN, since the former is clearly more prone to personalize item recommendation than the latter. In summary, we may conclude that our system is able to exploit the information provided by the graphical structure. Moreover, regularization via graph clustering seems to be of significant help.

Future work will consider experiments against different methods for sharing contextual and feedback information in a set of users, such as the feature hashing technique of [81].

Part V
Conclusions

7.4 Conclusions and future works

In this thesis, we presented three different problems: node classification, link classifications and networks of bandits, motivated by important industrial applications. We proposed at least one new algorithm with a strong theoretical background for each of the proposed problems, and we tested these algorithms on synthetic and real-world datasets, comparing their performances with the state of the art. In most of the cases our algorithms shown a significant improvement over the state of the art in accuracy and speed.
Nevertheless, there are multiple possible extensions to the research work we presented in this thesis.

In general, there are two common extensions to all these problems: the first extension is the generalization of our algorithms to directed graphs. For the network of bandit it is trivial, but for the node and link classification problem is not and requires a significant amount of work. The generalized problem is interesting from a practical point of view since it will allow to model asymmetric preferences which are common in online social networks.
The second, is the extension to partially known graph topologies. This setting has been partially explored in [23] for the node classification problem, but completely unexplored for the other two problems .

In the following subsections, we describe some of the extensions we consider among the most interesting in detail.

7.4.1 Node Classification

An important extension, is the generalization of the Shazoo algorithm to the multiclass case. We conjecture that the upper bound presented for the binary case can hold also for the multiclass case.
Extensions of Mucca and Shazoo to the multilabel classification and to ranking should be quite easy from the practical point of view, but providing strong guarantees as for the classification problem is not straightforward.

7.4.2 Link Classification

The link classification problem is still quite unexplored from the machine learning point of view and allows plenty of different new research directions. The first possible extension is the generalization of the presented algorithms to the case of weighted graphs. This will allow to include side information

that is usually available in practice into the graph. For example: two users in a social network may have very few contacts, so in that case their relation can not be considered as strong as users having daily contacts.
Another interesting extension, partially explored in [14], is the extension to graphs with multiple labels on the edges. For example, this perfectly models the reality of users being in contact with each other for different reasons: business, hobbies, sport, school, etc. and, just to cite an example, it can provide precious information for the node classification algorithms.

7.4.3 Networks of Bandits

This is probably the most recent among the problems we tackled, and during the review phase of this thesis we obtained some interesting preliminary results. GOB.Lin is relying on the graphical structure provided as input to improve its predictions, but in some cases the similarity information is not available in advance or its signal is too noisy. The CLUB algorithm, presented in [46], learns at the same time a model of each node and the similarity structure among the nodes. Clearly, the algorithm does not perform better than GOB.Lin if GOB.Lin has a perfect similarity network available. Moreover, CLUB requires extra assumptions on the distribution of the data.
At the moment, one of the main problems of GOB.Lin is scalability. The problem has been addressed only to a certain extent in the current work. Another possible extension, is a distributed GOB.Lin-like algorithm that works with a limited number of updates, exploiting the techniques actually used for the Selective Sampling setting (i.e. see [21]).

Part VI
Appendices

Appendix A
Node Classification

A.1 Proofs regarding the SHAZOO algorithm

Proof of Theorem 3.1

Proof. Pick any $E' \subseteq E$ such that $\xi(M) = |E'|$. Let F be the forest obtained by removing from T all edges in E'. Draw an independent random label for each of the $|E'|+1$ components of F and assign it to all nodes of that component. Then any online algorithm makes in expectation at least half mistake per component, which implies that the overall number of online mistakes is $(|E'|+1)/2 > \xi(M)/2$ in expectation. On the other hand, $\Phi^W \leq M$ clearly holds by construction.

Proof of Theorem 3.5

We first give additional definitions used in the analysis, then we present the main ideas, and finally we provide full details.

Recall that, given a labeled tree $(T, \boldsymbol{y})$, a **cluster** is any maximal subtree whose nodes have the same label. Let $\mathcal{C}$ be the set of all clusters of T. For any cluster $C \in \mathcal{C}$, let M_C be the subset of all nodes of C on which SHAZOO makes a mistake. Let $\overline{C}$ be the subtree of T obtained by adding to C all nodes that are adjacent to a node of C. Note that all edges connecting a node of $\overline{C} \setminus C$ to a node of C are ϕ-edges. Let $E^{\phi}_{\overline{C}}$ be the set of ϕ-edges in $\overline{C}$ and let $\Phi_{\overline{C}} = \left|E^{\phi}_{\overline{C}}\right|$. Let $\Phi^{W}_{\overline{C}}$ be the total weight of the edges in $E^{\phi}_{\overline{C}}$. Finally, recall the notation $R^W_L = \sum_{(i,j)\in L} \frac{1}{W_{i,j}}$, where L is any line graph.

Recall that an **in-cluster line graph** is any line graph that is entirely contained in a single cluster. The main idea used in the proof below is to bound $|M_C|$ for each $C \in \mathcal{C}$ in the following way. We partition M_C into

$\mathcal{O}(|E'_{\overline{C}}|)$ groups, where $E'_{\overline{C}} \subseteq E_{\overline{C}}$. Then we find a set $\mathcal{L}_C$ of edge-disjoint in-cluster line graphs, and create a bijection between lines in $\mathcal{L}_C$ and groups in M_C. We prove that the cardinality of each group is at most $m_L = \min\Big\{|L|, 1 + \big\lfloor \ln\big(1 + \Phi^W R_L^W\big)\big\rfloor\Big\}$, where $L \in \mathcal{L}_C$ is the associated line. This shows that the subset M_T of nodes in T which are mispredicted by SHAZOO satisfies

$$|M_T| = \sum_{C \in \mathcal{C}} |M_C| \le \sum_{C \in \mathcal{C}} \sum_{L \in \mathcal{L}_C} m_L = \sum_{L \in \mathcal{L}_T} m_L$$

where $\mathcal{L}_T = \bigcup_{C \in \mathcal{C}} \mathcal{L}_C$. Then we show that

$$\sum_{C \in \mathcal{C}} \sum_{(i,j) \in E'_{\overline{C}}} w_{i,j} = \mathcal{O}\big(\Phi^W\big) \ .$$

By the very definition of ξ, and using the bijection stated above, this implies

$$|\mathcal{L}_T| = \sum_{C \in \mathcal{C}} |\mathcal{L}_C| = \mathcal{O}\left(\sum_{C \in \mathcal{C}} |E'_{\overline{C}}|\right) = \mathcal{O}\big(\xi(\Phi^W)\big) \ ,$$

thereby resulting in the mistake bound contained in Theorem 2.

The details of the proof require further notation.

According to SHAZOO prediction rule, when i_t is not a fork and $C(H(i_t)) \not\equiv \emptyset$, the algorithm predicts y_{i_t} using the label of any $j \in C\big(H(i_t)\big)$ closest to i_t. In this case, we call j an **r-node** (reference node) for i_t and the pair $\{j, (j,v)\}$, where (j,v) is the edge on the path between j and i_t, an **rn-direction** (reference node direction). We use the shorthand notation i^* to denote an r-node for i. In the special case when all connection nodes i of the hinge tree containing i_t have $\Delta(i) = 0$ (i.e., $C(H(i_t)) \equiv \emptyset$), and i_t is not a fork, we call any closest connection node j_0 to i_t an r-node for i_t and we say that $\{j_0, (j_0, v)\}$ is a rn-direction for i_t. Clearly, we may have more than one node of M_C associated with the same rn-direction. Given any rn-direction $\{j, (j,v)\}$, we call **r-line** (reference line) the line graph whose terminal nodes are j and the first (in chronological order) node $j_0 \in V$ for which $\{j, (j,v)\}$ is a rn-direction, where (j,v) lies on the path between j_0 and j.[1] We denote such an r-line by $L(j,v)$.

In the special case where $j \in C$ and $j_0 \notin C$ we say that the r-line is associated with the ϕ-edge of $E^{\phi}_{\overline{C}}$ included in the line-graph. In this case we denote such an r-line by $L(u,q)$, where $(u,q) \in E^{\phi}_{\overline{C}}$. Figure A.1 gives a pictorial example of the above concepts.

We now cover M_C (the subset of all nodes of $C \in \mathcal{C}$ on which SHAZOO makes a mistake) by the following subsets:

- M_C^F is the set of all forks in M_C.

[1] We may also have $v \equiv j_0$.

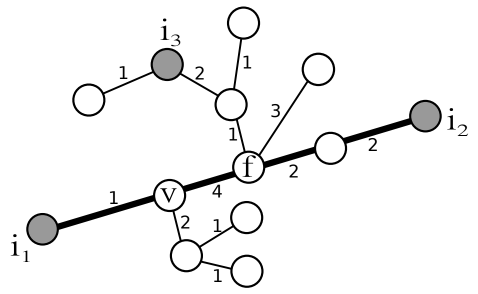


Fig. A.1: We illustrate an example of r-node, rn-direction and r-line. The numbers near the edge lines denote edge weights. In order to predict y_{i_2}, SHAZOO uses the r-node i_1 and the rn-direction $\{i_1, (i_1, v)\}$. After observing y_{i_2}, the hinge line connecting i_1 with i_2 (the thick black line) is created, which is also an r-line, since at the beginning of step $t = 2$ the algorithm used $\{i_1, (i_1, v)\}$. In order to predict y_{i_3}, we still use the r-node i_1 and the rn-direction $\{i_1, (i_1, v)\}$. After the revelation of y_{i_3}, node f becomes a fork.

- M_C^{in} is the subset of M_C containing the nodes i whose reference node i^* belongs to C (if i is a fork, then $i^* = i$). Note that this set may have a nonempty intersection with the previous one.
- M_C^{out} is the subset of M_C containing the nodes i such that i^* does not belong to C.

Two other structures that are relevant to the proof:

- $\mathcal{F}$ is the subset of all forks $f \in V_C$ such that $\Delta(f) \leq 0$ at some step t. Since we assume the cluster label is $+1$ (see below), and since a fork $i_t \in V_C$ is mistaken only if $\Delta(i_t) \leq 0$, we have $M_C^F \subseteq \mathcal{F}$.
- $C^{F'}$ is the subset of all nodes in M_C that, when revealed, create a fork that belongs to $\mathcal{F}$. Since at each time step at most one new fork can be created,[2] we have $|C^{F'}| \leq |\mathcal{F}|$.

The proof of the theorem relies on the following sequence of lemmas that show how to bound the number of mistakes made on a given cluster $C = (V_C, E_C)$. A major source of technical difficulties, that makes this analysis different and more complex than those of TreeOpt and wta, is that on a weighted tree the value of $\Delta(i)$ on forks i can potentially change after each prediction.

[2] In step t a new fork j is created when the number of edge-disjoint paths connecting j to the labeled nodes increases. This event occurs only when a new hinge line $\pi(i_t, f)$ is created. When this happens, the only node for which the number of edge-disjoint paths connecting it to labeled nodes gets increased is the terminal node j of the newly created hinge line.

Without loss of generality, from now on we assume all nodes in C are labeled $+1$. Keeping this assumption in mind is crucial to understand the arguments that follow.

For any node $i \in V_C$, let $\overline{\Delta}(i)$ be the value of $\Delta(i)$ when all nodes in $\overline{C} \setminus C$ are revealed.

Lemma A.1. *For any fork f of C and any step $t = 1, \ldots, n$, we have $\overline{\Delta}(f) \leq \Delta(f)$.*

Proof. For the sake of contradiction, assume $\overline{\Delta}(f) > \Delta(f)$. Let T^f be the maximal subtree of T rooted at f such that no internal node of T^f is revealed. Now, consider the cut given by the edges of $E_{\overline{C}}^{\phi}$ belonging to the hinge lines of T^f. This cut separates f from any revealed node labeled with -1. The size of this cut cannot be larger than $\Phi_{\overline{C}}^{W}$. By definition of $\Delta(\cdot)$, this implies $\Delta(f) \leq \Phi_{\overline{C}}^{W}$. However, also $\overline{\Delta}(f)$ cannot be larger than $\Phi_{\overline{C}}^{W}$. Because

$$\overline{\Delta}(i_t) \leq \sum_{(i,j) \in E_{\overline{C}}^{\phi}} W_{i,j} = \Phi_{\overline{C}}^{W}$$

must hold independent of the set of nodes in V_C that are revealed before time t, this entails a contradiction.

Let now $\xi_{\overline{C}}$ be the restriction of ξ on the subtree $\overline{C}$, and let D_C be the set of all distinct rn-directions which the nodes of M_C^{in} can be associated with. The next lemmas are aimed at bounding $|\mathcal{F}|$ and $|D_C|$. We first need to introduce the superset D'_C of D_C. Then, we show that for any C both $|D'_C|$ and $|\mathcal{F}|$ are linear in $\xi_{\overline{C}}(\Phi_{\overline{C}}^{W})$.

In order to do so, we need to take into account the fact that the sign of Δ for the forks in the cluster can change many times during the prediction process. This can be done via Lemma A.1, which shows that when all labels in $\overline{C} \setminus C$ are revealed then, for all fork $f \in C$, the value $\Delta(f)$ does not increase. Thus, we get the largest set D_C when we assume that the nodes in $\overline{C} \setminus C$ are revealed before the nodes of C.

Given any cluster C, let $\sigma_{\overline{C}}$ be the order in which the nodes of $\overline{C}$ are revealed. Let also $\sigma'_{\overline{C}}$ be the permutation in which all nodes in C are revealed in the same order as $\sigma_{\overline{C}}$, and all nodes in $\overline{C} \setminus C$ are revealed at the beginning, in any order. Now, given any node revelation order $\sigma_{\overline{C}}$, D'_C can be defined by describing the three types of steps involved in its incremental construction supposing $\sigma'_{\overline{C}}$ was the actual node revelation order.

1. After the first $|\overline{C} \setminus C| = \Phi_{\overline{C}}$ steps, D'_C contains all node-edge pairs $\{i, (i,j)\}$ such that i is a fork and (i,j) is an edge laying on a hinge line of $\overline{C}$. Recall that no node in C is revealed yet.
2. For each step $t > 0$ when a new fork f is created such that $\Delta(f) \leq 0$ just after the revelation of y_{i_t}, we add to D'_C the three node-edge pairs $\{f, (f,j)\}$, where the (f,j) are the edges contained in the three hinge lines terminating at f.

3. Let s be any step where: (i) A new hinge line $\pi(i_s, i_s^*)$ is created, (ii) node i_s^* is a fork, and (iii) $\Delta(i_s^*) \le 0$ at time $s-1$. On each such step we add $\{i_s^*, (i_s^*, j)\}$ to D'_C, for j in $\pi(i_s, i_s^*)$.

It is easy to verify that, given any ordering $\sigma_{\overline{C}}$ for the node revelation in $\overline{C}$, we have $D_C \subseteq D'_C$. In fact, given an rn-direction $\{i,(i,j)\} \in D_C$, if (i,j) lies along one of the hinge lines that are present at time 0 according to $\sigma'_{\overline{C}}$, then $\{i,(i,j)\}$ must be included in D'_C during one of the steps of type 2 above, otherwise $\{i,(i,j)\}$ will be included in D'_C during one of the steps of type 2 or type 3.

As announced, the following lemmas show that $|D'_C|$ and $|\mathcal{F}|$ are both of the order of $\xi_{\overline{C}}(\Phi^W_{\overline{C}})$.

Lemma A.2. *(i) The total number of forks at time $t = \Phi_{\overline{C}}$ is $\mathcal{O}\big(\xi(\Phi^W_{\overline{C}})\big)$. (ii) The total number of elements added to D'_C in the first step of its construction is $\mathcal{O}\big(\xi(\Phi^W_{\overline{C}})\big)$.*

Proof. Assume nodes are revealed according to $\sigma'_{\overline{C}}$. Let C' be the subtree of $\overline{C}$ made up of all nodes in $\overline{C}$ that are included in any path connecting two nodes of $\overline{C} \setminus C$. By their very definition, the forks at time $t = \Phi_{\overline{C}}$ are the nodes of $V_{C'}$ having degree larger than two in subtree C'. Consider C' as rooted at an arbitrary node of $\overline{C} \setminus C$. The number of the leaves of C' is equal to $|\overline{C} \setminus C| - 1$. This is in turn $\mathcal{O}\big(\xi_{\overline{C}}(\Phi^W_{\overline{C}}\big)$ because

$$\sum_{(i,j)\in E^{\phi}_{\overline{C}}} w_{i,j} = \mathcal{O}\big(\xi_{\overline{C}}(\Phi^W_{\overline{C}})\big) \ .$$

Now, in any tree, the sum of the degrees of nodes having degree larger than two cannot is at most linear in the number of leaves. Hence, at time $t = \Phi_{\overline{C}}$ both the number of forks in C and the cardinality of D'_C are $\mathcal{O}\big(\xi_{\overline{C}}(\Phi^W_{\overline{C}})\big)$.

Let now Γ^T_t be the minimal cutsize of T consistent with the labels seen before step $t+1$, and notice that Γ^T_t is nondecreasing with t.

Lemma A.3. *Let t be a step when a new hinge line $\pi(i_t, q)$ is created such that $i_t, q \in V_C$. If just after step t we have $\Delta(q) \le 0$, then $\Gamma^T_t - \Gamma^T_{t-1} \ge w_{u,v}$, where (u,v) is the lightest edge on $\pi(i_t, q)$.*

Proof. Since $\Delta(q) \le 0$ and $\pi(i_t, q)$ is completely included in C, we must have $\Delta(q) \le 0$ just before the revelation of y_{i_t}. This implies that the difference $\Gamma^T_t - \Gamma^T_{t-1}$ cannot be smaller than the minimum cutsize that would be created on $\pi(i_t, q)$ by assigning label -1 to node q.

Lemma A.4. *Assume nodes are revealed according to $\sigma'_{\overline{C}}$. Then the cardinality of $\mathcal{F}$ and the total number of elements added to D'_C during the steps of type 2 above are both linear in $\xi_{\overline{C}}(\Phi^W_{\overline{C}})$.*

Proof. Let $\mathcal{F}_0$ be the set of forks in V_C such that $\overline{\Delta}(f) \leq 0$ at some time $t \leq |V|$. Recall that, by definition, for each fork $f \in \mathcal{F}$ there exists a step t_f such that $\Delta(f) \leq 0$. Hence, Lemma A.1 implies that, at the same step t_f, for each fork $f \in \mathcal{F}$ we have $\overline{\Delta}(f) \leq 0$. Since $\mathcal{F}$ is included in $\mathcal{F}_0$, we can bound $|\mathcal{F}|$ by $|\mathcal{F}_0|$, i.e., by the number of forks $i \in V_C$ such that $\Delta(i) \leq 0$, under the assumption that $\sigma'_{\overline{C}}$ is the actual revelation order for the nodes in $\overline{C}$.

Now, $|\mathcal{F}_0|$ is bounded by the number of forks created in the first $|\overline{C} \setminus C| = \Phi_{\overline{C}}$ steps, which is equal to $\mathcal{O}\big(\xi(\Phi^W_{\overline{C}})\big)$ plus the number of forks f created at some later step and such that $\Delta(f) \leq 0$ right after their creation. Since nodes in $\overline{C}$ are revealed according to $\sigma'_{\overline{C}}$, the condition $\Delta(f) > 0$ just after the creation of a fork f implies that we will never have $\Delta(f) \leq 0$ in later stages. Hence this fork f belongs neither to $\mathcal{F}_0$ nor to $\mathcal{F}$.

In order to conclude the proof, it suffices to bound from above the number of elements added to D'_C in the steps of type 2 above. From Lemma A.3, we can see that for each fork f created at time t such that $\Delta(f) \leq 0$ just after the revelation of node i_t, we must have $|\Gamma^T_t - \Gamma^T_{t-1}| \geq w_{u,v}$, where (u, v) is the lightest edge in $\pi(i_t, f)$. Hence, we can injectively associate each element of $\mathcal{F}$ with an edge of E_C, in such a way that the sum of the weights of these edges is bounded by $\Phi^W_{\overline{C}}$. By definition of ξ, we can therefore conclude that the total number of elements added to D'_C in the steps of type 2 is $\mathcal{O}\big(\xi(\Phi^W_{\overline{C}})\big)$.

With the following lemma we bound the number of nodes of $M^{\text{in}}_C \setminus C^{F'}$ associated with every rn-direction and show that one can perform a transformation of the r-lines so as to make them edge-disjoint. This transformation is crucial for finding the set $\mathcal{L}_T$ appearing in the theorem statement. Observe that, by definition of r-line, we cannot have two r-lines such that each of them includes only one terminal node of the other. Thus, let now F_C be the forest where each node is associated with an r-line and where the parent-child relationship expresses that (i) the parent r-line contains a terminal node of the child r-line, together with (ii) the parent r-line and the child r-line are not edge-disjoint. F_C is, in fact, a forest of r-lines. We now use $m_{L(j,v)}$ for bounding the number of mistakes associated with a given rn-direction $\{i, (j, v)\}$ or with a given ϕ-edge (j, v). Given any connected component T' of F_C, let finally $m_{T'}$ be the total number of nodes of $M^{\text{in}}_C \setminus C^{F'}$ associated with the rn-directions $\{i, (i, j)\}$ of all r-lines $L(i, j)$ of T'.

Lemma A.5. *Let C be any cluster. Then:*

(i) The number of nodes in $M^{\text{in}}_C \setminus C^{F'}$ associated with a given rn-direction $\{j, (j, v)\}$ is of the order of $m_{L(i,j)}$.

(ii) The number of nodes in $M^{\text{out}}_C \setminus C^{F'}$ associated with a given ϕ-edge (u, q) is of the order of $m_{L(u,q)}$.

(iii) Let $L(j_r, v_r)$ be the r-line associated with the root of any connected component T' of F_C. $m_{T'}$ must be at most of the same order of

$$\sum_{L(j,v)\in\mathcal{L}(L(j_r,v_r))} m_{L(j,v)} + |V_{T'}|$$

where $\mathcal{L}(L(j_r,v_r))$ is a set of $|V_{T'}|$ edge-disjoint line graphs completely contained in $L(j_r,v_r)$.

Proof. We will prove only (i) and (iii), (ii) being similar to (i). Let i_t be a node in $M_C^{\text{in}} \setminus C^{F'}$ associated with a given rn-direction $\{j,(j,v)\}$. There are two possibilities: (a) i_t is in $L(j,v)$ or (b) the revelation of y_{i_t} creates a fork f in $L(j,v)$ such that $\Delta(f) > 0$ for all steps $s \ge t$. Let now $i_{t'}$ be the next node (in chronological order) of $M_C^{\text{in}} \setminus C^{F'}$ associated with $\{j,(j,v)\}$. The length of $\pi(i_{t'}, i_t)$ cannot be smaller than the length of $\pi(i_{t'}, j)$ (under condition (a)) or smaller than the length of $\pi(f,j)$ (under condition (b)).

This clearly entails a dichotomic behaviour in the sequence of mistaken nodes in $M_C^{\text{in}} \setminus C^{F'}$ associated with $\{j,(j,v)\}$. Let now p be the node in $L(j,v)$ which is farthest from j such that the length of $\pi(p,j)$ is not larger than Φ^W. Once a node in $\pi(p,j)$ is revealed or becomes a fork f satisfying $\Delta(f) > 0$ for all steps $s \ge t$, we have $\Delta(j) > 0$ for all subsequent steps (otherwise, this would contradict the fact that the total cutsize of T is Φ^W). Combined with the above sequential dichotomic behavior, this shows that the number of nodes of $M_C^{\text{in}} \setminus C^{F'}$ associated with a given rn-direction $\{j,(j,v)\}$ can be at most of the order of

$$\min\left\{|L(j,v)|,\, 1 + \left\lfloor \log_2\left(\frac{R^W_{L(j,v)} + (\Phi^W)^{-1}}{(\Phi^W)^{-1}}\right)\right\rfloor\right\} = m_{L(j,v)}\ .$$

Part (iii) of the statement can be now proved in the following way. Suppose now that an r-line $L(j,v)$, having j and j_0 as terminal nodes, includes the terminal node j' of another r-line $L(j',v')$, having j' and j'_0 as terminal nodes. Assume also that the two r-lines are not edge-disjoint. If $L(j',v')$ is partially included in $L(j,v)$, i.e., if j'_0 does not belong to $L(j,v)$, then $L(j',v')$ can be broken into two sub-lines: the first one has j' and k as terminal nodes, being k the node in $L(j,v)$ which is farthest from j'; the second one has k and j'_0 as terminal nodes. It is easy to see that $L(j,v)$ must be created before $L(j',v')$ and j_0 is the only node of the second sub-line that can be associated with the rn-direction $\{j',(j',v')\}$. This observation reduces the problem to considering that in T' each r-line that is not a root is completely included in its parent.

Given an r-line $L(u,q)$ having u and z as terminals, we denote by $m_{\pi(u,z)}$ the quantity $m_{L(u,q)}$.

Consider now the simplest case in which T' is formed by only two r-lines: the parent r-line $L(j_p,v_p)$, which completely contains the child r-line $L(j_c,v_c)$. Let s be the step in which the first node u of $L(j_p,v_p)$ becomes a hinge node. After step s, $L(j_p,v_p)$ can be vieved as broken in two edge-disjoint sublines having $\{j_p,u\}$ and $\{j_0,u\}$ as terminal node sets, where j_0 is one of the terminal of $L(j_p,v_p)$. Thus,

$$m_{T'} \leq \max_{u \in V_{L(j_p,v_p)}} m_{\pi(j_p,u)} + m_{\pi(u,j_0)} + 1 \ .$$

Generalizing this argument for every component T' of F_C, and using the above observation about the partially included r-lines, we can state that, for any component T' of F_C, $m_{T'}$ is of the order of

$$\max_{u_1,\dots,u_N \in V_{L(j_p,v_p)}} \Big(m_{\pi(j_p,u_1)} + m_{\pi(u_N,j_0)} + \sum_{k=1}^{N-1} m_{\pi(u_k,u_{k+1})} + 2|V_{T'}| \Big)$$

where $N = |V_{T'}| - 1$. This entails that we can define $\mathcal{L}(L(j_r,v_r))$ as the union of $\{\pi(j_p,u_1), \pi(u_N,j_0)\}$ and $\bigcup_{k=1}^{N-1} \pi(u_k,u_{k+1})$, which concludes the proof.

Lemma A.6. *The total number of elements added to D'_C during steps of type 3 above is of the order of $\xi_{\overline{C}}(\Phi^W_{\overline{C}})$.*

Proof. Assume nodes are revealed according to $\sigma'_{\overline{C}}$, and let s be any type-3 step when a new element is added to D'_C. There are two cases: (a) $\Delta(i^*_s) \leq 0$ at time s or (b) $\Delta(i^*_s) > 0$ at time s.

Case (a). Lemma A.3 combined with the fact that all hinge-lines created are edge-disjoint, ensures that we can injectively associate each of these added elements with an edge of E_C in such a way that the total weight of these edges is bounded by $\Phi^W_{\overline{C}}$. This in turn implies that the total number of elements added to E_C is $\mathcal{O}\big(\xi_{\overline{C}}(\Phi^W_{\overline{C}})\big)$.

Case (b). Since we assumed that nodes are revealed according to $\sigma'_{\overline{C}}$, we have that $\Delta(i^*_s)$ is positive for all steps $t > s$. Hence we have that case (b) can occur only once for each of such forks i^*_s. Since this kind of fork belongs to $\mathcal{F}$, we can use Lemma A.4 and conclude that (b) can occur at most $|\mathcal{F}| = \mathcal{O}\big(\xi_{\overline{C}}(\Phi^W_{\overline{C}})\big)$ times.

Lemma A.7. *With the notation introduced so far, we have $|D_C| = \mathcal{O}\big(\xi_{\overline{C}}(\Phi^W_{\overline{C}})\big)$.*

Proof. Combining Lemma A.2, Lemma A.4, and Lemma A.6 we immediately have $D'_C = \mathcal{O}\big(\xi_{\overline{C}}(\Phi^W_{\overline{C}})\big)$. The claim then follows from $D_C \subseteq D'_C$.

We are now ready to prove the theorem.

Proof of Theorem 3.5. Let F_T be the union of F_C over $C \in \mathcal{C}$. Using Lemma A.7 we deduce $|V_{F_C}| = \Phi_{\overline{C}} + \mathcal{O}\big(\xi_{\overline{C}}(\Phi^W_{\overline{C}})\big) = \mathcal{O}\big(\xi_{\overline{C}}(\Phi^W_{\overline{C}})\big)$, where the term $\Phi_{\overline{C}}$ takes into account that at most one r-line of F_C may be associated with each ϕ-edge of $\overline{C}$.

By definition of $\xi(\cdot)$, this implies $|V_{F_T}| = \mathcal{O}\big(\xi(\Phi^W)\big)$. Using part (i) and (ii) of Lemma A.5 we have $|M_T| \leq |M^F_C| + |M^{\text{in}}_C| + |M^{\text{out}}_C| \leq |\mathcal{F}| + |C^{F'}| + \sum_{L \in V_{F_T}} m_L \leq \sum_{L \in V_{F_T}} m_L + \mathcal{O}\big(\xi(\Phi^W)\big)$.

Let now $\mathcal{T}(F_T)$ be the set of components of F_T. Given any tree $T' \in \mathcal{T}(F_T)$, let $r(T')$ be the r-line root of T'. Recall that, by part (iii) of Lemma A.5 for any tree $T' \in \mathcal{T}(F_T)$ we can find a set $\mathcal{L}(r(T'))$ of $|V_{T'}|$

edge-disjoint line graphs all included in $r(T')$ such that $m_{T'}$ is of the order of $\sum_{L\in\mathcal{L}_{T'}(r(T'))} m_L + |V_{T'}|$. Let now $\mathcal{L}'_T$ be equal to $\cup_{T'\in\mathcal{T}(F_T)}\mathcal{L}(r(T'))$. Thus we have

$$|M_T| = \mathcal{O}\left(\sum_{L\in\mathcal{L}'_T} m_L + |V_{F_T}| + \xi(\Phi^W)\right) = \mathcal{O}\left(\sum_{L\in\mathcal{L}'_T} m_L + \xi(\Phi^W)\right) \ .$$

Observe that $\mathcal{L}'_T$ is not an edge disjoint set of line graphs included in T only because each ϕ-edge may belong to two different lines of $\mathcal{L}'_T$. By definition of m_L, for any line graphs L and L', where L' is obtained from L by removing one of the two terminal nodes and the edge incident to it, we have $m_{L'} = m_L + \mathcal{O}(1)$. If, for each ϕ-edge shared by two line graphs of $\mathcal{L}'_T$, we shorten the two line graphs so as no one of them includes the ϕ-edge, we obtain a new set of edge-disjoint line graphs $\mathcal{L}_T$ such that $\sum_{L\in\mathcal{L}'_T} m_L = \sum_{L'\in\mathcal{L}_T} +\xi(\Phi^W)$. Hence, we finally obtain $|M_T| = \mathcal{O}\Big(\sum_{L'\in\mathcal{L}_T} m_{L'} + \xi(\Phi^W)\Big) = \mathcal{O}\Big(\sum_{L'\in\mathcal{L}_T} m_{L'}\Big)$, where in the last equality we used the fact that $m_{L'} \geq 1$ for all line graphs L'. $\square$

A.2 Experimental results

In this section we report the detailed results of our experiments with node classification algorithms. Some of these results have been plotted and reported in Chapter 5.
Each table refers to experiments on one dataset, on the first row we reported the size of the training set. In the first column the name of the algorithm and concatenated with a "+" the spanning tree (eventually) used in combination with the algorithm.
Please remember that algorithms are denoted by k*algorithmName are committees of k spanning trees aggregating predictions via a majority vote.

Predictors	2,5%	5%	10%	25%	33%	50%
GTG-ESS	5,09 (0,4)	4,61 (0,3)	4,23 (0,15)	3,98 (0,18)	3,91 (0,24)	3,57 (0,24)
LABPROP	7,08 (0,63)	5,72 (0,4)	4,8 (0,09)	4,16 (0,18)	3,99 (0,22)	3,61 (0,2)
GPA+RST	45,73 (8,61)	42,71 (7,31)	32,83 (4,33)	27,01 (3,83)	29,51 (4,44)	23,54 (3,32)
GPA+MST	8,61 (3,89)	11,75 (4,35)	7,45 (1,43)	9,21 (4,09)	8,1 (2,46)	7,21 (3,27)
GPA+BFST	54,73 (2,83)	47,4 (2,73)	40,45 (4,91)	30,03 (2,54)	30,12 (3,55)	25,71 (1,92)
WTA+RST	32,08 (2,49)	26,56 (1,12)	21,15 (0,89)	15,42 (0,67)	13,8 (0,43)	11,81 (0,55)
WTA+MST	11,51 (0,88)	9,92 (1,29)	7,85 (0,52)	5,96 (0,5)	5,49 (0,26)	4,83 (0,32)
WTA+BFST	45,98 (2,69)	38,28 (1,71)	30,46 (0,77)	22,3 (0,91)	19,96 (0,78)	17,69 (0,58)
SHAZOO+RST	24,2 (2,57)	18,62 (0,93)	14,16 (1,02)	10,42 (0,44)	9,28 (0,56)	7,89 (0,58)
SHAZOO+MST	6,04 (0,3)	5,74 (0,41)	4,82 (0,33)	3,89 (0,25)	3,66 (0,17)	3,32 (0,21)
SHAZOO+BFST	42,8 (1,81)	33,05 (2,36)	23,75 (1,49)	16,24 (1,11)	14,26 (0,98)	12,22 (0,75)
MUCCA+RST	24,1 (2,22)	18,38 (1,23)	13,97 (0,85)	10,08 (0,51)	9,11 (0,56)	7,67 (0,54)
MUCCA+MST	5,92 (0,4)	5,47 (0,35)	4,53 (0,39)	3,82 (0,22)	3,52 (0,23)	3,2 (0,21)
MUCCA+BFST	40,21 (2,6)	31,66 (2,76)	23,21 (1,41)	15,93 (1,09)	14,09 (1,05)	12,12 (0,75)
3*WTA+RST	24,15 (1,5)	17,43 (1,13)	12,96 (0,52)	9,23 (0,45)	8,21 (0,34)	7,14 (0,32)
3*SHAZOO+RST	16,82 (1,07)	11,91 (0,8)	8,82 (0,33)	6,68 (0,42)	6,08 (0,29)	5,44 (0,18)
3*MUCCA+RST	16,62 (1,18)	11,69 (0,66)	8,52 (0,34)	6,48 (0,37)	5,88 (0,23)	5,32 (0,12)
7*WTA+RST	13,66 (1,01)	9,45 (0,48)	7,15 (0,27)	5,75 (0,23)	5,33 (0,27)	4,77 (0,24)
7*SHAZOO+RST	10,01 (0,78)	7,36 (0,37)	5,91 (0,31)	4,86 (0,24)	4,54 (0,19)	4,14 (0,23)
7*MUCCA+RST	9,84 (0,87)	7,17 (0,37)	5,6 (0,24)	4,68 (0,23)	4,39 (0,19)	4,05 (0,28)
11*WTA+RST	10,34 (1,21)	7,63 (0,48)	6,12 (0,24)	4,88 (0,18)	4,66 (0,21)	4,16 (0,23)
11*SHAZOO+RST	8,32 (0,49)	6,41 (0,41)	5,3 (0,22)	4,43 (0,22)	4,21 (0,15)	3,78 (0,25)
11*MUCCA+RST	8,07 (0,57)	6,22 (0,45)	5,06 (0,19)	4,27 (0,18)	4,09 (0,14)	3,71 (0,27)

Table A.1: In this table we show the averaged classification error rates (percentages) achieved by the various algorithms on the USPS-0 dataset. Algorithms are trained ten times over a random subset of 2,5%, 5%, 10%, 25%, 33% and 50% of the total number of nodes and tested on the remaining ones. GTG-ESS and LABPROP are used as yardsticks for the comparison. Standard deviations are reported in parenthesis.

Predictors	2,5%	5%	10%	25%	33%	50%
GTG-ESS	5,43 (0,4)	4,72 (0,18)	4,31 (0,16)	4,04 (0,12)	3,87 (0,14)	3,66 (0,18)
LABPROP	15,07 (4,26)	6,74 (0,64)	4,95 (0,23)	4,19 (0,14)	3,91 (0,09)	3,68 (0,15)
GPA+RST	78,11 (3,28)	75,71 (3,82)	70,97 (4,02)	59,15 (3,41)	60,13 (3,64)	56,22 (3,48)
GPA+MST	74,14 (2,07)	68 (4,43)	62,8 (2,41)	54,15 (3,47)	50,57 (3,5)	47,88 (5,21)
GPA+BFST	74,11 (1,42)	69,54 (2,52)	62,68 (1,73)	54,43 (2,39)	51,62 (1,94)	48,28 (3,35)
WTA+RST	62,26 (1,59)	53,49 (1,24)	43,66 (1,06)	31,38 (0,94)	27,65 (0,75)	23,37 (0,68)
WTA+MST	61,94 (1,45)	53,99 (1,56)	44,46 (0,98)	33,38 (0,62)	30,39 (0,62)	26,14 (0,47)
WTA+BFST	70,03 (1,72)	63,72 (1,36)	55,5 (1,52)	44,81 (1,42)	42,17 (1,18)	38,3 (0,79)
SHAZOO+RST	55,5 (1,53)	43,71 (1,65)	32,6 (1,35)	21,35 (0,68)	18,8 (0,48)	15,83 (0,57)
SHAZOO+MST	57,57 (2,3)	46,69 (1)	36,21 (1,13)	24,3 (0,51)	20,86 (0,7)	17,19 (0,57)
SHAZOO+BFST	63,7 (2,89)	54,1 (2,59)	43,91 (1,31)	31,79 (1,11)	29,73 (1,18)	26,39 (0,86)
MUCCA+RST	54,93 (2)	43,54 (1,6)	32,53 (1,15)	21,31 (0,7)	18,65 (0,64)	15,81 (0,62)
MUCCA+MST	59,14 (2,19)	48,28 (0,72)	37,51 (0,97)	25,13 (0,66)	21,74 (0,72)	17,81 (0,58)
MUCCA+BFST	63,2 (2,69)	53,55 (2,65)	43,75 (1,15)	31,75 (1,06)	29,65 (1,23)	26,37 (0,89)
3*WTA+RST	54,27 (1,62)	42,99 (1,09)	31,71 (0,5)	19,97 (0,52)	17,22 (0,47)	13,66 (0,39)
3*SHAZOO+RST	46,44 (2,43)	32,48 (0,93)	21,65 (0,9)	12,89 (0,27)	11,08 (0,5)	9,08 (0,33)
3*MUCCA+RST	45,79 (2,28)	32,29 (0,84)	21,33 (0,56)	12,76 (0,42)	10,91 (0,37)	8,94 (0,32)
7*WTA+RST	37,95 (1,8)	24,68 (0,97)	15,02 (0,56)	8,69 (0,32)	7,59 (0,19)	6,28 (0,13)
7*SHAZOO+RST	28,12 (2,34)	15,88 (0,79)	9,65 (0,56)	6,39 (0,34)	5,69 (0,23)	5,08 (0,24)
7*MUCCA+RST	27,37 (2,29)	15,69 (0,94)	9,3 (0,59)	6,29 (0,3)	5,6 (0,28)	4,98 (0,25)
11*WTA+RST	29,39 (1,25)	17,21 (0,83)	10,1 (0,34)	6,35 (0,27)	5,61 (0,14)	4,89 (0,16)
11*SHAZOO+RST	20,66 (1,87)	11,25 (0,75)	7,27 (0,36)	5,18 (0,22)	4,76 (0,23)	4,25 (0,18)
11*MUCCA+RST	19,88 (2,06)	10,86 (0,83)	6,98 (0,32)	5,01 (0,17)	4,66 (0,29)	4,17 (0,22)

Table A.2: In this table we show the averaged classification error rates (percentages) achieved by the various algorithms on the USPS-10 dataset. Algorithms are trained ten times over a random subset of 2,5%, 5%, 10%, 25%, 33% and 50% of the total number of nodes and tested on the remaining ones. GTG-ESS and LABPROP are used as yardsticks for the comparison. Standard deviations are reported in parenthesis.

Predictors	2,5%	5%	10%	25%	33%	50%
GTG-ESS	5,48 (0,46)	4,8 (0,35)	4,37 (0,28)	4,03 (0,17)	3,77 (0,14)	3,67 (0,22)
LABPROP	44,56 (7,83)	16,94 (3,17)	6,5 (0,54)	4,24 (0,17)	3,99 (0,18)	3,73 (0,19)
GPA+RST	84,73 (1,09)	82,21 (1,58)	80,44 (2,07)	73,75 (2,58)	73,68 (4,45)	69,68 (3,63)
GPA+MST	84,47 (1,88)	81,92 (1,08)	78,27 (1,66)	72,35 (2,32)	70,91 (1,78)	67,52 (2,21)
GPA+BFST	79,9 (1,8)	76,61 (2,12)	72,74 (2,71)	67,1 (1,09)	64,47 (2,67)	61,39 (2,15)
WTA+RST	75,7 (1,46)	67,46 (1,41)	59,12 (1,44)	45,21 (1,17)	40,72 (1,06)	35,15 (1,25)
WTA+MST	78,53 (1,11)	72,16 (0,96)	65,08 (0,7)	54,31 (0,72)	50,81 (0,79)	46,49 (0,29)
WTA+BFST	77,55 (1,13)	73 (1,2)	66,5 (0,99)	58,77 (0,53)	55,87 (0,79)	52,32 (0,96)
SHAZOO+RST	70,25 (1,85)	60,46 (1,4)	48,7 (1,62)	34,62 (1,06)	30,24 (0,96)	25,42 (1,25)
SHAZOO+MST	75,78 (1,37)	68,12 (0,8)	58,97 (1,35)	45,13 (0,61)	40,67 (0,66)	34,95 (0,73)
SHAZOO+BFST	73,76 (2,52)	65,2 (1,95)	56,81 (1,57)	45,16 (0,97)	41,91 (0,92)	38,11 (0,53)
MUCCA+RST	70,27 (2,14)	60,1 (1,55)	48,56 (1,59)	34,58 (0,97)	30,28 (0,98)	25,5 (1,29)
MUCCA+MST	76,81 (1,27)	69,3 (1,05)	60,44 (1,54)	46,46 (0,57)	41,81 (0,68)	35,66 (0,66)
MUCCA+BFST	73,4 (2,31)	64,97 (1,91)	56,46 (1,61)	45,05 (0,98)	41,81 (0,85)	38,05 (0,48)
3*WTA+RST	69,99 (0,98)	60,73 (1,17)	49,35 (0,97)	33,87 (0,77)	29,17 (0,61)	23,54 (0,77)
3*SHAZOO+RST	63,91 (1,67)	51,8 (1,6)	38,17 (1,11)	23,56 (0,88)	19,63 (0,57)	15,58 (0,51)
3*MUCCA+RST	63,55 (1,69)	51,65 (1,81)	37,99 (1,04)	23,49 (0,59)	19,78 (0,45)	15,73 (0,56)
7*WTA+RST	59,78 (1,09)	45,45 (1,08)	31,13 (0,55)	16,77 (0,58)	13,52 (0,28)	10,39 (0,49)
7*SHAZOO+RST	50,94 (1,53)	34,12 (1,48)	20,46 (0,66)	11,04 (0,6)	9,04 (0,35)	7,48 (0,32)
7*MUCCA+RST	50,53 (1,84)	33,73 (1,51)	20,16 (0,75)	10,88 (0,56)	9,04 (0,32)	7,55 (0,3)
11*WTA+RST	54,24 (2,39)	37,37 (1,04)	22,82 (0,56)	11,03 (0,36)	8,85 (0,4)	7,14 (0,32)
11*SHAZOO+RST	44,19 (2,15)	25,92 (0,83)	14,2 (0,48)	7,7 (0,49)	6,48 (0,4)	5,56 (0,3)
11*MUCCA+RST	43,79 (2,34)	25,4 (0,86)	13,94 (0,48)	7,58 (0,45)	6,4 (0,33)	5,55 (0,28)

Table A.3: In this table we show the averaged classification error rates (percentages) achieved by the various algorithms on the USPS-25 dataset. Algorithms are trained ten times over a random subset of 2,5%, 5%, 10%, 25%, 33% and 50% of the total number of nodes and tested on the remaining ones. GTG-ESS and LABPROP are used as yardsticks for the comparison. Standard deviations are reported in parenthesis.

Predictors	2,5%	5%	10%	25%	33%	50%
gtg-ess	42,94 (3,04)	36,1 (2,76)	32,86 (1,56)	24,42 (1,05)	22,35 (1,45)	19,86 (0,85)
labprop	50,98 (6,98)	39,19 (5,21)	32,83 (3,54)	24,16 (1,03)	22,09 (1,48)	19,81 (0,66)
gpa+rst	61,52 (5,04)	49,9 (3,52)	54,19 (7,61)	46,42 (6,2)	44,01 (3,26)	44,57 (6,71)
gpa+mst	57,14 (5,44)	53,49 (5,75)	47,77 (7,34)	41,95 (5,85)	39,63 (8,15)	36,67 (5,24)
gpa+bfst	59,37 (9,74)	52,62 (8,13)	52,33 (9,54)	45,16 (4,44)	43,01 (7,65)	38,9 (4,34)
wta+rst	54,39 (3,38)	47,82 (2,31)	42,28 (2,33)	33,56 (1,17)	30,88 (2,01)	27,79 (1,39)
wta+mst	52,13 (3,66)	44,81 (1,53)	41,25 (1,76)	33,19 (1,44)	30,67 (0,97)	26,54 (1,28)
wta+bfst	54,77 (4,35)	48,68 (2,59)	42,81 (1,97)	35,09 (1,64)	33,18 (1,25)	29,75 (1,22)
shazoo+rst	49,02 (3,57)	40,93 (3,94)	35,04 (1,83)	26,72 (1,43)	24,65 (1,35)	20,98 (0,82)
shazoo+mst	48,65 (4,96)	41,84 (2,52)	37,46 (2,03)	27,55 (1,09)	25,19 (1,49)	21,59 (1,15)
shazoo+bfst	53,38 (6,27)	44,56 (4,35)	36,91 (2,48)	28,44 (1,68)	26,6 (1,34)	22,94 (1,04)
mucca+rst	48,65 (3,02)	41,21 (2,67)	35,42 (1,89)	26,48 (1,28)	24,72 (1,76)	21,14 (0,88)
mucca+mst	46,79 (4,53)	41,2 (2,03)	36,68 (1,78)	27,63 (1,1)	25,36 (1,36)	21,68 (1,13)
mucca+bfst	51,86 (5,69)	43,24 (3,59)	36,97 (2,44)	29,01 (1,88)	26,93 (1,21)	22,98 (1,24)
3*wta+rst	49,92 (3,32)	42,95 (1,59)	37,15 (1,72)	29,31 (0,94)	26,29 (1,09)	24 (1,27)
3*shazoo+rst	48,99 (3,27)	40,43 (3,21)	34,75 (1,46)	26,13 (0,95)	23,02 (1,16)	20,29 (1,46)
3*mucca+rst	47 (3,87)	39,29 (3,45)	34,99 (0,81)	26,21 (0,98)	23,51 (1,04)	20,65 (1,53)
7*wta+rst	47,08 (3,04)	39,45 (1,04)	33,73 (1,94)	25,88 (1)	23,64 (1,04)	21,21 (1,09)
7*shazoo+rst	47,48 (3,16)	39,41 (3,16)	32,88 (1,96)	25,6 (0,92)	22,38 (0,89)	19,75 (1,58)
7*mucca+rst	45,04 (3,22)	37,83 (2,82)	32,99 (1,81)	25,36 (0,87)	22,61 (0,79)	20,01 (1,54)
11*wta+rst	44,68 (3,36)	38,04 (2,14)	32,58 (1,42)	25,9 (1,15)	22,78 (0,71)	20,37 (0,83)
11*shazoo+rst	47,17 (2,91)	38,09 (3,18)	32,66 (1,69)	25,22 (0,81)	22,33 (1,01)	19,58 (1,42)
11*mucca+rst	44,07 (3,18)	37,18 (2,54)	32,15 (1,51)	25,13 (0,66)	22,33 (0,89)	19,84 (1,5)

Table A.4: In this table we show the averaged classification error rates (percentages) achieved by the various algorithms on the COAUTHOR dataset. Algorithms are trained ten times over a random subset of 2,5%, 5%, 10%, 25%, 33% and 50% of the total number of nodes and tested on the remaining ones. gtg-ess and labprop are used as yardsticks for the comparison. Standard deviations are reported in parenthesis.

Predictors	2,5%	5%	10%	25%	33%	50%
GTG-ESS	30,58 (2,8)	26,33 (2,13)	20,93 (1,47)	17,37 (0,71)	16,93 (1,06)	15,32 (1,03)
LABPROP	40,09 (16,52)	26,48 (2,2)	20,45 (1,61)	16,76 (0,8)	16,72 (0,82)	15,24 (0,91)
GPA+RST	60,22 (7,81)	58,67 (5,97)	44,77 (5,92)	41,83 (5,16)	41,61 (8,16)	37,64 (4,85)
GPA+MST	59,91 (6,38)	52,18 (6,07)	45,38 (5,84)	39,35 (3,88)	39,61 (6,6)	35,04 (5,22)
GPA+BFST	60,24 (5,95)	53,65 (4,13)	45,36 (6,58)	39,29 (8,17)	38,83 (7,15)	33,76 (2,34)
WTA+RST	50,75 (2,85)	43,45 (2,18)	37,59 (1,61)	29,51 (0,86)	27,74 (0,81)	25,68 (1,16)
WTA+MST	54,92 (2,88)	48,85 (2,58)	40,02 (0,85)	31,88 (1,15)	30,18 (1,23)	27,22 (1,02)
WTA+BFST	55,46 (2,53)	48,12 (2,48)	39,71 (2,44)	32,67 (1,18)	30,16 (1,1)	27,84 (1,27)
SHAZOO+RST	46,38 (5,2)	38,4 (3,16)	31,7 (1,69)	23,62 (0,82)	21,92 (0,6)	19,53 (0,85)
SHAZOO+MST	50,07 (3,06)	42,21 (3,2)	33,41 (2,32)	24,22 (1,54)	23,44 (1,77)	20,93 (1)
SHAZOO+BFST	52,63 (5,25)	41,55 (3,56)	32,07 (2,14)	24,87 (1,18)	23,09 (0,84)	20,8 (1,18)
MUCCA+RST	46,36 (4,93)	39,18 (2,61)	31,65 (1,06)	24,16 (0,98)	22,37 (0,7)	19,88 (1,19)
MUCCA+MST	48,03 (2,86)	40,76 (2,93)	32,74 (2,02)	24,55 (1,57)	23,77 (1,84)	21,16 (1,13)
MUCCA+BFST	51,64 (5,53)	40,85 (3,29)	32,08 (2,22)	25,02 (1,31)	23,24 (0,99)	20,88 (0,92)
3*WTA+RST	43,83 (3,18)	36,85 (1,32)	31,09 (1,44)	24,67 (1,01)	22,92 (0,55)	20,74 (0,67)
3*SHAZOO+RST	44,06 (4,49)	34,04 (0,9)	28,71 (1,28)	21,31 (0,75)	19,76 (0,75)	18,04 (0,85)
3*MUCCA+RST	43,32 (4,59)	33,85 (1,22)	28,68 (1,26)	21,49 (0,91)	19,94 (0,97)	18,19 (0,84)
7*WTA+RST	35,47 (2,14)	29,31 (1,42)	25,02 (0,33)	20,05 (0,86)	18,67 (0,57)	17,57 (1,01)
7*SHAZOO+RST	37,42 (2,54)	30,13 (1,88)	25,3 (1,17)	19,22 (0,98)	18,13 (0,7)	16,59 (0,71)
7*MUCCA+RST	35,44 (2,82)	29,37 (1,78)	25,25 (1,18)	19,26 (1,09)	18,07 (0,87)	16,42 (0,62)
11*WTA+RST	31,63 (2,18)	27,42 (2,03)	23,34 (1,34)	18,72 (0,8)	17,64 (0,46)	16,58 (1,24)
11*SHAZOO+RST	36,02 (2,05)	28,54 (2,05)	23,88 (1,3)	18,88 (0,92)	17,81 (0,65)	16,41 (0,81)
11*MUCCA+RST	33,92 (2,1)	27,88 (1,98)	23,99 (1,14)	18,75 (1)	17,77 (0,78)	16,22 (0,61)

Table A.5: In this table we show the averaged classification error rates (percentages) achieved by the various algorithms on the CORA dataset. Algorithms are trained ten times over a random subset of 2,5%, 5%, 10%, 25%, 33% and 50% of the total number of nodes and tested on the remaining ones. GTG-ESS and LABPROP are used as yardsticks for the comparison. Standard deviations are reported in parenthesis.

Predictors	2,5%	5%	10%	25%	33%	50%
gtg-ess	39,08 (1,44)	37,97 (1,43)	35,34 (0,72)	32,62 (0,27)	31,98 (0,4)	30,46 (0,32)
labprop	51,21 (1,53)	47,45 (1,11)	40,6 (1,42)	34,29 (1)	32,19 (0,95)	30,23 (0,2)
gpa+rst	57,53 (2,23)	54,31 (3,42)	53,24 (3,68)	53,57 (5,46)	51,49 (3,55)	49,09 (2,66)
gpa+mst	58,48 (6,91)	57,05 (5,43)	56,4 (5,68)	49,8 (4,76)	56,94 (8,72)	48,94 (6,27)
gpa+bfst	60,4 (5,15)	55,53 (3,11)	54,15 (1,92)	52,52 (4,86)	51,63 (2,59)	49,74 (3,42)
wta+rst	52,97 (1,34)	50,24 (0,96)	47,74 (0,42)	44,28 (0,35)	43,24 (0,45)	41,86 (0,49)
wta+mst	51,52 (1,55)	48,93 (0,74)	46,01 (0,72)	42,74 (0,42)	41,66 (0,34)	39,37 (0,29)
wta+bfst	56,09 (1,21)	52,68 (0,83)	50,55 (0,68)	47,31 (0,48)	46,46 (0,62)	45,28 (0,82)
shazoo+rst	52,95 (1,24)	48,2 (0,89)	45,39 (0,57)	41,09 (0,42)	39,46 (0,52)	37,97 (0,39)
shazoo+mst	58,08 (3,95)	53,49 (2,73)	51,15 (2,87)	46,27 (1,86)	44,97 (0,56)	41,19 (0,97)
shazoo+bfst	53,15 (1,7)	49,55 (1,37)	45,58 (1,21)	42,61 (0,67)	42,14 (0,93)	40,74 (0,71)
mucca+rst	52,31 (1,67)	47,76 (0,8)	45,47 (0,58)	41,21 (0,46)	39,67 (0,44)	38,03 (0,46)
mucca+mst	55,93 (3,58)	51,49 (2,41)	49,35 (2,52)	45,22 (1,72)	44,28 (0,6)	40,82 (1,04)
mucca+bfst	52,6 (1,42)	49,33 (1,28)	45,53 (1,14)	42,6 (0,65)	42,21 (0,95)	40,78 (0,69)
3*wta+rst	50,27 (0,9)	46,51 (0,89)	43,97 (0,65)	40,17 (0,34)	38,97 (0,37)	37,37 (0,32)
3*shazoo+rst	50,68 (1,14)	46,33 (1,34)	42,82 (0,8)	38,13 (0,35)	36,68 (0,42)	35,15 (0,46)
3*mucca+rst	49,55 (0,81)	45,49 (1,05)	42,52 (0,6)	37,99 (0,32)	36,71 (0,41)	35,17 (0,41)
7*wta+rst	44,37 (0,86)	41,24 (0,49)	39,07 (0,44)	35,54 (0,26)	34,41 (0,27)	32,89 (0,3)
7*shazoo+rst	47,04 (0,8)	42,9 (0,99)	39,26 (0,67)	35,16 (0,38)	33,87 (0,22)	32,52 (0,33)
7*mucca+rst	45,23 (0,78)	41,66 (0,85)	38,76 (0,58)	34,97 (0,3)	33,8 (0,23)	32,57 (0,27)
11*wta+rst	42,73 (1,12)	39,44 (0,73)	37,18 (0,42)	34,11 (0,34)	32,87 (0,21)	31,35 (0,34)
11*shazoo+rst	45,89 (0,92)	41,69 (1)	37,94 (0,53)	34,28 (0,29)	32,78 (0,35)	31,69 (0,47)
11*mucca+rst	43,75 (0,93)	40,34 (0,87)	37,32 (0,46)	34,04 (0,22)	32,73 (0,33)	31,68 (0,37)

Table A.6: In this table we show the averaged classification error rates (percentages) achieved by the various algorithms on the IMDB dataset. Algorithms are trained ten times over a random subset of 2,5%, 5%, 10%, 25%, 33% and 50% of the total number of nodes and tested on the remaining ones. gtg-ess and labprop are used as yardsticks for the comparison. Standard deviations are reported in parenthesis.

Predictors	2,5%	5%	10%	25%	33%	50%
GTG-ESS	22,62 (0,64)	21,11 (0,64)	19,6 (0,34)	18,37 (0,27)	18,15 (0,27)	17,91 (0,32)
LABPROP	24,49 (0,98)	22,11 (1,96)	19,75 (0,34)	18,4 (0,29)	18,17 (0,3)	17,92 (0,32)
GPA+RST	51,2 (4,69)	49,45 (7,66)	42,2 (4,77)	39,62 (5,74)	38,94 (5,86)	35,65 (3,24)
GPA+MST	43,1 (4,73)	40,57 (2,46)	35,88 (4,63)	32,44 (2,17)	29,29 (2,21)	30,01 (4,26)
GPA+BFST	45,22 (4,79)	42,31 (5,73)	40,76 (3,93)	32,69 (2,37)	31,08 (2,66)	30,8 (3,82)
WTA+RST	37,89 (1,27)	34,26 (0,76)	31,33 (0,44)	27,09 (0,54)	26,35 (0,37)	25,5 (0,31)
WTA+MST	38,02 (1,36)	35,5 (0,73)	31,61 (0,59)	27,86 (0,35)	27,11 (0,24)	26,12 (0,31)
WTA+BFST	39,94 (1,62)	35,88 (1,16)	32,69 (0,66)	28,4 (0,3)	27,26 (0,56)	26,33 (0,47)
SHAZOO+RST	40,68 (2,09)	32,35 (2,1)	27,15 (0,69)	22,12 (0,52)	21,28 (0,35)	20,38 (0,38)
SHAZOO+MST	32,28 (1,06)	29,35 (1,18)	26,16 (0,55)	22,23 (0,55)	21,4 (0,35)	20,52 (0,25)
SHAZOO+BFST	36,35 (2,21)	30,29 (1,77)	27,36 (0,99)	22,57 (0,41)	21,54 (0,41)	20,86 (0,3)
MUCCA+RST	40,47 (2,25)	32,74 (1,64)	27,49 (0,76)	22,3 (0,56)	21,43 (0,4)	20,49 (0,43)
MUCCA+MST	32,47 (1,09)	29,82 (1,16)	26,4 (0,59)	22,44 (0,57)	21,6 (0,4)	20,65 (0,28)
MUCCA+BFST	36,37 (2,32)	30,74 (1,69)	27,67 (1,05)	22,75 (0,42)	21,75 (0,4)	20,95 (0,29)
3*WTA+RST	32,63 (1,28)	28,96 (0,54)	26,05 (0,36)	22,78 (0,28)	22,24 (0,34)	21,53 (0,46)
3*SHAZOO+RST	36,97 (1,79)	28,12 (0,92)	23,89 (0,46)	20,19 (0,22)	19,93 (0,32)	19,23 (0,43)
3*MUCCA+RST	35,96 (1,52)	28 (0,9)	24,21 (0,44)	20,33 (0,26)	20,05 (0,36)	19,24 (0,42)
7*WTA+RST	26,7 (0,69)	24,46 (0,49)	22,73 (0,28)	20,37 (0,21)	19,77 (0,31)	19,14 (0,38)
7*SHAZOO+RST	31,14 (2,03)	24,43 (0,81)	21,64 (0,31)	19,32 (0,22)	19,1 (0,3)	18,64 (0,4)
7*MUCCA+RST	29,81 (1,48)	24,3 (0,85)	21,8 (0,3)	19,43 (0,24)	19,2 (0,31)	18,65 (0,39)
11*WTA+RST	24,98 (0,57)	23,18 (0,34)	21,51 (0,28)	19,55 (0,34)	19,15 (0,21)	18,43 (0,42)
11*SHAZOO+RST	28,75 (1,83)	23,22 (0,79)	20,95 (0,32)	19,05 (0,21)	18,9 (0,31)	18,55 (0,4)
11*MUCCA+RST	27,38 (1,5)	23,12 (0,71)	21,05 (0,36)	19,1 (0,21)	18,93 (0,29)	18,54 (0,41)

Table A.7: In this table we show the averaged classification error rates (percentages) achieved by the various algorithms on the PUBMED dataset. Algorithms are trained ten times over a random subset of 2,5%, 5%, 10%, 25%, 33% and 50% of the total number of nodes and tested on the remaining ones. GTG-ESS and LABPROP are used as yardsticks for the comparison. Standard deviations are reported in parenthesis.

Appendix B
Link Classification

B.1 Proofs regarding the treeCutter algorithm

Proof of Theorem 6.4

Proof. A common tool shared by all three implementations is a preprocessing step.

Given a subtree T' of the input graph G we preliminarily perform a visit of all its vertices (e.g., a depth-first visit) tagging each *node* by a binary label y_i as follows. We start off from an arbitrary node $i \in V_{T'}$, and tag it $y_i = +1$. Then, each adjacent vertex j in T' is tagged by $y_j = y_i \cdot Y_{i,j}$. The key observation is that, after all nodes in T' have been labeled this way, for any pair of vertices $u, v \in V_{T'}$ we have $\pi_{T'}(i,j) = y_i \cdot y_j$, i.e., we can easily compute the parity of $\mathrm{P}_{T'}(u,v)$ in *constant* time. The total time taken for labeling all vertices in $V_{T'}$ is therefore $\mathcal{O}(|V_{T'}|)$.

With the above fast tagging tool in hand, we are ready to sketch the implementation details of the three algorithms.

Part 1. We draw the spanning tree T of G and tag as described above all its vertices in time $\mathcal{O}(|V|)$. We can execute the first 6 lines of the pseudocode in Figure 5 in time $\mathcal{O}(|E|)$ as follows. For each subtree $T_i \subset T$ rooted at i returned by extractTreelet, we assign to each of its nodes a pointer to its root i. This way, given any pair of vertices, we can now determine whether they belong to same subtree in constant time. We also mark node i and all the leaves of each subtree. This operation is useful when visiting each subtree starting from its root. Then the set $\mathcal{T}$ contains just the roots of all the subtree returned by extractTreelet. This takes $\mathcal{O}(|V_T|)$ time. For each $T' \in \mathcal{T}$ we also mark each edge in $E_{T'}$ so as to determine in constant time whether or not it is part of T'. We visit the nodes of each subtree T' whose root is in $\mathcal{T}$, and for any edge (i,j) connecting two vertices of T', we predict in constant time $Y_{i,j}$ by $y_i \cdot y_j$. It is then easy to see that the total time it takes to compute these predictions on all subtrees returned by EXTRACTTREELET is $\mathcal{O}(|E|)$.

To finish up the rest, we allocate a vector $\boldsymbol{v}$ of $|V|$ records, each record v_i storing only one edge in E_G and its label. For each vertex $r \in \mathcal{T}$ we repeat the following steps. We visit the subtree T' rooted at r. For brevity, denote by $\mathrm{root}(i)$ the root of the subtree which i belongs to. For any edge connecting the currently visited node i to a node $j \notin V_{T'}$, we perform the following operations: if $v_{\mathrm{root}(j)}$ is empty, we query the label $Y_{i,j}$ and insert edge (i,j) together with $Y_{i,j}$ in $v_{\mathrm{root}(j)}$. If instead $v_{\mathrm{root}(j)}$ is not empty, we set (i,j) to be part of the test set and predict its label as

$$\hat{Y}_{i,j} \leftarrow \pi_T(i,z') \cdot Y_{z',z''} \cdot \pi_T(z'',j) = y_i \cdot y_{z'} \cdot Y_{z',z''} \cdot y_{z''} \cdot y_j ,$$

where (z',z'') is the edge contained in $v_{\mathrm{root}(j)}$. We mark each predicted edge so as to avoid to predict its label twice. We finally dispose the content of vector $\boldsymbol{v}$.

The execution of all these operations takes time overall linear in $|E|$, thereby concluding the proof of Part 1.

Part 2. We rely on the notation just introduced. We exploit an additional data structure, which takes extra $\mathcal{O}(|V|)$ space. This is a heap H whose records h_i contain references to vertices $i \in V$. Furthermore, we also create a link connecting i to record h_i. The priority key ruling heap H is the degree of each vertex referred to by its records. With this data structure in hand, we are able to find the vertex having the highest degree (i.e., the *top* element of the heap) in constant time. The heap also allows us to execute in logarithmic time a *pop* operation, which eliminates the *top* element from the heap.

In order to mimic the execution of the algorithm, we perform the following operations. We create a star S centered at the vertex referred to by the top element of H connecting it with all the adjacent vertices in G. We mark as "not-in-use" each leaf of S. Finally, we eliminate the element pointing to the center of S from H (via a pop operation) and create a pointer from each leaf of S to its central vertex. We keep creating such star graphs until H becomes empty. Compared to the creation of the first star, all subsequent stars essentially require the same sequence of operations. The only difference with the former is that when the top element of H is marked as not-in-use, we simply pop it away. This is because any new star that we create is centered at a node that is not part of any previously generated star. The time it takes to perform the above operations is $\mathcal{O}(|V| \log |V|)$.

Once we have created all the stars, we predict all the test edges the very same way as we described for TREECUTTER (labeling the vertices of each star, using a set $\mathcal{T}$ containing all the star centers and the vector $\boldsymbol{v}$ for computing the predictions). Since for each edge we perform only a constant number of operations, the proof of Part 2 is concluded.

Part 3. TREELETSTAR(k) can be implemented by combining the implementation of TREECUTTER with the implementation of STARMAKER. In a first phase, the algorithm works as TREECUTTER, creating a set $\mathcal{T}$ containing the roots of all the subtrees with diameter bounded by k. We label all the vertices of each subtree and create a pointer from each node i to $\mathrm{root}(i)$. Then, we visit all these subtrees and create a graph $G' = (V', E')$ having the following properties: V' coincides with $\mathcal{T}$, and there exists an edge $(i, j) \in E'$ if and only if there exists at least one edge connecting the subtree rooted at i to the subtree rooted at j. We also use two vectors $\boldsymbol{u}$ and $\boldsymbol{u}'$, both having $|V|$ components, mapping each vertex in V to a vertex in V', and viceversa. Using H on G', the algorithm splits the whole set of subtrees into stars of subtrees. The root of the subtree which is the center of each star is stored in a set $\mathcal{S} \subseteq \mathcal{T}$. In addition to these operations, we create a pointer from each vertex of S to r. For each $r \in \mathcal{S}$, the algorithm predicts the labels of all edges connecting pairs of vertices belonging to S using a vector $\boldsymbol{v}$ as for TREECUTTER. Then, it performs a visit of S for the purpose of relabeling all its vertices according to the query set edges that connect the subtree in the center of S with all its other subtrees. Finally, for each vertex of $\mathcal{S}$, we use vector $\boldsymbol{v}$ as in TREECUTTER and STARMAKER for selecting the query set edges connecting the stars of subtrees so created and for predicting all the remaining test edges.

Now, G' is a graph that can be created in $\mathcal{O}(|E|)$ time. The time it takes for operating with H on G' is $\mathcal{O}(|V'| \log |V'|) = \mathcal{O}\left(\frac{|V|}{k} \log \frac{|V|}{k}\right)$, the equality deriving from the fact that each subtree with diameter equal to k contains at least $k+1$ vertices, thereby making $|V'| \leq \frac{|V|}{k}$. Since the remaining operations need constant time per edge in E, this concludes the proof.

Appendix C
Networks of Bandits

C.1 Proofs regarding the GOB.Lin algorithm

Proof of Theorem 7.1

Proof. Define

$$\widetilde{U} = A_{\otimes}^{1/2} U \qquad \text{and} \qquad U = (\boldsymbol{u}_1^\top, \boldsymbol{u}_2^\top, \ldots, \boldsymbol{u}_n^\top)^\top \in \mathbb{R}^{dn} .$$

Let then t be a fixed time step, and introduce the following shorthand notation:

$$\boldsymbol{x}_t^* = \operatorname*{argmax}_{k=1,\ldots,c_t} \boldsymbol{u}_{i_t}^\top \boldsymbol{x}_{t,k} \qquad \text{and} \qquad \boldsymbol{\phi}_t^* = \operatorname*{argmax}_{k=1,\ldots,c_t} \widetilde{U}^\top \widetilde{\boldsymbol{\phi}}_{t,k} .$$

Notice that, for any k we have

$$\widetilde{U}^\top \widetilde{\boldsymbol{\phi}}_{t,k} = U^\top A_{\otimes}^{1/2} A_{\otimes}^{-1/2} \boldsymbol{\phi}_{i_t}(\boldsymbol{x}_{t,k}) = U^\top \boldsymbol{\phi}_{i_t}(\boldsymbol{x}_{t,k}) = \boldsymbol{u}_{i_t}^\top \boldsymbol{x}_{t,k} .$$

Hence we decompose the time-t regret r_t as follows:

$$\begin{aligned}
r_t &= \boldsymbol{u}_{i_t}^\top \boldsymbol{x}_t^* - \boldsymbol{u}_{i_t}^\top \boldsymbol{x}_{t,k_t} \\
&= \widetilde{U}^\top \widetilde{\boldsymbol{\phi}}_t^* - \widetilde{U}^\top \widetilde{\boldsymbol{\phi}}_{t,k_t} \\
&= \widetilde{U}^\top \widetilde{\boldsymbol{\phi}}_t^* - \boldsymbol{w}_{t-1}^\top \widetilde{\boldsymbol{\phi}}_t^* + \boldsymbol{w}_{t-1}^\top \widetilde{\boldsymbol{\phi}}_t^* + \text{CB}(\widetilde{\boldsymbol{\phi}}_t^*) - \text{CB}(\widetilde{\boldsymbol{\phi}}_t^*) - \widetilde{U}^\top \widetilde{\boldsymbol{\phi}}_{t,k_t} \\
&\le \widetilde{U}^\top \widetilde{\boldsymbol{\phi}}_t^* - \boldsymbol{w}_{t-1}^\top \widetilde{\boldsymbol{\phi}}_t^* + \boldsymbol{w}_{t-1}^\top \widetilde{\boldsymbol{\phi}}_{t,k_t} + \text{CB}(\widetilde{\boldsymbol{\phi}}_{t,k_t}) - \text{CB}(\widetilde{\boldsymbol{\phi}}_t^*) - \widetilde{U}^\top \widetilde{\boldsymbol{\phi}}_{t,k_t},
\end{aligned}$$

the inequality deriving from

$$\boldsymbol{w}_{t-1}^\top \widetilde{\boldsymbol{\phi}}_{t,k_t} + \text{CB}(\widetilde{\boldsymbol{\phi}}_{t,k_t}) \ge \boldsymbol{w}_{t-1}^\top \widetilde{\boldsymbol{\phi}}_{t,k} + \text{CB}(\widetilde{\boldsymbol{\phi}}_{t,k}), \qquad k = 1, \ldots, c_t.$$

At this point, we rely on [1] (Theorem 2 therein with $\lambda = 1$) to show that

$$\left|\widetilde{U}^\top \widetilde{\boldsymbol{\phi}}_t^* - \boldsymbol{w}_{t-1}^\top \widetilde{\boldsymbol{\phi}}_t^*\right| \le \mathrm{CB}(\widetilde{\boldsymbol{\phi}}_t^*) \qquad \text{and} \qquad \left|\boldsymbol{w}_{t-1}^\top \widetilde{\boldsymbol{\phi}}_{t,k_t} - \widetilde{U}^\top \widetilde{\boldsymbol{\phi}}_{t,k_t}\right| \le \mathrm{CB}(\widetilde{\boldsymbol{\phi}}_{t,k_t})$$

both hold simultaneously for all t with probability at least $1-\delta$ over the noise sequence. Hence, with the same probability,

$$r_t \le 2\,\mathrm{CB}(\widetilde{\boldsymbol{\phi}}_{t,k_t})$$

holds uniformly over t. Thus the cumulative regret $\sum_{t=1}^T r_t$ satisfies

$$\begin{aligned}
\sum_{t=1}^T r_t &\le \sqrt{T\sum_{t=1}^T r_t^2} \\
&\le 2\sqrt{T\sum_{t=1}^T \Big(\mathrm{CB}(\widetilde{\boldsymbol{\phi}}_{t,k_t})\Big)^2} \\
&\le 2\sqrt{T\left(\sigma\sqrt{\ln\frac{|M_T|}{\delta}} + \|\widetilde{U}\|\right)^2 \sum_{t=1}^T \widetilde{\boldsymbol{\phi}}_{t,k_t}^\top M_{t-1}^{-1}\widetilde{\boldsymbol{\phi}}_{t,k_t}}\ .
\end{aligned}$$

Now, using (see, e.g., [38])

$$\sum_{t=1}^T \widetilde{\boldsymbol{\phi}}_{t,k_t}^\top M_{t-1}^{-1}\widetilde{\boldsymbol{\phi}}_{t,k_t} \le (1 + \max_{k=1,\dots,c_t}\|\widetilde{\phi}_{t,k}\|^2)\ln|M_T|\ ,$$

with

$$\begin{aligned}
\max_{k=1,\dots,c_t}\|\widetilde{\phi}_{t,k}\|^2 &= \max_{k=1,\dots,c_t}\phi_{i_t}(\boldsymbol{x}_{t,k})A_\otimes^{-1}\phi_{i_t}(\boldsymbol{x}_{t,k}) \\
&\le \max_{k=1,\dots,c_t}\|\phi_{i_t}(\boldsymbol{x}_{t,k})\|^2 \\
&= \max_{k=1,\dots,c_t}\|\boldsymbol{x}_{t,k}\|^2 \\
&\le B^2\ ,
\end{aligned}$$

along with $(a+b)^2 \le 2a^2 + 2b^2$ applied with $a = \sigma\sqrt{\ln\frac{|M_T|}{\delta}}$ and $b = \|\widetilde{U}\|$ yields

$$\sum_{t=1}^T r_t \le 2\sqrt{T\left(2\sigma^2\ \ln\frac{|M_T|}{\delta} + 2\|\widetilde{U}\|^2\right)\ (1+B^2)\ln|M_T|}\ .$$

Finally, observing that

$$\|\widetilde{U}\|^2 = U^\top A_\otimes U = L(\boldsymbol{u}_1,\dots,\boldsymbol{u}_n)$$

gives the desired bound.

References

1. Yasin Abbasi-Yadkori, Dávid Pál, and Csaba Szepesvári. Improved algorithms for linear stochastic bandits. In *NIPS*, pages 2312–2320, 2011.
2. Jacob Abernethy, Olivier Chapelle, and Carlos Castillo. Graph regularization methods for web spam detection. *Machine Learning*, 81(2):207–225, 2010.
3. Lada A. Adamic and Natalie Glance. The political blogosphere and the 2004 u.s. election: divided they blog. In *Proceedings of the 3rd international workshop on Link discovery*, LinkKDD '05, pages 36–43, New York, NY, USA, 2005. ACM.
4. Omar Ali, Giovanni Zappella, Tijl De Bie, and Nello Cristianini. An empirical comparison of label prediction algorithms on automatically inferred networks. In *ICPRAM (2)*, pages 259–268, 2012.
5. Kareem Amin, Michael Kearns, and Umar Syed. Graphical models for bandit problems. In *Uncertainty in Artificial Intelligence: Proceedings of the Twenty-Seventh Conference*. ACM, 2011.
6. Daniar Asanov. Algorithms and methods in recommender systems, 2011.
7. Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *The Journal of Machine Learning Research*, 3:397–422, 2003.
8. Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multi-armed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.
9. Mikhail Belkin, Irina Matveeva, and Partha Niyogi. Regularization and semi-supervised learning on large graphs. In *Learning theory*, pages 624–638. Springer, 2004.
10. Yoshua Bengio, Olivier Delalleau, and Nicolas Le Roux. Label propagation and quadratic criterion. In *Semi-Supervised Learning*, pages 193–216. MIT Press, 2006.
11. Avrim Blum and Shuchi Chawla. Learning from labeled and unlabeled data using graph mincuts. In *Proceedings of the 18th International Conference on Machine Learning*. Morgan Kaufmann, 2001.
12. Toine Bogers. Movie recommendation using random walks over the contextual graph. In *CARS'10: Proceedings of the 2nd Workshop on Context-Aware Recommender Systems*, 2010.
13. Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM, 2011.
14. Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Antti Ukkonen. Chromatic correlation clustering. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1321–1329. ACM, 2012.

15. Iván Cantador, Peter Brusilovsky, and Tsvi Kuflik. 2nd workshop on information heterogeneity and fusion in recommender systems (HetRec 2011). In *Proceedings of the 5th ACM Conference on Recommender Systems*, RecSys 2011. ACM, 2011.
16. Stéphane Caron and Smriti Bhagat. Mixing bandits: a recipe for improved cold-start recommendations in a social network. In *Proceedings of the 7th Workshop on Social Network Mining and Analysis*, page 11. ACM, 2013.
17. Stéphane Caron, Branislav Kveton, Marc Lelarge, and Smriti Bhagat. Leveraging side observations in stochastic bandits. In *The 28th Conference on Uncertainty in Artificial Intelligence*, pages 142–151, 2012.
18. Dorwin Cartwright and Frank Harary. Structure balance: A generalization of Heider's theory. *Psychological review*, 63(5):277–293, 1956.
19. Carlos Castillo, Debora Donato, Aristides Gionis, Vanessa Murdock, and Fabrizio Silvestri. Know your neighbors: Web spam detection using the web topology. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 423–430. ACM, 2007.
20. Giovanni Cavallanti, Nicolò Cesa-Bianchi, and Claudio Gentile. Linear algorithms for online multitask classification. *Journal of Machine Learning Research*, 11:2597–2630, 2010.
21. N. Cesa-Bianchi, C. Gentile, and F. Orabona. Robust bounds for classification via selective sampling. In *Proceedings of the 26th International Conference on Machine Learning*. Omnipress, 2009.
22. Nicolò Cesa-Bianchi, Claudio Gentile, and Fabio Vitale. Fast and optimal prediction of a labeled tree. In *Proceedings of the 22nd Annual Conference on Learning Theory*, 2009.
23. Nicolo Cesa-Bianchi, Claudio Gentile, and Fabio Vitale. Predicting the labels of an unknown graph via adaptive exploration. *Theoretical computer science*, 412(19):1791–1804, 2011.
24. Nicolò Cesa-Bianchi, Claudio Gentile, Fabio Vitale, and Giovanni Zappella. A linear time active learning algorithm for link classification. *Advances in Neural Information Processing Systems*, pages 1619–1627.
25. Nicolò Cesa-Bianchi, Claudio Gentile, Fabio Vitale, and Giovanni Zappella. Random spanning trees and the prediction of weighted graphs. In *Proceedings of the 27th International Conference on Machine Learning*, 2010.
26. Nicolò Cesa-Bianchi, Claudio Gentile, Fabio Vitale, and Giovanni Zappella. A correlation clustering approach to link classification in signed networks. In *Proceedings of the 25th conference on learning theory (COLT 2012)*, 2012.
27. Nicolò Cesa-Bianchi, Claudio Gentile, Fabio Vitale, and Giovanni Zappella. Random spanning trees and the prediction of weighted graphs. *Journal of Machine Learning Research*, 14:1005–1039, 2013.
28. Nicolò Cesa-Bianchi, Claudio Gentile, and Giovanni Zappella. A gang of bandits. In *NIPS*, 2013.
29. Nicolò Cesa-Bianchi, Claudio Gentile, and Giovanni Zappella. A gang of bandits. *arXiv preprint arXiv:1306.0811*, 2013.
30. Nicolò Cesa-Bianchi and Gabor Lugosi. *Prediction, Learning and Games*. Cambidge University Press, 2006.
31. Kai-Yang Chiang, Nagarajan Natarajan, Ambuj Tewari, and Inderjit S Dhillon. Exploiting longer cycles for link prediction in signed networks. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1157–1162. ACM, 2011.
32. Eunjoon Cho, Seth A Myers, and Jure Leskovec. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1082–1090. ACM, 2011.

33. Wei Chu, Lihong Li, Lev Reyzin, and Robert E Schapire. Contextual bandits with linear payoff functions. In *Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011.
34. Marie-Christine Costa, Lucas Létocart, and Frédéric Roupin. Minimal multicut and maximal integer multiflow: a survey. *European Journal of Operational Research*, 162(1):55–69, 2005.
35. Koby Crammer and Claudio Gentile. Multiclass classification with bandit feedback using adaptive regularization. *Machine Learning*, 90(3):347–383, 2013.
36. Elias Dahlhaus, David S Johnson, Christos H Papadimitriou, Paul D Seymour, and Mihalis Yannakakis. The complexity of multiway cuts. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 241–251. ACM, 1992.
37. Varsha Dani, Thomas P Hayes, and Sham M Kakade. Stochastic linear optimization under bandit feedback. In *COLT*, pages 355–366, 2008.
38. O. Dekel, C. Gentile, and K. Sridharan. Robust selective sampling from single and multiple teachers. In *COLT*, pages 346–358, 2010.
39. Inderjit S Dhillon, Yuqiang Guan, and Brian Kulis. Weighted graph cuts without eigenvectors a multilevel approach. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 29(11):1944–1957, 2007.
40. David Easley and Jon Kleinberg. *Networks, crowds, and markets*, volume 8. Cambridge Univ Press, 2010.
41. M. Elkin, Y. Emek, D.A. Spielman, and S.-H. Teng. Lower-stretch spanning trees. *SIAM Journal on Computing*, 38(2):608–628, 2010.
42. Aykut Erdem and Marcello Pelillo. Graph transduction as a noncooperative game. *Neural Computation*, 24(3):700–723, 2012.
43. Theodoros Evgeniou and Massimiliano Pontil. Regularized multi–task learning. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '04, pages 109–117, New York, NY, USA, 2004. ACM.
44. Giuseppe Facchetti, Giovanni Iacono, and Claudio Altafini. Computing global structural balance in large-scale signed social networks. *Proceedings of the National Academy of Sciences*, 108(52):20953–20958, 2011.
45. Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.
46. C. Gentile, S. Li, and G. Zappella. Online Clustering of Bandits. *ArXiv e-prints*, January 2014.
47. Quanquan Gu, Charu Aggarwal, Jialu Liu, and Jiawei Han. Selective sampling on graphs for classification. 2013.
48. Quanquan Gu and Jiawei Han. Towards active learning on graphs: An error bound minimization approach. In *ICDM*, pages 882–887, 2012.
49. Ramanthan Guha, Ravi Kumar, Prabhakar Raghavan, and Andrew Tomkins. Propagation of trust and distrust. In *Proceedings of the 13th international conference on World Wide Web*, pages 403–412. ACM, 2004.
50. Ido Guy, Naama Zwerdling, David Carmel, Inbal Ronen, Erel Uziel, Sivan Yogev, and Shila Ofek-Koifman. Personalized recommendation of social software items based on social relations. In *Proceedings of the Third ACM conference on Recommender systems*, pages 53–60. ACM, 2009.
51. Frank Harary. On the notion of balance of a signed graph. *Michigan Mathematical Journal*, 2(2):143–146, 1953.
52. Fritz Heider. Attitudes and cognitive organization. *The Journal of psychology*, 21(1):107–112, 1946.
53. Mark Herbster and Guy Lever. Predicting the labelling of a graph via minimum p-seminorm interpolation. In *Proceedings of the 22nd Annual Conference on Learning Theory*. Omnipress, 2009.
54. Mark Herbster, Guy Lever, and Massimiliano Pontil. Online prediction on large diameter graphs. In *Advances in Neural Information Processing Systems*, pages 649–656, 2008.

55. Mark Herbster, Massimiliano Pontil, and Sergio R Galeano. Fast prediction on a tree. In *Advances in Neural Information Processing Systems*, pages 657–664, 2008.
56. Yuheng Hu, Ajita John, Dorée Duncan Seligmann, and Fei Wang. What were the tweets about? topical associations between public events and twitter feeds. In *ICWSM*, 2012.
57. Giovanni Iacono and Claudio Altafini. Monotonicity, frustration, and ordered response: an analysis of the energy landscape of perturbed large-scale biological networks. *BMC systems biology*, 4(1):83, 2010.
58. Soummya Kar, H Vincent Poor, and Shuguang Cui. Bandit problems in networks: Asymptotically efficient distributed allocation rules. In *Decision and Control and European Control Conference (CDC-ECC), 2011 50th IEEE Conference on*, pages 1771–1778. IEEE, 2011.
59. Frank R Kschischang, Brendan J Frey, and H-A Loeliger. Factor graphs and the sum-product algorithm. *Information Theory, IEEE Transactions on*, 47(2):498–519, 2001.
60. Jérôme Kunegis, Andreas Lommatzsch, and Christian Bauckhage. The slashdot zoo: mining a social network with negative edges. In *Proceedings of the 18th international conference on World wide web*, pages 741–750. ACM, 2009.
61. Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Predicting positive and negative links in online social networks. In *Proceedings of the 19th international conference on World wide web*, pages 641–650. ACM, 2010.
62. Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Signed networks in social media. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1361–1370. ACM, 2010.
63. Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th International Conference on World Wide Web*, pages 661–670. ACM, 2010.
64. Russell Lyons and Yuval Peres. Probability on trees and networks, 2005.
65. Shie Mannor and Ohad Shamir. From bandits to experts: On the value of side-observations. In *NIPS*, pages 684–692, 2011.
66. Paolo Massa and Paolo Avesani. Trust-aware bootstrapping of recommender systems. In *ECAI Workshop on Recommender Systems*, pages 29–33. Citeseer, 2006.
67. S Thomas McCormick, MR Rao, and Giovanni Rinaldi. Easy and difficult objective functions for max cut. *Mathematical Programming*, 94(2-3):459–466, 2003.
68. Charles A Micchelli and Massimiliano Pontil. Kernels for multi–task learning. In *Advances in Neural Information Processing Systems*, pages 921–928, 2004.
69. Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. The MIT Press, 2012.
70. Corrado Monti, Matteo Zignani, Alessandro Rozza, Adam Arvidsson, Giovanni Zappella, and Elanor Colleoni. Modelling political disaffection from twitter data. In *Proceedings of the Second International Workshop on Issues of Sentiment Discovery and Opinion Mining*, page 3. ACM, 2013.
71. John Nash. Non-cooperative games. *The Annals of Mathematics*, 54(2):286–295, 1951.
72. Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, New York, NY, USA, 2007.
73. Alan Said, Ernesto W De Luca, and Sahin Albayrak. How social relationships affect user similarities. In *Proceedings of the 2010 Workshop on Social Recommender Systems*, pages 1–4, 2010.
74. Aleksandrs Slivkins. Contextual bandits with similarity information. *Journal of Machine Learning Research – Proceedings Track*, 19:679–702, 2011.
75. Daniel A Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926, 2011.
76. BT Swapna, Atilla Eryilmaz, and Ness B Shroff. Multi-armed bandits in the presence of side observations in social networks. 2013.

77. Balázs Szörényi, Róbert Busa-Fekete, István Hegedus, Róbert Ormándi, Márk Jelasity, and Balázs Kégl. Gossip-based distributed stochastic bandit algorithms. In *Proceedings of the 30th International Conference on Machine Learning*, 2013.
78. Narseo Vallina-Rodriguez, Salvatore Scellato, Hamed Haddadi, Carl Forsell, Jon Crowcroft, and Cecilia Mascolo. Los twindignados: The rise of the indignados movement on twitter. In *Proceedings of the 2012 ASE/IEEE International Conference on Social Computing and 2012 ASE/IEEE International Conference on Privacy, Security, Risk and Trust*, SOCIALCOM-PASSAT '12, pages 496–501, Washington, DC, USA, 2012. IEEE Computer Society.
79. Fabio Vitale, Nicolò Cesa-bianchi, Claudio Gentile, and Giovanni Zappella. See the tree through the lines: The shazoo algorithm. In *Advances in Neural Information Processing Systems*, pages 1584–1592, 2011.
80. Jörgen W Weibull. *Evolutionary game theory.* MIT press, 1997.
81. Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1113–1120. ACM, 2009.
82. David Bruce Wilson. Generating random spanning trees more quickly than the cover time. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 296–303. ACM, 1996.
83. Giovanni Zappella. A scalable multiclass algorithm for node classification. *MLG Workshop at ICML*, 2012.
84. Giovanni Zappella, Alexandros Karatzoglou, and Linas Baltrunas. Games of friends: a game-theoretical approach for link prediction in online social networks. In *Workshops at the Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.
85. Yan-Ming Zhang, Kaizhu Huang, and Cheng-Lin Liu. Fast and robust graph-based transductive learning via minimum tree cut. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 952–961. IEEE, 2011.
86. Xiaojin Zhu, Zoubin Ghahramani, and John Lafferty. Semi-supervised learning using gaussian fields and harmonic functions. In *ICML*, volume 3, pages 912–919, 2003.