REPORT

Mining the social graph

- project deliverable 4.3

Authors: Chryssanthi Iakovidou, Panagiotis Galopoulos, Symeon Papadopoulos, Yiannis Kompatsiaris, Barbara Guidi, Andrea Michienzi, Laura Ricci, Fabrizio Baiardi, Pau Pamplona, Estella Oncins, Rolf Nyffenegger

Confidentiality: Public

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825585.

HELIOS Commercial Exploitation Requirements Gathering	
project name HELIOS	**Grant agreement #** 825585
Authors Chryssanthi Iakovidou (CERTH), Panagiotis Galopoulos (CERTH), Symeon Papadopoulos (CERTH), Yiannis Kompatsiaris (CERTH), Barbara Guidi (UNIPI), Andrea Michienzi (UNIPI), Laura Ricci (UNIPI), Fabrizio Baiardi (UNIPI), Pau Pamplona (UAB), Estella Oncins (UAB), Rolf Nyffenegger (SWISS TXT)	**Pages** 93
Reviewers Javier Marín Morales (UPV), Kristina Kapanova (TCD)	
Keywords Social network analysis, graph mining, distributed graph analysis, community detection, graph neural networks, interaction prediction	**Deliverable identification code** D4.3
Summary In this deliverable we explore state-of-the art and new approaches for mining heterogeneous social graphs in decentralized time-evolving systems. Such approaches can discover patterns of relations and interactions driven by the underlying preferences of HELIOS users, which in turn can be used to improve the quality of HELIOS applications. In particular, we investigate existing graph mining practices, such as graph clustering and graph neural networks, which can mine the time-evolving organization and preferences of social media users and propose novel protocols and adaptations that help adopt them in the decentralized setting of HELIOS. To this end, we propose a novel community detection protocol for discovering the community structure of decentralized networks, as well as a novel decentralized adaptation of temporal graph neural networks of mining user preferences. The latter can recommend future interactions with similarly high efficacy to their centralized counterpart in social network datasets with similar properties as those of the envisioned HELIOS networks and is deployed in a graph recommendation module that facilitates social recommendation and analysis tasks. To support integration of that module with the rest of the HELIOS platform, we also improve the Contextual Ego Network management library.	
Confidentiality	Public
25/06/2020 Written by Chryssanthi Iakovidou	
Project Coordinator's Contact address Ville Ollikainen, ville.ollikainen@vtt.fi, +358 400 841116	
Distribution HELIOS project partners, subcontractors and the Project Officer	

Table of Contents

1 Introduction **8**

2 Preliminaries **10**

2.1 Social Network Graphs 10
2.2 Graph Theory Concepts 10
2.3 User Embeddings 13

3 Related Work **15**

3.1 Diffusing Note Relevance 15
Personalized PageRank 15
Graph signal processing 16
3.2 Community Detection 18
Centralized Community Detection 18
Decentralized Community Detection 19
3.3 Graph Neural Networks 20
GNN Architecture 20
GNNs for Edge Prediction 23
3.4 Mining Temporal Graphs 25

4 Community Detection in Temporal DSNs **28**

4.1 Decentralized Dynamic Community Detection Challenges 28
4.2 Choosing a Decentralized Community Detection Approach 29
4.3 Overview of the solution 30
4.3 Computing communities 32
Join algorithm 32
Leave algorithm 32
4.4 Protocol for community management 34
Node join 34
Community birth 35
Node leave 35
Moderator leave 36
4.5 Moderator Election Strategies 37

5 Decentralized Social Graph Recommendations **38**

5.1 Preliminary Investigation 39
Personalized PageRank Mining 40
GNN Mining 41
Predicting Static Relations 42
5.2 Evolving User Preferences for Rediscovering Interactions 45
The need to account for temporal preference evolution 45
Temporal degradation of training example importance 46
5.3 From Distributed to Federated Learning 49
5.4 Decentralized Interaction Mining 52
Aggregating device losses 52

Regularizing towards exchanged embeddings 53
Plausible deniability and differential privacy 55
Experiments 56

6 Mining the Heterogeneous Social Graph in HELIOS 58

6.1 Extending the CEN Management Library 58
Moving to a reference-based API 59
Enabling modular development 63
Failsafe recovery 64
Automated instantiation design patterns 65
6.2 Social Graph Recommendation Module 65
Native implementation of matrix operations 66
Mining graph interactions 67
GNNMiner 67
6.3 Peer-to-Peer Simulation 70
Asserting the correctness of the social graph recommendation module 70
Demonstrating the social graph recommendation mining 70

7 Practical Application 72

7.1 Graph Mining in the HELIOS Platform 72
7.2 Relation to Use Cases 72
7.3 User Feedback on the Social Graph Recommendation Module 73

8 Conclusions and Future Work 76

8.1 Conclusions 76
8.2 Future Work 76

Annex I – CEN Management Library API 78

Annex II – Graph Recommendation Module API 83

References 88

List of figures

Figure 1. Example of directed (left) and undirected (right) graphs 10
Figure 2. The dynamic community events that define the lifecycle of a community. 13
Figure 3. The importance assigned to k hops away for different graph filters. 17
Figure 4. Viewing time as a graph. 18
Figure 5. Example neural network architectures for signal processing. 21
Figure 6. GNN architecture based on message passing through graph neighbors. 22
Figure 7. Multilayer GNN architecture relaxing the iterative scheme. 22
Figure 8. GNN architecture for link prediction given a graph convolution operation. 24
Figure 9. LSTM neural layer and the *tanh* activation function used to avoid vanishing gradients... 26
Figure 10. Multiplayer and multigraph view of the same temporal graph. 27
Figure 11. Steps of the node join protocol. 33
Figure 12. Steps of the node leave protocol. 34
Figure 13. Example of linear edge prediction in a graph. 39
Figure 14. An example graph for link prediction. 43
Figure 15. Example of non-positionally-aware representations. 45
Figure 16. Using temporal GNNs to recommend older interactions. 48
Figure 17. A decentralized machine learning scheme from the viewpoint of an ego device that continuously exchanges parameters with all its alters, even if no interactions are ongoing with some of them. 49
Figure 18. Our proposed federated learning scheme. Contrary to Figure 17. The device exchanges parameters (sync) only with the currently interacting alter. 52
Figure 19. Regularization of the first two elements of $H(v|u,T)$ towards zeros and towards the embedding of the neighbor. 54
Figure 20. Comparing the decentralized and non-decentralized versions of the temporal GNN over occurring interactions. 57
Figure 21. UML diagram of classes involved in dynamic storage and loading operations. 59
Figure 22. Theoretical organization of the serializer into a reference pool of objects that can be serialized into files. 60
Figure 23. How the organization of Figure 22 translates to an example in-memory organization of CEN references. 61
Figure 24. The contents of a serialized CEN node. 62
Figure 25. Saving storage space. 63
Figure 26. Parameter exchange scheme between the graph recommendation module of HELIOS devices. 65
Figure 27. The most important data structures and operations of the GNN mining module pertaining to the GNNMiner class. 68
Figure 28. Example outcome of delegating the graph recommendation module’s data to be stored in the CEN management library. 69

List of tables

Table 1. Well-known link scoring functions. 24
Table 2. Information used by the three decentralized community detection approaches 29
Table 3. Social network graphs involved in experiments. 40
Table 4. Number of PPR iterations that account for 99% of random walks. 41
Table 5. AUC for relation recommendation averaged across 5 experiments. 44
Table 6. Primitives of the CEN management library. 58
Table 7. Prospective questions of user feedback 75
Table 8. Package *eu.h2020.helios_social.core.contextualegonetwork* entities. 78
Table 9. *ContextualEgoNetwork* class methods. 79
Table 10. *CrossModuleComponent* abstract class methods. 80
Table 11. *Context* class methods (extends *CrossModuleComponent*). 80
Table 12. *Node* class methods (extends *CrossModuleComponent*). 81
Table 13. *Edge* class methods (extends *CrossModuleComponent*). 81
Table 14. *Interaction* class methods. 81
Table 15. *ContextualEgoNetworkListener* interface methods. 82
Table 16. Package *eu.h2020.helios_social.modules.socialgraphmining* entities. 83
Table 17. *Measure* class methods. 84
Table 18. *SocialGraphMiner* abstract class methods. 84
Table 19. *GNN.GNNMiner* class methods (extends *SocialGraphMiner*). 85
Table 20. *GNN.ContextTrainingExampleData* class methods (used by *GNN.GNNMiner*). 86
Table 21. GNN.*TrainingExample* class methods (used by *GNN.GNNMiner*). 86
Table 22. GNN.*GNNNodeData* class methods (used by *GNN.GNNMiner*). 86
Table 23. *SwitchableMiner* class methods (extends *SocialGraphMiner*). 87

List of Acronyms

Acronym	Description
CEN	Contextual Ego Network
CrossE	Cross-Entropy Loss
DHR	Discovery Hit Rate
DSN	Decentralized Social Network
GNN	Graph Neural Network

1 Introduction

HELIOS is a platform for creating and managing Decentralized Social Networks (DSNs), with the goal of democratizing content production, promotion and monetization while letting users retain control over data ownership and dissemination. In this deliverable, we explore algorithms that mine user behavior to bring social analysis and recommendation capabilities to the decentralized applications and protocols developed in the HELIOS platform. Mining algorithms could enrich user experience by suggesting social actions, such as getting in touch with connections who are still of interest, and providing patterns of social organization that describe the social roles of users. They could also analyze social behaviour to support the decisions of other mechanisms, such as selecting who to send information to so that fast propagation is achieved.

User behavior can be broadly categorized as either active or passive [1,2]. The latter refers to users only consuming content but not engaging in social actions; hence, it seldomly leaves behind data that can be utilized by graph mining algorithms. Instead, in this task we focus on mining the active behavior of social media users, which refers to their performed social actions. These actions can be organized into edges of graphs -called social graphs- whose nodes represent social media users. Mining social graphs can be considered equivalent to mining the user actions they capture.

We recognize that there are two different types of mineable social actions social media users can engage in: forming relations and performing interactions. Relations represent long-term links between them, such as being friends or acquaintances, and are typically bidirectional in the sense that both involved parties need to acknowledge them. Therefore, mining relations can reveal important organizational properties of the social graph, such as community structures that can be exploited for information diffusion, preferences pertaining to the structural role of users within the graph or whether the graph is a small world one, i.e. whether visiting a user from any other can be achieved within a few number of hops.

On the other hand, interactions refer to repeatable social actions, such as sending private or public messages. Since they repeat over time, these form temporal graphs whose edges can appear multiple times and last only as long as the respective action (often, only for the brief instance at which the action is transmitted between user devices). Interactions are often directed, in the sense that it is important to differentiate between their sender and their receiver. It must be pointed out that, for users to interact in social media platforms, they need to have already formed relations; as an edge case, acquaintance relations can be implicitly formed the first time two users interact. In this deliverable we tackle the task of understanding and modelling the latent time-evolving user preferences that drive their interactions. These preferences can be used both to recommend interactions with “old” connections that may be missed when users engage with their “favorite” connections on a regular basis and to recommend relations between users of similar preferences.

The challenges of performing the above-described types of graph mining within the HELIOS platform lie in its decentralized nature. In particular, state-of-the-art graph mining algorithms are not well-suited to decentralized platforms, either because their computations cannot be split between many devices, or because in doing so they fail to account for the intrinsic evolution of such platforms over time. Previously, researchers have proposed distributed graph mining algorithms in which user devices mine (a local area of) the social graph by constantly communicating with either a central service or their graph neighbors until they arrive at a desired solution (see Section 3). However, these approaches are not applicable to fully DSNs, where there is no central service and critical communications can be disrupted by users going offline at irregular intervals.

In this deliverable, we start by investigating popular graph mining algorithms pertaining to the organization of users into community structures and their preferences, placing emphasis on the algorithms that can also capture the temporal dynamics of social networks. These include community detection for mining user relations and graph neural networks for mining user interactions. We then select the most promising of those algorithms and devise novel information exchange protocols as well as comprehensive adjustments that allow them to be used in DSNs, such as the applications developed within the HELIOS platform. In particular, in this deliverable we describe two new types of decentralized graph mining algorithms; decentralized dynamic community detection (see Section 4), which uses a super-peer approach to detect and manage user communities through time, and decentralized temporal graph neural networks (see Section 5), which capture the temporal evolution of user preferences as these are reflected by the preferences of who they currently interact with [3,4].

As we mentioned before, the algorithms and protocols proposed in this deliverable enable graph mining that can improve HELIOS applications, such as those to be developed within the scope of the use cases first described Deliverable 2.1 and revised in Deliverable 2.6. For example, in Use Case A, action recommendations can help users wade through an otherwise large number of possibilities, such as selecting people to connect to in new environments. The matchmaking between users in Use Case B could also be enriched through an understanding of latent user preferences that drive their actions. Finally, in the marathon setting of Use Case C, graph mining could help recommend social actions, such as speaking with fans of similar athletes. At the same time, community detection can be used to support the operations of the HELIOS peer-to-peer network. For instance, the matchmaking of Use Cases A and B could lead people to organize in stable communities according to their interests, which can hence support the diffusion of information between users of common interests.

2 Preliminaries

2.1 Social Network Graphs

A common formalism that enables the analysis of social networks is to model them as graphs $G = (V, E)$,whose nodes V comprise their users and edges $E = V \times V$ the social actions between them, such as relations and interactions. For example, in Figure 1 we show a visual representation of graphs with nodes $V = \{A, B, C, D\}$ and edges $E = \{(A, B), (A, C), (A, D), (C, D)\}$. As we can see in that figure, graphs can be either directed or undirected, depending on whether the element of the edge tuples E are ordered or not respectively, i.e. an edge (A, B) of an undirected graph, is considered the same as (B, A). In the case of interactions, multiple edges could exist between the same two nodes. For example, the edge (A, B) could occur twice - once per every interaction between those users.

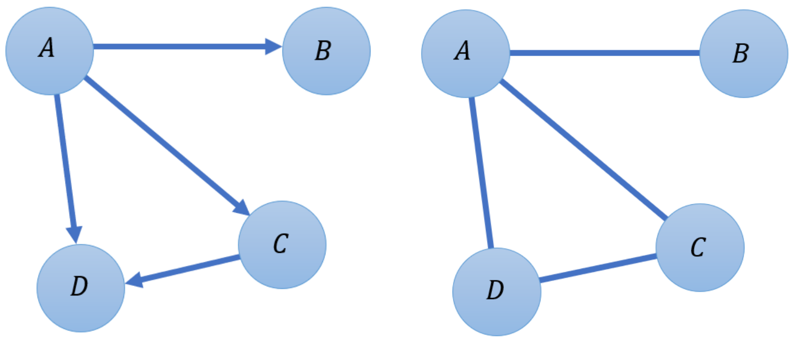


Figure 1. Example of directed (left) and undirected (right) graphs

Usually, social interactions, such as sending or reacting to messages, are modeled as edges of directed graphs, which provide a clear distinction between who the senders and receivers are. For example, in Figure 1 the edge (A, B) of the directed graph could depict that user Ahas messaged user B but not the other way around. On the other hand, the edges of undirected graphs would correspond to modeling bidirectional relations, such as users being friends. Although specific types of social actions could deviate from this formulation (e.g. parent relations need distinguish between the parent and the child), throughout this deliverable we assume the undirected modeling of social actions, unless stated otherwise.

In the above example, the graphs are static in the sense that their edges do not change over time. However, graph nodes can also exhibit dynamism in that they can join or leave the graph and create or remove edges. This concept was described in detail in Deliverable 4.2, alongside existing models for managing dynamic graphs in the literature, as well as the one adopted by HELIOS. As an example, interaction graphs are often dynamic in that new interactions occur over time.

2.2 Graph Theory Concepts

Graph nodes exhibit various types of structures and properties that are of interest to social network mining. In this subsection we present some of the most widely used ones that pertain to the research and development actions of this deliverable.

Ego network
The ego network is a common structure in the fields of complex network analysis and distributed systems that represents a local view of each node's placement in the graph, as first described in Deliverable 4.1. In particular, the ego network of a node u comprises that node and all its graph neighbors, as well as the edges between the starting node and its neighbors as well as between

the neighbors. In this deliverable, we will call uthe ego of its ego network and its neighbours as its alters within the ego network.

Paths

Paths are a pivotal concept of graph theory, which is used to define most network properties or complex structures. In the case of undirected static graphs, a path vu from a source node v to a destination node u is defined as a sequence of nodes $(v, v_1, \ldots, v_n, u)$, such that each node is a neighbor to its predecessor and its successor in the sequence, and there are no duplicates in the sequence. In the case of directed static graphs, the nodes in the path should also be connected with at least one edge from the previous to the next one. In dynamic graphs, there is no commonly agreed view of paths and the one followed in this task will be described later on in subsection 3.4.

Some widely adopted graph theory concepts pertaining to paths are listed below:

- *Path length.* The number of edges in a path.
- *Random walks.* Random walks are processes used to iteratively create paths arising in the graph by selecting an existing path or node and forming a new path by (randomly) visiting one of the last node's neighbors.
- *Connected graphs.* A graph is said to be connected if there exists a path from every one of its nodes to every other.
- *Node distance.* The length of the shortest path between two nodes.
- *Node eccentricity.* The longest distance between the node and any other node.
- *Graph diameter.* The largest node eccentricity in the graph.
- *Small world graphs.* Both offline [5] and online [6] social networks often show a very low diameter which increases only logarithmically with respect to their number of nodes. Since node distance tends to be short between nodes of those graphs, they are often referred to as small world graphs.

The clustering coefficient

The clustering coefficient measures the extent at which nodes of a graph are tightly connected to each other. It has been shown that, in social networks and other real-world graphs, nodes tend to show values of clustering coefficients which are higher with respect to the ones found in random networks, thus forming groups of tightly interconnected nodes with a density that is higher than the one of the whole graph.

Usually two definitions are used in the literature: the *global* clustering coefficient and the *local* clustering coefficient. The global coefficient is based on the notion of *triangle* and *triplet*. A triangle is a graph made of three nodes and three edges, where each node is connected to the other two, whereas a triplet is a triangle with a missing edge. The global clustering coefficient of a graph can then be defined as:

$$CC_{global}(G) = {}^{3T}\!/_{t} \tag{2.1}$$

where Tis the number of triangles in the graph and tis the number of triplets. The global clustering coefficient expresses how many triplets are part of triangles. The local clustering coefficient is instead a measure of how well the neighbours of a reference node u are connected. The local coefficient is defined as:

$$CC_{local}(G) = {}^{|CN(u)|}\!/_{d_u(d_u - 1)} \tag{2.2}$$

where $|CN(u)|$ is the number of neighbours of node u that are connected, and d_u is the degree of node that node, i.e. its total number of neighbors. We can estimate the global clustering coefficient of the graph by averaging the local clustering coefficients of all of its nodes. Also the clustering coefficient was used to determine whether a graph is a small world or not [7].

Node centrality

Centrality is a general concept of how important nodes are within a graph’s structure. Its study has gained a lot of traction in the field of social network analysis because it can uncover users that play a key role in information dissemination, such as influencers. Many centrality measures have been proposed in the literature, the most well-known of which we explore here.

Degree centrality. The most intuitive measure of node centrality is with how many neighbours nodes are connected with. This measure’s distribution over graph nodes often follows a power law, where exponentially more nodes have low (rather than high) degree centrality, regardless of the number of graph nodes.

More advanced centrality measures can be defined using the concept of node distance. Such measures include the closeness centrality, which identifies the importance of nodes based as the sum of the reciprocal of the distances from a node to all the others in the graph, under the assumption that more central nodes should be closer, on average, to all the other nodes of the graph. One can also identify central nodes based on how many include them, a concept referred to as betweenness centrality. The intuition in this case is that nodes with the higher centrality should be frequently visited when moving between pairs of nodes.

Finally, other measures of centrality focus on the structural positioning of nodes. For example, k-decomposition [8,9] assumes that finding nodes with at least *k* neighbors who also have at least *k* neighbors for increasingly larger *k* reveals the hierarchical organization of graphs around central highly-connected components. Whereas Google’s PageRank algorithm [10], which will be explained in more detail in subsection 3.1, considers as more central the graph nodes that would be visited more often when one randomly hops through the graph’s edges.

Community structure

The community structure of graphs is one of the most studied properties of both social media and complex networks [11,12,13], because it can reveal important information concerning the structure of the network. There is no universally accepted formal definition of what a community is. The lack of consensus among researchers on a formal definition has brought the proliferation of several definitions of community and, along with them, to the development of a huge number of methods to detect the communities depending on the definition chosen. For instance, if we decide to identify the community structure with the partition of the graph with the highest modularity, we can then define several algorithms that search for that partition. Moreover, we can also define other algorithms that can try to approximate the best solution. For the rest of the document, we will adopt the following intuitive and abstract definition of community provided by Fortunato et al. [14]:

> *“A community is a set of entities where each entity is closer to the other entities within the community than to the entities outside it.”*

This definition lets us be general enough when talking about communities and how they can be extracted or used. In the case of static networks, there are a plethora of methods in the literature, based on a number of different ideas [15]. Given their importance, researchers started to grow interest in their study also from a temporal point of view, that is, taking into account that the graph may change over time. As for the static case, a consensus on the definition of the concept was not achieved, thus leaving freedom to the development of dynamic community detection algorithms.

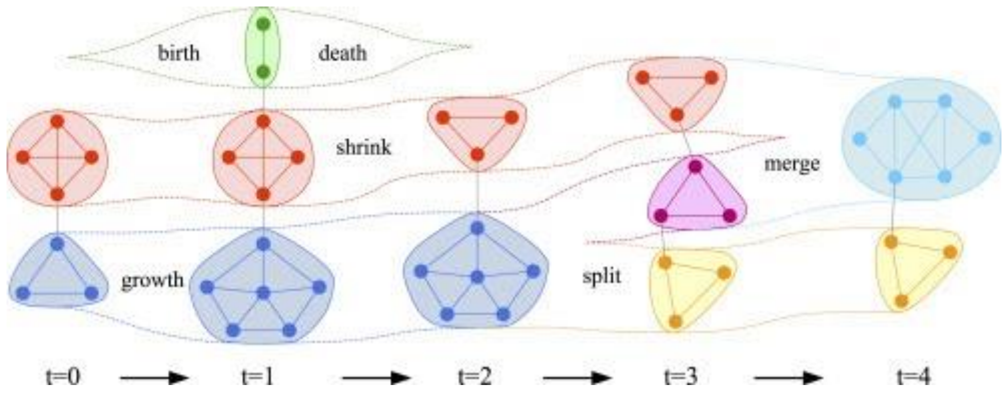


Figure 2. The dynamic community events that define the lifecycle of a community.

Dynamic communities were shown to have a complex lifecycle which is characterized by a set of events, as shown in Figure 2. The events can be summarized as follows:

- *Birth.* A new community is born when the network topology matches the condition for a community to be created. While, in principle, there is no lower limit for the number of nodes required to form a community, in many approaches at least three nodes are required;
- *Growth.* New nodes join an existing community, increasing its size;
- *Merge.* Two communities are merged together into a single one;
- *Split.* A community divides into two or more independent communities;
- *Shrink.* A community decreases in size when some nodes leave it;
- *Death.* If a community completely loses its structure, it ends up in dissolving, thus marking its death.

By studying the community lifecycle, one can gather even more information about the user behaviour in the social network. Communities with a recurrent lifecycle uncover repetitive behaviour which may be connected to the daily/weekly cycle, while unexpected events can help in the detection of abnormal activity gathering around unexpected situations.

2.3 User Embeddings

A common practice when mining data of high dimensionality, such as data organized as graphs [16], is mapping data points to lower dimensional spaces (e.g. that comprise hundreds instead of tens of thousands of parameters) while preserving their underlying relations. This way, low-dimensional representations can replace the respective high-dimensional ones as inputs to machine learning models. In turn, this reduces the time needed to train and run those models by a large margin, as at best that time would be proportional to the number of dimensions (also called features) in training data. The found low-dimensional representations are called the *embeddings* of the respective data points, in the sense that they embed them in the lower dimensional space.

Data embeddings were popularized thanks to natural language processing and synthesis systems [17,18], where they are often superior to similar techniques for reducing dimensionality. Furthermore, they provide a much-celebrated semantic understanding of underlying linguistic models, where conceptual combinations or differences of words are preserved in the latent low-dimensional space they are projected into. A characteristic example of this phenomenon is that the embedding obtained from large English corpora for the word “queen” tends to lie closely to the outcome of the embedding vector operations “king - man + woman”.

Motivated by these results, embeddings have also been adopted to graph mining tasks, for example to represent user and item preferences, which can help recommend matching graph nodes [19,20]. In this case, they have in large part replaced previous approaches that discovered underlying representations of nodes by factoring or decomposing the graph’s adjacency matrix into a product of simpler ones that exhibit helpful properties. These kinds of approaches focus on decomposing the node representations of the adjacency matrix to their principal components and projecting them in low dimensional spaces that retain the most important components. Although this line of thought is conceptually different to graph embeddings -to the extent that they have been previously considered different areas of research- recent findings have shown that embeddings for reconstructing graph edges are effectively approximations to low dimensional projections obtained by spectral decomposition. Compared to principal component analysis, embedding mechanisms run fastly, can be easily generalized to temporal graphs and can be deployed with unknown similarity objectives. Therefore, in this deliverable we recognize that embeddings are better-suited to enabling the machine learning aspects of HELIOS.

In general, extracting embeddings within the scope of graph mining is often thought of as capturing the graph's structure, in the sense that similar embeddings can help reconstruct edges by connecting the respective nodes [21,22]. Given this formulation, graph node embeddings enable prediction tasks, such as recommending missing edges based on their endpoint similarities in the embedding space. A more advanced understanding that moves beyond the scope of providing an unsupervised understanding of the graph arises in the general framework of graph neural networks (see Subsection 2.3). In this case, embeddings are trained alongside an encompassing model, such as a learned non-linear similarity function, so that more emphasis is placed on the structural characteristics best-suited to the prediction task. In the scope of social networks, embedding-based recommendations can be used to suggest promising relations and interactions between users.

In the literature, there are three well-known approaches for finding node representations in time-invariant graphs [22]; these are graph factorization, random walk-based embeddings and deep embeddings. All three approaches can be considered variations of the same scheme, in which node embeddings are extracted so that their similarities can respectively reconstruct the graph, the probability of following particular random walks within the graph, which is equivalent to reconstructing node neighborhoods up to a given number of hops away, or augmenting the latter by using multilayer neural network architectures (see subsection 3.3).

As a general framework to understanding node embeddings, we consider a number of processes $N_k(u)$ that yield sets of nodes satisfying different notions of structural or temporal proximity to nodes u. For example, $N_{1\,hop}$ may comprise the neighbors of each node, which reside one hop away in the graph's structure [23] and $N_{2\,hop}$ may comprise the neighbors of each node's neighbors, which reside two hops away in the graph's structure (i.e. can be reached by a path of length 2). The embedding mechanism then tries to find node representations that preserve the outcome of one or more of these processes. As a result, those processes deeply characterize the nature of the embedding mechanism, as well as what kind of information it captures.

For example, the $N_{1\,hop}$ proximity mentioned before focuses on preserving the immediate neighbors of nodes but potentially forsakes temporal information that could be provided alongside those nodes. In general, it has been shown [24] that making embeddings accurately predict the $N_{k\,hop}$ neighborhoods of nodes with weights that are exponentially degrading with respect to k preserves which subgraphs of the graph have identical internal structure (i.e. there exists a mapping between their nodes so that edges also match), a property also known as isomorphism. Embeddings can also learn to preserve any similarity measure between nodes as long as an adequately strong objective function of node similarity is selected [25].

3 Related Work

The core engagement mechanism of social networks lies on interactions and long-term relations between users, such as sending messages and becoming friends. Due to the growth of social media however, the social behavior enabled by these mechanisms has grown to be increasingly complex and span many interpersonal relations. As a result, social media users may encounter trouble in selecting who to interact or form relations with, as their prefered social actions could be drowned within the sheer number of available options. Furthermore, users of social networks tend to group together based on the mechanisms that influence how they interact or form relations, where formed groups can often be identified with community detection algorithms. The study of the community structure was proved to be useful for a number of applications, such as acquaintance sorting [26], for the influence maximization problem [27,28], recommendation systems [29,30], link prediction [31,32], and detection of users with similar motivation [33] or needs [34]. To address these points, existing social media platforms have introduced a variety of tools that extract patterns of user behavior or their organization into communities.

In HELIOS we aim to model both the structural and the temporal aspect of the heterogeneous social graph that comprises user relations within different contexts, as described in Deliverable 4.2. The goal of this modelling is to enhance user experience by recommending new graph between HELIOS nodes, such as devices and smart objects. In this deliverable we focus on the preferences that arise from static relations between users and enrich it by taking into account dynamic user behavior, as captured in their interactions between over time.

In addition to the above mining tasks, any machine learning performed in HELIOS encounters the limitations of its decentralized architecture. This means that there exists no central service to guide user devices toward a common objective by communicating with all of them. Instead, those devices can only exchange information at the irregular intervals users activate them (e.g. go online) and are at best aware only of the devices they have previously communicated with. In terms of graph theory, this means that devices effectively know their *ego network*, in which the ego device has communicated at least once with one of its alters' devices.

3.1 Diffusing Note Relevance

Graph mining approaches often aim to extract information that is captured in the graph's structure and how this pertains to a set of example nodes. Example nodes may be one or many and often share some metadata attributes that are diffused throughout the rest of graph nodes through their edges. Traditionally this setting is used to predict the relevance of nodes to the attributes shared by the examples. However, it can also be used to recommend additional edges by finding the relevance of all graph nodes to a given one and selecting the most relevant ones to form an edge to. Since rank diffusion can be thought of as a propagation of a relatedness score throughout the graph, this method realizes the concept of forming edges between well-connected nodes, i.e. that are linked through many paths.

Personalized PageRank

The graph mining task of ranking nodes based on their relevance to an attribute of interest is often implemented through strategies that diffuse the relevance of known example nodes to the rest of graph nodes based on their structural proximity [35-38]. These types of methods usually follow a random walk with restart formulation, in which an iterative Markov process starts from a random example node and, at each step, either jumps to the neighbors of the currently visited node or teleports back to a new randomly selected example node. For example, they are predominantly used in the domain of collaborative filtering, in which recommender systems aim to suggest items to users based on their common preferences with other users [39-41].

The original motivation of rank diffusion strategies stems from Google's PageRank algorithm [10] which estimates the probability of a random web surfer to arrive at a page of the web's hyperlink graph. To ensure convergence while capturing the importance of graph nodes, this method considers that the surfer randomly clicking on hyperlinks has a chance to start from a new fully random web page-node at each step. This strategy can be adapted to accommodate personalized preferences of the random surfer by setting their restarting web pages-nodes to belong to a set of known preferences. In terms of attribute mining, the random surfer is a Markov process and personal preferences correspond to nodes that implicitly define a shared attribute of interest. The probability of the random surfing process arriving on nodes can be considered as the relevance of these nodes to this attribute of interest. At the same time, these probabilities are effectively a diffusion of the original preferences, in the sense that they are larger for graph nodes that are repeatedly encountered by more paths starting from the prefered nodes.

The most well-known random walk with restart strategy is personalized PageRank [42-44], in which the probability of jumping from a node v to a neighbor u is proportional to the corresponding element $W[u,v]$ of a normalization of the network's adjacency matrix. If we additionally organize the example nodes into a personalization vector s whose elements $s[v] \geq 0$ represent the known relevances of nodes v to a metadata group, the ranks $r[v]$ of network nodes v are calculated through personalized PageRank converge to the solution of the following linear system:

$$r = aWr + (1-a)s \tag{3.1}$$

where $1-a$ is the teleportation probability to a random example node, also called restart probability on merit that it restarts the random walk. The adjacency matrix normalization that models the previous random walk description probabilistically distributes each node's jump probability to its neighbors, i.e. satisfies $\sum_{v\in V} W[u,v] = 1$. If M is the graph adjacency matrix whose elements $M[u,v] \in \{0,1\}$ correspond to the binary existence or weight of edges (they are zero if the edge (u,v) does not exist) and D is the diagonal matrix of node degrees, i.e. whose diagonal elements are obtained as $D[u,u] = \sum_{v\in V} M[u,v]$ and other elements are zero, the aforementioned normalization can be written as:

$$W = MD^{-1} \Leftrightarrow W[u,v] = \frac{M[u,v]}{\sum_{k\in V} M[u,k]} \tag{3.2}$$

More recent works often adopt a symmetric normalization that satisfies $M[u,v] = M[v,u]$:

$$W_{symm} = D^{-1/2}MD^{-1/2} \Leftrightarrow W[u,v] = \frac{M[u,v]}{\sqrt{\sum_{k\in V} M[u,k]}\sqrt{\sum_{k\in V} M[k,]}} \tag{3.3}$$

This normalization pertains to the normalized Laplacian operator $L = I - W_{symm}$, which is the graph equivalent to discrete derivation [44-46]. Furthermore its symmetric nature makes it suited to modelling bidirectional relations of undirected graphs and lets it satisfy known bounds of its information diffusion rate [46,47].

Graph signal processing

The simplest numerical approach for solving the personalized PageRank formula (2.1) is the power method, which starts from the rank distribution of the personalization vector s and iterates the formula by plugging in previously estimated r until convergence. From a theoretical standpoint, this yields the following computational scheme for personalized PageRank:

$$r = (10a)s + (1-a)aWs + (1-a)a^2W^2s + \cdots = (1-a)(I-aW)^{-1}s \tag{3.4}$$

An important aspect of this solution is that multiplication with the normalized adjacency matrix W can be interpreted as a propagation of node ranks one hop away. For example, the expression Wr

propagates node ranks r to their immediate neighbors. By extension, multiplication of node ranks with W^k can be interpreted as propagation of node ranks k hops away. Given these observations, personalized PageRank can be considered a weighted aggregation of processes that propagate the personalization vector k hops away within the graph with weights $a_k = (1-a)a^k$.

This analysis can been generalized into what is known as the domain of graph signal processing [48-51]. This domain defines the concept of graph filters H as functions of the normalized adjacency matrix that perform a weighted aggregation of rank propagation within the graph. In particular, given weighting scheme akfor capturing the importance of nodes at k hops away from the personalization nodes, graph filters can be expressed as:

$$H(W) = \sum_{k=0}^{\infty} a_k W_k \tag{3.5a}$$

These filters can then be applied on personalization vectors s to diffuse their information throughout the graph's topology:

$$r = H(W)s \tag{3.5b}$$

For example, a popular graph filter often used as an alternative to personalized PageRank is Heat Kernels [52]. This filter places strong emphasis to propagations only few hops away by selecting weights $a_k = e^{-t}t^{kj}/k!$ for some parameter t. The differences between personalized PageRank and this scheme are demonstrated in Figure 3, in which we can see that the former performs a monotonic degradation of hop importances but the latter focuses on a window of hops deemed important. Using the equations (3.5), the weights of the HeatKernel filter yield the following transformation of the "graph signal" s:

$$r = e^{-t(I-W)s} \tag{3.6}$$

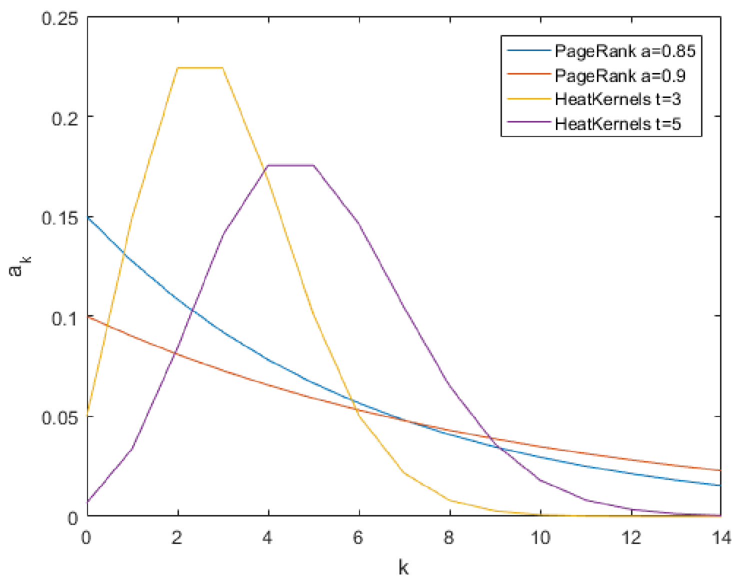


Figure 3. The importance assigned to $\boldsymbol{k}$ hops away from the query vectors for different graph filters.

Graph filters are often defined in the domain of graph signal processing to facilitate a variety of machine learning tasks. These are based on understanding graph filters as transformations of the graph's *spectrum*, i.e. the eigenvalues of its adjacency matrix. In particular, if W is a normalization of the graph's adjacency matrix it can be decomposed to its Jordan normal form $W = U\Lambda U^{-1}$, where U is the orthonormal matrix comprising W 's eigenvectors and Λ a diagonal matrix of the respective (generalized if W is not obtained by symmetric normalization) eigenvalues [53]. Then, graph filters can be rewritten as transformations of the eigenvalues:

$$W^k = U\Lambda^k U^{-1} \Rightarrow H(W) = UH(\Lambda)U^{-1} \quad (3.6)$$

This formula helps express diffusion of a seed vector's ranks as $H(W)s = UH(\Lambda)(U^{-1}s)$ which can be viewed as a transformation of the seed vector to the graph's implicit spectral domain through the operation$U^{-1}s$, application of the filter $H(\Lambda)$ and a transformation of the outcome back to the graph's domain by left multiplication with U. Generalizing the theory of time signal processing that arises when time is considered a linear graph, as shown Figure 4, these steps are equivalent to a Fourier transformation to the graph's spectral domain, a convolution with a graph filter in that domain and an inverse transformation.

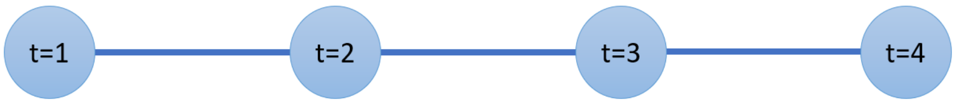


Figure 4. Viewing time as a graph. In this case, graph signal processing becomes time signal processing.

This analysis enables a variety of machine learning tasks. One of these is limiting the number of computations needed to calculate the outcome of graph filters by keeping only the most important eigenvalues, i.e. by selecting $H_\theta(\lambda) = \{\lambda \; if \; \lambda \geq \theta, 0 \; otherwise\}$ a small enough threshold θ that yields a good approximation $W' = UH(\Lambda)U^{-1}$ of the adjacency matrix, given that the error of using this approximation in propagating ranks can be bound by:

$$\|H(W) - W\| \leq \|U\|\theta/\lambda_{max}$$

where λ_{max} the largest eigenvalue of W (it assumes value of 1 if the latter is symmetrically normalized) and the matrix norm is defined as $\|U\| = \max_{s!=0} \frac{\|Us\|}{\|s\|}$. If $H_\theta(\Lambda)$ has a lot of zeros while still achieving a small error, then the rows of U^{-1} and columns of U that correspond to the zeroed eigenvalues do not need to be computed, as their values would be absorbed by those zeroes.

Another useful graph property is the Cheeger inequality [47,54,55] which relates the rate at which information is diffused in a graph by bounding around the difference of the maximal and second maximal eigenvalues, a quantity called the spectral gap of the graph.

3.2 Community Detection

Community structure is one of the main properties of Social Networks, which is widely studied in the literature. Community detection is a process able to facilitate the discovery of important information, by exploiting the structure of the network. Considering the scenario in which the process is applied, we can divide the studies concerning the community detection in centralized and decentralized approaches.

Centralized Community Detection
Several community detection algorithms are proposed in the literature [15]. They are mainly based on different definitions of community. The absence of a general formal definition is the key in the proliferation of different definitions and different algorithms. The most popular algorithms rely on a pure structure-based approach. In the following, we present the most common community detection algorithms.

Label propagation [56]. This algorithm is based on the assumption that a node's community should depend on the communities of its neighbours. This is done through an iterative labeling process. At the start, each node is labeled with a unique label, and iteratively labels of the nodes are updated.

The label of a node is updated according to the most frequent label among its neighbours. The label update stops when some conditions are met, usually when either no labels are changed in a full update iteration, or when a set number of iterations are completed. A very important trait of this algorithm is the almost-linear time complexity.

Infomap [57,58]. This is an algorithm which is based on random walks. The amount of information to describe a random walk is modeled through what the authors call the *map equation*, which is optimized in order to obtain the community structure. The algorithm initially assigns each node to its own community. In the subsequent steps nodes are moved to the community of the neighbour for which the decrease of the map equation is maximum, if any.

Walktrap [59]. This is also a community detection approach based on random walks. This approach is based on the intuition that random walks are likely to get trapped into sets of densely connected nodes. Moreover, the approach returns a hierarchical community structure. Initially each node belongs to a different community. Then the pair of communities which minimizes a measure based on the distances among the nodes of the two communities are merged together, until only one community exists. The partition with highest modularity is returned as the final community structure.

Greedy modularity [60]. This method is also called Clauset-Newman-Moore greedy modularity maximization and, as its name suggests, aims to optimize a measure called modularity [61], which evaluates the modular structure of a partition of the graph. Modularity measures how many intra-community edges are in the partitions compared to the expected ones as if the network is random. Nodes are initially assigned to their own community. Then, through an iterative procedure, the pair of communities for which the modularity score of the partition would increase the most are merged together. The procedure ends when merging any pair of communities will result in a decrease of modularity.

Louvain [62]. The method aims to find the partition with the highest modularity value using an iterative procedure based on the contribution of each node to the modularity score. Each node starts in its own community and then the community structure is updated as follows. In the iterative procedure, nodes are considered one by one and they are moved to a community such that the modularity score of the partition increases. Only communities of the neighbours are considered and the one which would increase the modularity score the most is chosen.

Decentralized Community Detection
In the latest years, there has been increasing attention to the management of communities in decentralized networks. We point out that in the literature there is sometimes confusion regarding the concept of community in social networks and that definitions change in different contexts [13]. Indeed in some studies we find that the word *community* or the saying *virtual community* identifies groups of people in a somewhat restricted social virtual place held together by shared interests and understandings, a sense of obligation and possibly trust. These communities can still be expressed as graphs, but usually the users can explicitly request or decide to be included in the communities. Instead, in the context of peer-to-peer systems, the concept of a community is generally used in reference to all users of the whole system and the topology of the network that describes it, rather than groups of users formed around more specific interests and understandings [63]. However, in a distributed social environment, the concept of community is more general than this. Indeed, communities can be formed by exploiting several pieces of information belonging to both domains, ranging from interactions, to the social overlay topology, common attributes, memberships, etc.

Some studies concerning the general aspect of communities, the community detection or node clustering, in dynamic peer-to-peer networks, have been recently presented. Many of these approaches are in principle adaptations of label propagations [64] in a distributed setting. For instance, in [65] authors propose a framework that allows an analytical study of the distributed community detection problem. A revised label propagation algorithm is proposed to model some features of opportunistic networks. A simpler approach is presented in [66], where the rule to update the labels of the nodes is based on a similarity metric. This approach seeks to achieve better results on modularity and to mitigate the wandering community effect. In [67], authors propose a distributed approach for local dynamic community detection and three implementation variants. In this case, the distributed nature of the algorithm induces a very weak consistency among the nodes of the network. Also, the node clustering problem has been tackled with distributed approaches [68]. A downside of this approach is the fact that it is difficult to deal with node dynamics, i.e. join and leave, because, due to the presence or absence of nodes, the clusters may differ a lot. A common technique to tackle dynamism is to add new nodes to existing clusters and to periodically run from scratch the distributed clustering algorithm. There are only a few interesting approaches for distributed communities detection applied on distributed social networks. In [69] the authors study the problem of community detection as a binary classification problem. Finally, in [70] the authors propose a decentralized protocol in order to manage community detection in a decentralized environment. The performances of the protocol show that it is not feasible to apply it as-it-is to decentralized environments, in particular for the definition of community.

3.3 Graph Neural Networks

GNN Architecture

Neural networks is a machine learning field that has seen explosive growth in recent years. Their growth has, in large part, been fueled by the widespread use of GPU computing, which provides a robust setting for parallelizing symmetric operations. At their core, neural networks architectures are structured in the form of hierarchically stacked layers, each of which performs data transformations, as demonstrated in Figure 5. These architectures can be deep in the sense that they involve multiple layers. Layer parameters, such as the elements of the matrices *Wi* in the example figure, are trained by back-propagating the output loss of how much they deviate from the target value. Mathematically, training schemes form various adaptations of the gradient descent method, in which parameter derivatives move down across the steepest derivative slope. An important variation of neural networks are convolutional neural networks, which deploy convolutional instead of multiplicative matrix layer input transformations to limit training to the few parameters on each convolutional filter.

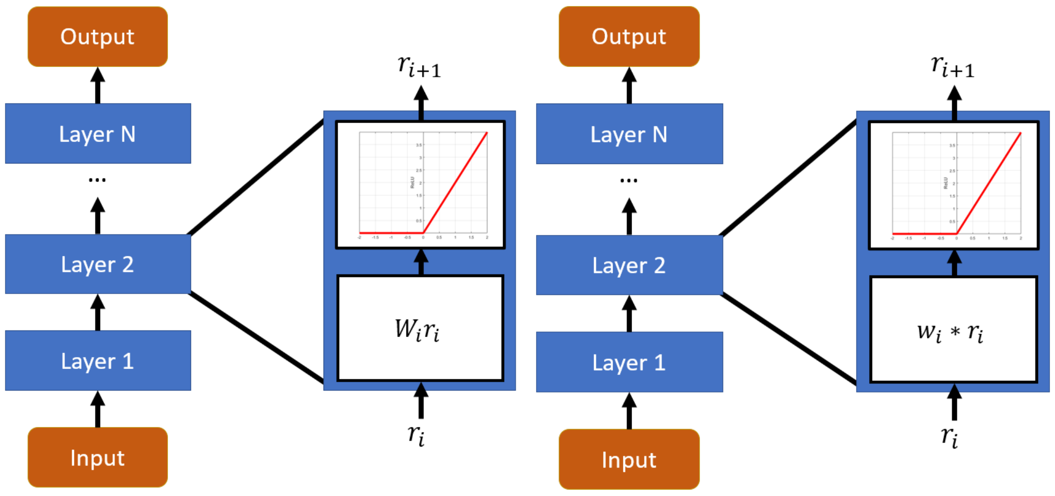


Figure 5. Example neural network architectures for signal processing. On the left one, each layer comprises a multiplication of its input $\boldsymbol{r_i}$ with a dense matrix $\boldsymbol{W_i}$ and a ReLU activation of the outcome to yield the output $\boldsymbol{r_{i+1}}$. On the right one, matrix multiplication is replaced with a convolutional operation with a filter vector $\boldsymbol{w_i}$ of few parameters.

An important theoretical property [71] of neural networks is that, if they comprise adequately many parameters, they can learn any function at any precision level. However, in practice, their efficacy is also compromised by the number of given training examples (millions of examples are often required to capture all feature relations), the available computational resources that could limit the number of training steps, and the selection of training hyper-parameters, such as learning rates.

The popularization of neural networks in a wide variety of domains, such as image processing and text mining, has also motivated their adoption in graph analysis and mining. This has created the emerging field of Graph Neural Networks (GNNs), which adapt the more traditional neural network architectures to work with graph-structured data as inputs.

Originally, GNN architectures were defined as message passing mechanisms, in which nodes send messages to their neighbors over the graph's edges. The neighbors in turn aggregate and transform the received messages before passing them on until the whole process arrives at a stable set of embedding-like representations for all nodes. In particular, each node *u* derives a vector representation *ru* by aggregating the received representations of its neighbors (e.g through a simple sum or average) and transforming the latter with a function H that is shared among all nodes, as demonstrated in Figure 6. Thanks to Banach's fixed point theorem [72], this procedure has a convergence point at which the representations of nodes do not change further. The final representations effectively capture the structural role of nodes within the graph and can be used to facilitate machine learning tasks, for example as inputs to a traditional neural network architecture.

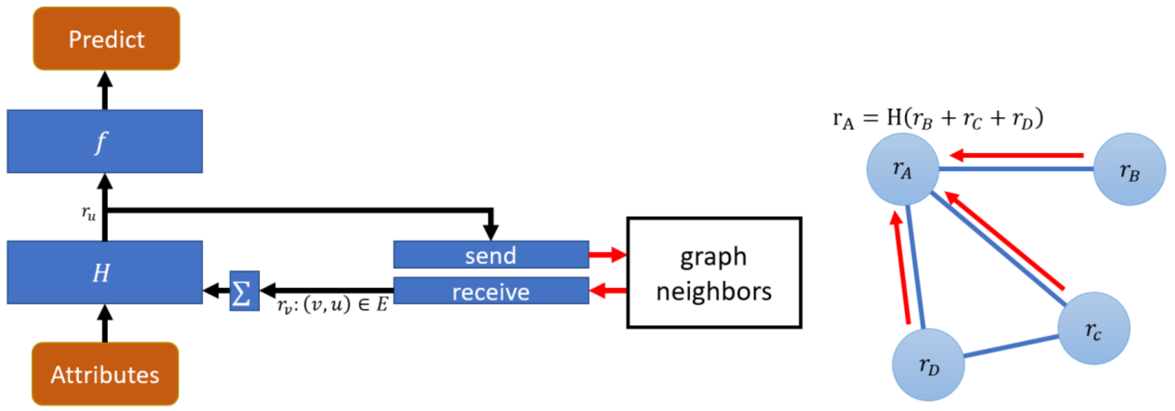


Figure 6. GNN architecture based on message passing through graph neighbors identified by edges $\boldsymbol{E}$ (left) and a visual demonstration of the latter (right). The parameters of the transformation function $\boldsymbol{H}$ are learned so that messages converge to embeddings $\boldsymbol{r_u}$ of nodes u that help facilitate a prediction task $\boldsymbol{f}(\boldsymbol{r_u})$.

An important bottleneck of this scheme is the computational cost of repeated message passing until convergence. This cost is encountered both when accumulating the training objective's partial gradients for message transformation parameters and after applying those gradients to improve parameter estimations, as the respective improved convergence point estimation needs be computed. To tackle this problem, more recent GNN approaches have moved to approximating the outcome of message passing with a feedforward (i.e. non-recursive) neural network equivalent. In particular, advances in deep learning that succeed in approximating a variety of functions have motivated the replacement of the transformation function with a multilayer neural network. The latter typically comprises several layers of aggregating previous layer representations of the previous layers across all neighbors, linearly transforming the aggregation outcome and passing the result through a (non-linear) activation function, as shown in Figure 7, where $r_u^{(i)}$ represents the representation found by node u at layer i. Commonplace activation functions are overviewed in subsection 5.1.

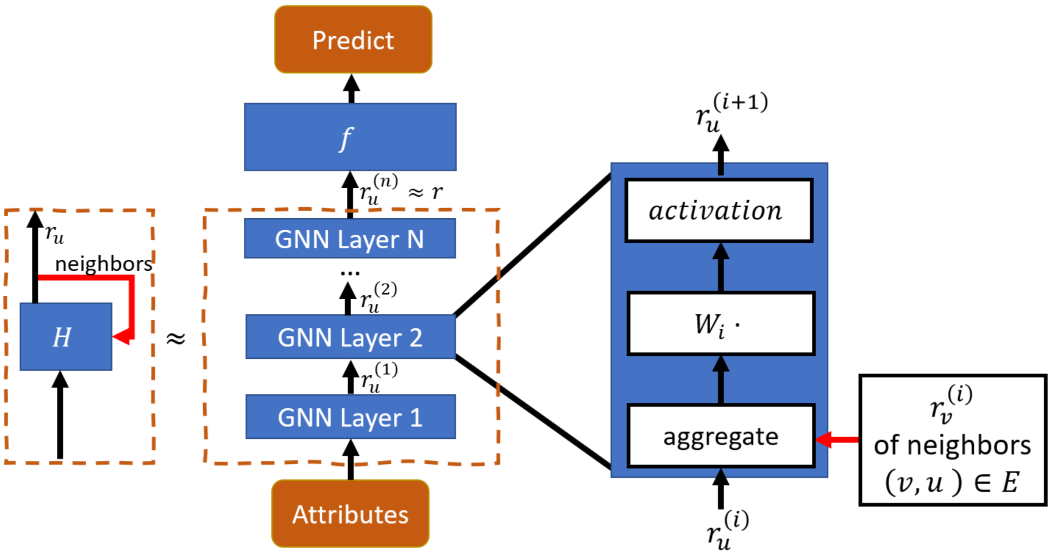


Figure 7. Multilayer GNN architecture relaxing the iterative scheme and the transformation function $\boldsymbol{H}$ with intermediate embedding $\boldsymbol{r_u^{(i)}}$.

If we consider an element-wise activation function, i.e. such that $\sigma(x)[u] = \sigma(x[u])$, and an aggregation mechanism that averages all neighbors $(v, u) \in E$ of nodes $u \in V$, these produce a variation of the scheme demonstrated in Figure 7 that computes layer representations as:

$$r_u^{(i+1)} = \sigma\left(W_{ego,i}\, r_u^{(i)} + \sum_{(v,u)\in E} \frac{W(v,u)}{|(v,u) \in E|}\, W_i r_v^{(i)}\right) \tag{3.7}$$

Where $W_{ego,i}$ and W_i are low-dimensional square matrices (e.g. 32×32 if all intermediate GNN layer representations are of 32 dimensions) that perform the previous layer's and neighbor aggregation transformations respectively. Then, if we gather the representations at layers i and across all nodes into matrices $R^{(i)}$ with columns $R^{(i)}[u] = r_u^{(i)}$, the above equation can be rewritten as:

$$R^{(i+1)} = \sigma\big(W_{ego,i}\, R^{(i)} + W_i W R^{(i)}\big) \tag{3.8}$$

where W is the column-wise normalization of the graph adjacency matrix presented in (2.2) that is responsible for performing the averaging aggregation. Given the aforementioned understanding in graph spectral theory of neighborhood aggregations as graph convolutions, this particular GNN architecture can be considered an extension of convolutional neural networks, in which the convolution operation is replaced with graph convolution.

To see which graph characteristics contribute to the final node representations, let us consider the simplest linear activation $\sigma(x) = x$. For this activation, if the GNN learns to directly propagate the aggregation with weights $W_i = \frac{a_k}{a_{k-1}} I$ and $\mathrm{W}_{\mathrm{ego,i}} = I$ for $i > 0$ and I the unit matrix and $W_0 = 0$, the representations at layer i can be written as:

$$R^{(i+1)} = (I + W_i W) R^{(i)} = \ldots = (a_0 I + a_1 W + a_2 W^2 + \ldots + a_\iota W^\iota) W_{ego,0} R^{(0)}$$

Hence, GNNs can be considered a generalization of graph signal processing filters of parameters a_i.to matrix-organized signals $R^{(0)}$ whose diagonals can comprise the elements of the respective vector signals.

GNNs for Edge Prediction

GNNs were originally developed with the idea of learning to predict node labels in sparsely-labeled graphs by learning an underlying propagation mechanism through the graph. However, recent works have used similar architectures to also learn missing or likely graph edges. This kind of objective aligns well with one of the goals of this deliverable to recommend relations or interactions of interest to users. Edge prediction tasks often aim to leverage only the structural information of the graph, the same as embeddings do. Given the nature of this task, it is often assumed that no metadata features (e.g. user preferences) are available as GNN inputs and nodes are assigned a one-hot encoding of a unique identifier as their features.

To this end, the commonly followed architecture is using GNN layers to derive node embeddings R and then compare the embeddings of node pairs through a (sometimes learned) similarity function *sim* . This function can then be used to predict new edges, as demonstrated in Figure 8.

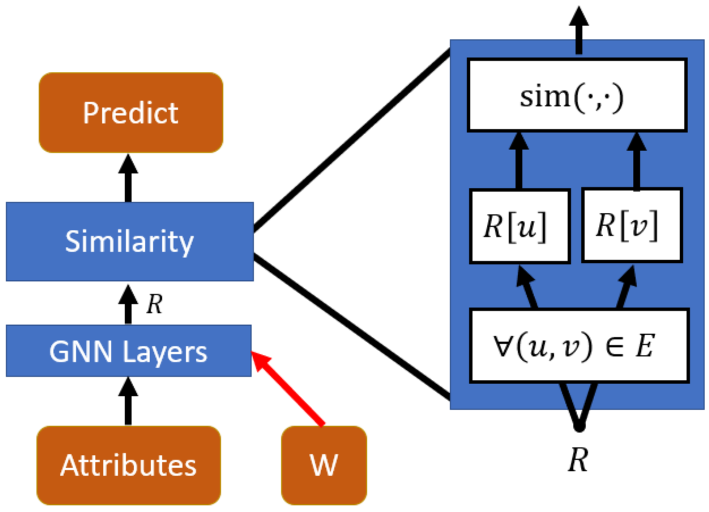


Figure 8. GNN architecture for link prediction given a graph convolution operation W, such as the graph adjacency matrix of equation (3.2). At the node similarity layer, each node pair that could potentially be linked is considered and a similarity between representations is computed.

An overview of some common similarity functions is presented in Table 1. When not otherwise stated, in this work we follow the triple dot similarity followed by DistMult and R-GCN approaches, as in principle others could be discovered by adding additional GNN layers. Usually, the outcome of similarity functions is further transformed to reside in the range $[0,1]$ by passing through a sigmoid activation function.

Table 1. Well-known link scoring functions, where $\langle .,.,. \rangle$ is the triple inner product, $\langle .,. \rangle$ the inner product, and $*\,\boldsymbol{\omega}$ convolution with a pattern $\boldsymbol{\omega}$. In all cases, $\boldsymbol{r}$ is a learned vector.

Approach		**Similarity Function**
TransE	[73]	$\|H[u]-H[v]+r\|$
DistMult	[74]	$\langle H[u], r, H[v] \rangle$
R-GCN	[75]	$\langle H[u], r, H[v] \rangle$
ConveE	[76]	$\langle H[u], ReLU(1D(ReLU([2D(H[v])\,;\,2D(r_i)] * \omega)W)) \rangle$

An important challenge when learning to predict graph edges is setting up a training scheme that scales well with the number of graph nodes, i.e. whose running time scales linearly or near-linearly with the number of graph edges. For example, let us consider the naive scenario in which the embeddings of all graph nodes are trained so that they attempt to predict whether edges exist between all node pairs in the graph, i.e. including those not connected. This would require setting up an objective that involves $|V|^2$ terms to calculate and partially derivative, where $|V|$ is the number of graph nodes. In practice, when graphs span a lot (e.g. millions) of nodes, such a strategy would require an intractable number of operations to calculate the objective function (and its partial derivatives).

A common way to efficiently address this problem is through a procedure called negative sampling, which selects only some of the non-edges between graph nodes to learn as non-existing. Usually, negative sampling selects the non-edge node pairs through fully random processes, for example by taking existing edges and randomly replacing one of their ends with another node of the graph that is not already linked with the remaining edge end. More sophisticated negative sampling approaches involve doing this permutation among structurally close node candidates. For example, it has been proposed that diffusion mechanisms use personalized PageRank to find the most structurally close nodes that can serve as negative samples.

As a final remark, neural networks are often trained towards minimizing a cross-entropy loss function [77] between real training data labels and predicted confidences of assuming those labels. This loss function penalizes severely confident mispredictions of data labels:

$$\begin{aligned} CrossE(real, predicted) = \\ &- \textstyle\sum_i real[i] \log_2 predicted[i] \\ &- \textstyle\sum_i (1 - real[i]) \log_2 (1 - predicted[i]) \end{aligned} \tag{3.9}$$

In the context of GNNs, training labels are either 1 or 0, depending on whether a node pair forms an edge and the predicted confidence is obtained as the outcome of the similarity function. To ensure fast convergence without exploding parameter values into high orders of magnitude, neural networks often introduce a regularization term that penalizes large values by penalizing their square with a small constant λ; that constant makes each parameter's involvement in the loss function negligible as long as it does not assume large values, but severely impacts its training process towards assuming smaller orders of magnitude otherwise. Formally, a loss function using regularization for GNNs can be written as:

$$loss = CrossE\big(\{1 \text{ if } (u,v) \in E, 0 \text{ otherwise}\}, \sigma(r_u \cdot r_v)\big) + \frac{\lambda}{N} \sum_{p \in params} p^2 \tag{3.10}$$

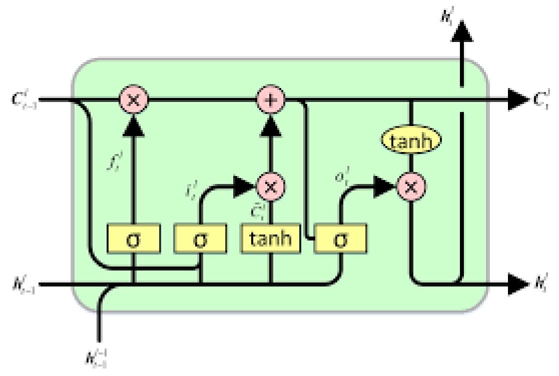
Where (u, v) are terated over all training examples and $params$ are the GNN parameters, such as matrix elements across all N layers of the GNN architecture (each layer may comprise multiple parameters). Then, training the architecture refers to obtaining parameters that (locally) minimize this loss function, for example by training each parameter through a gradient descend scheme:

$$\Delta p = -\gamma \, {}^{\partial loss}/_{\partial p} \tag{2.11}$$

where Δp is the change induced to parameter p and γ is the learning rate. To improve convergence speed compared to constant learning rates, in this deliverable we consider first inducing large changes and only later on fine-tuning learned parameter values by assuming adaptive learning rates of the form $\gamma = 0.9^{epoch}$ where epoch is the number of times we have passed over training data to update parameters. Other methods for parameter optimization, such as the popular Adam optimizer [78], tend to improve only convergence speed compared to this practice and hence lie outside the scope of this deliverable.

3.4 Mining Temporal Graphs

An important aspect of machine learning algorithms is capturing the temporal aspect of given data. In traditional neural network architectures, this problem is solved by processes involving some kind of memory of past interactions. The most widespread of such approaches is Long Short-Term Memory (LSTM), which learns to aggregate previous outputs to the input of each layer [79,80]. A typical LSTM architecture is shown in Figure 9. An important aspect of this architecture is the usage of the *tanh* function, of which the derivative is bounded but tends to avoid values of small magnitude that would require a disproportionately large number of repetitions to accumulate into meaningful parameter updates during training, a problem known in the literature for other activation functions as vanishing gradient.



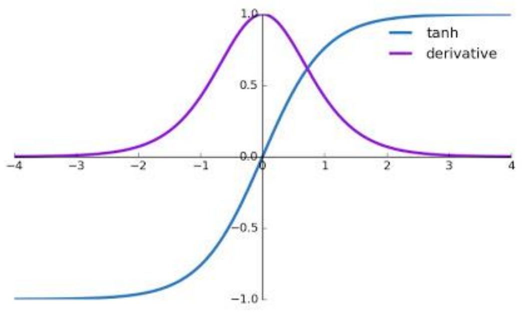


Figure 9. LSTM neural network layer (left) and the *tanh* activation function (right) used to avoid vanishing gradients.

When applied to GNNs, LSTM layers can be considered from the viewpoint of learning embeddings that are aggregated in the near future. However, it can be argued that temporal patterns in graphs are intrinsically tied to the notion of temporal paths that explicitly capture information dissemination over space (i.e. between nodes) and time. There are multiple ways to model the time-evolving aspect of temporal paths, some of which involve a more granular understanding of the graph's evolution than the others. There are two main approaches in this regard:

a) Multilayer temporal graphs. The first approaches on temporal graph mining aimed to capture the state of the graph within a short timeframe and express the evolution of mined characteristics over different points in time. This field matured alongside the (unrelated at the time) field of neural networks to consider a multilayer view, where each layer represents a time slot [81]. Time slots could be either separate points in time at which the graph's structure is captured or evenly-spaced periods of time between which node behavior is aggregated into one graph (e.g. by integrating edge weights or providing an edge if it has been formed during that period). This concept of graph nodes being connected with edges within a designated period of time is also referred to as belonging to both the structural and temporal neighborhood of each other [82].

b) Labeled temporal graphs. A more recent approach to temporal graph mining involves understanding the more intricate dynamics that arise from the causal formation of subsequent edges given previous ones. For example, workplace messages between executives may trigger trickle-down mechanisms so that directives are passed lower in the hierarchy. Labeled graphs aim to avoid losing the intermediate steps involved in this type of information propagation, which may not be captured by multilayer temporal representations. To this end, all edges are considered to connect the same set of nodes and are augmented with a temporal information label, such as their creation time. This representation is particularly useful for modeling interactions that occur over time.

Labeled temporal representations of graphs provide a holistic understanding of the graph. For example, in Figure 10 the multiplayer representation of the example graph at each individual time slot may not yield any meaningful insights towards node C, even if that node could be considered a potential motivation of forming the edge ED at that time.

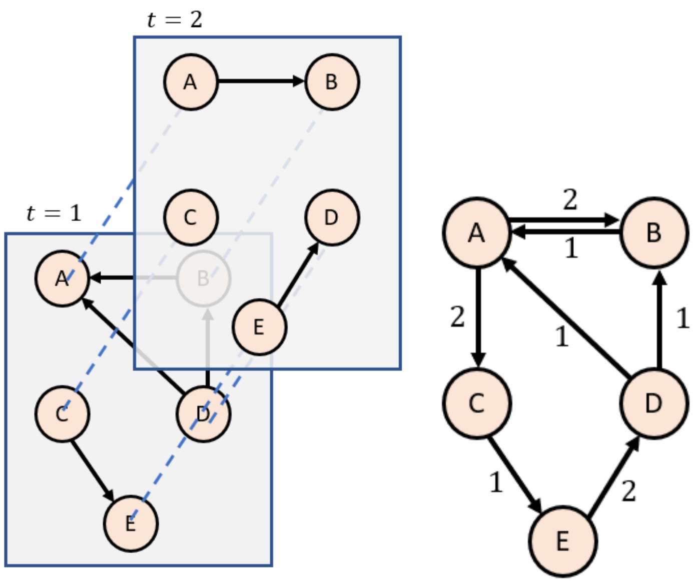


Figure 10. Multiplayer (left) and multigraph (right) view of the same temporal graph.

On the surface, the more intricate understanding of temporal graph evolution appears more difficult to devise algorithms for, which is the reason that multiplayer representations were the first ones to be considered in the field of graph mining. However, the more recent formulation of node embeddings as methods to preserve random walks has enabled mining of these models through embedding extraction processes, such as those detailed in subsection 2.3, which aim to preserve the graph structure, as captured by temporal random walks.

4 Community Detection in Temporal DSNs

In this section, we present a decentralized protocol for community detection and for the management of local communities, targeted to the HELIOS scenario. In particular, we present the principal novelty of the protocol created for peer-to-peer networks based on the definition of a social overlay, such as HELIOS. In the decentralized setting of HELIOS, the social overlay of each node can only contain partial, local information about the whole network. This partial knowledge is modeled by the nodes with the usage of the Contextual Ego Network (CEN), introduced in Deliverable 4.1. Therefore, the solution we propose is local, which means that it exploits only the local information of a node in a DSN and this is in line with the HELIOS scenario, because the nodes' knowledge is limited by the information included in the CEN. To have communities that are local to a specific user, we execute one instance of the protocol for each user, which will work within the user's CEN. Users belonging to the CEN will cooperate to extract and maintain the communities as its topology changes over time. In order to take into account the multilayer nature of the CEN, we foresee to extract the communities at the level of the context, that is, limited by the context boundaries in order to have communities that are meaningful from the contextual relations point of view. In order to partially overcome synchronization and distributed consensus problems, the solution we propose is based on the definition of a set of super-peers which execute a sequential algorithm for detecting the communities and, periodically, synchronize among themselves to possibly merge communities. It uses a Temporal Trade-off approach [83] to manage the evolution of communities, which is well suited to discover and manage communities varying over time in a distributed system. Finally, it includes a load balancing mechanism which tries to even the burden of the management of the communities among all the nodes.

4.1 Decentralized Dynamic Community Detection Challenges

The detection of the dynamic communities in decentralized scenarios, such as the one of HELIOS, is a very complex task and presents a number of challenges and problems to be faced.

First and foremost, one should be aware that in a DSN there are two kinds of dynamism which have different impacts on the system: *social* and *infrastructural*. Social dynamism is linked to the social relationships of the users and their impact on the social overlay. For instance, if users *u* and *v* establish a new friendship relation, they should add each other to their respective CEN, and the update should be propagated to the relevant neighbours such that they can update their CEN as well. On the other hand, the infrastructural dynamism is related to the availability of the nodes in the network overlay. One must be aware that in this particular scenario, not all nodes are available at all times, but instead nodes may be available only for a short period of time per day, which may be linked to the activity of people outside the platform (work, day/night cycle, etc.). Infrastructural dynamism also encompasses the cases in which we have unexpected failures of nodes. Infrastructural dynamism has an enormous impact on the task of community detection, because node churn rates are high in DSN scenarios, and each time a node appears or disappears from the network overlay, existing communities should be updated accordingly and new communities can form.

If we consider the graph on which the task must be performed, we can model its dynamics either by considering edge-based dynamism or node-based dynamism. In the first case the graph will change in terms of single edges, while in the latter case the graph will change in terms of nodes and the edges connected to them. Edge dynamism is usually preferred because it has a finer grain with respect to node dynamism, but can have a non-negligible impact on the algorithm used for community detection. If we contextualise the two dynamisms in the scenario of HELIOS, we clearly see that edge dynamism models very well the social dynamism, because social relationships usually change one at a time. Node dynamism is instead better suited for modelling the infrastructural dynamism, because the network overlay changes in terms of nodes available in the network.

Another very important aspect to handle is the choice of the algorithm and the protocol structure to be used in the solution. As we saw in subsection 3.2, there are a plethora of approaches in the literature, using different features, and based on various intuitions. While the optimization of measures, and their approximations, are quite popular, not all of them can be applied in the scenario of HELIOS because they usually require a high amount of computational and storage resources even for centralized approaches. But the HELIOS setting imposes even harder limitations. Devices of the users will have limited resources, therefore simple and possibly local definitions of community are to be preferred. The communication may not be always flawless and stable, so it becomes imperative to develop a solution that is self-adapting and self-healing. Finally the disparity of the nodes in the network should be taken into account in order to balance the burden of the solution according to each node's capabilities.

4.2 Choosing a Decentralized Community Detection Approach

While community detection has been widely studied in static complex networks, the interest is quickly growing also for dynamic networks. This is because dynamic networks better model the dynamic nature of current complex networks such as social networks, economic networks and many more. Indeed, the time-evolving nature of social network, especially when considering spontaneous networks arising from peer-to-peer/opportunistic contacts makes it even harder to formally define what a community is. A first very abstract definition of dynamic communities has been proposed in [83], where the classical definition of a community as a set of closely correlated nodes is refined by taking into account that the graph topology can change over time.

Dynamic community detection algorithms can be broadly categorized as [83]:

a) Instant-optimal community detection. The communities are discovered by considering only the current state of the network. The network's evolution is seen as a series of successive snapshots, each representing the state of the network at a particular instant.

b) Temporal trade-off community detection. The communities identified at a given time depend on the state of the network at previous times, up to the initial known state.

c) Cross-time community detection. These methods use past, current and future information with respect to the current time at which communities are identified.

In Table 2 we summarize these three types of community detection algorithms. We point out that the information used by approaches deeply affects their usability in different settings.

Table 2. Information used by the three decentralized community detection approaches

Approach	**State of the network in the past**	**Current state of the network**	**State of the network in the future**
Instant optimal	NO	**YES**	NO
Temporal trade-off	**YES**	**YES**	NO
Cross-time	**YES**	**YES**	**YES**

In particular, cross-time approaches detect communities at a given time instance using all the information pertaining to the graph, namely its current state alongside past and future events (i.e. that have occured and will occur afterwards). The greatest limitation of such approaches is that they require access to future information to detect communities. Since in our scenario communities have to be computed in real time future information is not available, which prevents us from using such an approach.

On the other hand, instant optimal approaches detect communities by considering only the state of the graph at the current time instant but not any past or future information. These approaches use the least amount of information and hence lead to defining simplified community detection processes. A significant shortcoming of these approaches is that communities are detected from scratch each time, since no past information is available. Therefore, a snapshot of the social graph must be collected in order to start the detection of the communities; whereas this may not be a significant issue in centralized graph mining, it is infeasible in decentralized, peer-to-peer architectures with heterogeneous low-end devices, where each node is constrained to a limited view of the graph. The scalability of the system may be highly affected by such an approach: in fact, it is not feasible to make one node collect the snapshot of the network and then detect communities in a large scale network. A mechanism which efficiently distributes the computation, will clearly introduce heavy synchronization phases. For these reasons, the Instant Optimal class of algorithms is not suited for our scenario.

Finally, temporal trade-off approaches exploit present and past information to detect communities at a given time instant. The term "past information" does not necessarily refer only to the state of the network at past instants of time but may also include previously detected communities. So, starting from pre-existing communities, a potentially scalable approach could keep these communities updated by observing only which nodes join or leave the network. Beyond the fact that such an approach is more efficient because the detection of communities is not made from scratch each time they are needed, it is also potentially more scalable. Indeed, to carry out the task it is not necessary to have at one’s disposal a complete snapshot of the network and communities can be reevaluated locally.

To summarize, we decided to choose an approach belonging to the temporal trade-off class, because of the better chances to obtain an efficient and scalable solution. The two key features of our approach emerging from this choice are: the ability to update communities locally and independently and the possibility to update communities on the fly, without detecting communities from scratch each time.

4.3 Overview of the solution

The solution we propose follows the temporal trade-off approach and adopts a node-based dynamism as opposed to the edge-based dynamism. The node dynamism can better model the infrastructural dynamism that is a critical aspect of decentralized online social networks, because the graph changes in terms of nodes switching from offline to online and vice-versa. The approach is decentralized, meaning that the computation and management of the communities is not condensed in a single node, or offloaded in an external resource such as a cloud infrastructure.

The nodes themselves must cooperate and communicate with a protocol in order to detect the communities. However, since obtaining synchronization in a completely distributed environment is a complex task [84], we decide to adopt a weak consistency model where nodes do not need to be aware of all the communities and the nodes inside them. Instead nodes will only aim for being part of a community. The solution follows a super-peer [85] approach because we appoint a node in each community to be responsible for keeping the community updated though time as the network changes. In the rest of the document, we will refer to these nodes as the *moderators* of the communities. In order to keep the community updated, the moderator should not only decide which nodes are part of the community and which are not, but it should also make sure that a new moderator is chosen when it leaves the community. Therefore, the moderators of communities are not fixed nodes through time, but instead each node can potentially become a moderator.

This flexibility of the approach makes the moderators not so easy to be detected, hence we also introduce a lookup service (e.g. a distributed hash table [86,87] stored across the decentralized graph or an equivalent mechanism) that nodes use to discover moderators. A node *m*, that is currently a moderator for a community of a node *e*, will register itself in the lookup service, using the identifier of node *e* as key. In this way, other nodes can easily retrieve the moderators of the active communities inside the CEN of their neighbours.

A single moderator per community is usually enough, but there is a chance that the community structure may end up in an inconsistent state. Indeed if a node unexpectedly fails and that node is also moderator for a community, the community may be still alive, but not working properly, because the node appointed as manager of the community is not available. To overcome this possible situation, we introduce the roles of *primary moderator* and *secondary moderator* of a community. The primary moderator of a community is the node that is operatively in charge of managing the community. The secondary moderator acts as a normal node, with the only addition that it keeps itself in contact with the primary moderator, so that, in case it detects the primary moderator failure, it can substitute the primary moderator and guarantee the correct behaviour of a community, restoring the optimal situation with two moderators. We only use two moderators per community because it was shown that using a single backup node yields satisfactory results [88].

One important novelty in this approach is that nodes will constantly try to be included in at least one community. In all the approaches we found in the literature, nodes will try to join communities only when specific conditions are met, which usually means when the node just switched its state from offline to online. But, at the end of a join procedure, a node does not necessarily end up in joining at least one community. This can happen because, for instance, not enough nodes are online at the same time, or their connections are not such that the node can be accepted in the communities. We also must consider that the network dynamism is present, and communities can dissolve if the key nodes leave the network, possibly leaving other nodes outside any communities. Since being part of at least one community is crucial, to overcome the issue of being left out of every community, nodes will check their status periodically and will try to always be included in at least one community at all times. In order to do so, a node will firstly try to join existing communities and, if none of them is able to accept it, the node will also try to form a community of its own.

The community detection can be an extremely time consuming task, and in some cases it is the result of very complex procedures. However, in the scenario of HELIOS, we must take into account that node’s devices do not necessarily boast high communication bandwidth (e.g. may be limited by the speed of their internet connection), memory or computing power, therefore the aim is to try to find a community structure that is meaningful, without the need of extremely complicated procedures. For this solution the leading structure for the definition of a community is the triangle (a set of three nodes fully interconnected), as this structure is the basis for many other community detection algorithms [89-93]. While this seems at first a very complex task, because it involves triangle counting algorithms, it is instead a much easier version of the problem because one node (the joining node) is fixed, and because once a triangle is found the procedure can stop. However, in the worst case scenario the check can take a non-negligible amount of time. For this reason, we also introduce another condition that, if met, it allows a node to be accepted in a community. This simpler condition is based on a relaxation of the concept of *triangle*, i.e. *triad* (three nodes connected by two edges), and consists of checking if a node has at least a number of neighbours already in the community, it can join the community.

To present the solution we will firstly describe what is the algorithm used by the moderators to manage the communities in the event a node requests to join or to leave the community. Then we present the protocol that the nodes follow.

4.3 Computing communities

In this subsection we introduce the algorithm used by the moderators to manage the communities. It is used by each moderator when it receives messages from the nodes joining/leaving the social overlay of a node to update the communities. The protocol executed by the peers will be described in subsection 4.5.

Join algorithm

We start by presenting the algorithm used to evaluate whether a node should be accepted into a community by its moderator when the node sends a join request. In the solution we use two building blocks for the definition of a community: *triangles* and *triads*. Let us consider a community and a requesting node. That node can be added to the community if and only if at least one of the following conditions are satisfied:

- *Triangle condition.* This condition is satisfied if the requesting node forms at least one triangle with two different nodes already belonging to the community. This is the case for node C in Figure 11.
- *Triad condition (or invitation condition).* This condition is satisfied when the number of neighbours of the requesting inside the community is larger than a fixed threshold, i.e. when the joining node forms a number of triads equal to the previous threshold with the additional constraint that the joining node must be the central node of the triad. This is the case for node B presented in Figure 11.

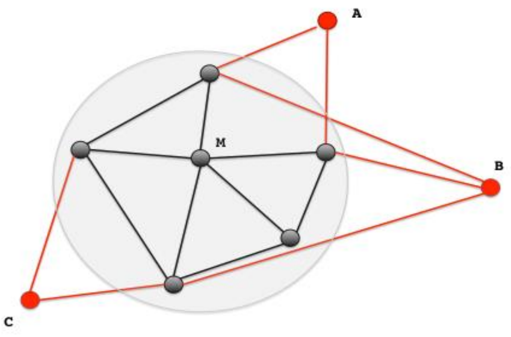
The triangle condition aims at keeping a high level of clustering among the nodes inside the community. On the other hand, the invitation condition aims at softening the triangle condition, making nodes that are well connected to the community to join it, even if they do not close a triangle. What we expect is that a well connected node *n* will close increasingly more triangles over time, so in time it will increase the clustering coefficient of the community, even if the insertion of *n* has lowered it temporarily. This fact should also bring to a situation where the number of communities is sensibly lower because a node is more easily accepted into existing communities, rather than creating new ones. Last, the invitation condition is also computationally less expensive with respect to the triangle condition. Indeed, it only requires to count the number of neighbours of the requesting that are inside the community. Furthermore, with a low threshold, it will often happen that whenever the triangle condition is satisfied, also the invitation condition is satisfied.

Leave algorithm

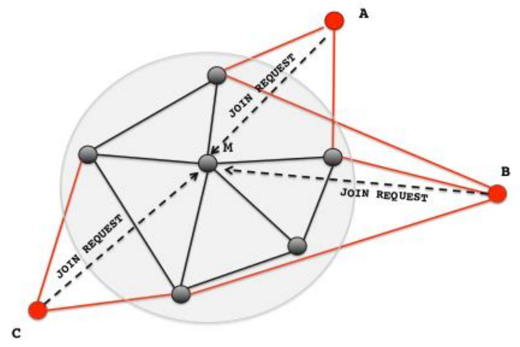
Let us now discuss what happens when a node *n* requests to be removed from a community because it is about to go offline. The very first step for a moderator is to handle the leaving of node *n*, which is simply done by removing the node from the community. After the node is removed the following step is a *check for membership* for a subset of nodes inside the community which are involved in the leaving of *n*. In this solution the check for membership is localized to guarantee a reduced computational cost. Rather than checking all nodes in the community, the check is performed only for the neighbours of *n*, sensibly reducing the number of controls to be made. The nodes to be checked are labeled as *unconfirmed* (see Figure 12). Then, any unconfirmed node is considered, one by one, and the moderator checks whether the node still belongs to the community. The check process consists in verifying if either joining condition holds, i.e. the triangle and the triad condition. If at least one of the conditions holds, the node is labeled as *confirmed* and excluded from further checks. This verification process is repeated until no more nodes can be labeled as confirmed. At this point, all unconfirmed nodes (nodes that were not confirmed) are removed from the original community, while all confirmed nodes will remain inside the community (see Figure 12). We clarify that in the case the set of unconfirmed nodes is not empty, we will not check for membership also the neighbours of these unconfirmed nodes.

After the check for membership process has finished, the original community can be split into several shards, where each shard may contain one or more nodes. To detect these shards, the connected components of the graph resulting from the removal of the leaving nodes are computed.

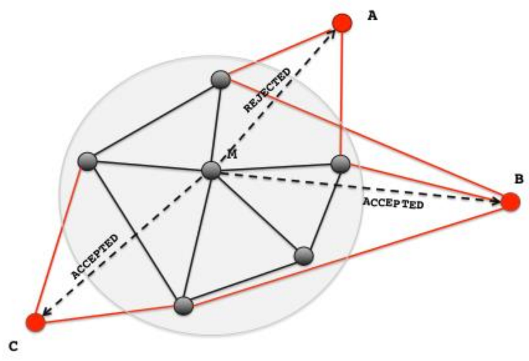
Each of the resulting components will be considered as a new standalone community. As we will see in the next section, if required, the moderator will notify the peers involved in the community update so that they can self organize in new communities.



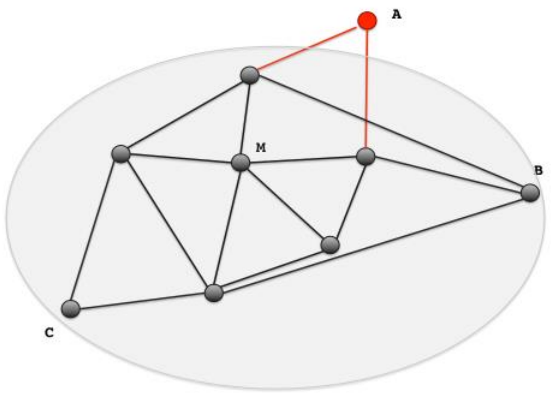
(a) Nodes $A, B$and C attempt to join the community moderated by m send a request to the moderator.



(b) Joining nodes



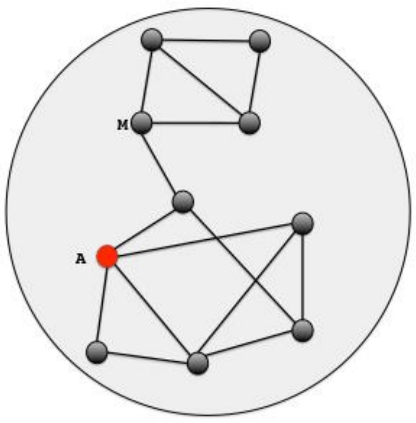
(c) Nodes B, and C get accepted because they satisfy one of the two conditions. Node A does not get accepted.



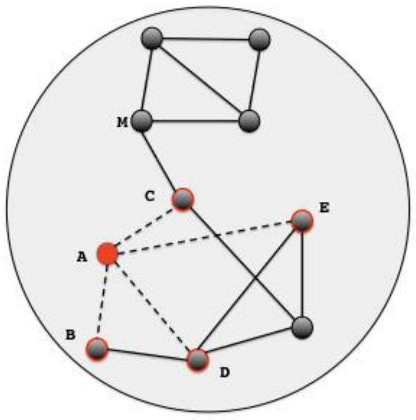
(d) The community structure after the protocol completes its execution.

.

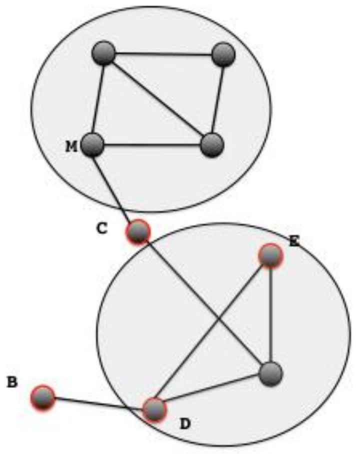
Figure 11. Steps of the node join protocol.



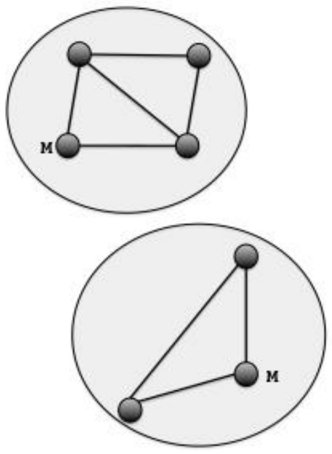
(a) The moderator manages the leave of node A.



(b) The neighbors of node A are marked as “uncomfirmed”.



(c) Nodes E and D are comfirmed and will remain in the community. Nodes C and D remain uncomfirmed and are hence left out.



(d) As a result, the original communty is split in two shards that work as two separate communities.

Figure 12. Steps of the node leave protocol.

4.4 Protocol for community management

In this section we present the steps of the protocol followed by the nodes and moderators. Moderators are passive and wait for other nodes to contact them in the case the node wants to join or leave the community managed by the moderator. A node executes the joining protocol when it finds itself not being part of any community, which can happen in two cases: the node just joined the network, or the node is removed from all the community (as result of the check for membership process explained above). Instead, a node executes the leaving protocol only in the case it is about to voluntarily leave the network. By the end of the section we also analyse the case in which a node that is currently moderator, primary or secondary, of a community leaves the network.

Node join

In the case of a node *n* joining an ego network of a node *e* as an alter, the aim of *n* is to get accepted into an existing community within the ego network of *e*. In order to do so, node *n* obtains the list of active moderators of the ego network it is joining from the moderators lookup service, and upon receiving this list, it sends a join request to each moderator. Thanks to the fact that the moderator knows the ego network of the node whose community it is moderating, it can check if the requesting node can join the community autonomously, without further information. Moreover, the joining node pings all its alters to check which of them is online, in case a new community needs to be defined from scratch. Finally, a timeout is set to restart the joining procedure in case it fails. When the list of moderators for the ego network of *e* is retrieved in the aforementioned lookup service, the joining node n contacts them to notify it is now online and to ask to be part of one of the communities managed by the moderators.

Upon receiving a notification from a joining node, a moderator evaluates the former's connections with other nodes inside each managed community, so as to assert whether the joining node can be inserted in that community based on the previously outlined join algorithm's conditions. Then, a message is sent back to the joining node to notify it on whether it can be inserted in a community or it cannot join any community because the joining conditions are not satisfied. In the first case, the moderator also accordingly updates its managed communities. In Figure 11, we show the join procedure executed by nodes A, B and C, with the messages exchanged between each of them and the moderator of a community.

It may happen that a node cannot join any community of any moderator, for instance when no communities have been previously defined and no moderator has been elected. When this happens, the joining node can detect if a new community can be built around itself, collecting information about its neighbours through the ping process detailed in the following paragraph.

Community birth

If a node cannot join any community, either because no community has already been created or due to not satisfying the criteria to be accepted in any comunity, it will try to create a community of its own, which leads to a community birth event. When a node joins the network, it also pings its neighbours on the social overlay. Thanks to the information received from the online neighbours, a it is able to detect whether it resides within a community structure. Upon receiving replies to ping messages from its neighbours, the node checks if there is at least a triangle of online nodes including itself. After having detected all the triangles with the online neighbours, the node *n* creates a new community with the nodes that appear in at least one triangle.

When the community has formed, the node birthing it becomes the primary moderator. Right after, the newly self-proclaimed moderator elects a secondary moderator by choosing a node among the nodes belonging to the newly-formed community, using one of the strategies described in subsection 4.5. To complete the community birth protocol, the primary moderator also registers itself as a moderator of a community in the lookup service, along with a reference to the secondary moderator, so that other nodes that will join the network in the future will be able to interact with the community. As the last step, to complete the community birth, all the nodes inside the community are notified of its birth by the primary moderator.

It must be noted that the join protocol and the community birth protocol can fail if not enough nodes are online (less than 3), or if they are connected in a way such that no community structure is present at this time. For this reason, if node *n*, after the completion of both protocols, is still not in any community, the node sets a timeout after which it will trigger a re-execution of the join protocol.

Node leave

A node voluntarily leaving the network should inform the primary moderator of the communities it is part of that it will be unavailable soon. Since each node stores a reference to the moderators of the communities it belongs to, the overhead to leave the network is minimal, requiring a single message. When the moderator *m* receives the message from a leaving node *n*, it has to update the community as described in subsection 4.3. As a result of the node leave algorithm, some nodes may be labeled as *unconfirmed*: these nodes should be excluded from the community as they are not well connected with the rest of the nodes which are instead rightfully in the community. To those unconfirmed nodes, the primary moderator *m* will send a message to inform them that they are no longer part of the community so that nodes can stay up-to-date with the communities it is part of.

In particular cases this may lead to a situation in which one (or more) nodes are still online and available but they are not inside any community. A node in this situation will be unable to have access to the services connected to the community membership, therefore it is its first priority to find a new community to join. To re-establish a regular situation, the node tries to become part of a community as if it just joined the network. The fact that a node actively tries to always be part of a community is a peculiar property of this solution. Indeed, many decentralized community detection algorithms in the literature do not have this feature, and nodes try to join communities only when they switch their status from offline to online. Instead, our solution enables nodes to actively search for new communities when they do not have one.

After this process, the community can simply shrink in size if the departure of the leaving node has not produced a split of the community. A more impactful scenario is when, after a node leaves the network, the community is divided into shards. In this case, the primary moderator *m* of the original community must make sure that each shard is able to progress in its lifecycle. If *m* still belongs to one of the shards resulting from the split, this shard inherits the original community and the corresponding primary moderator.

All the nodes that do not belong to this shard will receive a message from the primary moderator *m* informing them that they are no longer part of that community. On the other hand, each shard of the original community is now a community of its own and will have both a primary and a secondary moderator. To this end, for each shard *s* of the original community, the moderator of the original community *m* chooses a random node *m'* among all the nodes inside *s* to be the new primary moderator of *s* (Figure 12) so that each shard can start behaving as a new community. Once a primary moderator *m'* is selected for *s*, *m* entrusts *m'* the community *s*. The primary moderators of the new communities, independently of each other, are in charge of notifying all the nodes in their respective shard about their identity, and register them as moderators in the lookup service, so that other nodes can find and join it. To complete the protocol, the new primary moderators elect a new secondary moderator for the community, such that the optimal situation is restored.

Moderator leave

In the previous section we described the protocol followed in the case generic nodes leave the network. However, there is a chance that the node leaving the network is moderator for some communities.

We start with the case in which the leaving node is the primary moderator for a community. This event can be treated as a normal node leave, as long as a new primary moderator can be elected to replace the leaving node-moderator. We perform the election of the new primary moderator in leaving primary moderator, since it has the most recently updated view of the community. This election can be performed quickly, which helps ensure that the previous primary moderator leaves as soon as possible and that no community is left without a primary moderator. The more time-consuming task of updating the community is delegated to the new primary moderator. The strategies we considered for the election of a (new) primary moderator are presented in subsection 4.5. Once the community is entrusted to the latter, we revert to the case in which a general node requests to leave the network, with the addition that the new primary moderator should register itself as such in the lookup service and the leaving moderator should also remove itself.

The case of a secondary moderator leaving the network is simpler. The primary moderator elects a new secondary moderator, updates the lookup service, and manages the leave of the secondary moderator as a normal node. The strategies we considered for the election of a (new) secondary moderator are presented in subsection 4.5 as well.

Finally, a moderator can also fail unexpectedly. To cope with this possibility we recommend that each pair of primary and secondary moderators of the same community run a ping-pong protocol. This way, they can preemptively detect each other's failure and perform the required actions to reestablish the optimal situation with two moderators.

4.5 Moderator Election Strategies

In this subsection we discuss two possible moderator election strategies and their implications on the protocol described in subsection 4.4.

The first strategy we discuss is the random election strategy. With this strategy, the primary moderator election corresponds to choosing a random node inside the community. In case a secondary moderator is needed, the moderator is chosen at random among the nodes inside the community, minus the primary moderator, because a node cannot be at the same time primary and secondary moderator of the same community. This strategy comes in handy in the case where it is crucial to save all possible resources. Indeed it requires no additional storage, since the community structure (i.e. a reference to the nodes inside the community) is already stored in the moderator for the proper functioning of the protocol, no additional communication, and a negligible amount of computation, depending on how the node is chosen at random.

However, having a different way to choose moderators can bring several benefits to the protocol, such as distributing in a smarter way the burden of being a moderator among the nodes. In particular, as a second strategy, we try to define one that is designed to use low communication, computational and storage resources of nodes to avoid introducing overheads. The approach we adopted involves the finding of the node, among the valid candidates, which minimizes a measure of communication load. The features we take into account for this task are the number of messages received by a node and the bandwidth of the corresponding device. We define the load L_{u^u} of a node u as:

$$L_u = \frac{M_u + 1}{B_u} \tag{4.1}$$

where M_u is the number of messages received the node during the last 5 minutes, and B_u is the bandwidth of node u. We add the constant term 1 to the number of messages received, so that nodes of high bandwidth are preferred, even when no messages are received. At the same time, the above formula selects the node between those of similar bandwidths that have received fewer messages and are hence expected to be moderators of fewer communities When a new primary moderator is needed, the election process establishes that, among all the nodes in the community, the one with the lowest load is chosen. If a secondary moderator is needed, the node with the lowest load is chosen as long as it is not the primary moderator of the community. Thanks to this mechanism, we aim to choose as moderators the nodes that can handle a high amount of traffic.

To implement this moderator election strategy, each node needs to keep track of the messages it receives so that it can easily evaluate its loading factor using (4.1). In addition it has to send a periodic update message to the moderators of the communities it is part of, such that the moderators have a clear view of the nodes' load. From the side of the moderator, it needs to keep track of the load factors of the nodes in the community using a data structure. Since the nodes send periodic updates about their load factor, but we do not expect frequent moderator elections, moderators use a hash addressed data structure for storing the load factors of nodes. This way we expect to pay constant time for the update of the load factor of each node, and the cost of sorting the load factors when an election is needed.

5 Decentralized Social Graph Recommendations

Currently, most large social network platforms, such as Facebook and Twitter, are accessed through internet endpoints that also expose centrally produced recommendations to users. This organization has raised concerns over the ownership and dissemination rights of user information, especially since that information is not under the direct control of the ones that provide it but needs to pass through a central gateway (i.e. the social network service) to reach others. Such concerns have motivated the development of DSN platforms, such as the ones outlined in Deliverable 4.1, in which users directly share information with desired recipients.

An important challenge that arises in DSNs, such as the ones developed in HELIOS, is adopting graph mining strategies to work without a central overseer. In the case of mining user preferences to provide future recommendations, a key problem is that existing algorithms often learn a common set of parameters that pertain to underlying mechanisms shared between all graph nodes. For example, they learn how to extract and match user latent preferences. Then, it is important to identify mining algorithms and communication protocols that enable a decentralized learning process, which runs on user devices without relying on any central entity. In practice, such algorithms would be constrained to viewing only small local subgraphs of social networks and this introduces the challenge of learning a high quality understanding of user behavior only through this kind of partially accessible information.

Before researching decentralized graph mining algorithms though, it is important to first identify the best centralized ones, as these could provide valuable insights into which practices work and which do not. To this end, in this section we first investigate popular mining algorithms for extracting latent users preferences in the form of embeddings that can explain their actions; mined preferences can be used in recommendation tasks, such as recommending new relations and interactions. For example, user preferences could help recommend new friends or old acquaintances.

A set of preliminary experiments in six real-world social network datasets reveal that shallow GNN architectures are the ones that best capture latent user preferences that drive the formation of relations. We also provide a literature-corroborated theorization of why this type of architecture best mines active user behavior, despite being conceptually simpler than some of its competitors. Hence, our subsequent analysis focuses on this type of architecture.

In addition to recommending relations, we also investigate the efficacy of shallow GNNs in understanding the preferences that drive user interactions by experimenting in the four interaction datasets. At first, we met limited success in predicting which of the older interactions users would revisit. However, we argue that the frequency at which interactions occur makes them heavily dependent on short-term evolution of user preferences (e.g. switch of focus) and that the selected GNN architecture should account for their temporal evolution, for example by placing more weight in the preferences revealed by more recent interactions. Indeed, if we also introduce this kind of dynamism, the top three recommendations of the selected GNN architecture are better than heuristic alternatives in recommending old interactions.

We then introduce the differences between centralized and decentralized architectures, while stressing the need for new practices that are lightweight enough from a communication overload perspective to be allowed in DSNs.

For the first formulation of GNNs that involves message passing between graph nodes, although this setting appears to mimic a decentralized structure, in which each node has its own view, the transformation process needs to be learned simultaneously across all nodes.

5.1 Preliminary Investigation

Before engaging in a more thorough discussion on mining decentralized temporal social networks, we recognize that different graph mining algorithms are better suited to different mining tasks, depending on the complexity of the employed machine learning architecture (e.g. in terms of GNNs the number of layers and type of the activation functions σ) and how well its intricacies capture the underlying domain modeled by the graph. For example, if node representations within a domain are driven by linear combinations of underlying real-world attributes, as demonstrated in Figure 13, using non-linear activation functions may introduce needless systemic complexity that could potentially overfit the learned model and hence fail to generalize to new examples.

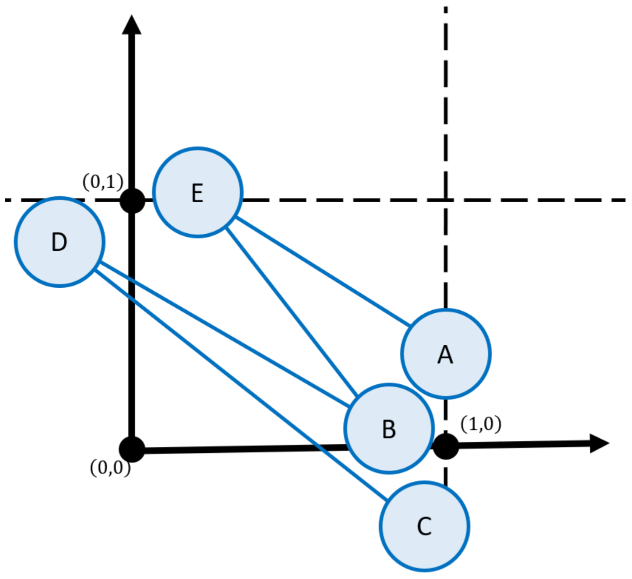


Figure 13. Example of linear edge prediction in a graph. Nodes $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ could be represented close to $(\mathbf{1}, \mathbf{0})$ and the nodes $\boldsymbol{D}, \boldsymbol{E}$ close to $(\mathbf{0}, \mathbf{1})$ so that edges could be predicted by 1-the dot product (which is a linear function) of their endpoints.

Given these concerns, we first investigate the most promising architectures for predicting user relations and interactions in social networks of similar characteristics to HELIOS. This way, we can select only the best-performing algorithms to make them account for the temporal evolution of user preferences and the decentralized structure of HELIOS.

An important aspect of this analysis that will be useful later on is assessing whether the small size of social networks impacts the efficacy of mining algorithms. As an edge case, we investigate whether utilizing only the immediate neighborhoods of nodes suffices for graph mining. But we are also interested in graphs of few nodes that arise during the initiation phase of social media platforms, such as HELIOS, since neural networks are in general notorious for needing huge volumes of data to exceed the accuracy of competing methods.

To this end, we conduct experiments on six social graphs; a Facebook static graph that comprises friendship relations, a Facebook temporal graph of wall-posting interactions between users, a Twitter static graph of user follows, a Twitter temporal graph of retweets, an SMS network of message sending interactions and an Email exchange within a large organization. The domains of these graphs span different types of relations and interactions, similar to those that could arise within HELIOS applications. Their respective characteristics are summarized in Table 3.

Table 3. Social network graphs involved in experiments.

Dataset		Source	Type	Nodes	Relations	Interactions
Friends	[94]	Facebook	Static	4,039	88,234	---
Wall	[95]	Facebook	Dynamic	63,731	817,090	1,269,502
Messages	[96]	Facebook	Dynamic	1,899	15,737	61,734
Follows	[94]	Twitter	Static	4,799	127,847	---
SMS	[97]	Copenhagen network	Dynamic	24,582	24,279	24,333
Email	[98]	Enron emails	Dynamic	92	755	1,148,072

Personalized PageRank Mining

The first type of mining we investigate is the well-established Personalized PageRank (PPR). As we mention in subsection 3.1, this graph diffusion scheme is affected by the rate *a* at which the importance of longer random walk degrades, where 1-a is the restart probability of the equivalent random walk with restart scheme. In particular, the lower the restart probability, the farther away seed node ranks are propagated. On the other hand, larger restart probabilities concentrate on recommending relations with nodes only a few hops away in the graph's structure [99]. To make our investigation inclusive, we consider a wide range of restart probabilities $1-a \in \{50\%, 30\%, 15\%, 1\%\}$.

Different restart probabilities need different number of iterations to converge to a robust node rank order [100], i.e. to arrive at ranks whose pairwise comparisons are not affected by additional iterations. Therefore, a different stopping point needs to be adopted when running personalized PageRank each time. An important barrier in doing so is that rank order robustness measures, such as numerical tolerance or rank order correlation with the previous iteration, are heavily influenced by the scheme's convergence speed and could be too lax or too strict with no way of assessing this.

To avoid time-consuming and potentially biased case-by-case empirical investigation of the best point to stop personalized PageRank, we developed a methodology for estimating that point [101]. In particular, we argue that node rank order is robust and we account for most random walks. To explore this property, we consider that the random walk with restart scheme characterizing personalized PageRank can be defined as several independent sub-processes $walker_k$, $k = 1...n$ that start from the given seed nodes and repeatedly perform random walks of exactly k steps before restarting. We also consider that the random walk with restart process spans intermediate steps $\ell = 1, ..., w$ of transitioning to the next node, where w is an arbitrarily large number. For example, w could be the number of steps required to infer node ranks within a tight numerical precision by using the law of large numbers to count how many arrive on each node. Then, the random walk with restart process can be equivalently formulated as selecting a sub-process walkerk at random and then performing *k* steps before restarting from a new one. To express this behavior at any intermediate step *l*, we employ the following random variables:

$$S(\ell) = \{1 \text{ if at iteration } \ell, 0 \text{ otherwise}\}$$
$$X_k(\ell, n) = \{1 \text{ if } \ell \text{ in } walker_k, 0 \text{ otherwise}\}$$
$$W_k(\ell, n) = \{1 \text{ if } walker_k \text{ restarts at } \ell, 0 \text{ otherwise}\}$$

Since each intermediate step occurs once, $E[S(\ell)] = 1$. Furthermore, the random walk sub-processes cannot be interrupted, which makes the probability of selecting one equal to the probability of performing a random walk of the respective length:

$$E[X_k(\ell,n)] = P(\text{the random walk restarts at iteration } k) = (1-a)a^{k-1}$$

where $1-a$ is the restart probability of the random walk at each step that should not be chosen for $\ell = 1, \ldots, k-1$ consecutive steps and then should be chosen for the last step. Additionally, intermediate steps ℓ correspond on average to random points within sub-processes k that perform random walks of that length, which makes the probability of restarting at that particular point:

$$E[Wk(n)] = w/k$$

Then, the number of random walks $W(n)$ up to iteration n is equal to the number of restarts:

$$W(n) = \sum_{\ell=1}^{w} \left(S(\ell) \sum_{k=1}^{n} X_k(\ell,n) W_k(\ell,n) \right)$$

Finally, the above random variables are independent to each other. Therefore, the expected value operator $E[\cdot]$ yields:

$$E[W(n)] = w \frac{1-a}{a} \sum_{k=1}^{n} \frac{a^k}{k}$$

This means that the fraction p of expected random walks considered at iteration n is:

$$p = \frac{E[W(n)]}{E[W(\infty)]} = \frac{1}{\ln(1-a)} \sum_{k=1}^{n} \frac{a^k}{k} \qquad (5.1)$$

This probabilistic analysis can help us designate a-priori a robust stopping point of PPR repetitions. For example, it can be used to show that 99% of random walks are accounted for at the number of iterations presented in Table 4. Although more intricate stopping points can be obtained if we consider the actual ranks arising in each iteration, our experiments revealed that the given number of iterations approximate the same graph diffusion outcome fairly well, i.e. with over 99.8% node rank order correlation with the eventual outcome.

Table 4. Number of PPR iterations that account for 99% of random walks.

Restart probability	Number of iterations
50%	5
30%	9
15%	17
1%	203

GNN Mining

The second type of graph mining we investigate is following a GNN architecture to understand user embeddings that affect the formation of graph edges. To do this, we empirically select 32-dimensional embeddings on all layers, 150 training epochs (i.e. updates to parameters) and a $\lambda = 0.01$ regularization parameter, which we then asserted to form pareto-optimal solutions (i.e. that cannot be improved further by changing any parameter) in all experiments detailed in this section if we train towards minimizing the regularized cross-entropy loss of (3.10). Therefore, we adopt these hyper-parameter values in all implemented GNN architectures.

On the other hand, the efficacy of GNNs is known to vary greatly, depending on the number of layers and type of layer activation. We explore the combination of N{1,2,3} layers and the following activation functions:

- *Linear activation.* Doesn't apply any activation function, i.e $\sigma(x) = x$. This corresponds to traditional node embedding approaches.that aim to find underlying representations that can directly reconstruct the adjacency matrix. It is well-known however, that non-linear functions often perform better in multilayer architectures.
- *ReLU activation.* Using rectified linear units $\sigma(x) = max(x, 0)$. These are similar to linear activations but introduce a form of pseudo-linearity.
- *Sigmoid activation.* $\sigma(x) = 1/(1 + e^{-x})$
- *tanh activation.* $\sigma(x) = (e^x - e^{-x})/(e^x + e^{-x})$Introduces non-linearity, while at the same time avoids the issue of gradients vanishing when being propagated to lower layers of the architecture [102].

To facilitate an investigation of many potential architectures, we employ NVIDIA's Tensorflow platform for Python, which speeds up matrix and vector computations by performing them in parallel over the graphic card's processing units. This way, the training time of machine learning models is reduced by more than 100x. Unfortunately, existing GNN libraries, such as Microsoft's tf-gnn[1] are focused on making predictions based on a small number (e.g. tens) of attributes and are not designed for predicting node links based on the high-dimensional one-hot encoding of node ids needed for unsupervised training (see subsection 3.4).

Due to the lack of existing tools suited to our investigation, we developed a generic GNN architecture for Python Tensorflow[2] that can be parameterized to replicate the investigated GNN architectures. In the same project, we also implemented graph diffusion strategies so that node scores are obtained in parallel for all example nodes.

Predicting Static Relations

We start by comparing the ability of graph mining approaches to reconstruct relations in a network. We remind that our goal is to select the best practices those approaches should follow and adapt only those to account for temporal information and the decentralized setting of HELIOS.

To do this, we first perform experiments by training the previously presented graph mining approaches on a random subset of 80% of all node combinations, which are assigned a label of 1 if they correspond to a graph edge and 0 if they do not. Then, we measure whether the trained algorithms can help predict the remainder 20% of node combinations. For example, if we consider the example graph of Figure 14, the labels of all potential node pairs are {AB: 1, AC: 1, AD: 1, AE: 0, AF: 0, BC: 0, BD: 0, BE: 0, BF: 0, CD: 1, CE: 1, CF: 0, DE: 0, DF: 1, EF: 1} and using 80% of them for training, could leave the randomly selected sample of {AC: 1, DE: 0, CF: 0} for evaluation.

[1] https://github.com/microsoft/tf-gnn-samples
[2] https://github.com/maniospas/gnn-test

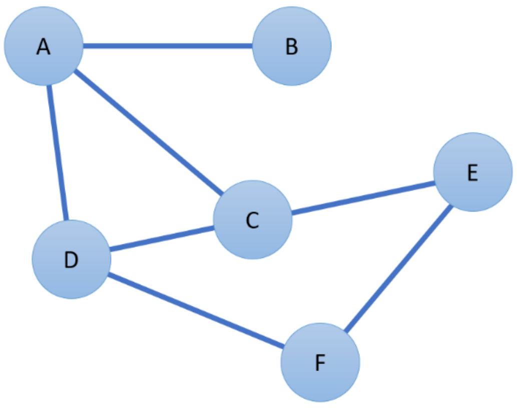


Figure 14. An example graph for link prediction.

To speed up training and evaluation time, we additionally follow a negative sampling methodology, in which we limit the number of node pairs to a randomly selected subset of at most 100 for each node (i.e. each node is paired with at most 100 others to form examples of non-existing edges).

Then, we evaluate whether the predicted probabilities of edges existing are assigned higher values for the withheld edges compared to non-existing edges using the Area Under Curve (AUC) measure. AUC effectively measures the portion of existing edges having higher scores than negative connections and is robust in that it is not affected by the potentially much smaller number of existing vs. non-existing edges. Formally, if we denote as $TPR(r)$ and $FPR(r)$ as the fraction of edges with predicted similarity greater than *r* and the fraction of non-edge node pairs with predicted similarity greater than *r*, AUC is defined as the area under the FPR-TPR plot:

$$AUC = \int_0^1 TPR(r)dFPR(r) \tag{5.2}$$

When nodes are assigned random similarities, AUC assumes a value of 50%, whereas perfectly assigning higher similarities to node pairs that correspond to edges yields 100% AUC.

As a final step, we ensure that results are not biased by the random initialization of training edges and neural parameters. To this end, we average the AUC evaluation over the test edges across 5 experiments. A summary of our evaluated graph mining algorithms and the obtained results is presented in Table 5.

In general, we can see that multilayer architectures fail to improve the predictive efficacy of employed single-layer GNNs. We attribute this finding to the fact that representations of GNNs are not positionally aware [103-105] in the sense that their representations can reconstruct graph structure but not longer paths. For example, “if two nodes reside in very different parts of the graph but have topologically the same (local) neighbourhood structure, they will have identical GNN structure” [105].

The failure of multilayer architectures in improving link prediction can be attributed to GNNs learning to both distinguish and match neighbors at distances up to the number of layers *N*. Therefore, if matching connected nodes is strongly favored by training algorithms, as often done by training objectives, the representations a given node can match are similar to nodes that reside up to twice that distance away. For example, in Figure 15 a two-layer architecture would make node C obtain similar representations not only to D and B but also A. However, due to that graph’s symmetry, these representations would pass through the edges AB’ and AD’ to the right-hand side, where the inverse process would require similar representations by C' to C. Only the edge BB' would provide a regulatory role to make the representations of the two sides.

Table 5. AUC for relation recommendation averaged across 5 experiments. The best mining methods for each graph (i.e. each column) are bolded. The standard deviation across each averaging was less than 5%.

		Static networks		Dynamic networks			
Algorithm		**Friends**	**Follows**	**Messages**	**Wall**	**Email**	**SMS**
PPR	a = 0.5	96%	64%	91%	90%	74%	36%
PPR	a = 0.7	96%	97%	90%	93%	77%	40%
PPR	a = 0.85	97%	93%	90%	89%	69%	20%
PPR	a = 0.99	96%	91%	89%	77%	71%	21%
GNN	N = 1, linear	97%	97%	93%	93%	85%	94%
GNN	N = 2, linear	98%	79%	50%	55%	50%	50%
GNN	N = 3, linear	50%	50%	50%	50%	50%	50%
GNN	N = 1, ReLU	98%	97%	91%	94%	93%	**99%**
GNN	N = 2, ReLU	50%	50%	50%	55%	50%	50%
GNN	N = 3, ReLU	50%	50%	50%	50%	50%	50%
GNN	N=1, sigmoid	98%	**98%**	93%	91%	92%	**99%**
GNN	N=2, sigmoid	90%	96%	93%	90%	68%	98%
GNN	N=3, sigmoid	86%	**98%**	93%	88%	42%	90%
GNN	N=1, tanh	**99%**	**98%**	**94%**	**96%**	**94%**	**99%**
GNN	N=2, tanh	98%	97%	93%	95%	84%	**99%**
GNN	N=3, tanh	97%	96%	93%	90%	69%	**99%**

On the other hand, if the GNN architecture is shallower (e.g. a single-layer one) then localized structural intricacies are easier to learn. This phenomenon has also been recognized by previous research on graph labeling under scarce data [104, 106], whose theoretical probing reveals that applying more than two layers of graph smoothing operations, such as graph convolution, tends to smoothen out the representations of graph nodes by biasing them towards a uniform distribution. Furthermore, is has been argued that simpler representations [107] can be more powerful when graph neural networks are trained under the same mechanism.

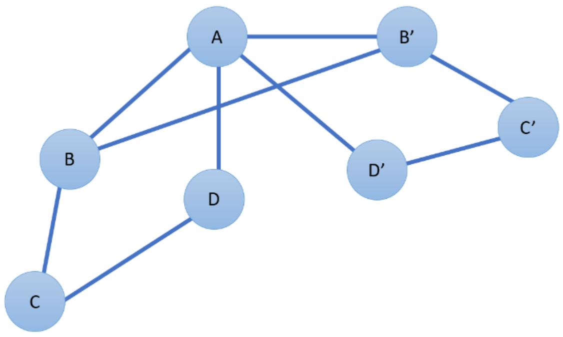


Figure 15. Nodes $\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}$ would obtain non-positionally-aware representations similar to $\boldsymbol{B}', \boldsymbol{C}', \boldsymbol{D}'$ respectively.

Effectively, this subsection identifies shallow GNN architectures of a single graph convolutional layer as the ones most suited for unsupervised learning of social network relations. Hence, our analysis also focuses on single-layer architectures. These can be considered an enrichment of node embeddings in that they also aim to learn the similarity function of the embedding space.

5.2 Evolving User Preferences for Rediscovering Interactions

The need to account for temporal preference evolution

Given the success of shallow GNN architectures in reconstructing graph relations by capturing latent user preference embeddings, we investigate whether they are equally powerful in recommending future interactions. An important point to consider when assessing their efficacy in this task is that social media users often engage in exchanges that can be conceptually considered part of larger interactions but which involve many lower-level ones provided by the social network application. For example, two users could exchange multiple messages in short order when holding a conversation. However, it is difficult to identify the precise point at which individual interactions can be grouped into thematically larger ones, in part because users often hold multiple of the latter at the same time or take the time to interact with a third person between their exchanges. At the same time, we argue that their most recent interactions are often fresh in their memory and there is no particular need to recommend those. For example, this frequently happens in the Messages, Wall, Email and SMS datasets, where 64%, 57%, 9% and 92% of user interactions respectively occur towards one of the last three interacted alters.

Based on the above, we recognize that, when users favor recent interactions, it is more important for interaction recommendation mechanisms to recommend “older” ones. Therefore, we propose adapting traditional recommendation assessment measures to favor rediscovery of older interactions. To this end, we focus on adapting the notion of Hit Rate of the top k recommendations ($HR@k$), which expresses whether the next occurring interaction the user performs (i.e. towards another user) is among the top k recommended ones. Our proposed adaptation avoids favoring ongoing larger thematic interactions between users (which are trivial to predict) by not aiming to recommend any of the last κ interactions each user has engaged in as either a source or a destination endpoint. To formally express this, we introduce a measure called Discovery Hit Rate of the k recommendations while ignoring the chronologically last κ ones ($DHR@k, \kappa$). If we consider a time-dependent interaction recommendation system $R(u, t)$ that recommends at times t who the node u should interact and a recommendation system $B(u, t)$ that prioritizes the last interactions of the same node before that time, this new measure can be formally expressed as:

$$DHR@k,\kappa = \frac{|\{(u,v,t) \in E: v \in \text{top } k \text{ of } R(u,t), v \notin \text{top } \kappa \text{ of } \mathrm{B}(u,t)\}|}{|(u,v,t) \in E: v \notin \text{top } \kappa \text{ of } \mathrm{B}(u,t)|} \quad (5.3)$$

where E are the graph's interactions (u,v,t) from nodes u to v at time t and $|\cdot|$ counts the number of set elements. Higher DHR values (i.e closer to 1) indicate perfect prediction of which older interactions to revisit. The parameter κ captures how old the revisiting interactions should be; when $\kappa = 0$ all interactions are allowed and $DHR@k,0 = HR@k$. In the case where $B(u,t)$ does not often predict (e.g. as happens in the Email dataset where users re-engage in their most recent interactons only 9% of the time), it holds that $DHR@k,\kappa \approx HR@k$.

Using this measure, we then conduct experiments on the Messages, Wall and Email datasets, in which we measure $DHR@3{,}3$ and $DHR@3{,}6$. Since the quality of recommendations could change over time, for example, as more users are inserted as graph nodes, we report the average DHR values across the last 1000 occurring interactions for which users had to select who to interact with between at least 9 others (we don't experiment on the SMS dataset, because its users don't form relations with that many people). The same methodology will be followed throughout the rest of this section, for example to assess the efficacy of decentralized GNNs researched later on.

In the first series of experiments, we utilize all interactions that have occured at times previous to t to train a single-layer GNN. This training is slightly different to the relation prediction GNNs of the previous subsection in that multiple edges (which correspond to multiple previous interactions) could exist between the same nodes. Then, the new model can be considered to capture user preferences over all past interactions. This model is compared with the heuristic baseline of favoring more recent interactions, regardless of who was their sender or receiver. The top κ predictions of this heuristic are the ones DHR ignores, but we assume that the interest in older interactions could also depend on their chronological order. For example, we expect that users would be more willing to interact with more recent acquaintances (e.g. from days prior) rather than exceptionally old ones (e.g. from months or years prior).

This first comparison is presented in Figure 16 alongside subsequent experiments. We can see that the centralized GNN architecture does not exhibit a high discovery rate of previous interactions, even failing to provide an improvement compared to our heuristic baseline for the Messages dataset. We attribute this finding to interaction-related user preferences being systematically related to the attention users place on their involved interactions, i.e. that users tend to interact with people more recently brought to their attention.

To model this type of preference evolution, we propose weighting the importance placed during training on accurate edge reconstruction so that earlier interactions -which may not drive user preferences later- are eventually forgotten. This does not affect whether old interactions would be re-recommended, as user preferences can still float towards them by new interactions that reflect similar preferences to the forgotten ones.

Temporal degradation of training example importance

To facilitate mining the temporal evolution of user preferences, we propose a new edge weighting mechanism that accounts for the temporal evolution of GNN parameters. In particular, we propose that each time a new interaction is introduced to the graph, the weight of accurately predicting previous ones is multiplied with a constant degrading factor $a \in [0,1]$. If we assume a discrete representation of times as ordinals $t = 1,2,3,\dots$ the weight w of each interaction (u,v,t) at time T can be theoretically expressed as:

$$w(T,t) = \{a^{T-t} \text{ if } t \leq T, 0 \text{ otherwise}\} \quad (5.4)$$

which prevents future interactions from being involved in the training scheme.
Unfortunately, weighting previous interactions does not suffice in setting up a GNN training scheme, as negative examples (i.e. interactions that do not occur and which training should learn to avoid recommending) also need to be weighted. At the very least, the negative sampling mechanism should account for the weights of positive examples so as not to train the architecture on a set of predominantly negative and disproportionately few positive examples. To this end, we propose an alternative method for providing weighted training data, in which negative examples are randomly generated as permutations of positive ones (i.e. by exchanging one of the edge endpoints with another random node of the graph) and are assigned the same weight as those.

In particular, we consider a list of training interactions D in which we place weighted labeled edges (u, v, t, w, l) where w are the training weights and $l \in \{0,1\}$ are binary labels of whether the edge is a positive or negative example at time t. When a new interaction (u, v, t) occurs, all the weights of tuples in D are multiplied with the aforementioned degrading factor a and three new tuples are added to it: a positive training example of the interaction $(u, v, t, 1,1)$ and two negative examples $(u, v', t, 1,0)$ and $(v', v, t, 1,0)$, where $v' \neq u, v$ is randomly sampled among the neighbors of u.

Given this new scheme of organizing weighted training positive and negative examples, the loss function of (3.10) can be adjusted to depend on the current time T when applied on a single-layer GNN that recommends interactions (u, v, T) with estimated probability $R(v|u, t)$:

$$loss(T) = \sum_{(u,v,t,w)\in D} w\, CrosE\big(l, R(v|u,t)\big) + \lambda \sum_{p\in params} p^2 \qquad (5.5)$$

where p are the GNN parameters *N* its number of layers and the cross-entropy loss of a single example is calculated as $CrossE(y, y) = -y \log_2 y - (1 - y) \log_2(1 - y)$. After each new interaction, we retrain towards this loss so that node preferences are guided towards understanding the most recent user preferences. Furthermore, we initialize GNN parameters and user preferences with those found during previous training steps, which helps preserve the understanding of previously understood user behavior that do not pertain to shifts in who users interact with. To understand how the above temporal-aware loss function relates to the original loss function (3.9), we can see that, when a new interaction (u, v, T) becomes available we can rewrite this as:

$$loss(T) = loss(0) + \sum_{t=1}^{T} w(T, t) \Delta_{(u,v,T)} loss$$

where the $\Delta_{(u,v,t)}$ operator designates the value change before and after accounting for a new interaction (u, v, t). The last expression is the discrete equivalent of integrating a time-degrading transformation of the loss's derivative. Hence $loss(T)$ can be considered to capture the evolution of an interaction recommendation model's original loss function as new interactions occur. A similar scheme is also followed by stochastic gradient descent optimization [108], in which training is applied on one example (in our case: one positive and its corresponding negative examples) at a time and still reaches a (locally) optimal point We finally investigate the applicability of this newly-proposed temporal GNN training scheme in discovering user interactions:

- *Favor Last Interactions.* The (strictly better than random) baseline proposed in the last subsection.
- *Static GNN.* A (centralized) single-layer GNN trained towards minimizing the loss (3.9).
- *Temporal GNN.* A (centralized) single-layer GNN trained towards minimizing the loss (5.5) . Through a perfunctory exploration of its efficacy for various degrading factors of example importances, we select $a = 0.5$. We also exclude training tuples with weights $w < 0.01$ from training, as at that point they only slightly affect the training's outcome. An empirical investigation leads us to adopting the well-performing regularization parameter $\lambda = 0.1$.

In Figure 16 we show a comparison between temporal GNN mining and the previous baselines in terms of DHR. We can see that taking advantage of the temporal component helps significantly outperform the baselines in discovering more interactions older than both the three or the six previous ones.

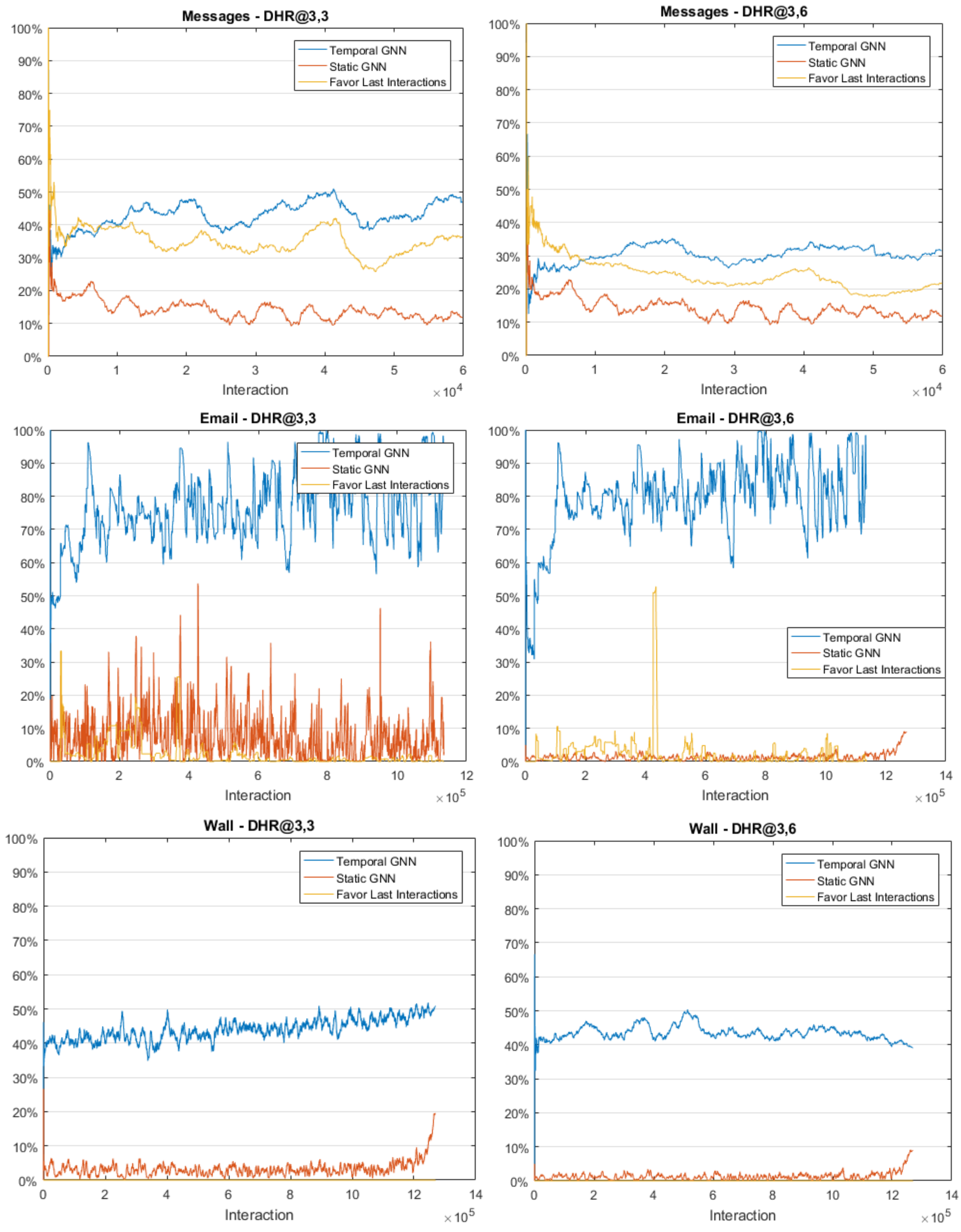


Figure 16. Using temporal GNNs to recommend older interactions. Plots show DHR@3,3 (y-axis in the left column) and DHR@3,6 (y-axis in the right column) over occurring interactions (x-axis).

5.3 From Distributed to Federated Learning

Having identified the most promising centralized GNN architectures for social graph mining, we now research how these can be deployed in the decentralized setting of HELIOS. In this respect, we remind that graph-based machine learning is a fairly new discipline and hence little work has been done to detach it from a central processing node. Hence, before tackling the decentralization of GNN architectures, we need to investigate the communication protocols they should follow, so as to limit their operations only to those provided by such protocols.

In subsection 3.5 we already mention that there exists a valid background on learning models in multiprocessor systems, a process referred to as distributed learning. Distributed learning often delegates computations across many devices that report back their results to a central processing node that is responsible for learning an updated model. To ensure privacy of individual device data, this process can even involve devices reporting only gradients instead of found examples to the central node, where they can be aggregated with stochastic gradient descent. Furthermore, this practice performs the bulk of computations in individual devices, which run in parallel and hence reduce proportionally the training time of machine learning models. An example of this process is outlined in Figure 17.

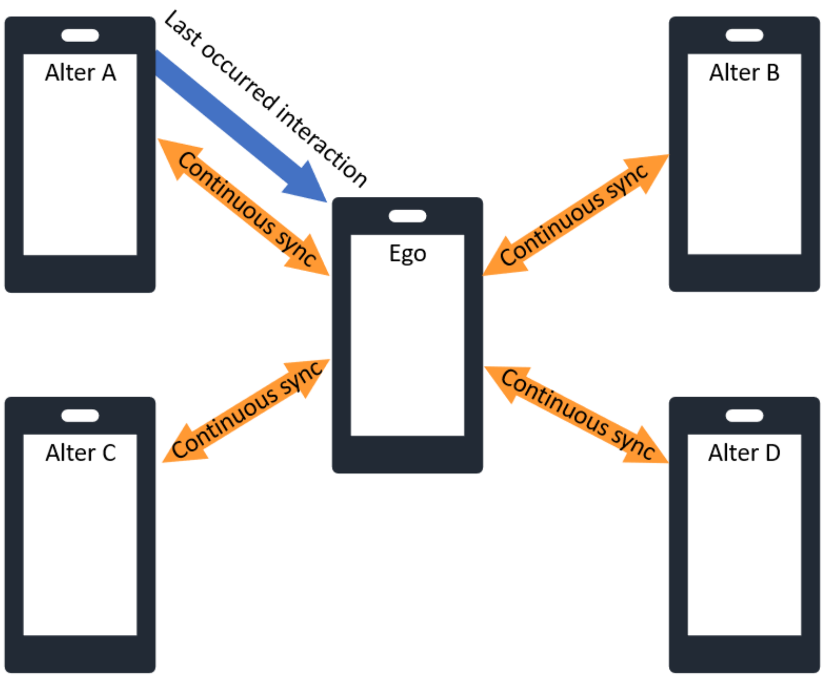


Figure 17. A decentralized machine learning scheme from the viewpoint of an ego device that continuously exchanges parameters (sync) with all its alters, even if no interactions are ongoing with the alters $\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}$.

In this setting it is obvious that, despite devices not needing knowledge of non-alters, a central service is needed to orchestrate the learning process. As a result, distributed systems are not truly decentralized. This shortcoming has motivated a natural extension of the previous process in which each device sees itself as a central service and aims to elicit a machine learning model. In terms of social networks, each user's device communicates with the alters's devices in order to drive a machine learning task.

This practice works well for machine learning algorithms for which a few training examples suffice to communicate their parameters, such as simple linear regression or rank diffusion models. In fact, constraining machine learning only on local areas of the network avoids the -often erroneousassumption that graph dynamics, such as the mechanisms that drive the formation of edges, are Independent and Identically Distributed (IID). Furthermore, when more complex machine learning models need to be learned, where we would expect individual devices to fail to learn parameters of adequate quality, we can organize the learning process so that neighboring devices simultaneously converge towards similar model parameters. For example, these can be exchanged and averaged over each device's alters.

Overall, the above-described extensions of distributed learning over decentralized systems can reproduce a variety of machine learning algorithms, such as the neural network architectures described in this work. However, when we try to apply them to DSNs, we find that they commonly exhibit three important shortcomings that come at odds with the dynamic, evolving and irregular aspects of user behavior:

a) Distributed learning cannot learn node representations that change over time
To help understand this claim, we point out that node embeddings are also part of the learned GNN parameters and should hence be communicated during parameter transfer. Hence, general-purpose distributed learning systems would make all users learn a copy of all other updated user embeddings. Doing this would require transferring all embeddings at each information exchange step; this would lead to an impossible overhead for large social networks with thousands or millions of users.

In practice, a large part of the described overhead can be avoided by embedding mechanisms that consider nodes at most a few edge hops away. In this case, nodes can store only the needed embeddings and exchange updates representations only with their neighbors that also need those nodes. This optimization strongly localizes embedding storage and transfer. However, it does not address the core problem of communicating the representations of a large number of nodes. Additionally, it introduces challenges of actually discovering who resides only a few hops away, for example by running one random walk mechanism originating from one user through the decentralized graph.

Given the above, both the notion of globally trained parameters and the embedding architecture become significantly deformed. Hence, we propose that, to learn structural node embeddings in decentralized settings, new algorithms and paradigms that differ from traditional GNN architectures should be developed. As an example, we point to the work of Kermarrec et al. [109], which sets up a model of local repulsion and attraction to guide originally random graph node embeddings into a multi-dimensional layout.

b) Distributed learning encounters failures when users go offline
This concern is even more prevalent in social networks built over peer-to-peer protocols that can connect physically proximal users (e.g. exchange messages via Bluetooth), who can go out of range. In distributed protocols, users select at random or all neighbors to exchange information with based on synchronous or asynchronous processes. However, their neighbors -or even the users themselves for that matter- may leave the network or the user's neighborhood for indeterminate periods of time and hence may not be there when their neighbors reach out to them or when the devices themselves would need to request information. We recognize three possible resolutions of this phenomenon:

- *Do not perform the information exchange.* This effectively ignores information for the sake of convenience and hence reduces the predictive capabilities of the learned embeddings. Furthermore, if users interact for brief periods of time that do not coincide with the timing of information exchanges, some or all alters may never be taken into account.
- *Buffer the information exchange* and perform it the next time the alter becomes available, for example when both users come online. This poses several technical challenges, such as handling accumulated exchanges and the communication overhead of sending the

buffered ones within a short timeframe. Furthermore, if the information exchange happens much later than its original timing, it may worsen the prediction at the arrived alter. Finally, buffered exchanges would need to be dropped after certain time periods to account for possible churn and this introduces some of the drawbacks of the previous solution. It must be stressed that buffering the information exchanges needed for distributed learning is much more resource intensive compared to buffering messages, as distributed learning needs to accumulate many (e.g. tens of) information exchanges with all alters every time a user's preferences change, such as on each of their interactions.

- *Exchange information only with the currently available alters.* In this case, extracted node embeddings would be biased towards accurate predictions between the users more often available. Furthermore, the extreme case of this solution is exchanging information with no alters if they all happen to be offline.

The above points suggest that performing information exchanges on-demand in DSNs is not possible unless constant network presence can be assumed. Besides the introduction of always-online super-peers, as we did in Section 4, the only way to guarantee information transfer when users go online or offline is for information exchanges to happen at (approximately) the same time as new changes are induced. In this work, we adopt this direction by proposing an embedding mechanism that sends and updates parameters only when new interactions occur.

c) Distributed learning requires time and space homogeneity

Often, distributed learning assumes that learned parameters, such as node embeddings, remain the same throughout training. Based on this assumption, their preferred method to guaranteeing the convergence of learned parameters is employing learning rates that are dampened through time. However, the notion of needing to converge to fixed values becomes meaningless when considering real-world scenarios in which node preferences may drift over time, as for example happens when trying to mine user preferences that drive their interactions. To make matters worse, learned parameters may also differ between different areas of the graph, for example because different certain groups of users adhere to different social norms.

Approaches that aim to remain aware of the above issues while providing decentralized solutions to machine learning tasks are commonly referred to as *federated learning*. Unfortunately, federated learning has been, up to now, constrained to simple algorithms [110] that are not suited to performing edge (e.g. relation or interaction) recommendations. As a result, our subsequent research actions aim to deliver federated learning implementations that can help understand temporal user preferences. In particular, we aim to adapt our previously proposed temporal GNN framework in a decentralized setting that does not suffer from the aforementioned shortcomings.

To this end, we finish this subsection by proposing an ideal federating learning protocol that is applicable to DSNs; this protocol consists of performing one cycle of parameter exchanges during every occurred interaction, as demonstrated in Figure 18. This ensures that machine learning parameters are exchanged at times when a communication channel (i.e. the one through which the interaction travels through) is known to be available and enforces a local view of the network, under which each device is only aware of its neighbors. Of course, these advantages effectively move the innovation burden from the communication protocol to the machine learning algorithms to be trained under such limited capabilities of accessing and adjusting information.

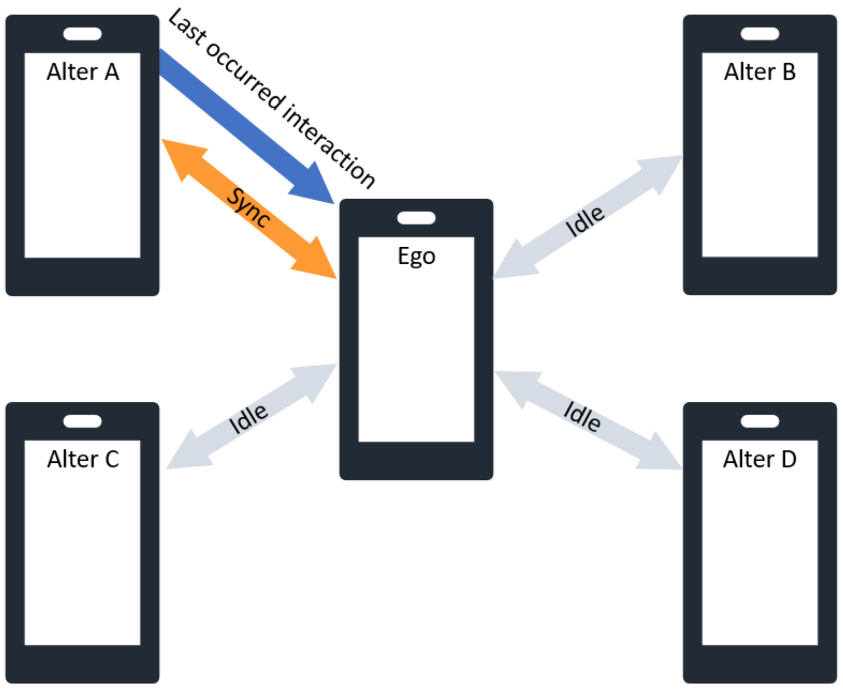


Figure 18. Our proposed federated learning scheme. Contrary to Figure 17. The device exchanges parameters (sync) only with the currently interacting alter.

5.4 Decentralized Interaction Mining

Aggregating device losses

In the previous subsection we explained that a federated scheme for training machine learning algorithms in DSNs should ideally perform parameter exchanges only between the devices involved in the interaction.

To set up a parameter exchange protocol that can help train a GNN architecture under this constraint, we first consider a localization of the temporal loss (5.5) on only one user's u device, on which a recommendation algorithm calculates estimated probabilities of performing interactions towards nodes v at times t as $R(v|u,T)$. Then, we analyze how a temporal GNN training scheme would work if it was forced to locally run only on this device with no additional communication; it would create a set of local training examples, which will annotate as $D[u]$, that are formed with the same weighting and negative sampling process as the previous training examples D but only on the interactions in which u is involved.

If we aggregate these training examples over all devices, the main difference compared to the previously centralized list of examples D is that the negative examples are sampled out of interactions with previous alters, which makes forget previous interactions faster. At the same time though, weight degradation occurs only if new examples are provided to the specific device and hence very old examples on devices that have not previously interacted with others may still be left with high weights. We theorize that, in practice, these two mechanisms work competitively to provide similar lengths of preference memory as in the centralized setting.

This can be formally corroborated in the specific case when user activity patterns remain unchanging over time and identically distributed between users; then it holds true that the average number of interactions between each user and the last interaction in the graph $E[\Delta T]$ is the same between users and that user interactions occur in statistically independent times. Hence, if a training example $(u, v, t, w, l) \in D[u]$ is stored in a user's device, adding the same example in a centralized list of holding examples as $(u, v, t, w_{centralized}, l) \in D$ would on average produce the following relation between their weights:

$$E[w] = w_{centralized} a^{-E[\Delta T]}$$

where a is the example degradation parameter shared between all users. Therefore, if we consider the decentralized loss $loss(u, T) = loss(T)$ trained on $D[u]$ for regularization parameter $\lambda(u) = \lambda a^{-E[\Delta T]}$ it holds that:

$$loss(T) = \alpha^{-E[\Delta T]} E\left[\sum_u loss(u, T[u])\right]$$

where $T[u]$ are the last time users u have been involved in interactions and $loss(T)$ is trained on the examples $D = \cup_u D[u]$. Since $a^{-E[\Delta T]}$ is effectively a constant, this means that the centralized loss minimizes the expected value $E[\cdot]$ of aggregating all decentralized losses over user devices.

Regularizing towards exchanged embeddings

The analysis of the previous paragraph reveals that, if the decentralized loss function is minimized on each user device, then the overall loss should also exhibit a (local) minimal point as their aggregation. Then, it remains to ensure that the found point lies near global minima. An important challenge to this end is that each device would learn to perform recommendations with different criteria than its neighbors that may overfit on training examples with little generalization capabilities towards the future. Effectively, different devices should produce similar understandings of which latent user preference is which; these preferences may not necessarily be understandable by humans, but they should be understandable by the GNN architecture in the same or similar ways. The centralized nature of GNN architectures systemically avoids the above discrepancy, since their intermediate node embeddings are implicitly constraining towards obtaining the same representations regardless of who they are compared to. However, in our decentralized setting different devices may learn different representations for the same node not only due to different training examples (which to some extent cannot be avoided) but also due to selecting different latent preference spaces to train on.

Our research has led us to the problem of constraining embeddings of the same nodes to lie in similar latent spaces in the sense that when they are similar, the users exhibit similar preferences under the understanding of all devices. To help train towards this property, we consider the representations $H(v|u, T)$ the device of user u learns for user vby inputting their one-hot encoded ids. An easy trick to first remove the (for the device) unknown nodes of the graph is to see that we can do away with the whole one-hot encodings and directly learn this feature representation, i.e. that we learn the outcome of the embedding mechanism. Then, prediction tasks for user u would consist of matching that user with the representations of their alters, i.e. to compare $H(u|u, T)$ with $H(v|u, T)$ through the GNN architecture.

To do this, we come back to the concept of regularization and how it constrains the order of magnitude of learned parameters by penalizing them when they obtain large values. Then, we propose that the parameters pertaining to the extraction of node embeddings could instead be regularized towards that node's understanding about itself, i.e. that $H(v|u, T)$ should be similar to $H(v|v, T-1)$; these two quantities could not be forced to be the same by assignment, but regularizing them with each other means that the regularization point when learning $H(v|u, T)$ is moved with $H(v|v, T-1)$ as the center. In other words, the former still learns new representations, but these lie close to the latter in both latent preference spaces.

The process of regularizing towards similar latent preference spaces of neighbors is visually demonstrated in Figure 19, where we can see that regularization effectively lets us move alongside minimizing the non-regularized part of the loss while also providing a solution that lies close to the one provided by the neighbors, which allows them to lie in similar latent preference spaces. If the same holds true between every pair of device and alter neighbor representations, then we can consider our proposed regularization to effectively align those spaces so that they are permutations of a common underlying one. At worst, the implicit understanding of that underlying latent preference space would change only slowly as we move away from each node.

This theorization leads us to defining the following loss function for training the embeddings of user devices:

$$loss(T) = \sum_{(u,v,t,w,l)\in D[u]} w\, CrosE\big(l, R(v|u,t)\big) + \lambda + \lambda[u] \sum_{v} \|H(v|u,T) - H(v|v,T-1)\|^2 + \sum_{p\in params} p^2 \tag{5.6a}$$

where $\|\cdot\|^2$ is the square of the L2 norm and is used to obtain the sum of all square differences between the learned and neighborhood representation elements. Then, all these local losses can be used to define the global loss function that the decentralized GNN algorithm implicitly aims to minimize at each point in time:

$$loss_{centralized}(T) = \sum_{u} loss(u, T[u]) \tag{5.6b}$$

Implementing this under the communication protocol of subsection 5.3 requires exchanging only the last representation each node has inferred about themselves $H(u|v,T-1) = H(v|v,T(v))$ with its interacting alter.

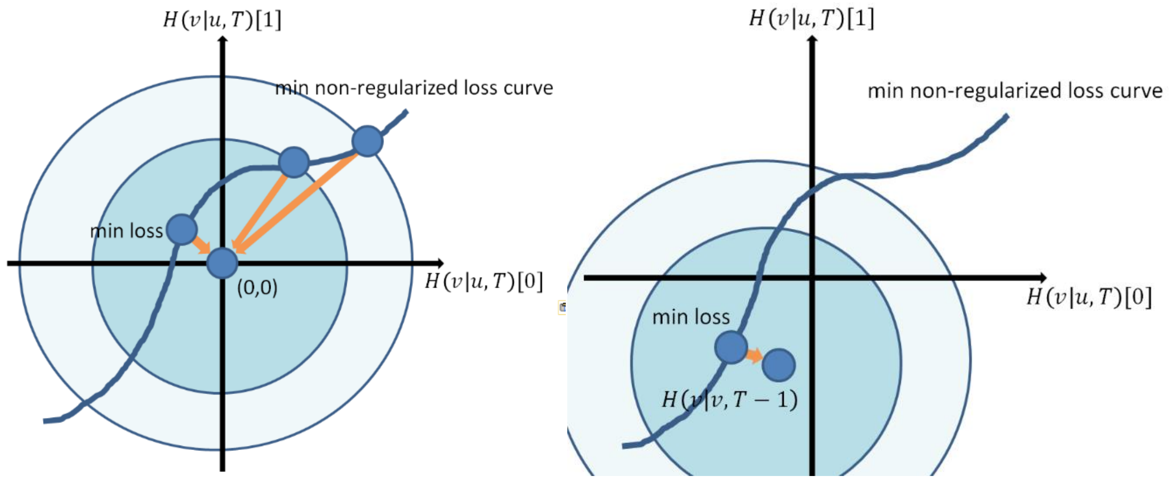


Figure 19. Regularization of the first two elements of $\boldsymbol{H(v|u,T)}$ towards zeros (left) and towards the embedding of the neighbor (right). The arrows show the attraction towards the regularization center (it becomes stronger the farther away from the embedding value currently is). The non-regularized version of the loss function is (approximately) minimized everywhere alongside the respective curve.

Plausible deniability and differential privacy

An important goal of the HELIOS platform is to provide users with control over how their personal information is disseminated to others. In the case of social graph mining, sensitive information pertains to their actions, such as who and when they interact. However, mined preferences of users partly encapsulate the mined preferences of their alters, which then can be sent to unrelated users. This type of information propagation is necessary to guide a uniform machine learning scheme throughout the graph. However, it is also important to avoid providing explicit understanding of user preferences (even if those are only latent ones) to non-alters.

Two theoretically-grounded concepts pertaining to privacy concerns are differential privacy [111,112], which measures how much information is revealed if multiple views of the same dataset are made publicly available, and plausible deniability [113,114] in which the user can deny interest in sensitive information with some probability. In the scope of interaction mining, these concepts translate to other users not being able to reconstruct one's interactions and for a user being able to deny that a set of mined preferences are the ones best matching them.

To address these concerns, we propose that, when sharing the preferences $H(v|v,T-1)$ with node neighbors, part of these can be obfuscated and replaced with random noise, as per the following formula:

$$\begin{aligned} H_{sent}(v|v,T-1) &= (1-privacy)H(v|v,T-1) \\ &+privacy[rand(\,),\dots rand(\,)]\|H(v|v,T-1)\| \end{aligned} \tag{5.7}$$

where H_{sent} are the representations sent to the alters, $privacy$ is a value within the range $[0,1]$ that corresponds to how much these representations are perturbed and $rand()$ is a process for creating random numbers. Hence, if someone tried to assign exact preferences $H(v|v,T-1)$ to users, these would at most be similar to the real ones by:

$$\|H_{sent}(v|v,T-1)-H(v|v,T-1)\|^2 \leq \|H(u|v,T-1)\|^2(1-privacy(1-E[rand(\,)]))$$

where $E[rand()]$ is the mean value of the random process. Hence, if $E[rand(\,)] \in (-\infty,1)$, positiveprivacy values help introduce plausible deniability over user preferences in that other users could be a better match to the permuted sent preferences. This deniability becomes even stronger when we consider that similar latent preferences are potentially shared between different users that reside in completely different areas of the graph.

The decentralized GNN architecture we propose also introduces differential privacy. In particular, even at the edge case where no plausible deniability mechanism has been introduced, it is impossible to reconstruct the (positive and negative) interaction examples that led the local copy of the temporal GNN running on a user's device to mine the particular preferences. To understand this claim, we point out that the number, chronological order and preference distribution of training examples all play an important role in selecting user preferences, where many different variations that would involve different users (among both alters and non-alters) can lead to mining the same preferences. As a result, it is impossible to reconstruct the interactions of a user through their preferences, unless all other users of the HELIOS platform explicitly collaborate by sharing their interactions with that user (or the knowledge that they have not interacted with them) and their mined latent preferences at the time of those interactions.

During the continuation of this task, a more formal investigation of plausible deniability and differential privacy over our federated learning scheme will be conducted, so as to numerically relate implementation parameters to the exact probabilistic definitions of these privacy concepts.

Experiments

Having defined a decentralized adaptation of the temporal GNN training scheme proposed in subsection 5.2 that follows the federated protocol suggested for learning on DSNs in subsection 5.4, we now explore its efficacy in recommending old interactions in social media graphs. To do this, we compare it against its centralized counterpart and the baseline of favoring more recent interactions. In particular, we perform experiments in which we compare the following three algorithms:

- *Temporal GNN.* The centralized temporal GNN for interaction recommendation, which degrades weights of previous relations by multiplying them with $a = 0.5$ every time a new relation occurs. This is the architecture proposed in subsection 5.2.
- *DecentralizedTemporal GNN.* The decentralized adaptation of the temporal GNN proposed in this subsection using regularization parameters $\lambda[u] = \lambda_{sync} = 0.1$. We also apply a plausible deniability value of $privacy = 10\%$, which yields almost identical results to not applying any privacy.
- *Favor Last Interactions.* The baseline method of recommending the most recent of older interactions first.

The outcome of these experiments is presented in Figure 20. In that figure, we can see that the decentralized GNN architecture tends to closely preserve the accuracy of the centralized one. In particular, it yields an average drop of less than 3% of the DHR values in all but the last subfigure. Furthermore, it significantly outperforms the compared baseline, which usually fails to present meaningful predictions among older interactions altogether in the Wall and Email datasets.

These results indicate that the proposed decentralized GNN architecture can help provide a high-quality understanding of evolving latent user preferences that can help match the attention placed on their alters.

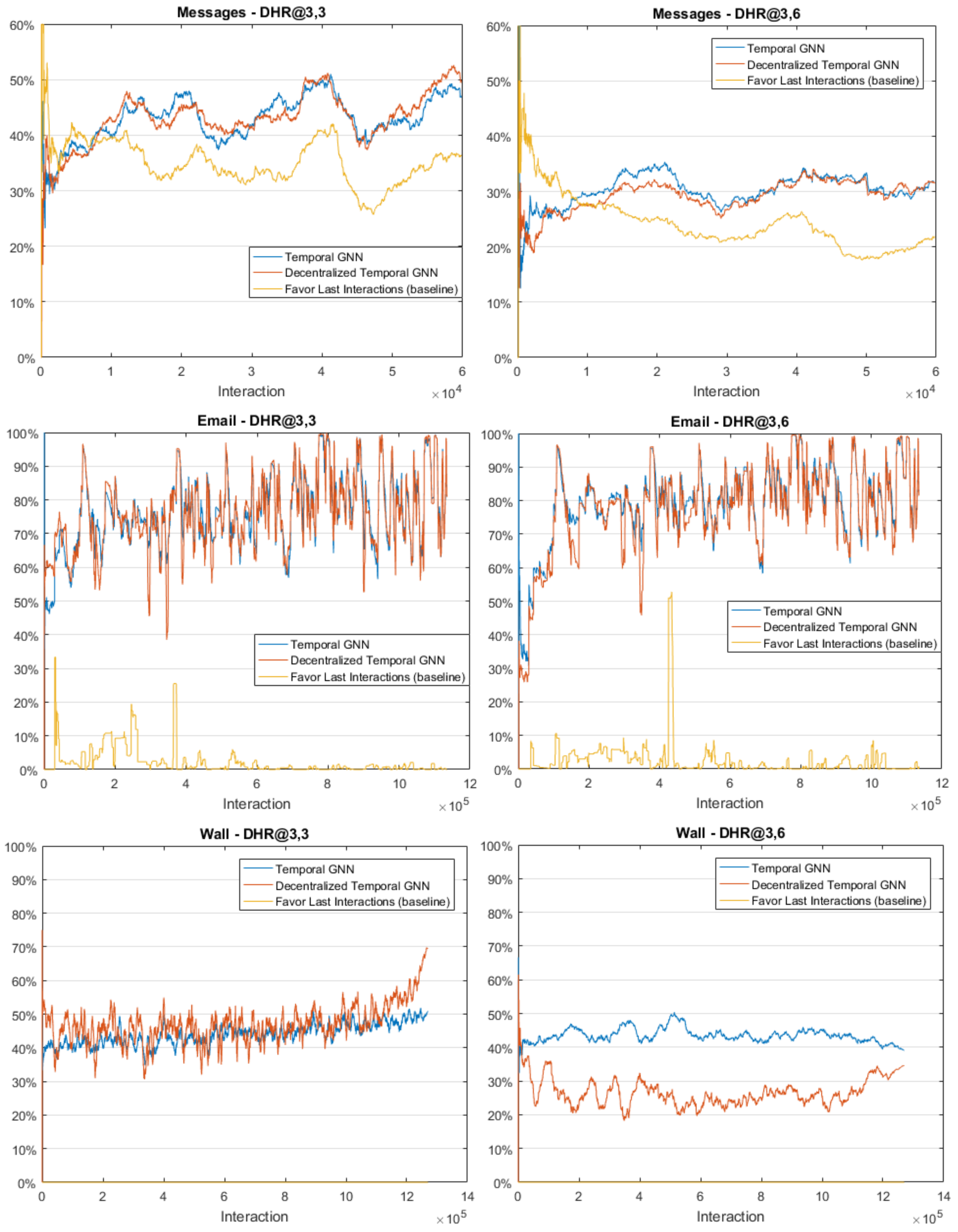


Figure 20. Comparing the decentralized and non-decentralized versions of the temporal GNN over occurring interactions (x-axis) with DHR@3,3 (y-axis in the left column) and DHR@3,6 (y-axis in the right column).

6 Mining the Heterogeneous Social Graph in HELIOS

Given the encouraging findings of our research, we implemented our proposed time-evolving DSN embedding protocol as a module of the HELIOS platform. In particular, we developed a graph recommendation module, which exposes this approach through a high-level Application Programming Interface (API) so that other HELIOS modules and applications can use it to enrich their functionality by mining the interactions occurring within the heterogeneous social network graph. This module provides context-specific recommendations based on mined user-specific preferences and hence uses the Contextual Ego Network (CEN) management library to attach information on these entities.

6.1 Extending the CEN Management Library

Deploying the decentralized graph mining algorithms developed in this deliverable to HELIOS devices requires the ability to manage data structures pertaining to the CEN, such as those defined in Deliverables 4.1 and 4.2. For convenience, these are summarized again in Table 6.

Table 6. Primitives of the CEN management library.

Primitive Type	Description
Node	Nodes of the heterogeneous social graph. These span both the user's device and its alters and contain a unique identifier with which the former recognizes them.
Context	The contexts each user has defined. Each context is effectively an ego graph that comprises some (but not necessarily all) of the user's alters, as well as edge relations between them and their interactions.
Edge	The relation edges formed between users in a context. If more than one types of relations are introduced by different modules, these can be differentiated by storing an appropriate flag object on the edge (see below for storing module-specific objects).
Interaction	Interactions that occur between users of the CEN. Each occurring interaction is effectively a temporal graph edge. Since interactions can only occur between related users, they are effectively tied to the respective relation edges.
ContextualEgoNetwork	The CEN object, which comprises the ego and alter nodes of a user, as well as their contexts.

A prototype library for managing the CEN has already been developed in the scope of the deliverables defining these data structures. The aim of this library is to be utilized by other HELIOS modules, such as the social graph recommendation module that is presented in this deliverable, and enable the development of applications that depend on the notion of the CEN and its relations and interactions. Additional needs arose during the development of this deliverable's algorithms, which led us to extending the library in the ways detailed in the rest of this subsection. Future development needs could also lead to introducing more yet-unforeseeable improvements.

An overview of the old combined with the new data structures is presented in the subsequent UML diagram of Figure 21. For the sake of conciseness, in that diagram we omit operations pertaining to the creation, addition and removal of objects, such as the ones tackled on the last point. A detailed view of all operations of the improved version of the library is provided in Annex I.

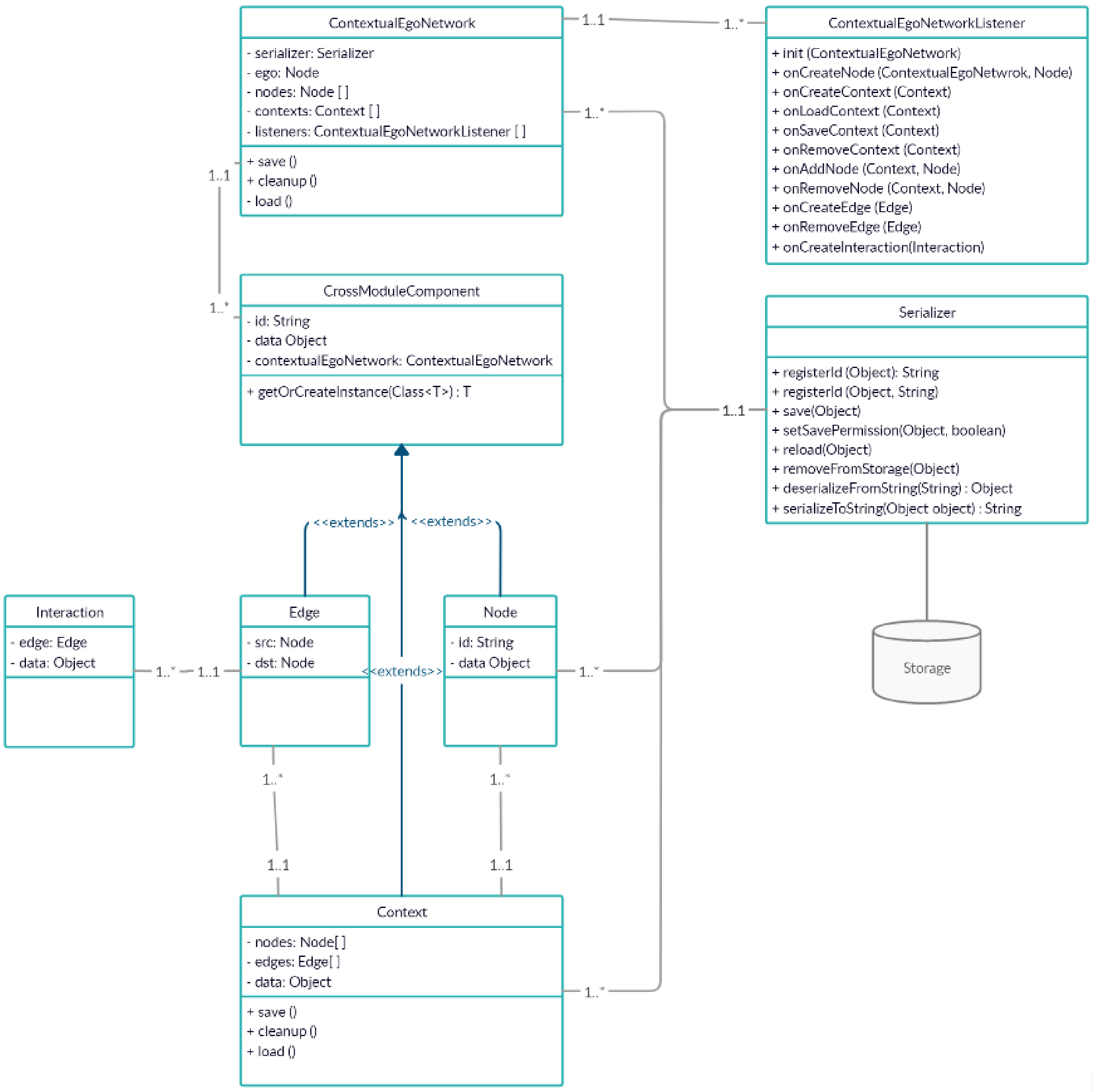

ContextualEgoNetwork
- serializer: Serializer
- ego: Node
- nodes: Node [ ]
- contexts: Context [ ]
- listeners: ContextualEgoNetworkListener [ ]
+ save ()
+ cleanup ()
- load ()
ContextualEgoNetworkListener
+ init (ContextualEgoNetwork)
+ onCreateNode (ContextualEgoNetwrok, Node)
+ onCreateContext (Context)
+ onLoadContext (Context)
+ onSaveContext (Context)
+ onRemoveContext (Context)
+ onAddNode (Context, Node)
+ onRemoveNode (Context, Node)
+ onCreateEdge (Edge)
+ onRemoveEdge (Edge)
+ onCreateInteraction(Interaction)
CrossModuleComponent
- id: String
- data Object
- contextualEgoNetwork: ContextualEgoNetwork
+ getOrCreateInstance(Class<T>) : T
Serializer
+ registerId (Object): String
+ registerId (Object, String)
+ save(Object)
+ setSavePermission(Object, boolean)
+ reload(Object)
+ removeFromStorage(Object)
+ deserializeFromString(String) : Object
+ serializeToString(Object object) : String
Storage
Interaction
- edge: Edge
- data: Object
Edge
- src: Node
- dst: Node
Node
- id: String
- data Object
Context
- nodes: Node[ ]
- edges: Edge[ ]
- data: Object
+ save ()
+ cleanup ()
+ load ()
<<extends>>
<<extends>>
<<extends>>
1..1
1..*
1..*
1..1
1..*
1..1
1..*
1..1
1..*
1..*
1..1
1..*
1..1
1..*

Figure 21. UML diagram of classes involved in dynamic storage and loading operations. Getter, construction and removal methods are omitted for clarity.

Moving to a reference-based API

The first version of the library implemented objects that provided identifier-based relational information. However, during the development of graph recommendation code, it was important to avoid logical bugs that would arise when querying for identifiers of wrong object types and to prevent calls; errors arising from these miscast types are very difficult to backtrace as the CENs of users grow in size and it becomes impossible for the developer to obtain a holistic understanding of all in-memory data structures. We pinpoint two ways to introduce identifier type checking: a) introducing constraints against all other stored identifiers and b) taking advantage of the typed nature of Java objects and introduce direct references between those. We elected to use the second approach, as the first one would require a lot of running overhead when running the code by making the library perform additional checks on the validity of identifiers.

An advantage of this organization is that it reduces the complexity of the code when many objects need to be accessed in sequential order. For example, in the new version of the library we can write *interaction.getEdge().getContext()* to obtain the context an interaction would occur in. Furthermore, our selected solution makes it easier to integrate future functionalities by implementing them as methods of their respective classes without potentially affecting the functionality of unrelated code entities.

The challenge that needs to be addressed when moving to this non-relational but reference-based logic of the CEN management library is the seamless integration of context loading without affecting the objects that reference them. Given that we also provide a system that does so, we also implement context cleanup to unload context data (i.e. nodes and edges) from the memory without destroying the context objects themselves. This way, context data can also be silently reloaded on-demand when its respective getter, creation or removal methods are called.

To support storage of reference-based memory serialization, we consider a pool of instantiated objects the serializer can reference, which we will call the *Reference Pool.* Objects in this pool are assigned unique local identifiers (UIDs) which are internally represented during serialization as an additional field *@id*. Objects of the reference pool are serialized to separate files whose storage path is determined by their UIDs and which can be loaded on-demand during subsequent runs of the application. The objects we put in this pool are the device's CEN, its nodes and its contexts. Other objects, such as edges and interactions within contexts, are saved as part of their encompassing structure. The serializer responsible for managing the pool of objects is tied to the specific CEN (i.e. it is different between different networks).

An important aspect of this structure is that any object can reference an object of the reference pool, as the latter can also be loaded in memory through its identifier. This does not hold true for objects outside the pool, which can only be referenced by ancestor objects when serialized and whose occurrences in parent objects are treated as different instances during deserialization, since at that point there is no other structure to yield previously instantiated objects instead of creating new ones.

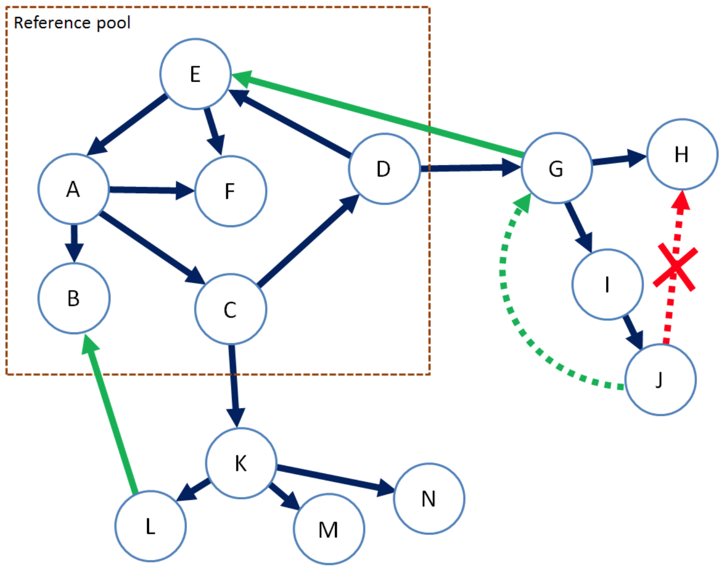


Figure 22. Theoretical organization of the serializer into a reference pool of objects that can be serialized into files. Arrows represent references towards other objects (i.e. that are declared as class field values).

References towards objects outside the reference pool (dashed arrows) are allowed only towards ancestor objects in the object hierarchy rooted on the pool, such as from J to G but not from J to H.

A serialization pool is typically not provided by serialization libraries, such as the one supporting the previous version of the CEN management library. This led us to provide our own serialization scheme. When doing so, we took care to avoid practices required by the previous serialization library that made mandatory the public exposition of functionality, such as direct setters to list primitives, that could induce catastrophic failure if used incorrectly (e.g. by removing previously stored information without notifying all node entities). Hence, our improved version of the library boasts not only ease but also correctness of use when integrated in other modules or applications of the HELIOS platform.

Furthermore, the data of new HELIOS modules, such as the graph recommendation module, can be potentially attached to CEN objects. Hence, our developed serialization scheme is made to be applicable on a wide range of source code object definitions whose instances can be attached to the CEN. An example of how the CEN data references are organized in-memory is shown in Figure 22. In that figure, green edges are those who link back to the reference pool and do not cause the loading of the respective objects unless the latter are explicitly requested. Furthermore, contexts are not fully loaded before their edges or nodes are explicitly requested for the first time.

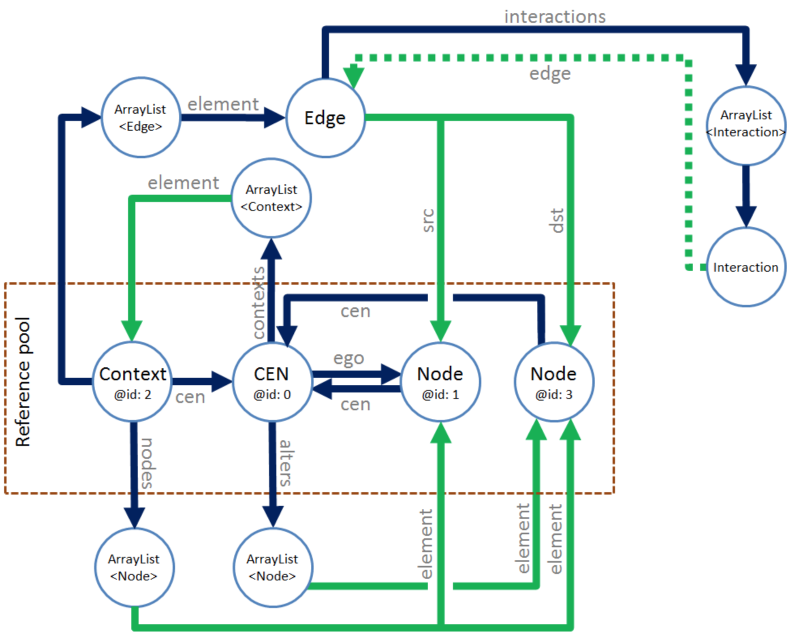
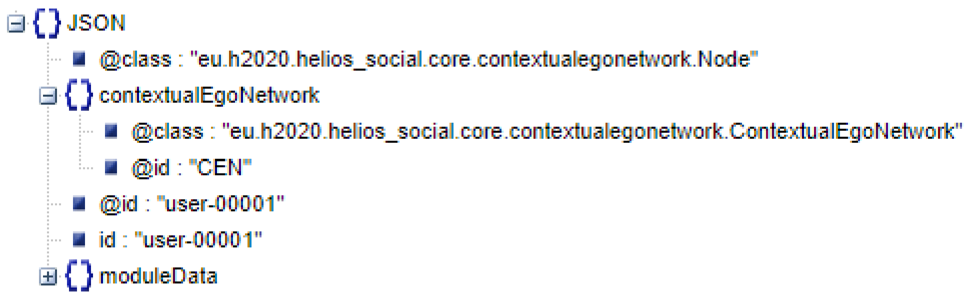


Figure 23. How the organization of Figure 22 translates to an example in-memory organization of CEN references.

When new entities are to be serialized, the serializer checks their encompassing objects for references to objects within the pool - if such references are found, the resulting fields of serialized classes are converted to a pair of serialization attributes:

{@class: PooledObjectClass, @id: UID}

as demonstrated in Figure 24. Besides dynamic serialization and deserialization, this structure also helps avoid circular object references (e.g. that object A references object B, which references C, which references A back) that would require high computational effort to identify.



```
JSON
    @class : "eu.h2020.helios_social.core.contextualegonetwork.Node"
    contextualEgoNetwork
        @class : "eu.h2020.helios_social.core.contextualegonetwork.ContextualEgoNetwork"
        @id : "CEN"
    @id : "user-00001"
    id : "user-00001"
    moduleData
```

Figure 24. The contents of a serialized CEN node.

It must be noted that objects not in the serialization pool are serialized to:

{@class: objectClassName, field1: field1, field2: value2, ...}

where both public and private fields are stored. We also provide the annotation *@Serializer.Serialization(enabled=false)* to place over fields the developers want to mark as not serializable - these are assigned *null* values during deserialization, unless the project entity's default constructor assigns another value. Including the object's class name *objectClassName* helps support the applicability of the serialization mechanism in polymorphic development environments, in which fields declared a base type but reference objects of any extending types.

Field values can either be new objects or primitives (e.g. strings, doubles, integers). Primitives values are either their string conversions (e.g. "10.0" for a double) if their instance is of the same type as the declared field type or:

{@class: primitiveClassName, value: primitiveStringValue}

when the primitive is stored to a generic *Object.* For example, storing double value of 10.0 in a class's field *Object data* would produce the following serialization within that class:

data: {@class: "java.lang.Double", value: "10.0"}

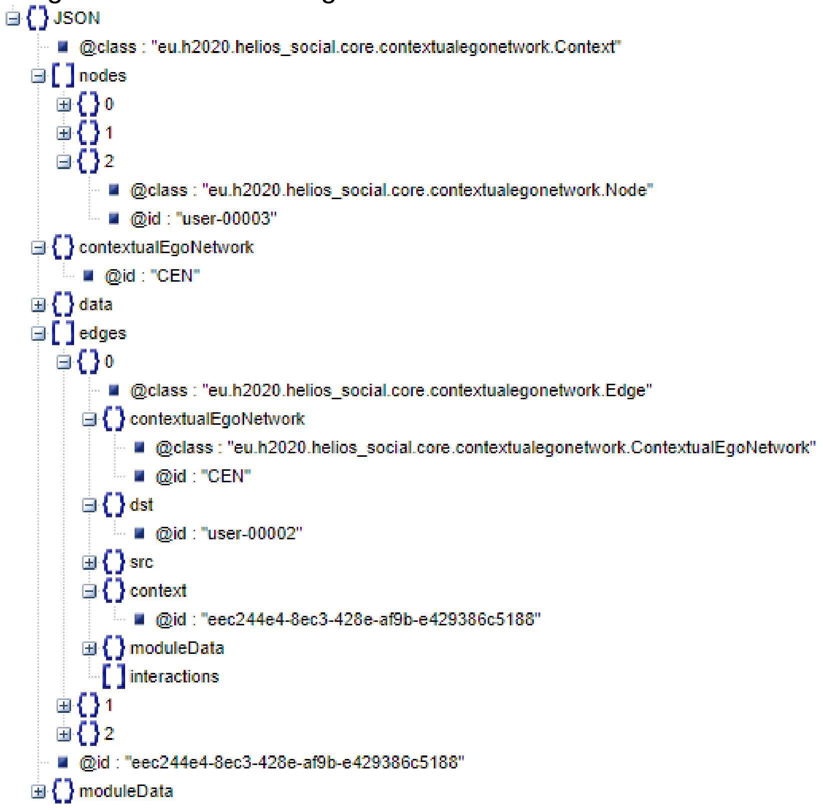
whereas a field *double doubleData* with the same value would be simply stored as:

doubleData: "10.0"

When deserializing objects with a serialization identifier, these are registered in the serializer's pool of objects. Otherwise, they are considered instances stored in their parent object's field. In the first case, the serialization identifier may not reference an object of the current serializer's pool. In this, a new object is created using the default constructor (i.e. the constructor with no arguments) of the *@class* attribute and its fields are read from the file corresponding to this identifier. At this point, we can see that storing the *@class* attribute is only needed for this first instantiation of each object. Since objects, such as nodes, can be referenced multiple times in a single file, we also adjusted the serialization process to remove that attribute when it is redundant.

On the other hand, objects with no UID are treated as single instances of a class that are not shared between different objects. This may introduce discrepancies between the behavior of deserialized objects, which are considered logic bugs. Unfortunately, checking for these is impossible in a system such as ours, that offers dynamic serialization and deserialization.

As an intermediate step of the serialization process, we use the JSON format[3] to organize the above hierarchical dependencies of classes and their fields that need to be stored in files and hence our implementation depends on the namesake Java library[4], which we used to manage our representations. This is the only external dependency of the CEN management library and, as a cross-platform solution, allows usage of the latter within Android applications. In Figure 25 we present a JSON representation of an example CEN serialization scheme. This process can be locally stored by being converted to a string.



```
{} JSON
    @class : "eu.h2020.helios_social.core.contextualegonetwork.Context"
  [] nodes
    {} 0
    {} 1
    {} 2
        @class : "eu.h2020.helios_social.core.contextualegonetwork.Node"
        @id : "user-00003"
  {} contextualEgoNetwork
      @id : "CEN"
  {} data
  [] edges
    {} 0
        @class : "eu.h2020.helios_social.core.contextualegonetwork.Edge"
      {} contextualEgoNetwork
          @class : "eu.h2020.helios_social.core.contextualegonetwork.ContextualEgoNetwork"
          @id : "CEN"
      {} dst
          @id : "user-00002"
      {} src
      {} context
          @id : "eec244e4-8ec3-428e-af9b-e429386c5188"
      {} moduleData
      [] interactions
    {} 1
    {} 2
    @id : "eec244e4-8ec3-428e-af9b-e429386c5188"
  {} moduleData
```

Figure 25. Saving storage space by saving only on *@class* attribute for each object in a file (e.g. the class of “user-0003” is found to be a HELIOS node when retrieving the list of nodes and its class needs not be retrieved again when it is assigned as the destination (dst) node of the context’s first edge

Enabling modular development

HELIOS modules and applications could need to attach information on CEN objects to be stored and loaded alongside them. For example GNNs often need to attach a local estimation of embeddings to the respective nodes of the library. Traditional development practices indicate that doing so requires either extending the classes of the CEN to accommodate additional data fields or implementing wrapping classes that reference the objects of the CEN. However, these practices complicate integration of multiple modules in the same project in that they need to explicitly depend on each other and incrementally extend each other’s classes; this would exponentially raise the cost of developing, maintaining or extending modules that depend on this library, since they all need to depend on each other.

[3] https://www.json.org
[4] https://mvnrepository.com/artifact/org.json/json

Given this concern, in this task we improve the context ego network management library by allowing its classes to accommodate instances other object types not yet known at the library's deployment time. In particular, we define an abstract class called *CrossModuleComponent* that facilitates storage of other modules' data structures and make all Nodes, Contexts and Edges of the CEN inherit its function. In particular, this class implements a method *CrossModuleComponent.getOrCreateInstance* which queries the object on whether an instance of the given class has been attached to it, which is returned. If no such instance is called, a publicly-accessible default (i.e. no-argument) constructor of the given class is used to create and attach a new instance first.

This implementation promotes usage of simple patterns for attaching information on the CEN. For example, a node may be queried to call a *getEmbeddings()* of a *GNNNodeData* class by:

```
node.getOrCreateInstance(GNNNodeData.class).getEmbeddings();
```

Failsafe recovery

The module provides a recovery mechanism that insures against premature application termination. In particular, it is important to ensure error-free running of the CEN management library when trying to load corrupted information, as for example happens when application crashes prevent reaching the *save* method of the CEN. A secondary but nonetheless important objective of this task is to also recover potentially unsaved information.

As a first step in these directions, we integrate an observer pattern within the CEN's organization [115]. This pattern allows listening for changes, such as creating new nodes or edges, and running the callbacks of specified objects when these occur. To this end, we provide a programming interface called *ContextualEgoNetworkListener* and allow objects that implement it to be registered in the CEN. Then, the objects pertaining to the latter retrieve all listeners and call the respective callbacks when the respective action occurs. For example the *onCreateEdge(Edge)* method of all listeners is called when a new edge is created in a context. For example, the previous functionality of assigning creation timestamps to objects is now implemented using a *listeners.CreationListener* that stores an object wrapping a single unix timestamp on created nodes, contexts and edges when these are created.

By taking advantage of the listener capabilities, we implement a data recovery mechanism that actively logs unsaved information in a separate file whenever a CEN action occurs. This log is cleared when the CEN is saved. Otherwise, if the application has been exited without saving, those actions are re-performed on its next run. We point out that we cannot continuously save the CEN or edited contexts, as these operations can prove computationally intensive for contexts that comprise many users, edges and interactions.

In particular, we implement the listener class *listener.RecoveryListener*, whose callbacks append the necessary information to a log file stored in the internal storage path. The pattern of adding recovery capabilities to a CEN is then as simple as adding the listener to it, as demonstrated in the example below:

```
ContextualEgoNetwork cen
   = ContextualEgoNetwork.createOrLoad("", "user-00001", null);
cen.addListener(new RecoveryListener());
```

Automated instantiation design patterns

As a final improvement to the CEN management library, we found out that during the development actions of this deliverable we frequently used patterns of the form:

```
if (object.hasValue(query))
  value = object.createValue(query);
else
  value = object.getValue(query);
```

These patterns were used to create contexts, alters or edges if they were not found in a designated object, such as the CEN or contexts. To enable ease of use to developers using the library, in this task we also provide ready implementations of those patterns that take the much simpler form:

```
value = object.getOrCreate(query);
```

6.2 Social Graph Recommendation Module

In this deliverable, we presented the outcomes of our research on novel decentralized graph mining algorithms that can mine the decentralized heterogeneous social network interactions between HELIOS users. In particular, we explored decentralized GNN architectures that aim to understand the evolution of user preferences over time and showed that these can help re-discover older interactions. To help utilize our proposed algorithms in social network applications developed in the HELIOS platform, we also implement them into a social graph recommendation module, an instance of which will run independently on each user's device.

An important aspect of this module is that it needs to exchange information with alter devices alongside real-world interactions, such as sent or received messages. In particular we foresee that a single cycle of parameter exchanges suffices to implement both the interaction mining algorithm previously proposed in this work, which depends on sending and receiving parameter estimations, as well as future algorithms of similar nature. If parameter exchanges occur only during the occurrence of new interactions, the communication overhead between devices needed to train mining algorithms is minimized: The communication protocol parameter exchanges follow whenever an interaction occurs and is visually demonstrated in Figure 26.

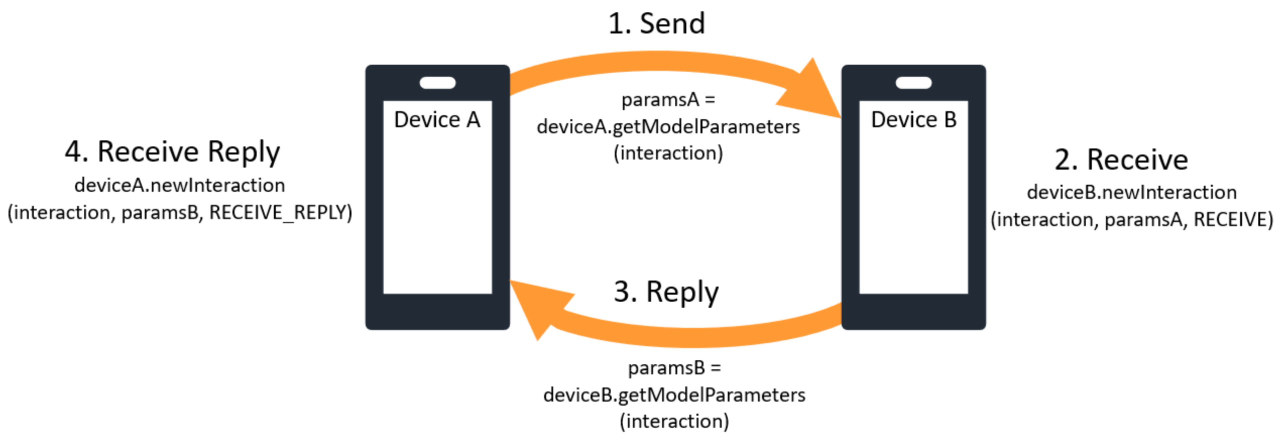

1. Send
Device A
paramsA = deviceA.getModelParameters (interaction)
Device B
2. Receive
deviceB.newInteraction (interaction, paramsA, RECEIVE)
3. Reply
paramsB = deviceB.getModelParameters (interaction)
4. Receive Reply
deviceA.newInteraction (interaction, paramsB, RECEIVE_REPLY)

Figure 26. Parameter exchange scheme between the graph recommendation module of HELIOS devices; device A sends mined parameters to device B and the latter replies with its own set of parameters.

At the time of this module's beta release, the burden of orchestrating the above operations, including a notification of the graph recommendation module that new interactions have occurred, falls on the one using the application.

The social graph recommendation module aims to learn user preferences, for example by representing them in latent low dimensional spaces, from the interactions it becomes aware of. To do so, it utilizes the improved CEN library to represent the relations and attached learned representations on nodes. It can then retrieve these to recommend new interactions to users.

Native implementation of matrix operations

The architectures designed in this deliverable do not exhibit high computational complexity, since their federated learning principles effectively allocate very few computations to each device.

Therefore, we deem that usage of additional libraries specifically tailored to performance computing, such as Tensorflow Lite[5], could only serve to complicate integration and maintenance of the graph recommendation module without providing noticeable boosts in performance. Introducing such systems can even produce adverse effects, such as increasing battery consumption through spikes of intense on-device operations that are too few to require explicit optimizations.

On the other hand, existing libraries of matrix (and vector) arithmetics, such as JScience, fail to introduce rigorous saliency checks at all stages of arithmetic operations. These are of particular importance during our research actions, as they facilitate early catching of logical bugs that arise during development of the machine learning models described in this work. For example, an issue that frequently occurs during development of GNNs is that, when small regularization is applied, gradients and parameters can explode into numerically large values through implicit feedback loops residing within the learned graph structures. These large values can then be exponentiated by neural activation functions (e.g. to calculate the derivative of a cross entropy loss) and this leads to an overflow that is converted to NaN arithmetics. These kinds of errors are hard to spot in the first place, unless disproportionally positive evaluations are spotted - due to NaN comparisons always yielding true. Even worse though, without catching them at the first time they occur, it is nigh impossible to identify the specific section of the code that first introduced them. To this end, base matrix operations should be equipped with error checking capabilities that directly pertain to such errors.

An equally important reason which led us away from building the graph recommendation module atop of existing arithmetic libraries is that these make running simulated experiments over existing networks too slow, since each simulated device is delegated its own copy of a CEN management and graph recommendation modules. For example, loading and unloading Tensorflow programs from the computer's graphics card (or worse, through a wrapper that runs on the CPU) at each simulated interaction introduces a significant overhead, whereas the general-purpose nature of matrix arithmetics means that they often neglect in-memory arithmetic operations that place back their outcome to one of their vectors, which ends up reducing memory allocations more than a five-fold.

Taking the above requirements into account, we started from implementing native matrix operations in the social graph miner's package *GNN.operations*. This package provides access both to matrix arithmetics (e.g. vertex operations, loss functions and derivatives) that are used in developing the GNNs of this deliverable.

[5] https://www.tensorflow.org/lite

Mining graph interactions

Before developing specific graph mining algorithms, we provide an abstraction that allows a uniform modeling scheme of interaction mining. In particular, we provide a programming abstract class called *SocialGraphMiner* that uses the source code entities of the CEN management library to define the signatures of the operations outlined in Figure 26:

```
public enum InteractionType {SEND, RECEIVE, RECEIVE_REPLY};

public void newInteraction(Interaction interaction,
     String neighborModelParameters, InteractionType interactionType);
public String getModelParameters(Interaction interaction);
public HashMap<Node, Double> recommendInteractions(Context context);
```

To distinguish between different steps of that scheme, we set up an *InteractionType* enumeration that is passed as a parameter at the *newInteraction* met. At any point, the *recommendInteractions* method can be used to obtain mined interaction recommendations for the device's user.

GNNMiner

In the beta release, we provide a *GNNMiner* extension of the basic *SocialGraphMiner* class that implements the proposed decentralized temporal GNN mining algorithm developed in Section 5. This algorithm is parameterized to enable potential adjustments of its hyper-parameters so that they can be tuned by user feedback to better match the characteristics of the social media applications developed on the HELIOS platform (see subsection 7.3). In later releases, we hope to present a construction factory pattern that enables the usage of different types of GNN mining architectures. As of the beta release, the miner aims to share the same received neighbor embedding estimations across all contexts but perform different training depending on a given (e.g. currently active) context of the CEN in which the last interaction has occurred. The efficacy of this type of mining across different contexts will be investigated in future work.

The developed miner is supported by three kinds of data structures; a *GNNNodeData* class that holds the currently calculated and received embedding tensors, a *ContextTrainingExampleData* class that holds a context-specific list of positive and negative training examples and a *TrainingExample* class that corresponds to the temporal (u,v,w,l) training example tuple described in subsection 5.2. Developers using the mining module need not necessarily be aware of these data structures, but they are made publicly visible to be attachable on the nodes and contexts of the CEN.

Given that the above structures are retrieved for all nodes and the current context of the CEN, the GNNMiner instance running on each user's device is trained towards the objective presented in subsection 5.4 by making use of our native implementation of matrix operations.

A simplified data organization scheme of the GNNMiner is presented as a UML diagram of Figure 27. For a more detailed description of this implementation's capabilities, refer to Annex II.

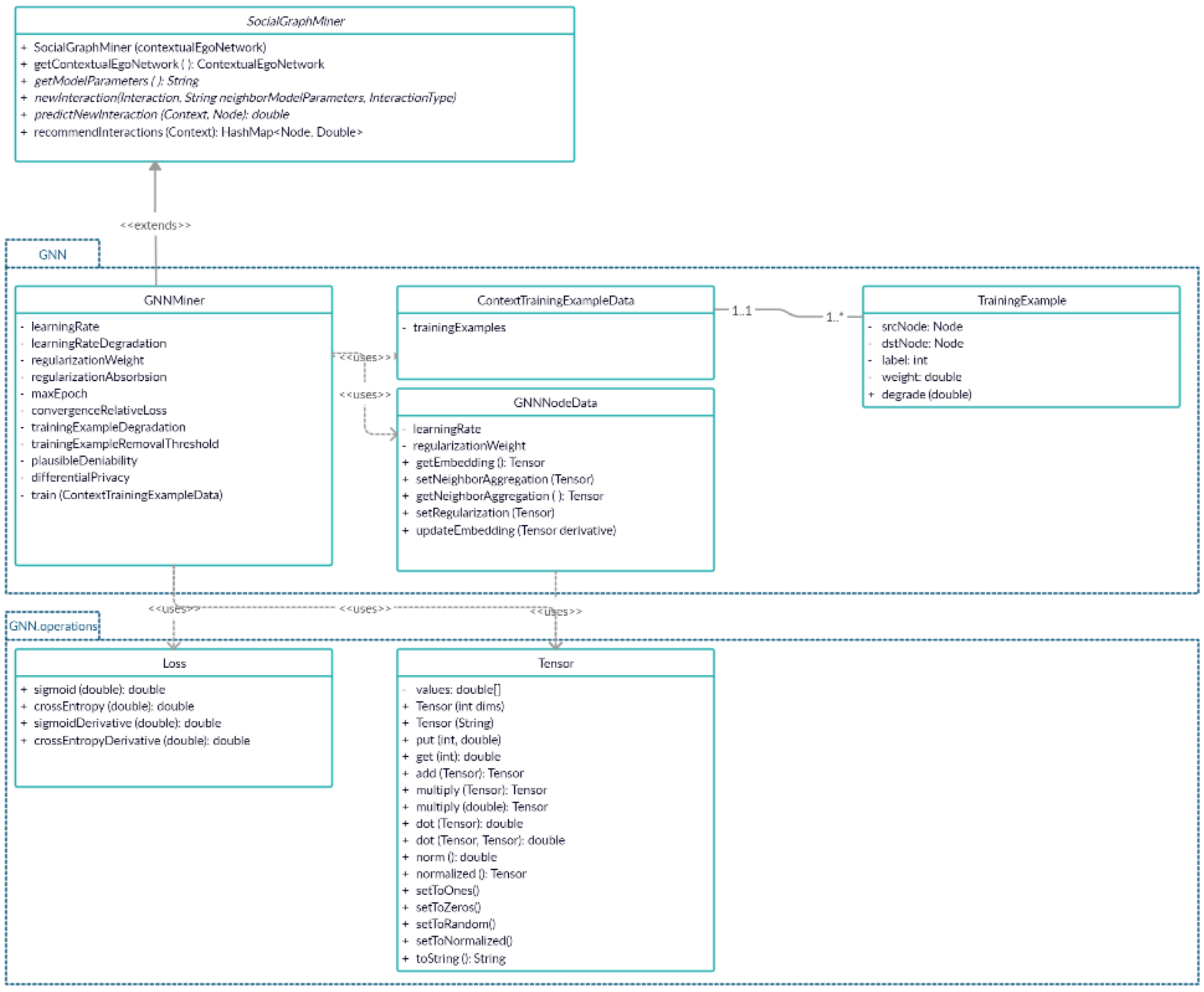
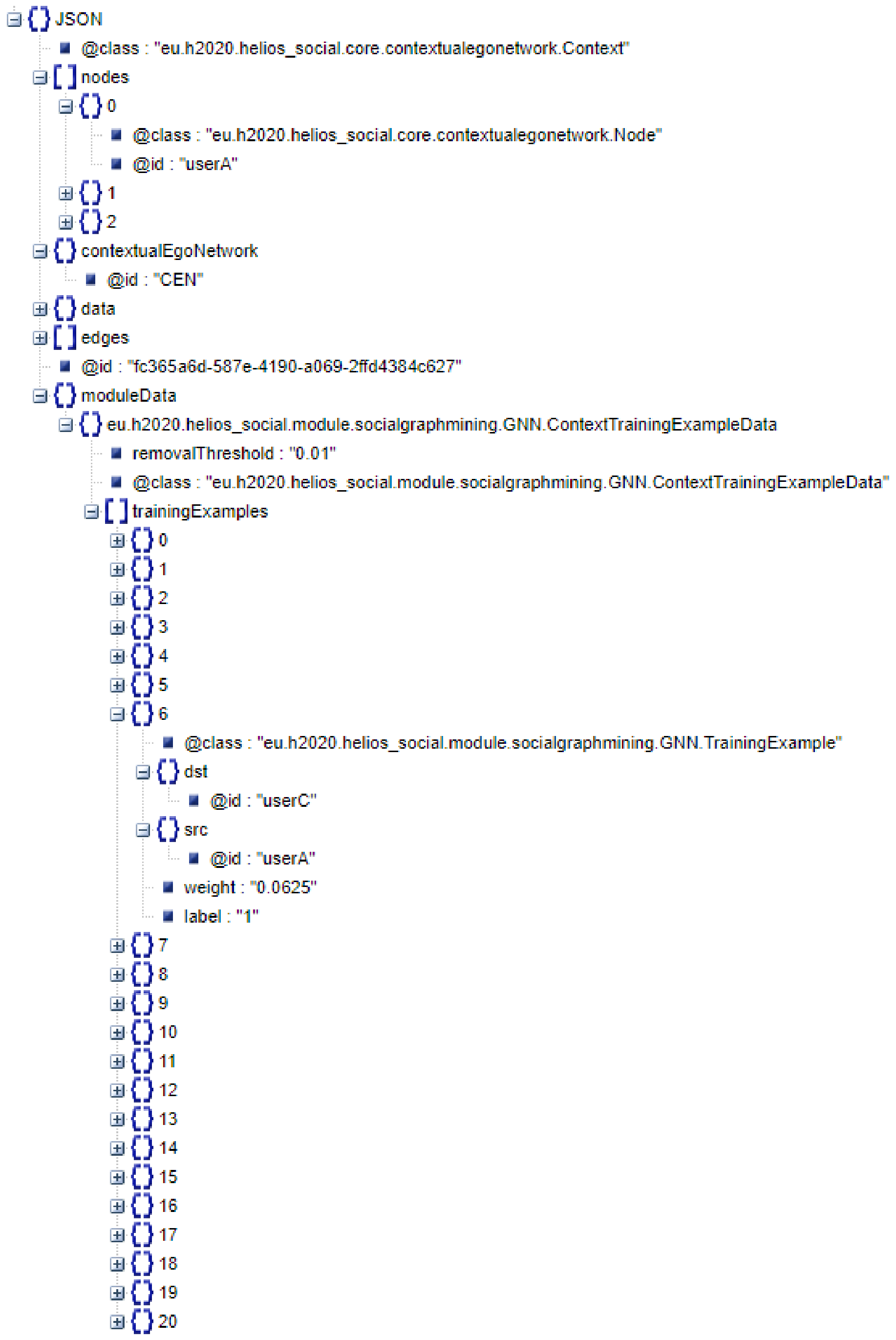

SocialGraphMiner
+ SocialGraphMiner (contextualEgoNetwork)
+ getContextualEgoNetwork ( ): ContextualEgoNetwork
+ getModelParameters ( ): String
+ newInteraction(Interaction, String neighborModelParameters, InteractionType)
+ predictNewInteraction (Context, Node): double
+ recommendInteractions (Context): HashMap<Node, Double>
<<extends>>
GNN
GNNMiner
- learningRate
- learningRateDegradation
- regularizationWeight
- regularizationAbsorbsion
- maxEpoch
- convergenceRelativeLoss
- trainingExampleDegradation
- trainingExampleRemovalThreshold
- plausibleDeniability
- differentialPrivacy
- train (ContextTrainingExampleData)
<<uses>>
<<uses>>
ContextTrainingExampleData
- trainingExamples
1..1
1..*
TrainingExample
- srcNode: Node
- dstNode: Node
- label: int
- weight: double
+ degrade (double)
GNNNodeData
- learningRate
- regularizationWeight
+ getEmbedding (): Tensor
+ setNeighborAggregation (Tensor)
+ getNeighborAggregation ( ): Tensor
+ setRegularization (Tensor)
+ updateEmbedding (Tensor derivative)
<<uses>>
<<uses>>
<<uses>>
GNN.operations
Loss
+ sigmoid (double): double
+ crossEntropy (double): double
+ sigmoidDerivative (double): double
+ crossEntropyDerivative (double): double
Tensor
- values: double[]
+ Tensor (int dims)
+ Tensor (String)
+ put (int, double)
+ get (int): double
+ add (Tensor): Tensor
+ multiply (Tensor): Tensor
+ multiply (double): Tensor
+ dot (Tensor): double
+ dot (Tensor, Tensor): double
+ norm (): double
+ normalized (): Tensor
+ setToOnes()
+ setToZeros()
+ setToRandom()
+ setToNormalized()
+ toString (): String

Figure 27. The most important data structures and operations of the GNN mining module pertaining to the GNNMiner class. The GNN.operations package is not needed by the developers using the module.



```
{} JSON
   @class : "eu.h2020.helios_social.core.contextualegonetwork.Context"
   [] nodes
      {} 0
         @class : "eu.h2020.helios_social.core.contextualegonetwork.Node"
         @id : "userA"
      {} 1
      {} 2
   {} contextualEgoNetwork
      @id : "CEN"
   {} data
   [] edges
   @id : "fc365a6d-587e-4190-a069-2ffd4384c627"
   {} moduleData
      {} eu.h2020.helios_social.module.socialgraphmining.GNN.ContextTrainingExampleData
         removalThreshold : "0.01"
         @class : "eu.h2020.helios_social.module.socialgraphmining.GNN.ContextTrainingExampleData"
         [] trainingExamples
            {} 0
            {} 1
            {} 2
            {} 3
            {} 4
            {} 5
            {} 6
               @class : "eu.h2020.helios_social.module.socialgraphmining.GNN.TrainingExample"
               {} dst
                  @id : "userC"
               {} src
                  @id : "userA"
               weight : "0.0625"
               label : "1"
            {} 7
            {} 8
            {} 9
            {} 10
            {} 11
            {} 12
            {} 13
            {} 14
            {} 15
            {} 16
            {} 17
            {} 18
            {} 19
            {} 20
```

Figure 28. Example outcome of delegating the graph recommendation module's data to be stored in the CEN management library.

6.3 Peer-to-Peer Simulation

Asserting the correctness of the social graph recommendation module

In Section 5 we proposed a novel temporal GNN architecture that mines user preferences through their interactions in decentralized settings. Following that, in this section we outline the development of a social graph recommendation module that implements the devised recommendation algorithm within the HELIOS platform. Given that this module will be provided for direct use by HELIOS applications, we need to demonstrate that it can be integrated in a true peer-to-peer environment. To this end, we experimented on an open source peer-to-peer simulation engine written in Java, called PeerSim [116], in which we simulated the social network datasets detailed in subsection 5.1. Our choice of the PeerSim engine was motivated by its ability to simulate large networks, with up to few millions of nodes, with customizable network delays and node churn. Furthermore, it allows a modular implementation of communication protocols so as to potentially simulate multiple HELIOS modules that rely on peer-to-peer communication.

Using PeerSim, we simulated user devices, each of which runs different instances of the CEN library and the social recommendation mining module and PeerSim communication protocol and simulated their interactions as messages using the network dataset. In this setting, we verified that running the recommendation module on the simulated devices indeed provides the same recommendations as in our experiments. In the next period, the organized simulation will become available as a testing platform on which to test different variations and hyper-parameters of social graph mining algorithms.

Demonstrating the social graph recommendation mining

Since PeerSim simulates a true peer-to-peer environment, in this subsection we focus on understanding how our module can be integrated in applications that aim to be deployed in such environments.

PeerSim supports two kinds of simulation models:

a) Cycle-driven simulation
In this simulation model, each node is scheduled regularly and each time it triggers the execution of the stack of protocols of that node. The number of times nodes should be scheduled is fixed and can be specified in a configuration file. In this simulation model there is no explicit exchange of messages, and communication is simulated by the execution of methods of the running protocols.

b) Event-driven simulation
In this simulation model, protocols communicate using messages, making it more realistic. The communication is modeled through send and receive primitives. In this simulation model one can also customize how the transport network is implemented to introduce network delays and message drops. Protocols in a node are triggered by sending a message to them and there is not a maximum nor a minimum amount of times a protocol can be executed.

For social network simulations we adopted a hybrid simulation model, where part of the communication protocol is executed at regular time intervals, and part of it is executed asynchronously upon interaction. In the current implementation of the social graph recommendation module, periodic execution is not needed, but it helps provide an abstraction for additional development of machine learning models that require regular updates of their parameters, for example to degrade the example weights over time instead of over interactions.

The interactions that occur between nodes of the social datasets are modeled as messages that activate each node’s communication protocols at the corresponding times. When an interaction triggers the protocol instance of a node, this initiates a CEN synchronization phase and a learning model update. In the first phase, the currently active context of the CEN of the node performing the interaction is updated and the graph mining module runs over the new interaction. At the end of this process, that module provides parameters to be exchanged and these are sent alongside the interaction to all its alters (including the interaction's recipient) that also comprise the interaction's recipient to be included in their ego networks. The interaction's recipient also performs training and sends back the learned parameters to the ego network. Experiments throughout this deliverable do not take into account ego network updates between alters that occur this way, but we provide this more generalized simulation setting for future research to investigate whether these can improve the efficacy of the GNN algorithm.

7 Practical Application

7.1 Graph Mining in the HELIOS Platform

HELIOS is a decentralized peer-to-peer social media platform that offers a number of functionalities on top of which developers can build their own social media applications. Its general architecture spans three different layers, as outlined in Deliverable 3.2:

Core Modules. This layer comprises the main components of the HELIOS platform, namely the peer-to-peer network exposed through the communication manager, the security and privacy manager, the context manager, the trust manager, the personal data storage manager and the profile manager. Finally, the CEN manager that was extended in this deliverable is also a core module.

Extension Modules. This layer extends the functionality of core modules to provide on-device data mining. These augment how users experience the decentralized platform by providing recommendations, rewarding opportunities and more functionalities pertaining to the analysis of social graphs. The social graph recommendation module presented in this deliverable, as well as the community detection module (described in Section 4) that will arise after the implementation of community detection algorithms, are extension modules.

HELIOS Applications (HApps). This layer comprises applications built on top of the HELIOS platform; after the planned beta release, core modules and their documentation will be made publicly available to enable the development of new social network applications.

7.2 Relation to Use Cases

To understand how graph mining can be used within real-world applications, in this subsection we propose potential adoptions of the algorithms and protocols developed in this task that explain how they can be used within HELIOS's use cases. In particular, three different use cases have been defined for the project, each of which is going to be translated to an application by their respective task based on key functionalities provided by HELIOS's core and extension modules. These include the functionalities developed in the scope of this deliverable.

Use Case A: HELIOS Connecting People. Task 5.3 is the leader of this Use Case and a preliminary version of the respective application, called Helios. TALK is already available. At its current state, this application focuses on group communications and offers the opportunity to the users to engage with new people in different contexts, organize communities based on common interests and connect with other people with similar interests. This task depends on the communication and social graph recommendation modules to not only provide friend recommendations, as existing social networks often do, but also recommendations of whom to interact with. The produced application records all user interactions through the CEN manager and forwards them to the social graph recommendation module, which provides recommendations to be displayed to the user. In that respect, Task 5.3 works closely with WP2 to define an appealing User Interface (UI) for displaying and help evaluate the appeal of such recommendations.

Use Case B: HELIOS Cultural Hub. Task 5.1 is the leader of this use case and its main goal is to augment cultural experiences so they become shared spaces of communication and open knowledge. This relies on ad-hoc and organic creation of the social graph to provide matchmaking opportunities to the users. Even though this use case does not explicitly require the social graph recommendation module for matchmaking, future releases of the respective application could leverage the contextual ego management module to record interactions between HELIOS users in the cultural places and the social graph recommendation module to provide additional friend recommendations. Furthermore, user interactions can support the definition of relations between users of similar interests. In turn, these can form communities whose structure can be exploited to spread interesting information among its components.

Use Case C: HELIOS Citizen Journalism. Task 5.4 is the leader of this use case and its main goal is to create an application that will allow citizen journalists as well as professional journalists to contribute videos and pictures anonymously. Through this application, it will be possible to publish data into a private relationship environment, or to provide content for people with similar interests. Furthermore, publishers have also access to the content via a separate video exchange platform in order to use the distributed content for their own media channels. A first (beta) version of the Citizen Journalist application is already available. In future releases, this application will integrate the media streaming module and the payments module to distribute content (such as video-sequences) and to reward citizen journalists for their contribution in diverse novel reward models. It will also interact with the HELIOS core components to take advantage of the possibilities of a social ego Network or the communication manager. The graph recommendation module could also be integrated in this application to understand social dynamics and user preferences. For example, it could identify communities of users characterized by common interests among those participating in an event, and suggest relations between them, so as to later spread information about those interests.

7.3 User Feedback on the Social Graph Recommendation Module

The social graph recommendation module, as provided in the beta release of HELIOS, aims to support decentralized systems that can recommend social actions to their users. In this task, the quality of such systems is asserted through their ability to satisfy theoretically grounded objectives, such as exhibiting high DHR values when recommending interactions. However, even when recommendations are accurate, they may not necessarily appeal to the users. For example, users may not know whether to accept or reject the recommendations or may find them irrelevant or of little value. Thus, it is critical that recommendation systems, such as the ones provided by the social graph recommendation module, are evaluated in real-world settings [117].

With this consideration in mind, we look to user research studies to help understand the user-perceived quality of the algorithms designed in this task. These studies are an essential part of the HELIOS design and development processes, since they help understand whether modules under development are relevant for the users and cover the intended needs, as well as pointing to potential improvements. Hence, we will employ user studies to gather feedback that can improve base recommendation algorithms, find aspects of user behavior which have not been modelled due to being highly specific (e.g. introducing interaction periodicity, such as repeating an interaction each Monday morning). They could also help obtain local feedback to guide the learning process, such as by tuning the hyper-parameters of the decentralized GNN algorithms running on each user's device, such as the temporal degradation rate of training example weights.

Gathering user feedback is part of the validation activities of the WP7, and its methodology has been defined in Deliverable 7.1. Three types of validation activities have been set to test the technologies under development in HELIOS and provide feedback to their developers; lab tests, trials and pilots. The validation of the social graph recommendation module developed in this task is foreseen to be assessed over two main scenarios that will be tested in trials and/or pilots: a) recommending interactions with already existing contacts and b) potential recommendation of new contacts.

The recommendation module in both scenarios will be fully automated and users will not be able to provide any manual input to modify the behaviour and performance of the system. These functionalities will not be available as they require a certain degree of technical expertise. For this reason, the user's attention regarding interaction and feedback with the recommendation module will be oriented to the user interface and the presentation of the information provided by the system. In order to obtain structured feedback, validation will be two-pronged, measuring relevance on a first stage and satisfaction on a second stage. WP7 is expected to gather specific feedback from users, that will be used by Task 4.3 to assess whether the predicted recommendations provided by the system are relevant or not. More specifically, the aim of involving end-users is to validate the relevance and satisfaction level of the recommendations provided by the HELIOS app to a) foster interaction (i.e. messaging, exchange of photos or other files) between existing contacts within an ego-network and b) extend their social network with new contacts.

By definition and to ensure an optimal quality of recommendations, the recommendation module requires several users to interact among them in a certain period of time in order to perform meaningful recommendations. Thus, to evaluate the recommendation module it is necessary to consider a validation scenario with minimum requirements already described in Deliverable 7.1. (i.e. lasting enough and with a relevant sample of users involved). For this reason, the evaluation of the recommendation module will be integrated during the Trials and Pilots (WP7) as an additional validation objective (jointly with other HELIOS services and technologies). These validation activities will allow users to rate the relevance and satisfaction levels of the recommendations in the foreseen scenarios.

In a first stage, relevance will be measured according to the performance of the recommendation module. Recommendations will be displayed to users through interfaces, such as the one being developed in T5.3 and users will only be able to accept or reject the recommendation. If the user accepts a recommendation it will be considered as relevant, on the contrary if the user rejects a recommendation it will be considered as non-relevant. This implicit data gathering will provide valuable quantitative data that could potentially be used as a feedback to improve the quality of the recommendations.

Satisfaction levels will be measured in a second stage and related to Task 5.3, where a user interface will be designed and validated. Task 2.3 will cooperate with Task 5.3 in order to align the design for a satisfactory user experience when the recommendation system is involved. The HELIOS recommendation module will provide some basic information or explanation of the recommendation to help users make a better decision. According to Gedikli et al. [118], explanations of recommendations help users make better decisions in contrast to recommendations without explanations, for example by increasing the transparency between the system and the user and also to increase user satisfaction. It is important to assist users in understanding why these items are presented to them. To date, many researchers have demonstrated that providing good explanations for recommendations could help inspire users' trust and satisfaction, increase users' involvement and educate users on the internal logic of the system [41,119,120].

Depending on the UI of an application, satisfaction levels with the recommendation module could be added through Likert scale statements in a 10-point rating scale, where 0 refers to strong disagreement, 10 to strong agreement and intermediate values to milder sentiments. Prospective questions pertaining to the integration of the graph recommendation module and which can be rated this way are demonstrated in Table 7.

Further qualitative feedback regarding the recommendation module might be gathered through post-test questionnaires (i.e. semi-structured interviews and focus groups) at the end of the trials and pilots described in Deliverable 7.1. Such qualitative information might be relevant to better understand what is the user's opinion of the recommendation system and to identify what they like and dislike or are expecting from the module, as well as potential aspects of the user's behavior that have not been modeled.

Table 7. Prospective questions of user feedback

Question	Rating
I am satisfied with the recommendation provided by the system	0-10
The information provided for the recommended contact is sufficient for me	0-10
It is easy for me to inform the system if I dislike/like/postpone the recommended contact	0-10
The interface helped me understand why the contact was recommended to me	0-10
The layout of the recommender interface is attractive and adequate	0-10

8 Conclusions and Future Work

8.1 Conclusions

In this deliverable, we explored graph mining algorithms that can help developers of HELIOS applications integrate community discovery and social action recommendation capabilities.

To do this, we began by researching promising state-of-the-art graph mining algorithms that can account for the temporal evolution of social networks and selected those that best suit the aforementioned graph mining tasks, namely community detection for graph structure analysis and graph neural networks for mining social actions. These algorithms rely on different forms of computation centralization, which make them inapplicable to the decentralized setting of HELIOS. To tackle this problem, we proposed novel adaptations and communication protocols that allow their adoption in the more challenging decentralized setting of HELIOS. When underlying profiling mechanisms are employed during this process, we take care to also introduce privacy-friendly practices that provide plausible deniability and differential privacy of user data.

From a practical standpoint, we provided a decentralized community detection protocol and a first implementation of a graph recommendation module that performs interaction recommendation in a fully decentralized manner. The dynamic community detection and management protocol we propose detects contextual local communities which can be used as support for the other modules in decision making, such as information diffusion strategies, recommendations, context categorization, and so on. Whereas the graph recommendation module extracts latent user preferences and leverages those to understand whom users want to interact with among older acquaintances. We also extended the CEN management library to better accommodate modular attachment of external data structures, such as those used to store the learned parameters of the social graph recommendation module.

Experiments on the graph recommendation module show that the top three recommendations provided by the algorithms of its beta release correctly comprise re-enacted older interactions more than 35% of the time. Hence, they are of similar efficacy to the centralized GNN mining architectures we found to best capture user preferences.

Finally, we pointed out the practical usage of our research within HELIOS's use cases and other future applications, as well as what kind of mechanisms these need to employ to gather user feedback over the quality of social graph mining algorithms.

8.2 Future Work

In addition to the work of this deliverable, there exist promising research and development directions that have yet to be fully explored but which we plan to address in the next period.

First, the implementation of the decentralized dynamic community discovery protocol is not yet finalized. It is currently undergoing an internal testing phase using peer-to-peer simulators, with all the features implemented and individually tested. Once the testing is finished, the protocol will be integrated in the platform and the results of the mining process of the module will be made available for the other modules. Moreover, we expect to receive feedback from the modules using this community detection module, and possibly update the protocol and the algorithm we initially developed for the module. We will investigate possible optimizations in the protocol such that computation, storage, and communication resources are saved whenever possible. Also the algorithm we presented may be subject to further optimizations and also consider various evolutions, such as more advanced community definitions, and algorithms to detect the defined structure.

Furthermore, the mined understanding of user preferences extracted through our proposed decentralized GNN algorithm can be used to perform relation recommendations. A prerequisite in doing so is that the node discovery mechanism of the communication module needs to be explicitly defined and integrated in the HELIOS core before we explore protocols and mining methods built on top of it that could help bring user attention to others who (by the definition of relations) are not their graph neighbors. In theory, the preferences extracted from user interactions could also be used for recommending relations. However, relations may be driven by non-captured preferences spanning larger windows of time. To enable experimentation with alternatives that can capture other types of preferences we have not yet considered, we also plan to provide a more general Java factory pattern to help construct different GNN architectures in future releases of the graph recommendation module. This can be used alongside the peer-to-peer simulation to identify promising alternatives to existing mining algorithms.

Given that we provide a general graph recommendation framework, we will also continue to research potential improvements to the involved mining algorithms and protocols. For example, we have already performed a preliminary investigation over signal processing practices that can improve the quality of recommendations of ranking algorithms based on information diffusion [121] and we could translate those to GNNs. Or we could move the recommendation algorithms to super-peers that manage communities and perform the mining on their managed communities, hence taking advantage of many more potential recommendations. An additional practical improvement is to further mitigate parameter exchanges of the recommendation parameter exchange protocol. That protocol has already reduced the communication overhead to exchanging parameters only when new interactions occur, but it could also potentially perform the exchanges only once every few interactions, when parameters have significantly changed from their previous values.

As a final remark, we recognize that it is of interest to explain the reasoning behind the mining outcome to users. However, the preferences mined with the social graph recommendation module cannot be directly explained in human terms, for example because they partially capture a machine-driven understanding of other users' actions. To solve this problem, we will work on identifying the preferences that contribute the most to recommendations and identify how much alters pertain to those. As an intermediate step that will be utilized by this mechanism, we have already researched unsupervised evaluation of graph diffusion algorithms [122] with which we aim to select the best diffusion algorithms for finding the alters most related to the latent preferences.

Annex I – CEN Management Library API

Details of the core module *eu.h2020.helios_social.core.contextualegonetwork*, which implements the improved version of the CEN management library.

Table 8. Package *eu.h2020.helios_social.core.contextualegonetwork* entities.

Class or Interface	Description
Context	This class implements a context of the Contextual Ego Network.
ContextualEgoNetwork	This class implements a Contextual Ego Network, which is the conceptual model of our Heterogeneous Social Graph.
ContextualEgoNetworkListener	An interface of listeners that can be added on a ContextualEgoNetwork to listen to structural changes.
CrossModuleComponent	This is a base class used by HELIOS components, such as Node and Edge that need to store data coming from multiple modules.
Edge	This class implements an edge of the social graph.
Interaction	This class implements a generic interaction between two entities in a context.
Node	This class implements a node in the social graph.
Serializer	This class supports dynamic object serialization, with the capability of reloading only parts of objects and saving only particular objects.
Utils	This class implements static error parsing and logging methods, with the ability to suppress errors that can be silently handled during deployment.
ContextualEgoNetworkListener implementations	
listeners.ActivityListener	This class implements a ContextualEgoNetworkListener that counts the number of ego and alter interactions over the course of the week and time of day.
listeners.AsyncRunListener	This class implements a wrapper for ContextualEgoNetworkListener instances that performs callbacks asynchronously to the main thread.
listeners.CreationListener	This class implements a ContextualEgoNetworkListener that assigns node, context and edge timestamps.
listeners.RecoveryListener	This class implements a ContextualEgoNetworkListener that automatically safeguards the contextual ego network from failing to call save before terminating the application.

Table 9. *ContextualEgoNetwork* class methods.

Method	Description
addListener	Attaches a ContextualEgoNetworkListener to the contextual ego network, which will be called on the respective events.
cleanup	Applies Context.cleanup on all contexts.
createOrLoad	Instantiates a ContextualEgoNetwork at the given storage path by creating a new ego node with the given data.
getAlters	Retrieves the alters (not including the ego).
getContextBySerializationId	Searches all ContextualEgoNetwork contexts for one with the same serializationId assigned to it during serialization
getContexts	Method to grant safe access to all contexts of the contextual ego networks
getCurrentContext	Method to return the current context.
getEgo	Retrieves the ego node.
getListeners	Obtain all listeners attached to the contextual ego network
getOrCreateContext	Returns a context that satisfies data.equals(context.getData()) .
getOrCreateNode	Searches for a node with the given id and, if no such node is found, creates a new one using the given data.
getPath	Retrieves the path folder in which the ego network is saved by its serializer.
getSerializer	Retrieves the Serializer responsible for saving and loading the ego network and its entities.
removeContext	Removes a given context from the ContextualEgoNetwork's contexts.
removeNodeIfExists	Removes a node from the contextual ego network given its serialization id, which is the same as Node.getId.
save	Makes the getSerializer save the contextual ego network.
setCurrent	Method to set a context as the current context.

Table 10. *CrossModuleComponent* abstract class methods.

Method	Description
assertSameContextualEgoNetwork	Checks that this component belongs to the given contextual ego network.
assertSameContextualEgoNetwork	Checks that this component belongs to the same ego network as the compared component
getContextualEgoNetwork	Retrieves the contextual ego network instance in whose hierarchy the component resides
getOrCreateInstance	Retrieves a given class's instance stored in the component.

Table 11. *Context* class methods (extends *CrossModuleComponent*).

Method	Description
addEdge	Creates an edge between two nodes of the social graph.
addNode	Adds a new node to the context.
addNodeIfNecessary	Adds a node to the context if it's not already part of it.
cleanup	Saves the context to a file and removes its data memory and it from the dynamic serializer (so that universal save does not save it anymore)
getData	Retrieves a unique data object stored in the context.
getEdge	Retrieves the edge (if it exists) between two nodes in the context.
getEdges	Retrieves a shallow copy of the context's edge list.
getInEdges	Retrieves the incoming edges of a given node.
getNodes	Retrieves a shallow copy of the context's node list.
getOrAddEdge	Retrieves the edge (if it exists) between two nodes of the social graph or creates it if it doesn't exist.
getOutEdges	Gets the outgoing edges of a given node
getSerializationId	Retrieves the id assigned to this context during serialization.
isLoaded	Checks whether the context has been loaded in memory.
load	Loads the context from the given serializer in memory.
removeEdge	Removes an edge between a source and a destination node.
removeNode	Removes a node and its edges from the context

removeNodeIfExists	Removes a node and its edges from the context if its part of it
save	If the context is loaded, it is serialized to a file.

Table 12. *Node* class methods (extends *CrossModuleComponent*).

Method	**Description**
getData	Retrieves the data object describing the node.
getId	Retrieves the identifier of the node used for serialization.

Table 13. *Edge* class methods (extends *CrossModuleComponent*).

Method	**Description**
addDetectedInteraction	Adds a new interaction with no duration on this edge at the current timestamp.
addInteraction	Creates and adds a new interaction on this edge.
getAlter	Retrieve the edge's endpoint that is not the ego of the edge context's ego network if getEgo finds an ego endpoint.
getContext	Retrieves the context the edge belongs to.
getDst	Retrieves the destination node of the edge.
getEgo	Retrieves the ego node of the edge context's ego network if that ego a member of the edge.
getInteractions	Retrieves a shallow copy of the edge's interaction list.
getSrc	Retrieves the source node of the edge.

Table 14. *Interaction* class methods.

Method	**Description**
getData	Retrieves the interaction's data.
getDuration	Retrieves the duration of the interaction.
getEdge	Retrieves the edge the interaction belongs to.
getEndTime	Retrieves the timestamp of the end of the interaction.
getStartTime	Retrieves the timestamp of the start of the interaction.
getType	Retrieves the class name of the interaction’s data.

Table 15. *ContextualEgoNetworkListener* interface methods.

Method	**Description**
init	Called when the listener is added to a contextual ego network.
onAddNode	Called when a node is added to a context (after the node is added).
onCreateContext	Called when a new context is created by the ContextualEgoNetwork.getOrCreateContext method (after the created context is added to the network).
onCreateEdge	Called when an edge is created in a context using the Context.addEdge method (after the edge is created).
onCreateInteraction	Called when an interaction is created (after it has been added to an edge).
onCreateNode	Called when a new node is created by the ContextualEgoNetwork.getOrCreateNode method (after the created node is added to the network).
onLoadContext	Called when the Context.load() method of a context is called.
onRemoveContext	Called when a context is removed from the contextual ego network by the ContextualEgoNetwork.removeContext method.
onRemoveEdge	Called when an edge is removed from a context using the Context.removeEdge(Node, Node) method.
onRemoveNode	Called when a node is removed from a context using the Context.removeNode.
onRemoveNode	Called when a node is removed from the contextual ego network by the ContextualEgoNetwork.removeNodeIfExists method.
onSaveContext	Called when the Context.save method of a context is called (after the save has completed).

Annex II – Graph Recommendation Module API

Details of the extension module's *eu.h2020.helios_social.modules.socialgraphmining* beta release, which implements the social graph recommendation algorithms described in this deliverable. Classes not needed to understand and work with the module are omitted.

Table 16. Package *eu.h2020.helios_social.modules.socialgraphmining* entities.

Class or Interface	Description
Measure	Provides an abstraction of evaluation measures used to assess implementations of SocialGraphMiner.
SocialGraphMiner	Provides an abstraction of the basic capabilities and requirements of graph mining algorithms.
Measure classes.	
measures.Accumulate	This Measure is used to average the outcome of a base measure over given frames of time.
measures.HitRate	This Measure provides a HitRate@k evaluation, provides a HitRate@k evaluation, which measures whether the occurred interactions lie among the top k recommendations provided by SocialGraphMiner.recommendInteractions.
GNN Implementation	
GNN.GNNMiner	This class provides an implementation of a SocialGraphMiner based on a Graph Neural Network (GNN) architecture.
GNN.ContextTrainingExampleData	This class provides a storage structure that organizes a list of TrainingExample data to be stored in the contextual ego network's contexts.
GNN.GNNNodeData	This class provides a storage structure that organizes an embedding and a regularization target of contextual ego network nodes.
GNN.TrainingExample	A class models a training example used in GNN architectures that can be written a tuple (src, dst, weight, label).
GNN.operations.Tensor	This class provides a native java implementation of Tensor functionalities.
GNN.operations.Loss	Provides computation and (partial) derivation of activation and cross-entropy loss functions.
SocialGraphMiner classes.	
SwitchableMiner	This class enables switching between different

	SocialGraphMiner implementations for performing predictions while training them at the same time.
RandomMiner	This class provides a SocialGraphMiner that recommends random interactions among previous ones.
RepeatAndReplyMiner	This class provides a SocialGraphMiner that re-recommends previous interactions with alters based on their chronological order.
DifferenceMiner	This class provides a SocialGraphMiner that wraps a miner predictions so as not to predict the top of a base miner's predictions (e.g. DHR@k,withhold of a miner is obtained if HitRate@k is calculated over the outcome of a new AdditionalDiscoveryMiner(miner, new RepeatAndReplyMiner(cen), withhold).

Table 17. *Measure* class methods.

Method	**Description**
evaluateSend	Supervised evaluation of interactions sent by each node

Table 18. *SocialGraphMiner* abstract class methods.

Method	**Description**
getContextualEgoNetwork	Retrieves the contextual ego network of the graph miner.
getModelParameters	Retrieves the parameters of the mining model that will be sent alongside the created interaction.
newInteraction	Makes the graph miner aware that a user received an interaction from another user with getModelParameters.
predictNewInteraction	Predicts the weight of performing a SEND interaction between the given context's ego and a destination node within a given context.
recommendInteractions	Calls predictNewInteraction to score the likelihood of interacting with all nodes of the given context.

Table 19. *GNN.GNNMiner* class methods (extends SocialGraphMiner).

Method	Description
getModelParameters	Retrieves the parameters of the mining model that will be sent alongside the created interaction.
newInteraction	Makes the graph miner aware that a user received an interaction from another user with SocialGraphMiner.getModelParameters.
predictNewInteraction	Predicts the weight of performing a SEND interaction between the given context's ego and a destination node within a given context.
setDeniability	Enables plausible deniability and differential privacy handling by permuting the ego and its alters' parameters with a random noise proportional to a given constant and their norm.
setLearningRate	The learning rate (default is 1) from which GNNMiner training starts.
setLearningRateDegradation	Performs a fixed degradation of the learning rate over training epochs by multiplying the latter with a given factor (default is 0.95) after each epoch.
setMaxTrainingEpoch	Limits the number of training epochs (default is 1000) over which to train the GNNMiner.
setMinTrainingRelativeLoss	When the GNNMiner is being trained, training stops at epochs where abs(previous epoch loss - this epoch loss) < convergenceRelativeLoss*(this epoch loss) where losses are weighted cross entropy ones.
setRegularizationAbsorbsion	Multiplies regularization tensors with this value before setting them as regularization; value of 1 (default) produces regularization of calculated alter embeddings towards the embeddings calculated on alter devices.
setRegularizationWeight	The regularization weight (default 0.1) to apply during training of the GNNMiner.
setTrainingExampleDegradation	Degrades example weights each time a new one is generated through newInteraction by calling ContextTrainingExampleData.degrade to multiply previous weights with the given degradation factor (default is 0.5).
setTrainingExampleRemovalThreshold	Sets the threshold weight at which old training examples are removed (default is 0.01).

Table 20. *GNN.ContextTrainingExampleData* class methods (used by *GNN.GNNMiner*).

Method	**Description**
degrade	Calls the TrainingExample.degrade operation for each TrainingExample in the data.
getTrainingExampleList	Grants direct access to a list of training examples to traverse of edit.

Table 21. *GNN.TrainingExample* class methods (used by *GNN.GNNMiner*).

Method	**Description**
degrade	Multiplies the weight of the training example with a given factor.
getDst	Retrieves the destination node of the training example interaction.
getLabel	Retrieves the binary label the training example interaction that indicates whether whether the training example is of an existing (1) or non-existing (0) interaction.
getSrc	Retrieves the destination node of the training example interaction.
getWeight	Retrieves the weight of the training example.

Table 22. *GNN.GNNNodeData* class methods (used by *GNN.GNNMiner*).

Method	**Description**
getEmbedding	Retrieves the embedding of the node.
getNeighborAggregation	An aggregation of the node's neighborhood embeddings in the social graph.
setLearningRate	Sets the learning rate (default is 1) of the updateEmbedding operation.
setNeighborAggregation	Sets a neighbor aggregation that can be retrieved with getNeighborAggregation.
setRegularization	Sets the regularization (default is a zero vector) of the updateEmbedding operation.

setRegularizationWeight	Sets the regularization weight (default is 0.1) of the updateEmbedding operation.
updateEmbedding	Performs the operation *embedding += (embedding-regularization)*learningRate*regularizationWeight-derivative*learningRate* that is a regularized gradient descent over a computed derivative, where the area of regularization is constrained towards the point set by setRegularization.

Table 23. *SwitchableMiner* class methods (extends *SocialGraphMiner*).

Method	**Description**
createMiner	Creates a miner of the given name for the given SocialGraphMiner class by calling its constructor that takes a social ego network as parameter.
getActiveMiner	Retrieves the active miner.
getCreatedMinerNames	Retrieves the names of all miners created through createMiner calls.
getMiner	Retrieves a miner previously created with createMiner by its given name.
getModelParameters	Overrides getModelParameters to send the parameters of all miners.
setActiveMiner	Gets a created miner with getMiner and, if such a miner is found, this is set as the active miner.

References

[1] Burke, M., Kraut, R., & Marlow, C. (2011). Social capital on Facebook: Differentiating uses and users. *Proceedings of the SIGCHI conference on human factors in computing systems,* pp. 571-580

[2] Brandtzæg, P. B. (2012). Social networking sites: Their users and social implications—A longitudinal study. *Journal of Computer-Mediated Communication, 17(4),* pp. 467-488

[3] McPherson, M., Smith-Lovin, L., & Cook, J. M. (2001). Birds of a feather: Homophily in social networks. *Annual review of sociology, 27(1),* pp. 415-444

[4] Berry, G., Sirianni, A., Weber, I., An, J., & Macy, M. (2020). Going beyond accuracy: estimating homophily in social networks using predictions. *arXiv preprint arXiv:2001.11171*

[5] Milgram, S. (1967). The small world problem. *Psychology today, 2(1),* pp. 60-67

[6] Backstrom, L., Boldi, P., Rosa, M., Ugander, J., & Vigna, S. (2012). Four degrees of separation. *Proceedings of the 4th Annual ACM Web Science Conference,* pp. 33-42

[7] Watts, D. J., & Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. *Nature, 393(6684),* pp. 440.

[8] Alvarez-Hamelin, J. I., Dall'Asta, L., Barrat, A., & Vespignani, A. (2006). Large scale networks fingerprinting and visualization using the k-core decomposition. *Advances in neural information processing systems,* pp. 41-50

[9] Montresor, A., De Pellegrini, F., & Miorandi, D. (2012). Distributed k-core decomposition. *IEEE Transactions on parallel and distributed systems*, *24*(2), pp. 288-300

[10] Page, L., Brin, S., Motwani, R., & Winograd, T. (1999). The pagerank citation ranking: Bringing order to the web. *Stanford InfoLab*

[11] Aynaud, T., Fleury, E., Guillaume, J. L., & Wang, Q. (2013). Communities in evolving networks: definitions, detection, and analysis techniques. *Dynamics On and Of Complex Networks, Volume 2,* pp. 159-200

[12] Cazabet, R., & Amblard, F. (2014). Encyclopedia of social network analysis and mining, chapter dynamic community detection.

[13] Papadopoulos, S., Kompatsiaris, Y., Vakali, A., & Spyridonos, P. (2012). Community detection in social media. *Data Mining and Knowledge Discovery, 24(3),* pp. 515-554

[14] Coscia, M., Giannotti, F., & Pedreschi, D. (2011). A classification for community discovery methods in complex networks. *Statistical Analysis and Data Mining: The ASA Data Science Journal, 4(5), pp. 512-546.*

[15] Fortunato, S. (2010). Community detection in graphs. *Physics reports, 486(3-5),* pp. 75-174

[16] Saul, L. K., Weinberger, K. Q., Ham, J. H., Sha, F., & Lee, D. D. (2006). Spectral methods for dimensionality reduction. *Semisupervised learning,* pp. 293-308

[17] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems,* pp. 3111-3119

[18] Kusner, M., Sun, Y., Kolkin, N., & Weinberger, K. (2015). From word embeddings to document distances. *International conference on machine learning*, pp. 957-966

[19] Schafer, J. B., Frankowski, D., Herlocker, J., & Sen, S. (2007). Collaborative filtering recommender systems. *The adaptive web, Springer, Berlin, Heidelberg*, pp. 291-324

[20] Sarwar, B., Karypis, G., Konstan, J., & Riedl, J. (2001). Item-based collaborative filtering recommendation algorithms. *Proceedings of the 10th international conference on World Wide Web*, pp. 285-295

[21] Yan, S., Xu, D., Zhang, B., Zhang, H. J., Yang, Q., & Lin, S. (2006). Graph embedding and extensions: A general framework for dimensionality reduction. *IEEE transactions on pattern analysis and machine intelligence*, *29*(1), pp. 40-51.

[22] Goyal, P., & Ferrara, E. (2018). Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, *151*, pp. 78-94.

[23] He, X., Cai, D., Yan, S., & Zhang, H. J. (2005). Neighborhood preserving embedding. *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1 (Vol. 2, IEEE)*, pp. 1208-1213

[24] Khan, A., Li, N., Yan, X., Guan, Z., Chakraborty, S., & Tao, S. (2011). Neighborhood based fast graph search in large networks. *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 901-912

[25] Tsitsulin, A., Mottin, D., Karras, P., & Müller, E. (2018). Verse: Versatile graph embeddings from similarity measures. *Proceedings of the 2018 World Wide Web Conference*, pp. 539-548

[26] Tchuente, D., Canut, M. F., Jessel, N., Péninou, A., & Sèdes, F. (2013). A community-based algorithm for deriving users' profiles from egocentrics networks: experiment on Facebook and DBLP. *Social Network Analysis and Mining, 3(3),* pp. 667-683

[27] Bozorgi, A., Haghighi, H., Zahedi, M. S., & Rezvani, M. (2016). INCIM: A community-based algorithm for influence maximization problem under the linear threshold model. *Information Processing & Management*, *52(6),* pp. 1188-1199.

[28] Wang, Y., Cong, G., Song, G., & Xie, K. (2010). Community-based greedy algorithm for mining top-k influential nodes in mobile social networks. *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1039-1048

[29] Qian, F., Zhang, Y., Zhang, Y., & Duan, Z. (2013). Community-based user domain model collaborative recommendation algorithm. *TSINGHUA science and technology*, *18*(4), pp. 353-359

[30] Lalwani, D., Somayajulu, D. V., & Krishna, P. R. (2015). A community driven social recommendation system. *IEEE International Conference on Big Data (Big Data)*, pp. 821-826

[31] Soundarajan, S., & Hopcroft, J. (2012, April). Using community information to improve the precision of link prediction methods. *Proceedings of the 21st International Conference on World Wide Web*, pp. 607-608

[32] Valverde-Rebaza, J., & de Andrade Lopes, A. (2013). Exploiting behaviors of communities of twitter users for link prediction. *Social Network Analysis and Mining*, *3(4),* pp. 1063-1074.

[33] Moody, J., & White, D. R. (2003). Structural cohesion and embeddedness: A hierarchical concept of social groups. *American sociological review*, pp. 103-127

[34] Krishnamurthy, B., & Wang, J. (2000). On network-aware clustering of web clients. *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 97-110

[35] Zhang, Z. K., Zhou, T., & Zhang, Y. C. (2010). Personalized recommendation via integrated diffusion on user–item–tag tripartite graphs. *Physica A: Statistical Mechanics and its Applications*, *389(1),* pp.179-186

[36] Ma, H., King, I., & Lyu, M. R. (2011). Mining web graphs for recommendations. *IEEE Transactions on Knowledge and Data Engineering*, *24(6),* pp. 1051-1064

[37] Sarkar, P., Moore, A. W., & Prakash, A. (2008). Fast incremental proximity search in large graphs
Proceedings of the 25th international conference on Machine learning, pp. 896-903

[38] Fouss, F., Pirotte, A., Renders, J. M., & Saerens, M. (2007). Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *IEEE Transactions on knowledge and data engineering*, *19(3),* pp. 355-369.

[39] Schafer, J. B., Frankowski, D., Herlocker, J., & Sen, S. (2007). Collaborative filtering recommender systems. *The adaptive web*, *Springer, Berlin, Heidelberg,* pp. 291-324

[40] Sarwar, B., Karypis, G., Konstan, J., & Riedl, J. (2001). Item-based collaborative filtering recommendation algorithms. *Proceedings of the 10th international conference on World Wide Web*, pp. 285-295

[41] Herlocker, J. L., Konstan, J. A., & Riedl, J. (2000). Explaining collaborative filtering recommendations. *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pp. 241-250

[42] Bahmani, B., Chowdhury, A., & Goel, A. (2010). Fast incremental and personalized pagerank. *Proceedings of the VLDB Endowment*, *4(3),* pp. 173-184

[43] Fogaras, D., Rácz, B., Csalogány, K., & Sarlós, T. (2005). Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, *2*(3), pp. 333-358

[44] Tong, H., Faloutsos, C., & Pan, J. Y. (2006). Fast random walk with restart and its applications. *Sixth international conference on data mining (ICDM'06), IEEE, pp. 613-622*

[45] Chung, F., & Zhao, W. (2010). PageRank and random walks on graphs *Fete of combinatorics and computer science, Springer, Berlin, Heidelberg,* pp. 43-62

[46] Bandeira, A. S., Singer, A., & Spielman, D. A. (2013). A Cheeger inequality for the graph connection Laplacian. *SIAM Journal on Matrix Analysis and Applications*, *34*(4), pp. 1611-1630

[47] Chung, F. (2005). Laplacians and the Cheeger inequality for directed graphs. *Annals of Combinatorics*, *9(1),* pp. 1-19

[48] Gavili, A., & Zhang, X. P. (2017). On the shift operator, graph frequency, and optimal filtering in graph signal processing. *IEEE Transactions on Signal Processing*, *65(23),* pp. 6303-6318

[49] Ortega, A., Frossard, P., Kovačević, J., Moura, J. M., & Vandergheynst, P. (2018). Graph signal processing: Overview, challenges, and applications. *Proceedings of the IEEE*, *106*(5), pp. 808-828

[50] Sandryhaila, A., & Moura, J. M. (2013). Discrete signal processing on graphs: Graph Fourier transform. *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 6167-6170

[51] Chen, S., Varma, R., Sandryhaila, A., & Kovačević, J. (2015). Discrete Signal Processing on Graphs: Sampling Theory. *IEEE transactions on signal processing*, *63(24)*, pp. 6510-6523

[52] Chung, F. (2007). The heat kernel as the pagerank of a graph. *Proceedings of the National Academy of Sciences*, *104*(50), pp. 19735-19740

[53] Johnson, R., & Zhang, T. (2007). On the effectiveness of Laplacian normalization for graph semi-supervised learning. *Journal of Machine Learning Research*, *8*(Jul), pp. 1489-1517

[54] Chung, F. (2007). Four proofs for the Cheeger inequality and graph partition algorithms. *Proceedings of ICCM* (Vol. 2), p. 378

[55] Miller, G. L., & Peng, R. (2013). Approximate maximum flow on separable undirected graphs. *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, *Society for Industrial and Applied Mathematics,* pp. 1151-1170

[56] Raghavan, U. N., Albert, R., & Kumara, S. (2007). Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, *76(3),* 036106.

[57] Rosvall, M., & Bergstrom, C. T. (2008). Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, *105*(4), pp. 1118-1123.

[58] Rosvall, M., Axelsson, D., & Bergstrom, C. T. (2009). The map equation. *The European Physical Journal Special Topics*, *178*(1), pp. 13-23

[59] Pons, P., & Latapy, M. (2005, October). Computing communities in large networks using random walks. *International symposium on computer and information sciences, Springer, Berlin, Heidelberg,* pp. 284-293

[60] Clauset, A., Newman, M. E., & Moore, C. (2004). Finding community structure in very large networks. *Physical review E*, *70*(6), 066111

[61] Newman, M. E. (2006). Modularity and community structure in networks. *Proceedings of the national academy of sciences*, *103(23),* pp. 8577-8582

[62] Blondel, V. D., Guillaume, J. L., Lambiotte, R., & Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, *2008*(10), 10008.

[63] Cheng, R., & Vassileva, J. (2005). User motivation and persuasion strategy for peer-to-peer communities. *Proceedings of the 38th annual Hawaii international conference on system sciences, IEEE*, pp. 193a-193a
[64] Raghavan, U. N., Albert, R., & Kumara, S. (2007). Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, *76*(3), 036106
[65] Clementi, A., Di Ianni, M., Gambosi, G., Natale, E., & Silvestri, R. (2015). Distributed community detection in dynamic graphs. *Theoretical Computer Science*, *584*, pp. 19-41
[66] Herbiet, G. J., & Bouvry, P. (2010). SHARC: community-based partitioning for mobile ad hoc networks using neighborhood similarity. *IEEE International Symposium on "A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*, pp. 1-9
[67] Hui, P., Yoneki, E., Chan, S. Y., & Crowcroft, J. (2007). Distributed community detection in delay tolerant networks. *Proceedings of 2nd ACM/IEEE international workshop on Mobility in the evolving internet architecture*, pp. 1-8
[68] Ramaswamy, L., Gedik, B., & Liu, L. (2005). A distributed approach to node clustering in decentralized peer-to-peer networks. *IEEE Transactions on Parallel and Distributed Systems*, *16(9)*, pp. 814-829
[69] Hu, P., & Lau, W. C. (2012). Localized algorithm of community detection on large-scale decentralized social networks. *arXiv preprint arXiv:1212.6323*.
[70] Guidi, B., Michienzi, A., & Ricci, L. (2018). Sonic-man: a distributed protocol for dynamic community detection and management. *IFIP International Conference on Distributed Applications and Interoperable Systems,* Springer, Cham, pp. 93-109
[71] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), pp. 436-444.
[72] Oltra, S., & Valero, O. (2004). Banach's fixed point theorem for partial metric spaces. *Rend. Istit. Mat. Univ. Trieste, 36,* pp. 17-26.
[73] Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., & Yakhnenko, O. (2013). Translating embeddings for modeling multi-relational data. *Advances in neural information processing systems*, pp. 2787-2795
[74] Yang, B., Yih, W. T., He, X., Gao, J., & Deng, L. (2014). Embedding entities and relations for learning and inference in knowledge bases. *arXiv preprint arXiv:1412.6575*.
[75] Schlichtkrull, M., Kipf, T. N., Bloem, P., Van Den Berg, R., Titov, I., & Welling, M. (2018). Modeling relational data with graph convolutional networks. *European Semantic Web Conference, Springer, Cham*, pp. 593-607
[76] Dettmers, T., Minervini, P., Stenetorp, P., & Riedel, S. (2018). Convolutional 2d knowledge graph embeddings. *Thirty-Second AAAI Conference on Artificial Intelligence*.
[77] Zhang, Z., & Sabuncu, M. (2018). Generalized cross entropy loss for training deep neural networks with noisy labels. *Advances in neural information processing systems*, pp. 8778-8788
[78] Zhang, Z. (2018). Improved Adam optimizer for deep neural networks. *IEEE/ACM 26th International Symposium on Quality of Service (IWQoS), IEEE,* pp. 1-2
[79] Huang, Z., Xu, W., & Yu, K. (2015). Bidirectional LSTM-CRF models for sequence tagging. *arXiv preprint arXiv:1508.01991*
[80] Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., & Schmidhuber, J. (2016). LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems*, *28(10),* pp. 2222-2232.
[81] Boccaletti, S., Bianconi, G., Criado, R., Del Genio, C. I., Gómez-Gardenes, J., Romance, M. & Zanin, M. (2014). The structure and dynamics of multilayer networks. *Physics Reports*, *544*(1), pp. 1-122.
[82] Yan, S., Xiong, Y., & Lin, D. (2018). Spatial temporal graph convolutional networks for skeleton-based action recognition. *Thirty-second AAAI conference on artificial intelligence*
[83] Rossetti, G., & Cazabet, R. (2018). Community discovery in dynamic networks: a survey. *ACM Computing Surveys (CSUR)*, *51*(2), pp. 1-37
[84] Fischer, M. J., Lynch, N. A., & Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, *32(2),* pp. 374-382.

[85] Yang, B. B., & Garcia-Molina, H. (2003). Designing a super-peer network. *Proceedings 19th international conference on data engineering (Cat. No. 03CH37405), IEEE,* pp. 49-60

[86] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., & Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, *31*(4), pp. 149-160

[87] Rowstron, A., & Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing, Springer, Berlin, Heidelberg,* pp. 329-350

[88] De Salve, A., Guidi, B., Mori, P., & Ricci, L. (2016). Distributed coverage of ego networks in f2f online social networks. *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, *IEEE*, pp. 423-431

[89] Palla, G., Derényi, I., Farkas, I., & Vicsek, T. (2005). Uncovering the overlapping community structure of complex networks in nature and society. *Nature, 435(7043)*, pp. 814-818

[90] Prat-Pérez, A., Dominguez-Sal, D., & Larriba-Pey, J. L. (2014). High quality, scalable and parallel community detection for large real graphs. *Proceedings of the 23rd international conference on World wide web*, pp. 225-236

[91] Serrour, B., Arenas, A., & Gómez, S. (2011). Detecting communities of triangles in complex networks using spectral optimization. *Computer Communications*, *34*(5), pp. 629-634

[92] Radicchi, F., Castellano, C., Cecconi, F., Loreto, V., & Parisi, D. (2004). Defining and identifying communities in networks. *Proceedings of the national academy of sciences*, *101*(9), pp. 2658-2663.

[93] Guidi, B., Michienzi, A., & Ricci, L. (2018). SONIC-MAN: a distributed protocol for dynamic community detection and management. *IFIP International Conference on Distributed Applications and Interoperable Systems, Springer, Cham,* pp. 93-109

[94] Leskovec, J., & Mcauley, J. J. (2012). Learning to discover social circles in ego networks. *Advances in neural information processing systems*, pp. 539-547

[95] Viswanath, B., Mislove, A., Cha, M., & Gummadi, K. P. (2009). On the evolution of user interaction in facebook. *Proceedings of the 2nd ACM workshop on Online social networks*, pp. 37-42

[96] Opsahl, T., & Panzarasa, P. (2009). Clustering in weighted networks. *Social networks*, *31*(2), pp. 155-163.

[97] Sapiezynski, P., Stopczynski, A., Lassen, D. D., & Lehmann, S. (2019). Interaction data from the copenhagen networks study. *Scientific Data*, *6*(1), pp. 1-10

[98] Shetty, J., & Adibi, J. (2004). The Enron email dataset database schema and brief statistical report. *Information sciences institute technical report, University of Southern California*, *4*(1), pp. 120-128

[99] Tan, X. (2017). A new extrapolation method for PageRank computations. *Journal of Computational and Applied Mathematics*, *313*, pp. 383-392

[100] Fujiwara, Y., Nakatsuji, M., Shiokawa, H., Mishima, T., & Onizuka, M. (2017). Fast ad-hoc search algorithm for personalized pagerank. *IEICE TRANSACTIONS on Information and Systems*, *100*(4), pp. 610-620

[101] Krasanakis, E., Papadopoulos, S. & Kompatsiaris, I. (2020). Stopping Personalized PageRank without an Error Tolerance Parameter. *Under review*

[102] Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, *6*(02), pp. 107-116

[103] Xu, K., Hu, W., Leskovec, J., & Jegelka, S. (2018). How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*

[104] Kipf, T. N., & Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*

[105] You, J., Ying, R., & Leskovec, J. (2019). Position-aware graph neural networks. *arXiv preprint arXiv:1906.04817*

[106] Li, Q., Han, Z., & Wu, X. M. (2018). Deeper insights into graph convolutional networks for semi-supervised learning. *Thirty-Second AAAI Conference on Artificial Intelligence*

[107] Shchur, O., Mumme, M., Bojchevski, A., & Günnemann, S. (2018). Pitfalls of graph neural network evaluation. *arXiv preprint arXiv:1811.05868*

[108] Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. *Proceedings of COMPSTAT'2010, Physica-Verlag HD,* pp. 177-186

[109] Kermarrec, A. M., Leroy, V., & Trédan, G. (2011). Distributed social graph embedding. *Proceedings of the 20th ACM international conference on Information and knowledge management*, pp. 1209-1214

[110] Lalitha, A., Kilinc, O. C., Javidi, T., & Koushanfar, F. (2019). Peer-to-peer federated learning on graphs. *arXiv preprint arXiv:1901.11173*.

[111] Dwork, C., & Roth, A. (2014). The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, *9*(3-4), pp. 211-407

[112] Abadi, M., Chu, A., Goodfellow, I., McMahan, H. B., Mironov, I., Talwar, K., & Zhang, L. (2016). Deep learning with differential privacy. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 308-318

[113] Bindschaedler, V., Shokri, R., & Gunter, C. A. (2017). Plausible deniability for privacy-preserving data synthesis. *arXiv preprint arXiv:1708.07975*

[114] Mac Aonghusa, P., & Leith, D. J. (2017). Plausible Deniability in Web Search—From Detection to Assessment. *IEEE Transactions on Information Forensics and Security*, *13*(4), pp. 874-887

[115] Gamma, E. (1995). Design patterns: elements of reusable object-oriented software. *Pearson Education India*

[116] Montresor, A., & Jelasity, M. (2009). PeerSim: A scalable P2P simulator. *IEEE Ninth International Conference on Peer-to-Peer Computing*, pp. 99-100

[117] Broder, A. Z., Charikar, M., Frieze, A. M., & Mitzenmacher, M. (2000). Min-wise independent permutations. *Journal of Computer and System Sciences, 60(3),* pp. 630-659

[118] Gedikli, F., Jannach, D., & Ge, M. (2014). How should I explain? A comparison of different explanation types for recommender systems. *International Journal of Human-Computer Studies*, *72*(4), pp. 367-382

[119] Tintarev, N., & Masthoff, J. (2007). Effective explanations of recommendations: user-centered design. *Proceedings of the 2007 ACM conference on Recommender systems*, pp. 153-156

[120] Tintarev, N., & Masthoff, J. (2007). A survey of explanations in recommender systems. *IEEE 23rd international conference on data engineering workshop*, pp. 801-810

[121] Krasanakis, E., Schinas, E., Papadopoulos, S., Kompatsiaris, Y., & Symeonidis, A. (2020). Boosted seed oversampling for local community ranking. *Information Processing & Management*, *57(2),* 102053

[122] Krasanakis, E., Papadopoulos, S., & Kompatsiaris I. (2020). LinkAUC: Unsupervised Evaluation of Multiple Network Node Ranks Using Link Prediction. *International Conference on Complex Networks and Their Applications*, *Springer, Cham,* pp. 3-14