

# Learning the similarity on very large graphs

Master thesis submitted as part of the  
Specialized Master in data science, Big data

**Timothée Defoin**

Promotor

Prof. Dr. Ir. Hugues Bersini

Co-promotor

Dr. Ir. Robin Devooght

Service

IRIDIA

Academic Year  
2018 - 2019



# Abstract

## Learning the similarity on very large graphs

With the rise of Internet and the social networks generating huge amount of data, large networks are becoming predominant in machine learning. One important problem encountered when dealing with those is the seed set expansion, in which we try to grow a given group from a known seed.

The aim of this work is to implement a method initially developed by Robin Devooght, Peter Staar and Costas Bekas. This method learns a similarity measure from the topology of the network, and is then used in the context of semi-supervised learning problems. We will then implement some variations of the method to see if the results can be improved.

A Python implementation of the algorithm and various benchmark methods has been realized in order to evaluate the method, and the way it has been done is explained.

The tests realized on various datasets confirmed the robustness of the method, outperforming the other methods in the vast majority of the cases. The implementation of little variation in the algorithms almost did not improve the results, but it still shed some light on where to continue if one wants to pursue this.

**Keywords:** Semi-supervised learning, similarity measure, graph, classification, seed set expansion, classification, label propagation



## Acknowledgements

I would like to thank Robin Devooght for the all the help he gave me in this work, using a lot of his free time to answer my questions. And also Mr.Bersini for giving me the opportunity to work with him.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Graphs and similarity measures</b>	<b>3</b>
2.1	Introduction to graphs . . . . .	3
2.1.1	Basic definitions . . . . .	3
2.1.2	Graph matrices . . . . .	4
2.2	Similarities . . . . .	7
2.2.1	Personalized PageRank (PPR) . . . . .	8
2.2.2	Normalized Laplacian exponential diffusion (Exp) . . . . .	8
<b>3</b>	<b>Semi-supervised learning</b>	<b>11</b>
3.1	Introduction . . . . .	11
3.1.1	Supervised and unsupervised learning . . . . .	11
3.1.2	Semi-supervised learning . . . . .	12
3.1.3	Common problems in semi-supervised learning . . . . .	12
3.2	Graph-based semi-supervised learning . . . . .	13
3.2.1	Label propagation . . . . .	13
3.2.2	Laplacian regularisation . . . . .	14
3.2.3	Dealing with large graphs . . . . .	15
<b>4</b>	<b>Learning the similarity</b>	<b>17</b>
4.1	Motivation . . . . .	17
4.2	Rationale of the method . . . . .	18
4.3	Seed set expansion . . . . .	20
<b>5</b>	<b>Implementation and results</b>	<b>23</b>
5.1	Python implementation . . . . .	23
5.1.1	The learning algorithm . . . . .	23
5.1.2	Benchmark methods . . . . .	24
5.2	Results for the seed set expansion . . . . .	25
5.2.1	Evaluation . . . . .	25
5.2.2	Datasets . . . . .	25
5.2.3	Methodology . . . . .	26

5.2.4	Results . . . . .	26
<b>6</b>	<b>Improvement perspectives</b>	<b>29</b>
6.1	Self-learning of the negative nodes . . . . .	29
6.2	Non linearity . . . . .	30
6.2.1	The approach . . . . .	30
6.2.2	Results . . . . .	31
<b>7</b>	<b>Conclusion</b>	<b>33</b>
	<b>Appendices</b>	<b>I</b>
<b>A</b>	<b>Python code</b>	<b>III</b>
A.1	Learning the similarity module . . . . .	III
A.2	Benchmark methods module . . . . .	V
A.3	Main function to generate the seed set expansion tests . . . . .	X



# 1

## Introduction

Graphs (also called networks) are structures composed of nodes and edges that naturally arise when one is studying components with pairwise relationships. This type of relationships can be found everywhere, from social structures with friendships between people to citations between documents, protein-protein interaction networks, and many more.

These structures offers a framework in which lots of techniques and algorithms are available to extract useful information from the network data. One example of this is similarity measures constructed from the topology of the graph, which quantify how similar two nodes are, and help to infer information about a node from the ones that are the most similar.

Similarity measures are methods that work and scale very well on large graphs, but their performance fluctuates a lot depending on the problem at hand, and there is no specific class of problems in which we know a similarity measure to perform consistently well. Having that in mind, Robin Devooght developed a new framework to compute similarity measures during his thesis at the ULB [1, Chapter 4], in collaboration with Peter Staar and Costas Bekas from the IBM research lab of Zurich. This method is called 'Learning the similarity', and offered very promising results when used on real life datasets.

The goal of this master thesis is to implement the method in an open source language framework and reproduce the promising results that were obtained. We will then attempt to improve the method using different approaches. Python was chosen for the implementation, as it is now widely used for machine learning and artificial intelligence purpose, in addition to its very rich ecosystem.

Chapters 2 and 3 will introduce the notions of graphs, similarity measures and semi-supervised learning necessary to understand the method.

In chapter 4, we explain the rationale behind the method, and how the algorithm works, following closely the presentation that was given in the thesis of Robin Devooght.

Chapter 5 is about the details of the implementation of the algorithm, and the results that are obtained when compared to benchmark methods. We will focus here on the seed set expansion

## 1. Introduction

---

problem.

And finally we finish with 2 ideas to improve the method, that is adding nonlinearities in the propagation of the labels and self-learning for the negative sampling.

## 2

# Graphs and similarity measures

This chapter will present an introduction to graph theory, giving the concepts necessary to understand the rest of this work. We will then move on to similarity measures, which provide a convenient way to solve a lot of graph related machine learning tasks.

## 2.1 Introduction to graphs

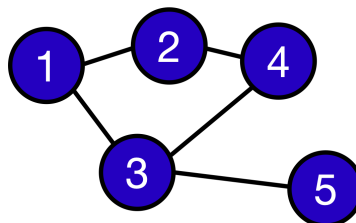
The aim of this section is to provide to the reader a brief summary of the basic concepts related to graphs that are used later on, and to fix the notations. This section was written following the introductory chapters of [2] and [3].

### 2.1.1 Basic definitions

A *graph* or *network*  $G = (\mathcal{V}, \mathcal{E})$  is a structure that is defined by providing:

- a set  $\mathcal{V}$  whose elements are called the *nodes* (or *vertices*) of the graph,
- a set  $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ , whose elements are pairs of nodes called *edges*.

Graphs are usually drawn using dots for the nodes, and joining a pair of dots with a line if there is an edge between the two corresponding nodes. A small example is given in fig. 2.1.



**Figure 2.1:** A simple undirected graph. The set of node is  $\mathcal{V} = \{1, 2, 3, 4, 5\}$  and the set of edges is  $\mathcal{E} = \{\{1, 2\}, \{1, 3\}, \{2, 4\}, \{3, 4\}, \{3, 5\}\}$ .

The *order* of a graph is its number of nodes  $n = |\mathcal{V}|$ , whereas the number of edges  $N = |\mathcal{E}|$  is called the *size* of the graph.

If the edges are ordered, that is  $(i, j)$  is a different edge than  $(j, i)$ , the graph is said to be *directed*, and the edges can also be denoted as  $i \rightarrow j$ . If there is no order, i.e.  $(i, j)$  and  $(j, i)$  corresponds to the same one edge, then the graph is said to be *undirected*, and the edges are denoted as  $i \leftrightarrow j$ . The use of a directed or undirected graph depends on whether the relationship has a direction or not. A co-authorship in a document network will be represented by undirected edges, whereas a citation network will be directed, since there is a notion of direction with a document citing another.

Two nodes are *adjacent* (or *connected*) if there is an edge between them. A node and an edge are *incident* when the edge is connected to the node.

A *weight*  $w_{ij}$  can be assigned to each edge  $(i, j)$ . This is useful if there is a degree of intensity in the relationship (for example the weight could be the number of common documents in a co-authorship network).

The *degree*  $d(i)$  of a node  $i$  in an unweighted undirected graph is given by the number of edges incident with it. The degree can be generalized to weighted graphs, for which it is given by the sum of the weights of the incident edges.

A *path* from node  $i$  to node  $j$  is a succession of connected nodes that starts at  $i$  and end in  $j$ . The length is given by the number of edges crossed in the unweighted case. If the edges are weighted, the length is then the sum of the weights of the encountered edges. The distance between two nodes in a graph is usually defined as the shortest path available between them, and the longest shortest path of a graph  $G$  is called its diameter.

In the rest of this work, all the graphs are unweighted and undirected unless specified otherwise.

### 2.1.2 Graph matrices

Matrices provide a neat way of representing a graph. They are also much more useful than the picture of a graph if we want to perform any type of computations. We will present in the following the ones that are most encountered when dealing with networks.

#### Adjacency matrix

The *adjacency matrix*  $\mathbf{A} = (a_{ij})_{n \times n}$  of a graph  $G$  is defined as

$$a_{ij} = \begin{cases} 1, & \text{if } (i, j) \subseteq \mathcal{E}, \\ 0, & \text{otherwise.} \end{cases} \quad (2.1)$$

This matrix captures all the structure of the graph in a useful algebraic representation. Furthermore, the elements of the powers  $\mathbf{A}^n$  of the adjacency matrix gives us the number of paths of length  $n$ . For instance, the elements  $[\mathbf{A}^2]_{ij}$  gives us the number of paths of length 2

between nodes  $i$  and  $j$ . Therefore, the quantity

$$S_n = \sum_{i=1}^n A^i \quad (2.2)$$

gives us the number of paths of length inferior or equal to  $n$ . Obviously, the adjacency matrix is symmetric for an undirected graph, as  $a_{ij} = a_{ji}$ ,  $\forall i, j$ .

### Degree matrix

The *degree matrix*  $\mathbf{D} = (d_{ij})_{n \times n}$  of a graph is a diagonal matrix given by

$$d_{ij} = \sum_k a_{ik} \delta_{ij} = d(i) \delta_{ij} \quad (2.3)$$

where  $\delta_{ij}$  is the Kronecker delta. The  $i$ -th elements of the diagonal gives the degree  $d(i)$  of node  $i$ , and it is often used to normalize matrices as a mean to penalize nodes with lots of edges.

### Transition probability matrix

Let us consider a random walker in a graph. At every increment in time, he jumps from the node he is in to one of the adjacent nodes randomly with equal probability. The probability he jumps from node  $i$  to  $j$  is then given by

$$p_{ij} = \frac{a_{ij}}{\sum_{k=1}^n a_{ik}} \quad (2.4)$$

In matrix form, this can be written as

$$\mathbf{P} = \mathbf{D}^{-1} \mathbf{A} \quad (2.5)$$

where  $\mathbf{D}^{-1}$  is the inverse of the degree matrix and  $\mathbf{P}$  is called the *transition probability matrix*. As a consequence of the Gershgorin circle theorem, it has a spectral radius  $\rho(\mathbf{P}) \leq 1$  (the spectral radius is the magnitude of the largest eigenvalue).

### Symmetrically normalized adjacency matrix

The *symmetrically normalized adjacency matrix* is defined as

$$\tilde{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}} \quad (2.6)$$

Its elements are therefore

$$\tilde{a}_{ij} = \begin{cases} \frac{1}{\sqrt{d(i)d(j)}}, & \text{if } (i, j) \in \mathcal{E}, \\ 0, & \text{otherwise.} \end{cases} \quad (2.7)$$

Because  $\tilde{\mathbf{A}}$  is similar to  $\mathbf{P}$  ( $\tilde{\mathbf{A}} = \mathbf{D}^{\frac{1}{2}} \mathbf{P} \mathbf{D}^{-\frac{1}{2}}$ ), it possess the same eigenvalues, and so the spectral radius is also equal to 1 or less.

### Laplacian matrix

The Laplacian matrix  $\mathbf{L}$  is defined as

$$\mathbf{L} = \mathbf{D} - \mathbf{A} \quad (2.8)$$

It is encountered in plenty of graph applications, such as label diffusion, graph spectral theory and many more. One important property of the Laplacian matrix is that it measures the *smoothness* of a vector  $\mathbf{y}$  on the graph. Indeed, we have

$$\mathbf{y}^T \mathbf{L} \mathbf{y} = \mathbf{y}^T (\mathbf{D} - \mathbf{A}) \mathbf{y} \quad (2.9)$$

$$= \sum_i y_i^2 \sum_j a_{ij} - \sum_{i,j} y_i a_{ij} y_j \quad (2.10)$$

$$= \frac{1}{2} \sum_{i,j} a_{ij} (y_i - y_j)^2 \quad (2.11)$$

This term gauges how close the values of adjacent nodes are. If  $\mathbf{y}^T \mathbf{L} \mathbf{y}$  is small, that means that when two nodes are connected (i.e.  $a_{ij} = 1$ ), their values  $y_i$  and  $y_j$  are close, and the vector  $\mathbf{y}$  is said to be *smooth*.

### Symmetrically normalized Laplacian matrix

Notice that in (2.14), nodes with a lot of edges (hubs) will carry more importance in the sum compared to less connected nodes. If we do not want this to happen, we can normalize the Laplacian (symmetrically), leading to

$$\Delta = \mathbf{I} - \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}} = \mathbf{I} - \tilde{\mathbf{A}} \quad (2.12)$$

Using this instead of the Laplacian in the definition of smoothness, we get

$$\mathbf{y}^T \Delta \mathbf{y} = \mathbf{y}^T \mathbf{D}^{-\frac{1}{2}} (\mathbf{D} - \mathbf{A}) \mathbf{D}^{-\frac{1}{2}} \mathbf{y} \quad (2.13)$$

$$= \frac{1}{2} \sum_{i,j} a_{ij} \left( \frac{y_i}{\sqrt{d(i)}} - \frac{y_j}{\sqrt{d(j)}} \right)^2 \quad (2.14)$$

We can see that now there is a discounting factor depending on the number of nodes, which will diminish the importance of hubs in the sum.

### Matrices of a simple graph

As a concrete example, we give below a table with the previous matrices for the very simple graph in fig.2.1.

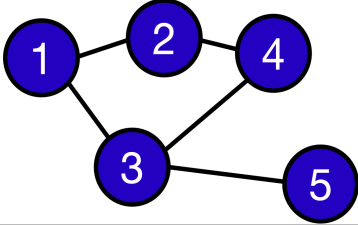
Graph	A	D
	$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$
$\tilde{A}$	P	L
$\begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{\sqrt{6}} & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 \\ \frac{1}{\sqrt{6}} & 0 & 0 & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{3}} \\ 0 & \frac{1}{2} & \frac{1}{\sqrt{6}} & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{3}} & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 \\ \frac{1}{3} & 0 & 0 & \frac{1}{3} & \frac{1}{3} \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 2 & -1 & -1 & 0 & 0 \\ -1 & 2 & 0 & -1 & 0 \\ -1 & 0 & 3 & -1 & -1 \\ 0 & -1 & -1 & 2 & 0 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix}$

Table 2.1: Graph matrices for a simple graph

## 2.2 Similarities

Defining a notion of similarity (or distance) between nodes of a graph is very useful in a lot of machine learning tasks. Indeed, once a good measure has been found, a lot of methods are available to solve clustering, classification or a whole lot of other problems.

Generally speaking, a similarity measure is a mapping  $s : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}$  between pair of nodes and a real scalar value. This can be written in a matrix form, where the elements  $s_{ij}$  of the *similarity matrix*  $S$  contain the similarity between node  $i$  and  $j$ . Note that there is no restrictions on the values of  $s_{ij}$ , similarities are a large class of objects which contain distances. Nevertheless, being so broad means that they do not enjoy some nice properties that distances possess.

The most basic similarity measure is the adjacency matrix itself. If two nodes are connected, they are similar, and if not, they are not. Of course, this is a very rough way of doing so. It does not account for the number of paths of length greater than one, even though it seems obvious that two nodes that share a lot of neighbours are probably pretty similar.

A little more sophisticated measure would be to use  $S_n$ , the number of paths of length inferior or equal to  $n$ . Even though it is slightly better, there are similarity measures that works much better than this. A great overview of such measures is given in [4].

We will present here two similarity measures that works well in practice: the personalized PageRank and the normalized Laplacian exponential diffusion.

### 2.2.1 Personalized PageRank (PPR)

The personalized PageRank similarity was first introduced by Pan et al. in [5], and is defined as

$$\mathbf{S}_{\text{PPR}} = (\mathbf{I} - (1 - \alpha)\mathbf{P}^T)^{-1} \quad (2.15)$$

This similarity is best thought in terms of random walkers roaming over adjacent nodes. Consider a random walker in node  $i$  at  $t = 0$ . Denoting with the vector  $\mathbf{u}$  the probability he jumps to another node in  $t = 1$  - assuming he must do so after one unit of time and has equal chances to take any incident edge - is given by

$$\mathbf{u}(1) = \mathbf{P}^T \mathbf{e}_i \quad (2.16)$$

where  $\mathbf{e}_i$  is the initial distribution of probability, that is 1 in node  $i$  and 0 elsewhere. Now, if we add the fact that the walk start back in the initial node with a probability  $\alpha$  at every step, then the probability is simply

$$\mathbf{u}(1) = (1 - \alpha)\mathbf{P}^T \mathbf{e}_i + \alpha \mathbf{e}_i \quad (2.17)$$

and if we continue, after time  $t$ , the probability is

$$\mathbf{u}(t) = (1 - \alpha)\mathbf{P}^T \mathbf{u}(t - 1) + \alpha \mathbf{e}_i \quad (2.18)$$

It can be shown that there is a steady state distribution for this random process, and it is given by

$$\mathbf{u}(\infty) = (\mathbf{I} - (1 - \alpha)\mathbf{P}^T)^{-1} \mathbf{e}_i. \quad (2.19)$$

This steady state distribution is what is used as a similarity. The elements of the column  $j$  of  $\mathbf{S}_{\text{PPR}}$  give the share of time the random walker spend on the different nodes starting from node  $j$ . The higher it is, the more similar the corresponding node is to node  $j$ .

### 2.2.2 Normalized Laplacian exponential diffusion (Exp)

Another widely used similarity measure is the symmetrically normalized exponential diffusion [6], given by

$$\mathbf{S}_{\text{exp}} = e^{-\alpha \Delta} \quad (2.20)$$

The following useful insight is given in [4]. Suppose each nodes has a quantity  $x_i(t)$  at each  $t$  that is diffused to other incident nodes with a rate of 1. After an small increment of time, the balance equation in node  $i$  is given by

$$x_i(t + \delta t) = x_i(t) + \sum_j a_{ij}(x_j(t) - x_i(t))\delta t \quad (2.21)$$



Taking the limit when  $\delta t \rightarrow 0$ , we get

$$\frac{dx_i(t)}{dt} = \sum_j a_{ij}x_j(t) - x_i(t) \sum_k a_{ik} \quad (2.22)$$

$$= \sum_j a_{ij}x_j(t) - \sum_j \delta_{ij}x_j(t)d(i) \quad (2.23)$$

$$= - \sum_j (\delta_{ij}d(i) - a_{ij}) x_j(t) \quad (2.24)$$

where  $d(i)$  is the degree of node  $i$ , and we can recognize the elements  $l_{ij} = \delta_{ij}d(i) - a_{ij}$  of the Laplacian matrix. This can be written in matrix form as  $\frac{d\mathbf{x}(t)}{dt} = -\mathbf{L}\mathbf{x}(t)$ . This system of differential equation is straightforward to solve

$$\mathbf{x}(t) = \exp(-\mathbf{L}t)\mathbf{x}_0 \quad (2.25)$$

with  $\mathbf{x}_0$  the initial conditions at  $t = 0$ .

This gives the intuition behind the definition of the Laplacian diffusion similarity that was first introduced by Kondor and Lafferty in [6]

$$\mathbf{S}_{LD} = \exp(-\alpha\mathbf{L}) \quad (2.26)$$

The quantity  $\mathbf{S}_{LD}\mathbf{e}_i$  (with  $\mathbf{e}_i$  the unit vector null everywhere except for label  $i$ ) represents the distribution after node  $i$  diffuses a unit quantity during a time  $t = \alpha$ , which means that every column  $j$  of the similarity matrix  $\mathbf{S}_{LD}$  represents a unit quantity diffusing from node  $j$  during a time  $t = \alpha$ .

As we mentionned in section 2.1.2, hubs will diffuse their quantity a lot more than poorly connected nodes. In order to avoid this, we can normalize the Laplacian, so that we end up with

$$\mathbf{S}_{\text{exp}} = \exp(-\alpha\mathbf{\Delta}) \quad (2.27)$$

the normalized Laplacian exponential diffusion kernel.



## 3

# Semi-supervised learning

In this chapter, we introduce the notion of semi-supervised learning, and how it is applied on graph data. In particular, there is a class of methods called *sum of similarities* that scales well and gives good results.

## 3.1 Introduction

Actually, more than 2.5 quintillion bytes of data is created everyday, and that pace is still accelerating [7]. With so much data being produced, the vast majority of it is 'raw', that is there is no meaningful labels associated to it. As a consequence of this, the need for methods able to leverage this type of data in addition to the labelled one is growing everyday. Such methods fall in the domain of semi-supervised learning, a middle-ground between complete supervised and unsupervised learning.

Chapelle et al. give a good introduction to the concept of supervised, unsupervised and semi-supervised learning in [8]. This section is a very brief summary of the introduction of their book.

### 3.1.1 Supervised and unsupervised learning

Machine learning tasks are usually divided into 2 main categories, that is unsupervised and supervised learning. We will briefly explain the difference between the two.

#### Supervised learning

In supervised learning, we are given a training set  $X = (x_1, \dots, x_n)$  for which we know the output of interest  $Y = (y_1, \dots, y_n)$ . The goal is use the training set to learn a mapping between  $X$  and  $Y$ , and to use this mapping to make predictions on unlabelled data.

#### Unsupervised learning

In contrast with the above, unsupervised learning deals with data  $X$  for which we have no output. It is usually thought as a first exploratory step in data mining, where we want to find an underlying structure in  $X$  to gain additional insight.

### 3.1.2 Semi-supervised learning

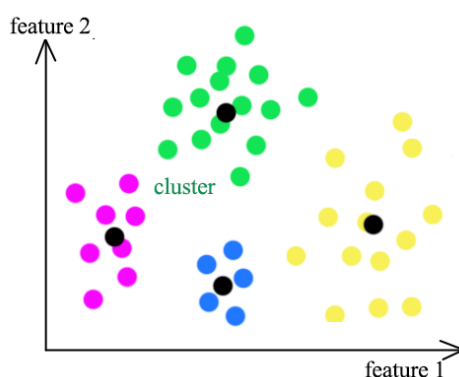
Semi-supervised learning lies in between the two previous paradigms. We have a (usually small) portion of data  $X_l$  that is labelled, and  $X_u$  which is the unlabelled portion of the data. The aim is to use at the same time the information of the labelled data and the information of the unlabelled data to get better results than with the labelled data alone.

### 3.1.3 Common problems in semi-supervised learning

We will describe here 3 very common problems found in semi-supervised learning (and in machine learning in general).

#### Clustering

Clustering is usually thought as an unsupervised learning problem. The goal of clustering is to group data into meaningful clusters. What is thought by meaningful is that samples inside the same cluster should be significantly more similar in some way than samples belonging to different clusters. Quantifying such a similarity between samples is a problem in itself, and will depend a lot on the problem at hand. An example of clustering is given in fig. 3.1.



**Figure 3.1:** An example of clusters resulting from the most common clustering algorithm: k-means clustering. Image coming from <https://jhui.github.io/assets/ml/kmean.png>

#### Binary classification

In the binary classification problem, we are given a dataset for which the output  $y_i$  of a labelled node  $i$  is equal to 1 or 0 depending on whether he belongs to the first class or the second, and the outputs of the unknown nodes are set to 0. This can be stored in the labelling vector  $\mathbf{y}$ . The endgame is to make a prediction  $\hat{\mathbf{y}} = f(\mathbf{y})$  using a classifier  $f$  that we have trained on

the known labels. As a side note, a multi-class problem can always be reduced to a succession of binary classifications by considering one class against the rest.

### Seed set expansion

The seed set expansion is a variation of the classification problem, for which we do not have negative examples. In social networks, we usually only have examples of people belonging to groups (that is true positives), but we do not know if someone does not belong in one (true negatives). This means that our starting point is a set of true positives, called the seed set, which we wish to expand. The absence of negative examples forbids us to use traditional classification methods that needs both type of examples to be effective. Nonetheless, there exists procedures to work around this, with the most obvious one being to create artificially negative examples to use the common classification techniques.

## 3.2 Graph-based semi-supervised learning

Often times, the data is organised as a network, and we wish to use the structure to help with different machine learning task. We saw in the previous chapter a couple of similarity measures that could help propagating labels to unlabelled nodes. We will now see two very important methods for graph-based semi-supervised learning, and how they relate to each other.

### 3.2.1 Label propagation

Just like the personalized PageRank could be seen as random walkers propagating the labels step by step until the steady state was reached, we can generally try to propagate the labels using any process we might deem appropriate to the problem at hand. This is what was done by Zhou et al. in [9]. The label are propagated step by step starting from  $\hat{\mathbf{y}}(0) = \mathbf{y}$  as follows

$$\hat{\mathbf{y}}(t) = \alpha \tilde{\mathbf{A}} \hat{\mathbf{y}}(t-1) + (1 - \alpha) \mathbf{y} \quad (3.1)$$

with  $\alpha \in (0, 1)$  and  $\tilde{\mathbf{A}}$  is the symmetrically normalized adjacency matrix. The predicted labels are diffused at each iteration while the known labels are reinforced.

Note that even though they chose to use  $\tilde{\mathbf{A}}$ , any other matrix  $\mathbf{M}$  modeling a sensible propagation can be used. As long as  $\alpha$  is chosen small enough so that  $\alpha \mathbf{M}$  has a spectral radius smaller than 1, the process will converge and we have a closed form expression of the stationary distribution

$$\hat{\mathbf{y}} = (\mathbf{I} - \alpha \mathbf{M})^{-1} \mathbf{y} \quad (3.2)$$

We can see that  $(\mathbf{I} - \alpha \mathbf{M})^{-1}$  can be interpreted as a similarity measure, and using  $\mathbf{M} = \tilde{\mathbf{A}}$  yields

$$\mathbf{S}_{\text{LLGC}} = (\mathbf{I} - \alpha \tilde{\mathbf{A}})^{-1} \quad (3.3)$$

One can notice how close that is to the PPR and Laplacian exponential diffusion (substituting  $\tilde{\mathbf{A}}$  by  $\mathbf{P}$ , we get the PPR similarity measure).

### 3.2.2 Laplacian regularisation

Another way of looking at graph-based semi-supervised learning methods, although a little less intuitive, is through a minimization problem. We follow the presentation given in [3, 8]. The usual least square minimization encountered in supervised learning is given by

$$\text{LS}(\hat{\mathbf{y}}) = \sum_{i \in \mathcal{V}_l} (\hat{y}_i - y_i)^2 \quad (3.4)$$

This can be written in matrix form as

$$\text{LS}(\hat{\mathbf{y}}) = (\hat{\mathbf{y}} - \mathbf{y})^T \Gamma (\hat{\mathbf{y}} - \mathbf{y}) \quad (3.5)$$

where  $\Gamma$  is a diagonal matrix with diagonal entries  $\gamma_{ii} = 1$  if node  $i$  is labelled and  $\gamma_{ii} = 0$  otherwise.

If we were using this single term in our optimization problem, then the solution would be only determined using the labeled nodes, since the values of the unlabelled nodes are not taken into account in  $\text{LS}(\hat{\mathbf{y}})$ . Therefore, a regularization term is added, so that the unlabelled nodes are used. A natural way of doing so is by adding a smoothness penalization term, which will spur connected nodes to share the same label. Such a term is given by

$$\frac{1}{2} \sum_{i,j \in \mathcal{V}} a_{ij} (\hat{y}_i - \hat{y}_j)^2 \quad (3.6)$$

As we have seen in chapter 2 when introducing the graph matrices, this can be written in matrix form using the Laplacian matrix of the graph as

$$\hat{\mathbf{y}}^T \mathbf{L} \hat{\mathbf{y}} \quad (3.7)$$

We then obtain the total cost function by using both terms

$$C(\hat{\mathbf{y}}) = (\hat{\mathbf{y}} - \mathbf{y})^T \Gamma (\hat{\mathbf{y}} - \mathbf{y}) + \lambda \hat{\mathbf{y}}^T \mathbf{L} \hat{\mathbf{y}} \quad (3.8)$$

where the  $\lambda$  parameter was added to adjust the relative weight of the smoothness term with respect to the least square term.

The solution  $\hat{\mathbf{y}}^*$  of the minimization problem

$$\underset{\hat{\mathbf{y}}}{\text{minimize}} C(\hat{\mathbf{y}}) \quad (3.9)$$

is obtained by taking the gradient with respect to  $\hat{\mathbf{y}}$  and setting it to 0, and we get

$$\frac{\partial}{\partial \hat{\mathbf{y}}} ((\hat{\mathbf{y}} - \mathbf{y})^T \Gamma (\hat{\mathbf{y}} - \mathbf{y}) + \lambda \hat{\mathbf{y}}^T \mathbf{L} \hat{\mathbf{y}}) \Big|_{\hat{\mathbf{y}}^*} = 0 \quad (3.10)$$

$$\Leftrightarrow 2(\Gamma(\hat{\mathbf{y}}^* - \mathbf{y}) + \lambda \mathbf{L} \hat{\mathbf{y}}^*) = 0 \quad (3.11)$$

$$\Leftrightarrow (\Gamma + \lambda \mathbf{L}) \hat{\mathbf{y}}^* = \Gamma \mathbf{y} \quad (3.12)$$

$$\Leftrightarrow \hat{\mathbf{y}}^* = (\Gamma + \lambda \mathbf{L})^{-1} \Gamma \mathbf{y} \quad (3.13)$$

Because  $y_i$  is set to 0 if node  $i$  is unlabelled, we have that  $\mathbf{\Gamma}\mathbf{y} = \mathbf{y}$ , and the expression of  $\hat{\mathbf{y}}^*$  reduces to

$$\hat{\mathbf{y}}^* = (\mathbf{\Gamma} + \lambda\mathbf{L})^{-1}\mathbf{y} \quad (3.14)$$

We can see that the result shares a lot of resemblance with what was obtained using an iterative approach. The matrix  $(\mathbf{\Gamma} + \lambda\mathbf{L})^{-1}$  can also be interpreted as a similarity.

### 3.2.3 Dealing with large graphs

Having to work with large graphs puts severe constraints on which method can be used. The complexity of algorithms should be no more than linear in the number of edges if we want to treat graphs with hundreds of millions of edges (and now for the largest graphs, even being linear in the number of edges is not enough). The previous methods can all be seen as similarity measures, and they are gathered under the name of *sum of similarities* (indeed, the prediction is the sum of the similarities to labelled nodes). These methods are linear in the number of edges, and therefore scale well on large graphs. The method that we will present in the next chapter also lies in this category.

Even though they work well, one should know that there exists other semi-supervised learning methods for large graphs, that do not rely on an underlying similarity matrix. One example of this is given in [10], where they added a second quantity of confidence which acts as second order information behind the labels.





## 4

# Learning the similarity

In this chapter, we will present the flexible method introduced by Robin Devooght, Peter Staar and Costas Bekas. This section follows closely the presentation given in the thesis of the former [1, Chapter 4].

## 4.1 Motivation

The method is based on the following observation. Most similarity matrices that works well in practice can be expressed as a sum of powers. The one we have seen in the previous sections can be expressed as follows

$$\mathbf{S}_{\text{PPR}} = \sum_i ((1 - \alpha)\mathbf{P}^T)^i \quad (4.1)$$

$$\mathbf{S}_{\text{LLGC}} = \sum_i (\alpha\tilde{\mathbf{A}})^i \quad (4.2)$$

$$\mathbf{S}_{\text{exp}} = \sum_i \frac{(-\alpha\Delta)^i}{i!} \quad (4.3)$$

Since the spectral radius of  $\mathbf{P}$  and  $\tilde{\mathbf{A}}$  is inferior or equal to 1, when multiplied by a coefficient that is less than 1, it is clear that these sums converges (and the exponential is guaranteed to converge also). Therefore, after some  $k$  that is big enough, the series can be truncated and give a very precise approximation

$$\mathbf{S}_{\text{PPR}} \approx \sum_{i=1}^k ((1 - \alpha)\mathbf{P}^T)^i \quad (4.4)$$

$$\mathbf{S}_{\text{LLGC}} \approx \sum_{i=1}^k (\alpha\tilde{\mathbf{A}})^i \quad (4.5)$$

$$\mathbf{S}_{\text{exp}} \approx \sum_{i=1}^k \frac{(-\alpha\Delta)^i}{i!} = \sum_{i=1}^k c_i \tilde{\mathbf{A}}^i \quad (4.6)$$

One can remark that all those similarities are expressed as a sum of powers of  $\tilde{\mathbf{A}}$  and  $\mathbf{P}^T$ . The idea is to suppose the similarity matrix to take the following expression

$$\mathbf{S} = c_0 \mathbf{I} + \sum_{i=1}^k (c_i (\mathbf{P}^T)^i + c_{k+i} \tilde{\mathbf{A}}^i) \quad (4.7)$$

This general form covers all the similarity measures that can be expressed as a sum of powers of  $\mathbf{P}^T$  and  $\tilde{\mathbf{A}}$ . This includes as we have seen earlier the personalized PageRank, Learning with Local and Global Consistency and the exponential diffusion.

Now the question is, how do we find the optimal coefficient? Let us start with the binary classification problem. Let  $\mathcal{A}$  be a set of nodes belonging to a community. A perfect similarity measure for our problem would be such that

$$\begin{aligned} \forall j \in \mathcal{A}, \forall \mathcal{S} \subseteq \mathcal{A} \setminus j : \sum_{i \in \mathcal{S}} s_{ij} &= |\mathcal{S}| \\ \forall j \notin \mathcal{A}, \forall \mathcal{S} \subseteq \mathcal{A} : \sum_{i \in \mathcal{S}} s_{ij} &= 0 \end{aligned} \quad (4.8)$$

Therefore, we would like our similarity measure to be as close as possible to this.

In the binary classification problem, we only know a subset  $\mathcal{K}$  of the nodes belonging to the community, and we want to find the labels of the rest of the nodes in  $\bar{\mathcal{K}}$ . The labelling vector  $\mathbf{y}$  is defined as

$$y_i = \begin{cases} 1 & \text{if } i \in \mathcal{A} \cap \mathcal{K} \\ 0 & \text{if } i \in \bar{\mathcal{A}} \cap \mathcal{K} \\ 0 & \text{otherwise.} \end{cases} \quad (4.9)$$

If we take the similarity measure  $\mathbf{S}$  as the perfect one defined in 4.8 divided by  $|\bar{\mathcal{K}}|$ , then the prediction vector will be flawless. Indeed, the elements of  $\hat{\mathbf{y}} = \mathbf{S}\mathbf{y}$  will be  $\hat{y}_i = 1$  if  $i \in \mathcal{A} \cap \bar{\mathcal{K}}$  and  $\hat{y}_i = 0$  if  $i \notin \mathcal{A} \cap \bar{\mathcal{K}}$ . We will now see how it is implemented for the form (4.7).

## 4.2 Rationale of the method

We define the the  $n \times (2k + 1)$  matrix  $\mathbf{F}(\mathbf{y})$  as follows

$$\mathbf{F}(\mathbf{y}) = \left( \mathbf{y}, \mathbf{P}^T \mathbf{y}, \dots, (\mathbf{P}^T)^k \mathbf{y}, \tilde{\mathbf{A}} \mathbf{y}, \dots, \tilde{\mathbf{A}}^k \mathbf{y} \right) \quad (4.10)$$

Each column corresponds to one of the elements of the basis of similarity matrix multiplied by  $\mathbf{y}$ . With this notation, the prediction vector is given by

$$\hat{\mathbf{y}} = \mathbf{F}(\mathbf{y}) \mathbf{c} \quad (4.11)$$

where  $\mathbf{c}$  is the  $2k + 1$  column vector containing the coefficient of the expansion (4.7).

If we knew the complete labelling  $\mathbf{y}^*$ , the best coefficients would be the one that minimizes the SSE as follows

$$\underset{\mathbf{c}}{\text{minimize}} \|\mathbf{y}^* - \mathbf{F}(\mathbf{y})\mathbf{c}\|^2. \quad (4.12)$$

In practice, the complete labelling  $\mathbf{y}^*$  is not known. We have the incomplete labelling  $\mathbf{y}$ . Of course, we do not want to simply solve

$$\underset{\mathbf{c}}{\text{minimize}} \|\mathbf{y} - \mathbf{F}(\mathbf{y})\mathbf{c}\|^2. \quad (4.13)$$

as the trivial solution  $\mathbf{F}(\mathbf{y}) = \mathbf{y}$  would then be obtained, without helping us much. We resort to supervised learning techniques, where we hide the labels and find the best coefficients that recovers the hidden labels. Let  $\mathbf{y}^i$  be the labelling  $\mathbf{y}$  except for the  $i$ -th component set to 0. We would like to recover the label of  $\mathbf{y}^i$ , that is  $[\mathbf{F}(\mathbf{y}^i)]_i = y_i$ . We want this to be true for every node in  $\mathcal{K}$ , which leads to the following minimization problem.

$$\underset{\mathbf{c}}{\text{minimize}} \sum_{i \in \mathcal{K}} (y_i - [\mathbf{F}(\mathbf{y}^i)\mathbf{c}]_i)^2 + \lambda \|\mathbf{c}\|^2 \quad (4.14)$$

where  $y_i$  is the  $i$ -th component of the labelling vector, and we added the regularization term  $\lambda \|\mathbf{c}\|^2$  to avoid over-fitting.

If we are dealing with a large graph and a fairly big number of known labels, computing every  $\mathbf{F}(\mathbf{y}^i)$  might take some time. In order to speed up the process, we can hide several labels at the same time so that we have less  $\mathbf{F}$  matrices to compute. We split  $\mathcal{K}$  into  $\sigma$  disjoint sets  $\mathcal{S}_i$ , and we define  $\mathbf{y}^{\mathcal{S}_i}$  as the labelling  $\mathbf{y}$  where the elements whose index  $i \in \mathcal{S}_i$  are set to 0. We then obtain the following problem

$$\underset{\mathbf{c}}{\text{minimize}} \sum_i \sum_{j \in \mathcal{S}_i} (y_j - [\mathbf{F}(\mathbf{y}^{\mathcal{S}_i})\mathbf{c}]_j)^2 + \lambda \|\mathbf{c}\|^2 \quad (4.15)$$

One should note that this corresponds to a  $\sigma$ -fold cross-validation whereas the first one was a leave-one-out approach (that is a maximal  $k$ -fold cross-validation). To simplify the notations, we define  $\text{rows}_{\mathcal{S}}(\mathbf{M})$  to be the submatrix of a matrix  $\mathbf{M}$  formed by selecting the rows whose index belongs to  $\mathcal{S}$ . We then define

$$\mathbf{F} = \begin{pmatrix} \text{rows}_{\mathcal{S}_1}(\mathbf{F}(\mathbf{y}^{\mathcal{S}_1})) \\ \text{rows}_{\mathcal{S}_2}(\mathbf{F}(\mathbf{y}^{\mathcal{S}_2})) \\ \vdots \\ \text{rows}_{\mathcal{S}_\sigma}(\mathbf{F}(\mathbf{y}^{\mathcal{S}_\sigma})) \end{pmatrix} \quad (4.16)$$

and

$$\mathbf{Y} = \begin{pmatrix} \text{rows}_{\mathcal{S}_1}(\mathbf{y}) \\ \text{rows}_{\mathcal{S}_2}(\mathbf{y}) \\ \vdots \\ \text{rows}_{\mathcal{S}_\sigma}(\mathbf{y}) \end{pmatrix} \quad (4.17)$$

We can then write (4.15) in the following form:

$$\underset{\mathbf{c}}{\text{minimize}} \|\mathbf{Y} - \mathbf{F}\mathbf{c}\|^2 + \lambda \|\mathbf{c}\|^2 \quad (4.18)$$

We compute the gradient of the previous objective function to minimize with respect to  $\mathbf{c}$  and set it to 0 to get the optimal solution  $\mathbf{c}^*$ , and we obtain

$$\frac{\partial}{\partial \mathbf{c}} (||\mathbf{Y} - \mathbf{F}\mathbf{c}||^2 + \lambda ||\mathbf{c}||^2) \Big|_{\mathbf{c}^*} = 0 \quad (4.19)$$

$$\Leftrightarrow -2\mathbf{F}^T(\mathbf{Y} - \mathbf{F}\mathbf{c}^*) + 2\lambda\mathbf{c}^* = 0 \quad (4.20)$$

$$\Leftrightarrow (\mathbf{F}^T\mathbf{F} + \lambda\mathbf{I})\mathbf{c}^* = \mathbf{F}^T\mathbf{Y} \quad (4.21)$$

$$\Leftrightarrow \mathbf{c}^* = (\mathbf{F}^T\mathbf{F} + \lambda\mathbf{I})^{-1}\mathbf{F}^T\mathbf{Y} \quad (4.22)$$

We just have to resolve a  $(2k + 1)$  linear system of equations to get the optimal coefficients  $\mathbf{c}^*$  of the ridge regression. Furthermore, the nodes have a small radius of influence, meaning we can use a fairly small  $k$ , making the resolution of the linear system very simple.

The algorithm used to compute  $\mathbf{c}^*$  is given below. The bottleneck of the algorithm is the construction of the multiple  $\mathbf{F}(\mathbf{y}^{\mathcal{S}_i})$  matrices. Every column of each of these matrices is computed by multiplying one of the similarity matrices with a vector. Thus, the complexity is  $\mathcal{O}(\sigma k N)$ , with  $\sigma$  the number of splits  $\mathcal{S}_i$ ,  $k$  the number of powers used in the expansion (4.7) and  $N$  is the number of edges.

---

**Algorithm 1** Learning the similarity

---

```

1: Input:  $\mathbf{A}, \mathcal{K}, \mathbf{y}, \mathbf{c}_0, \lambda, \sigma, k$ 
2: Compute  $\tilde{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$  and  $\mathbf{P}^T = \mathbf{A}\mathbf{D}^{-1}$ 
3: Split  $\mathcal{K}$  into  $\sigma$  sets  $\mathcal{S}_i$ 
4: Initialize the  $|\mathcal{K}| \times (2k + 1)$  matrix  $\mathbf{F}$ 
5: for  $i = 1 \rightarrow \sigma$  do
6:   Compute  $\mathbf{y}^{\mathcal{S}_i}$ 
7:   for  $j = 1 \rightarrow k$  do
8:     Compute  $\tilde{\mathbf{A}}^j \mathbf{y}^{\mathcal{S}_i}$ 
9:      $\mathbf{F}(\mathcal{S}_i, j) \leftarrow \text{rows}_{\mathcal{S}_i}(\tilde{\mathbf{A}}^j \mathbf{y}^{\mathcal{S}_i})$ 
10:    Compute  $(\mathbf{P}^T)^j \mathbf{y}^{\mathcal{S}_i}$ 
11:     $\mathbf{F}(\mathcal{S}_i, j) \leftarrow \text{rows}_{\mathcal{S}_i}((\mathbf{P}^T)^j \mathbf{y}^{\mathcal{S}_i})$ 
12:   end for
13: end for
14: Compute  $\mathbf{Y}$ 
15: Compute  $\mathbf{M} \leftarrow \mathbf{F}^T\mathbf{F} + \lambda\mathbf{I}$ 
16: Compute  $\mathbf{b} \leftarrow \mathbf{F}^T\mathbf{Y}$ 
17: Solve  $\mathbf{M}\mathbf{c}^* = \mathbf{b}$ 
18: return  $\mathbf{c}^*$ 

```

---

## 4.3 Seed set expansion

In the seed set expansion, the previous set  $\mathcal{K}$  is only composed of positive labels. This means that  $\mathbf{Y}$  is a vector of one, and the solution to this will not contain useful information.

To proceed, we have to introduce negative examples artificially. Because the size of the communities are much smaller than the size of the whole network, one can make the approximation to consider all the nodes in  $\bar{\mathcal{K}}$  to be equal to 0. This works well in practice because it will be right for most of the nodes. We then have an additional term in the minimization problem that becomes

$$\underset{\mathbf{c}}{\text{minimize}} \quad \|\mathbf{Y} - \mathbf{F}\mathbf{c}\|^2 + \|\text{rows}_{\bar{\mathcal{K}}}(\mathbf{F}(\mathbf{y}))\mathbf{c}\|^2 + \lambda\|\mathbf{c}\|^2. \quad (4.23)$$

The solution is simply given by

$$\mathbf{c}^* = (\mathbf{F}^T\mathbf{F} + \text{rows}_{\bar{\mathcal{K}}}(\mathbf{F}(\mathbf{y}))^T\text{rows}_{\bar{\mathcal{K}}}\mathbf{F}(\mathbf{y}) + \lambda\mathbf{I})^{-1}\mathbf{F}^T\mathbf{Y}. \quad (4.24)$$

and the algorithm to solve this is exactly the same as before.

In the following, we will mainly focus on the resolution of such problems using the similarity learning.



## 5

# Implementation and results

We will discuss here the implementation in Python of the model presented in the last chapter, and present the results obtained on numerous datasets.

## 5.1 Python implementation

Python is an open source language, widely used in the IA and ML community. It has a lot of support and libraries useful to perform a lot of tasks related to those (notably NumPy for numerical computations of arrays and SciPy to handle sparse matrix and a whole lot of other functionalities). It is also a very straightforward and flexible language, making it appropriate for prototyping models, which is why we chose to use it when implementing our model.

The implementation of the learning model and the benchmark methods is briefly explained in the following sections.

### 5.1.1 The learning algorithm

When dealing with a graph composed of tens or hundreds of thousands of nodes, it is clear that we can not simply store the adjacency in a big array without saturating the RAM of a single computer. Fortunately, the *scipy.sparse* package handles nicely sparse matrix. We used the CSR format as it is the most efficient for matrix multiplication.

The first step is to compute the adjacency matrix  $\mathbf{A}$  in CSR format from the data (usually a .txt file filled with pairs of nodes). We then compute the two matrices  $\tilde{\mathbf{A}}$  and  $\mathbf{P}^T$  for our base of similarity matrices.

We can then compute the  $\mathbf{F}$  matrices needed for the algorithm given a labelling vector  $\mathbf{y}$ ,  $\tilde{\mathbf{A}}$ ,  $\mathbf{P}^T$  and the order of the expansion  $k$ . Note that we do not need to calculate the power of  $\tilde{\mathbf{A}}$  and  $\mathbf{P}^T$ , we just have to compute the columns incrementally, i.e. when we have the column  $\tilde{\mathbf{A}}^i \mathbf{y}$ , we multiply it by  $\tilde{\mathbf{A}}$  and get  $\tilde{\mathbf{A}}^{i+1} \mathbf{y}$ , which we do until we get to the  $k$ -th power.

```

1 def F_matrix(k, A, B, y):
2     """returns the F matrix to k-th degree from A_tilde, P and y
3     """
4     F = np.empty((len(y), 2*k+1))
5     F[:, 0] = y
6     x1 = A.dot(y)
7     F[:, 1] = x1
8     x2 = B.dot(y)
9     F[:, 2] = x2
10    for x in range(1, k):
11        x1 = A.dot(x1)
12        F[:, 2*x+1] = x1
13        x2 = B.dot(x2)
14        F[:, 2*x+2] = x2
15    return F

```

The group of interest is split in a 10% training set and a 90% test set using the *shuffle* function of the *random* package, and the incomplete labelling  $\mathbf{y}$  is generated. The last thing needed is the hidden labellings  $\mathbf{y}^{S_i}$ . For large groups, the leave-one-out approach considerably increases the computational time required, and there is no noticeable increase in performance, which is why we chose to use a 5-fold split. The rest of the algorithm is straightforward. The details of the code can be found in the appendix.

### 5.1.2 Benchmark methods

The benchmark methods used are the same than in the original work, there was no need to change this since they are still the most efficient large scale methods based on the sum of similarities, and furthermore the results can be compared to assess the validity of the implementation. These benchmarks were presented before, and are given by

#### Personalized PageRank

$$\mathbf{S}_{PPR} = (\mathbf{I} - (1 - \alpha)\mathbf{P}^T)^{-1} \quad (5.1)$$

#### Exponential diffusion

$$\mathbf{S}_{exp} = e^{-\alpha\Delta} \quad (5.2)$$

#### Learning with local and global consistency (LLGC)

$$\mathbf{S}_{LLGC} = (\mathbf{I} - \alpha\tilde{\mathbf{A}})^{-1} \quad (5.3)$$

For the PPR and LLGC, computing the inverse matrix is not possible. We then have two possibilities. The first is to compute the prediction vector  $\hat{\mathbf{y}}$  by solving the linear system

$$\mathbf{S}^{-1}\hat{\mathbf{y}} = \mathbf{y} \quad (5.4)$$



so that we do not have to compute the inverse. This can be done using the *spsolve* function of the SciPy library. Unfortunately, this was too heavy computationally as soon as the graphs gets moderately big. The second possibility is to truncate the Taylor expansion at the  $k$ -th order, which is what we did. We chose to stop the expansion at order 16, just like the similarity learning to be consistent. The precision starts plateauing around  $k = 10$  for all the methods anyway.

## 5.2 Results for the seed set expansion

### 5.2.1 Evaluation

To compare the results of the different methods, we used the 3 same metrics than those used originally.

#### Precision at 100 (prec100)

This is the fraction of true positives in the first 100 of the ranking. This metric tells us how precise we are at the beginning of the ranking, with no care about what happens after.

#### Accuracy (acc)

This is the fraction of true positives in the first  $k = |\bar{\mathcal{K}}|$  elements of the ranking. This gives us a more global view of how accurate the prediction is. Note that we need to know an estimate of the size of the community to compute this quantity. Even though it is a reasonable metric, it still does not take into account what happens behind the  $k$ -th element of the ranking.

#### Normalized Discounted Cumulated Gain (NDCG)

The NDCG considers the whole prediction into account. The higher up in the ranking the true positive is, the higher his contribution to the score. It is given by

$$\text{NDCG} = \frac{1}{\text{IDCG}} \sum_{i=1}^{n_{\text{test}}} \frac{2^{y_{\pi_i}} - 1}{\log(i + 1)} \quad (5.5)$$

where  $\pi_i$  is the index of the  $i$ -th element in the ranking, and IDCG is the ideal discounted cumulated gain for normalizing purpose, given by

$$\text{IDCG} = \sum_{i=1}^{|\bar{\mathcal{K}}|} \frac{1}{\log(i + 1)} \quad (5.6)$$

### 5.2.2 Datasets

We tried to use a diverse panel of datasets. Unfortunately, there is not a lot of large graphs with labelled communities, and so most of them were already used in the work of Robin Devooght.

	size	order	groups	type
BlogCatalog	10312	333983	29	Blogs
Flickr	80513	5899882	195	Image Hosting
DBLP	317080	1049866	149	Co-authorship
Amazon	334863	925872	295	Item sales
Youtube	1134890	2987624	136	Video Sharing

**Table 5.1:** *Datasets characteristics. DBLP, Amazon and YouTube provides from <http://snap.stanford.edu/data>, and BlogCatalog and Flickr from [http://leltang.net/social\\_dimension.html](http://leltang.net/social_dimension.html)*

BlogCatalog, Amazon, DBLP and YouTube originally had more communities with them, but we selected the ones that were above a given size (100 members for BlogCatalog and Youtube, 1000 members for Amazon and DBLP). The communities of DBLP are formed by research papers that were published in a given journal or conference. For Amazon, they are product categories, and YouTube, groups that are user created.

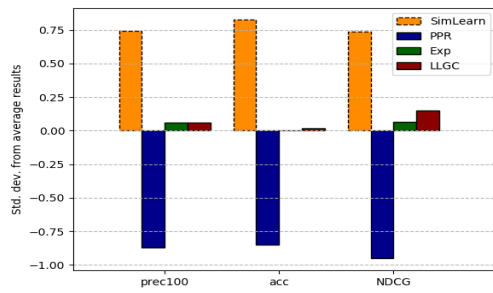
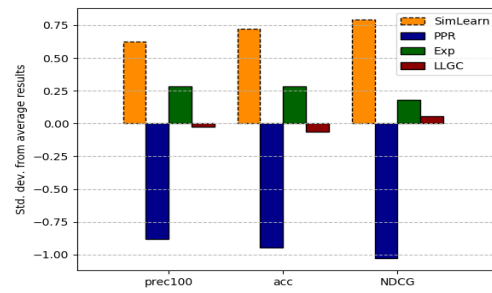
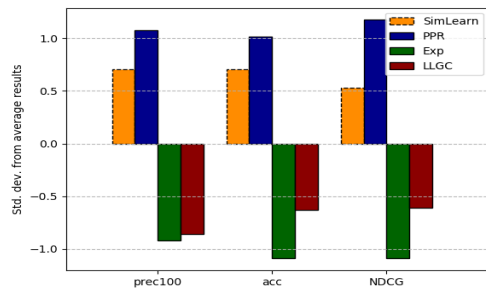
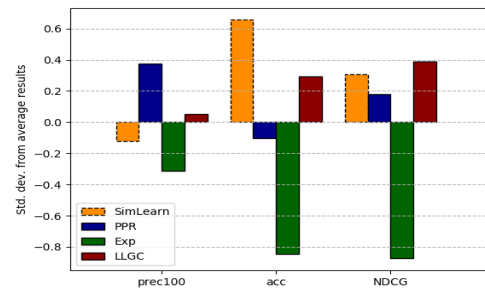
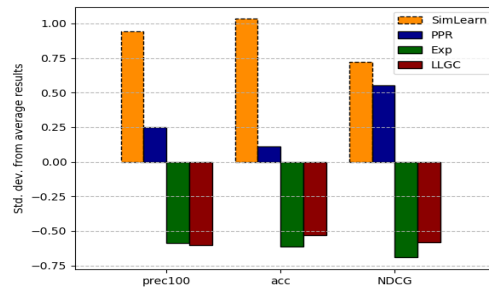
### 5.2.3 Methodology

For each dataset, 30 random communities are drawn from the data (29 for BlogCatalog and 15 for YouTube). Each community is treated as an independent seed set expansion problem. For one community, we randomly generate a seed set with a training ratio of 10%, compute the prediction with the similarity learning algorithm and the 3 benchmark methods, and evaluate the results with the 3 metrics that were presented above. This is performed 10 times, and then the results are averaged.

Because the results can vary a lot from community to community, one should be careful before averaging the results over different ones, because one "accident" of a method can influence a lot its total mean. One way to avoid this is to standardize the result on one community using the total mean and standard deviation of all the methods on this community, before averaging over the communities. This is what was done in the original work and has been reproduced here.

### 5.2.4 Results

The results are given in fig.5.1. We can see that globally, the similarity learning is outperforming by quite a margin the other methods. The only dataset where it was uniformly beaten by one of the benchmarks is DBLP, on which the personalized PageRank is performing better. Even though the PPR is better on one dataset, we can see that it performs horrendously on others such as Flickr and BlogCatalog, confirming how those preconstructed similarity measures can have fluctuating success, whereas the similarity learning is very steady in its performance across various types of datasets.

(a) *BlogCatalog*(b) *Flickr*(c) *DBLP*(d) *Amazon*(e) *YouTube*

**Figure 5.1:** Comparison of the similarity learning algorithm with 3 benchmark methods over 5 datasets. Results are standardized before being averaged over different communities.



## 6

# Improvement perspectives

In this chapter, we will present two approaches we tried to improve the results of the method.

### 6.1 Self-learning of the negative nodes

At the beginning of the seed set expansion, we decided to consider all labelled node as negatives. One idea is to use the first predictions resulting from the similarity learning method to make a more educated guess of the negative examples. Because the method is working well, we expect to have a fairly high concentration of false negative at the top of the ranking compared to the rest. Therefore, we tried removing from the negative samples the 50 highest ranked nodes, and perform the algorithm a second time with a slightly reduced penalty term.

This variation was tested following the same methodology than before. The only difference is that we directly compared the relative improvement and averaged over all communities. The results are given in [fig.6.1](#).

The results are mixed. We gained quite a bit of precision on DBLP, but this was lost on BlogCatalog and Flickr, while Amazon is unfazed by the change. This is surprising, because the `prec100` on Amazon is usually very high, meaning we almost only remove false negatives from the penalty term. Intuitively, one would think that we would initially gain some precision when doing so.

Note that this kind of self-learning techniques are known since a long time in machine learning. Recently, an analysis of a similar self-learning method has been performed in [11]. What they did is use a common label propagation technique and iteratively assigned new nodes in the seeds based on the ranking generated at each step. They had good results using those, which is surprising when we see the results we got when doing something fairly similar. Nonetheless, this could simply be because the similarity learning algorithm is already very optimized.

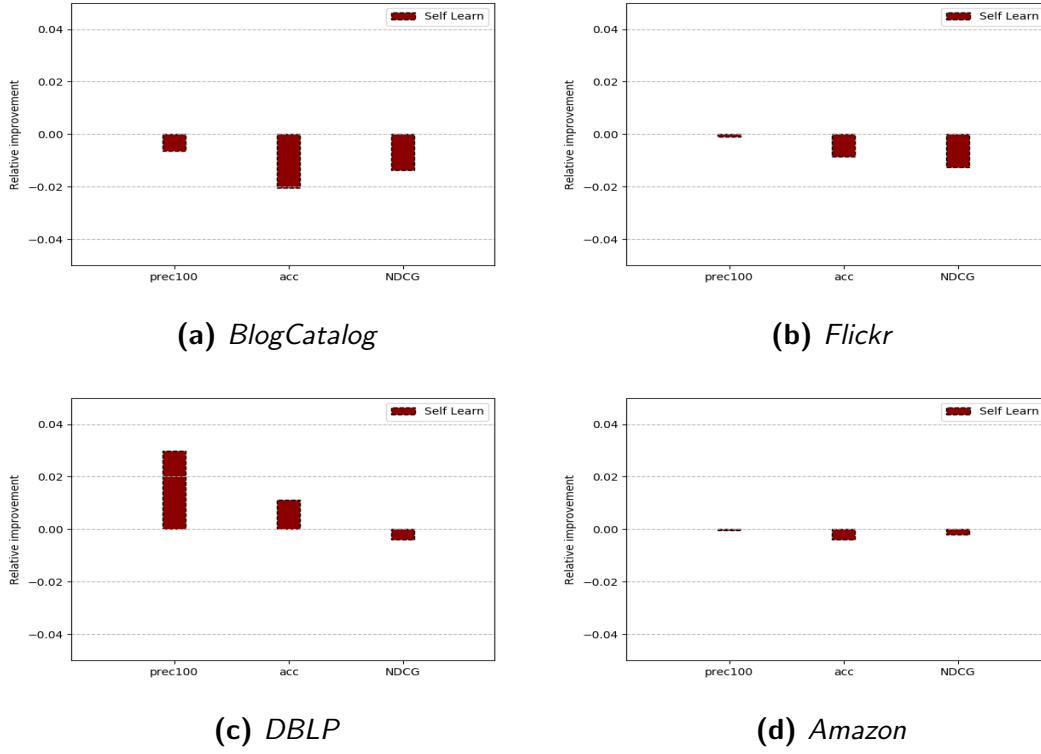


Figure 6.1: Relative improvement of the method after a one step self-learning on the negative nodes

## 6.2 Non linearity

Even though the method is very efficient and covers a wide family of similarity, one limitation is that it only considers linear propagation. One could ask himself if adding a small non-linearity in the propagation might help in the seed set expansion problem.

### 6.2.1 The approach

We tried to add a nonlinear propagation in the  $\mathbf{F}$  matrix. The total  $\mathbf{F}$  matrix becomes

$$\mathbf{F}_{\text{tot}}(\mathbf{y}) = (\mathbf{F}(\mathbf{y}), \mathbf{F}_{\text{nl}}(\mathbf{y})) \quad (6.1)$$

with  $\mathbf{F}(\mathbf{y})$  the usual linear part seen before, and the nonlinear part is given by

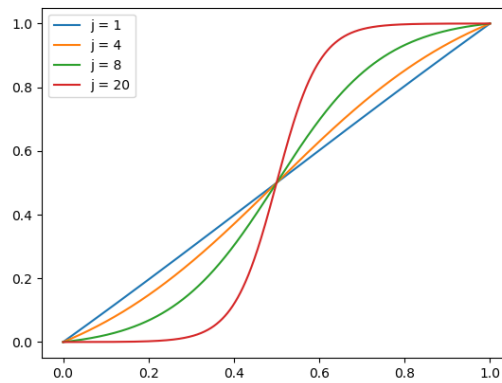
$$\mathbf{F}_{\text{nl}}(\mathbf{y}) = \left( (\mathbf{f} \circ \mathbf{P}^T)(\mathbf{y}), \dots, (\mathbf{f} \circ \mathbf{P}^T)^k \mathbf{y}, (\mathbf{f} \circ \tilde{\mathbf{A}})(\mathbf{y}), \dots, (\mathbf{f} \circ \tilde{\mathbf{A}})^k(\mathbf{y}) \right) \quad (6.2)$$

where  $\mathbf{f}(\mathbf{y})$  is a nonlinear vectorial function, the  $\circ$  denotes the composition of the functions and  $(f \circ \mathbf{M})^n$  is  $n$  iterations of the function  $(f \circ \mathbf{M})$ . The method stays exactly the same, we compute the  $4 \times k + 1$  coefficients  $\mathbf{c}^*$  that minimizes the squared error on the hidden labels predictions.

For  $\mathbf{f}$  we chose to use the sigmoid function applied to each element, that is

$$[\mathbf{s}(\mathbf{y})]_i = \frac{1}{1 + e^{-jy_i}} \quad (6.3)$$

where  $j$  is the parameter controlling the slope of  $s(x)$ . We translated the sigmoid so that when  $x = 0.5$ , the output is also equal to 0.5. This is given by  $s_t(x) = s(x - 0.5)$ . We then rescaled the function so that it is equal to 0 when  $x = 0$  and to 1 when  $x = 1$ . This can be done  $s_{ts}(x) = (s_t(x) - s_t(0))/(s_t(1) - s_t(0))$ . The shape of this function for different values of the parameter  $j$  is given in fig. 6.2.

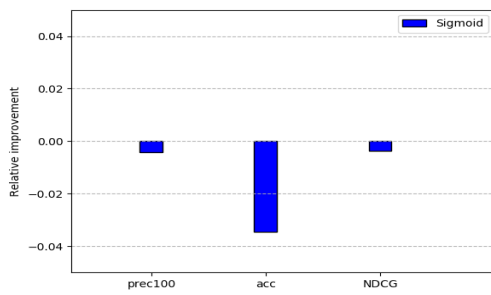
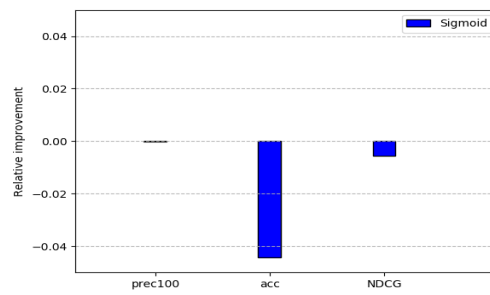
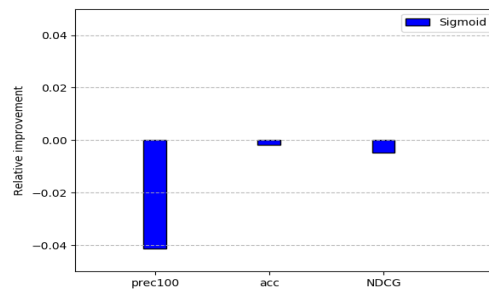


**Figure 6.2:** *The translated and rescaled sigmoid for different values of  $j$*

We chose a moderate slope of  $j = 8$  to perform the tests. We did the same procedure for 3 datasets than what was done in chapter 5, except that we averaged on the relative improvement compared to the standard algorithm.

### 6.2.2 Results

The results are given in the figures below. Even though the difference is not big, the addition of nonlinearities is counterproductive. There are still several leads that could be investigated. First, if nonlinearities can help, they surely will have a small effect, and we can expect their coefficient to be pretty small. Maybe using a stronger regularization coefficient just for the corresponding coefficients would be beneficial. Secondly, we could try other coefficients, or other nonlinear functions that might work better. Finally, the nonlinear propagation is completely separated from the linear part, combining them might lead to better results. One can note that this could also be done for the two matrices of the base, which are also each used separately from each other.

(a) *BlogCatalog*(b) *DBLP*(c) *Amazon***Figure 6.3:** *Relative improvement of the method with the addition of a nonlinearity*



## 7

# Conclusion

Let us remind that the goal of this thesis was the implementation of the 'Learning the similarity' method, reproduction of the promising results obtained with it, and if possible, improvements of the results by adding an original part.

We first started by introducing concepts related to graph and semi-supervised learning on those. A non-negligible part of this work has been reading and understanding how some of those methods work to better understand the similarity learning method.

The implementation was realized in Python, taking advantage of the numerous libraries that are available for such problems. The similarity learning algorithm and the benchmark methods are working as intended, and a comparison of the methods has been done on 5 real life datasets.

The results that were obtained confirmed the robustness of the method. Besides delivering good results, the similarity learning does not need any fine tuning of the parameters (and often when playing around with the model we actually worsen the results), scales well, and requires few computational time.

When trying to improve the results, the self-learning of negative samples gave some good results on the DBLP dataset, but slightly worsen the results for the 3 others, while the non-linearities had a detrimental effect on the results. Both methods could still be inspected in a lot of way, for example by tuning parameters involved in the method.

Another very interesting investigation would be to see if there is a way to meaningfully merge the two matrices intervening in the method, as they are propagating the labels completely independently.



# Bibliography

- [1] Robin Devooght. *Similarity measures on graphs and novel methods for collaborative filtering*. PhD thesis, Université Libre de Bruxelles, 2017.
- [2] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, August 2005.
- [3] François Fouss, Marco Saerens, and Masashi Shimbo. *Algorithms and Models for Network Data and Link Analysis*. Cambridge University Press, New York, NY, USA, 1st edition, 2016.
- [4] François Fouss, Kevin Francoise, Luh Yen, Alain Pirotte, and Marco Saerens. An experimental investigation of kernels on graphs for collaborative recommendation and semisupervised classification. *Neural Netw.*, 31:53–72, July 2012.
- [5] Jia-Yu Pan, Hyung-Jeong Yang, Christos Faloutsos, and Pinar Duygulu. Automatic multimedia cross-modal correlation discovery. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 653–658, New York, NY, USA, 2004. ACM.
- [6] Risi Imre Kondor and John D. Lafferty. Diffusion kernels on graphs and other discrete input spaces. In *Proceedings of the Nineteenth International Conference on Machine Learning*, ICML '02, pages 315–322, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.
- [7] How much data do we create every day? <https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/>.
- [8] Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien. *Semi-Supervised Learning*. The MIT Press, 1st edition, 2010.
- [9] Dengyong Zhou, Olivier Bousquet, Thomas Navin Lal, Jason Weston, and Bernhard Schölkopf. Learning with local and global consistency. In *Proceedings of the 16th International Conference on Neural Information Processing Systems*, NIPS'03, pages 321–328, Cambridge, MA, USA, 2003. MIT Press.
- [10] Matan Orbach and Koby Crammer. Graph-based transduction with confidence. volume 7524, pages 323–338, 09 2012.
- [11] Eliav Buchnik and Edith Cohen. Bootstrapped graph diffusions: Exposing the power of nonlinearity. *CoRR*, abs/1703.02618, 2017.

# Appendices



# A

## Python code

The code and the files used are available at my github <https://github.com/tdefoin/Similarity>.

### A.1 Learning the similarity module

```
1
2 import math
3 import numpy as np
4 import pandas as pd
5 import os
6 import scipy
7 import scipy.sparse
8 import copy
9 import random
10 import time
11 from time import perf_counter
12
13
14
15 def AP_from_A(A, n_nodes):
16     """returns the markovian transition and symetrically
17     normalised adjacency matrix from the edges data and number of
18     nodes"""
19
20     degrees = scipy.sparse.csr_matrix.sum(A, axis = 0)
21     degrees_inv = 1/degrees
22
23     D_inv = scipy.sparse.spdiags(degrees_inv, 0, n_nodes, n_nodes
24     , format = "csr")
25     D_inv_sqrt = scipy.sparse.spdiags(np.sqrt(degrees_inv), 0,
26     n_nodes, n_nodes, format = "csr")
27
28     A_tilde = D_inv_sqrt.dot(A).dot(D_inv_sqrt)
29     P_transpose = A.dot(D_inv)
```

```

25
26     return A_tilde , P_transpose
27
28
29
30 def get_group_indices(groups_list , group_number):
31     """from group edges, returns an array containing the indices
    corresponding to the group number selected"""
32     file = open(groups_list , 'r')
33     lines=file.readlines()
34     group_indices = np.fromstring(lines[group_number] , sep="\t" ,
    dtype = int)
35
36     return group_indices
37
38
39
40 def F_matrix(k, A, B, y):
41     """returns the F matrix to k-th degree from A_tilde, P and y
    """
42     F = np.empty((len(y) , 2*k+1))
43     F[:,0] = y
44     x1 = A.dot(y)
45     F[:,1] = x1
46     x2 = B.dot(y)
47     F[:,2] = x2
48     for x in range(1,k):
49         x1 = A.dot(x1)
50         F[:,2*x+1] = x1
51         x2 = B.dot(x2)
52         F[:,2*x+2] = x2
53
54     return F
55
56
57
58
59 def get_sim_score(A, n_nodes , groups , group_number , coef_regu ,
    coef_pen , k , n_split):
60     """returns predictions, indices rom the training group,
    indices from the testing group, the residuals and the
61     coefficients of the F matrix"""
62     AP = AP_from_A(A, n_nodes)
63     A_tilde = AP[0]
64     P = AP[1]
65     group = get_group_indices(groups , group_number)
66     random.shuffle(group)
67     group_split = np.array_split(group , n_split)

```

```

68     train_group = group_split[0]
69     test_group = np.delete(group, np.arange(len(train_group)))
70
71     target = np.zeros(n_nodes)
72     target[train_group]=1
73
74     nsplit = 5
75     seed_split = np.array_split(train_group, nsplit)
76     F = np.empty((n_nodes, 2*k+1))
77     F = F_matrix(k, A_tilde, P, target)
78     F_restricted = np.delete(F, train_group, axis = 0)
79
80     F_training = np.empty((len(train_group), 2*k+1))
81     count = 0
82     for x in range(nsplit):
83         hide_seed = copy.deepcopy(target)
84         hide_seed[seed_split[x]]=0
85         hide_F = F_matrix(k, A_tilde, P, hide_seed)
86         for i in range(len(seed_split[x])):
87             F_training[count+i,:] = hide_F[seed_split[x][
88 i],:]
89             count = count + len(seed_split[x])
90
91     Lambda = coef_regu
92     M = Lambda*np.identity(2*k+1) + np.transpose(F_training).dot(
93 F_training) + coef_pen*np.transpose(F_restricted).dot(F_restricted
94 )
95     b = np.transpose(F_training).dot(np.ones(len(train_group)))
96     c = np.linalg.solve(M, b)
97     y_hat = F.dot(c)
98     y_hat[train_group]=-np.inf
99
100     return y_hat, test_group, train_group, c

```

## A.2 Benchmark methods module

```

1 import math
2 import numpy as np
3 import pandas as pd
4 import os
5 import scipy
6 import scipy.sparse
7 import scipy.sparse.linalg
8 import copy
9 import random
10

```



```

11
12
13 def AP_from_A(A, n_nodes):
14     """returns the markovian transition and symetrically
    normalised adjacency matrix from the edges data and number of
    nodes"""
15     degrees = scipy.sparse.csr_matrix.sum(A, axis = 0)
16     degrees_inv = 1/degrees
17
18     D_inv = scipy.sparse.spdiags(degrees_inv, 0, n_nodes, n_nodes
    , format = "csr")
19     D_inv_sqrt = scipy.sparse.spdiags(np.sqrt(degrees_inv), 0,
    n_nodes, n_nodes, format = "csr")
20
21     A_tilde = D_inv_sqrt.dot(A).dot(D_inv_sqrt)
22     P_transpose = A.dot(D_inv)
23
24     return A_tilde, P_transpose
25
26
27
28 def get_group(groups_list, group_number):
29     """from group edges, returns an array containing the indices
    corresponding to the group number selected"""
30     file = open(groups_list, 'r')
31     lines=file.readlines()
32     group_indices = np.fromstring(lines[group_number], sep="\t",
    dtype = int)
33     return group_indices
34
35
36
37 def get_PPR_score(A, n_nodes, groups, group_number, taylor_order,
    n_split):
38
39     alpha = np.array([0.05,0.1,0.15,0.2])
40     AP = AP_from_A(A, n_nodes)
41     P = AP[1]
42     group = get_group(groups, group_number)
43     random.shuffle(group)
44
45     group_split = np.array_split(group, n_split)
46
47     #5-fold split for the cross validation
48     split_cv = np.array_split(group_split[0], 5)
49
50     target = np.zeros(n_nodes)
51     target[group_split[0]]=1

```

```

52
53     #Compute the NDCG for each hidden 5-fold for every parameter in
range alpha and chose the best
54     NDCG_mean = np.empty((len(alpha)))
55     for i in range(len(alpha)):
56         NDCG = np.empty(5)
57         for j in range(5):
58             y_cv = copy.deepcopy(target)
59             y_cv[split_cv[j]] = 0
60             taylor_k = copy.deepcopy(y_cv)
61             for z in range(taylor_order):
62                 taylor_k = ((1-alpha[i])*P).dot(taylor_k)
63                 y_cv = y_cv + taylor_k
64             y_cv[split_cv[(j+1)%5]]=-np.inf #set to -infinity so that
known labels are at the end of the ranking
65             y_cv[split_cv[(j+2)%5]]=-np.inf
66             y_cv[split_cv[(j+3)%5]]=-np.inf
67             y_cv[split_cv[(j+4)%5]]=-np.inf
68             x = np.arange(len(split_cv[j]))
69             IDCG = np.sum(1/np.log(x+2))
70             sum_NDCG = 0
71             z1 = np.empty(len(split_cv[j]), dtype = int)
72             z2 = np.argsort(-y_cv)
73             for z in range(len(split_cv[j])):
74                 z1[z] = np.argwhere(z2 == split_cv[j][z])
75             sum_NDCG = np.sum(1/np.log(z1+2))
76             NDCG[j] = sum_NDCG/IDCG
77         NDCG_mean[i] = np.mean(NDCG)
78     alpha_cv = alpha[np.argmax(NDCG_mean)]
79     print(alpha_cv)
80
81     y_hat = copy.deepcopy(target)
82     taylor_k = copy.deepcopy(target)
83     for i in range(taylor_order):
84         taylor_k = ((1-alpha_cv)*P).dot(taylor_k)
85         y_hat = y_hat + taylor_k
86
87     test_group = np.delete(group,np.arange(len(group_split[0])))
88     y_hat[group_split[0]]=-np.inf
89
90     return y_hat, test_group, group_split[0]
91
92
93
94 def get_exp_score(A, n_nodes, groups, group_number, taylor_order,
n_split):
95
96     alpha = np.array([0.1,0.5,1,2,10])

```

```

97 AP = AP_from_A(A, n_nodes)
98 A_tilde = AP[0]
99 Delta = scipy.sparse.identity(n_nodes, format = "csr") - A_tilde
100 group = get_group(groups, group_number)
101 random.shuffle(group)
102
103 group_split = np.array_split(group, n_split)
104
105 split_cv = np.array_split(group_split[0], 5)
106
107 target = np.zeros(n_nodes)
108 target[group_split[0]] = 1
109
110 NDCG_mean = np.empty((len(alpha)))
111 for i in range(len(alpha)):
112     NDCG = np.empty(5)
113     for j in range(5):
114         y_cv = copy.deepcopy(target)
115         y_cv[split_cv[j]] = 0
116         taylor_k = copy.deepcopy(y_cv)
117         for z in range(taylor_order):
118             taylor_k = (-(alpha[i]/(z+1))*Delta).dot(taylor_k)
119             y_cv = y_cv + taylor_k
120             y_cv[split_cv[(j+1)%5]] = -np.inf
121             y_cv[split_cv[(j+2)%5]] = -np.inf
122             y_cv[split_cv[(j+3)%5]] = -np.inf
123             y_cv[split_cv[(j+4)%5]] = -np.inf
124             x = np.arange(len(split_cv[j]))
125             IDCG = np.sum(1/np.log(x+2))
126             sum_NDCG = 0
127             z1 = np.empty(len(split_cv[j]), dtype = int)
128             z2 = np.argsort(-y_cv)
129             for z in range(len(split_cv[j])):
130                 z1[z] = np.argmax(z2 == split_cv[j][z])
131             sum_NDCG = np.sum(1/np.log(z1+2))
132             NDCG[j] = sum_NDCG/IDCG
133     NDCG_mean[i] = np.mean(NDCG)
134 alpha_cv = alpha[np.argmax(NDCG_mean)]
135 print(alpha_cv)
136
137 y_hat = copy.deepcopy(target)
138 taylor_k = copy.deepcopy(target)
139 for i in range(taylor_order):
140     taylor_k = (-(alpha_cv/(i+1))*Delta).dot(taylor_k)
141     y_hat = y_hat + taylor_k
142
143 test_group = np.delete(group, np.arange(len(group_split[0])))
144 y_hat[test_group] = -np.inf

```

```

145
146     return y_hat, test_group, group_split[0]
147
148
149
150 def get_LLGC_score(A, n_nodes, groups, group_number, taylor_order,
151                    n_split):
152     alpha = np.array([0.1, 0.3, 0.5, 0.7, 0.9])
153     AP = AP_from_A(A, n_nodes)
154     A_tilde = AP[0]
155     group = get_group(groups, group_number)
156     random.shuffle(group)
157
158     group_split = np.array_split(group, n_split)
159
160     split_cv = np.array_split(group_split[0], 5)
161
162     target = np.zeros(n_nodes)
163     target[group_split[0]] = 1
164
165     NDCG_mean = np.empty((len(alpha)))
166     for i in range(len(alpha)):
167         NDCG = np.empty(5)
168         for j in range(5):
169             y_cv = copy.deepcopy(target)
170             y_cv[split_cv[j]] = 0
171             taylor_k = copy.deepcopy(y_cv)
172             for z in range(taylor_order):
173                 taylor_k = (alpha[i] * A_tilde).dot(taylor_k)
174                 y_cv = y_cv + taylor_k
175             y_cv[split_cv[(j+1)%5]] = -np.inf
176             y_cv[split_cv[(j+2)%5]] = -np.inf
177             y_cv[split_cv[(j+3)%5]] = -np.inf
178             y_cv[split_cv[(j+4)%5]] = -np.inf
179             x = np.arange(len(split_cv[j]))
180             IDCG = np.sum(1/np.log(x+2))
181             sum_NDCG = 0
182             z1 = np.empty(len(split_cv[j]), dtype=int)
183             z2 = np.argsort(-y_cv)
184             for z in range(len(split_cv[j])):
185                 z1[z] = np.argmax(z2 == split_cv[j][z])
186             sum_NDCG = np.sum(1/np.log(z1+2))
187             NDCG[j] = sum_NDCG/IDCG
188         NDCG_mean[i] = np.mean(NDCG)
189     alpha_cv = alpha[np.argmax(NDCG_mean)]
190     print(alpha_cv)
191

```

```

192     y_hat = copy.deepcopy(target)
193     taylor_k = copy.deepcopy(target)
194     for i in range(taylor_order):
195         taylor_k = (alpha_cv*A_tilde).dot(taylor_k)
196         y_hat = y_hat + taylor_k
197
198     test_group = np.delete(group,np.arange(len(group_split[0])))
199     y_hat[group_split[0]]=-np.inf
200
201     return y_hat , test_group , group_split[0]

```

## A.3 Main function to generate the seed set expansion tests

```

1 import math
2 import numpy as np
3 import pandas as pd
4 import scipy
5 import scipy.sparse
6 import scipy.sparse.linalg
7 import copy
8 import random
9 import importlib
10 import sim_learning
11 import benchmark_methods_cross
12 import time
13 import sys
14 import random
15 import matplotlib.pyplot as plt
16 from time import perf_counter
17
18
19
20 def A_from_edges(data_edges , n_nodes):
21
22     nedges = len(data_edges)
23     row_ind = np.zeros(2*nedges , dtype = int)
24     col_ind = np.zeros(2*nedges , dtype = int)
25
26     row_ind[:nedges] = data_edges[:,0]
27     row_ind[nedges:] = data_edges[:,1]
28     col_ind[:nedges] = data_edges[:,1]
29     col_ind[nedges:] = data_edges[:,0]
30

```

```

31     A_csr = scipy.sparse.csr_matrix((np.ones(2*nedges, dtype =
float),(row_ind,col_ind)), shape=(n_nodes,n_nodes))
32     return A_csr
33
34
35
36 def get_prec100(results, test_group):
37
38     in_group= np.empty(shape = 0, dtype = bool)
39     for i in np.argsort(-results)[:100]:
40         in_group = np.append(in_group, i in test_group)
41     return np.mean(in_group)
42
43
44
45 def get_accuracy(results, test_group):
46
47     in_group= np.empty(shape = 0, dtype = bool)
48     for i in np.argsort(-results)[:len(test_group)]:
49         in_group = np.append(in_group, i in test_group)
50     accuracy = np.mean(in_group)
51     return np.mean(in_group)
52
53
54
55 def get_NDCG(results, test_group):
56
57     x = np.arange(len(test_group))
58     IDCG = np.sum(1/np.log(x+2))
59     sum_NDCG = 0
60     y = np.empty(len(test_group), dtype = int)
61     z = np.argsort(-results)
62     for i in range(len(test_group)):
63         y[i] = np.argmax(z == test_group[i])
64     sum_NDCG = np.sum(1/np.log(y+2))
65     NDCG = sum_NDCG/IDCG
66     return NDCG
67
68
69
70 number_nodes=10312
71 edges = np.empty([333983,2], dtype=int)
72 file = open(r'Datasets\blogcat_ungraph.txt', 'r')
73 row = 0
74 for line in file:
75     edges[row] = np.fromstring(line, sep="\t")
76     row += 1
77 groups = r'Datasets\blogcat_contiguous_large_groups.txt'

```

```

78 A = A_from_edges(edges, number_nodes)
79
80
81
82 group_sel=np.arange(29)
83 random.shuffle(group_sel)
84 n_groups = 29
85 prec100= np.empty((n_groups,10,4))
86 accuracy= np.empty((n_groups,10,4))
87 NDCG= np.empty((n_groups,10,4))
88 for i in range(n_groups):
89     for j in range(10):
90         results = sim_learning.get_sim_score(A, number_nodes,
91         groups, group_sel[i], 1e-02, 1, 16, 10)
92         prec100[i,j,0] = get_prec100(results[0], results[1])
93         accuracy[i,j,0] = get_accuracy(results[0], results[1])
94         NDCG[i,j,0] = get_NDCG(results[0], results[1])
95         results = benchmark_methods_cross.get_PPR_score(A,
96         number_nodes, groups, group_sel[i], 1e-02, 1, 16, 10)
97         prec100[i,j,1] = get_prec100(results[0], results[1])
98         accuracy[i,j,1] = get_accuracy(results[0], results[1])
99         NDCG[i,j,1] = get_NDCG(results[0], results[1])
100         results = benchmark_methods_cross.get_exp_score(A,
101         number_nodes, groups, group_sel[i], 1e-02, 1, 16, 10)
102         prec100[i,j,2] = get_prec100(results[0], results[1])
103         accuracy[i,j,2] = get_accuracy(results[0], results[1])
104         NDCG[i,j,2] = get_NDCG(results[0], results[1])
105         results = benchmark_methods_cross.get_LLGC_score(A,
106         number_nodes, groups, group_sel[i], 1e-02, 1, 16, 10)
107         prec100[i,j,3] = get_prec100(results[0], results[1])
108         accuracy[i,j,3] = get_accuracy(results[0], results[1])
109         NDCG[i,j,3] = get_NDCG(results[0], results[1])
110     print(i)
111
112 result_prec100 = np.empty((4,n_groups))
113 for i in range(n_groups):
114     mean = np.mean([np.mean(prec100[i,:,1]), np.mean(prec100[i
115     ,:,2]), np.mean(prec100[i,:,3]), np.mean(prec100[i,:,4])])
116     std = np.std([prec100[i,:,1], prec100[i,:,2], prec100[i,:,3],
117     prec100[i,:,4]])
118     result_prec100[0,i] = (np.mean(prec100[i,:,1])–mean)/std
119     result_prec100[1,i] = (np.mean(prec100[i,:,2])–mean)/std
120     result_prec100[2,i] = (np.mean(prec100[i,:,3])–mean)/std
121     result_prec100[3,i] = (np.mean(prec100[i,:,4])–mean)/std
122 score_prec100 = np.empty(4)
123 for i in range(4):
124     score_prec100[i]=np.mean(result_prec100[i,:])
125

```

```

120 result_acc = np.empty((4, n_groups))
121 for i in range(n_groups):
122     mean = np.mean([np.mean(accuracy[i, :, 1]), np.mean(accuracy[i, :, 2]),
123                     np.mean(accuracy[i, :, 3]), np.mean(accuracy[i, :, 4])])
124     std = np.std([accuracy[i, :, 1], accuracy[i, :, 2], accuracy[i, :, 3],
125                  accuracy[i, :, 4]])
126     result_acc[0, i] = (np.mean(accuracy[i, :, 1]) - mean) / std
127     result_acc[1, i] = (np.mean(accuracy[i, :, 2]) - mean) / std
128     result_acc[2, i] = (np.mean(accuracy[i, :, 3]) - mean) / std
129     result_acc[3, i] = (np.mean(accuracy[i, :, 4]) - mean) / std
130 score_acc = np.empty(4)
131 for i in range(4):
132     score_acc[i] = np.mean(result_acc[i, :])
133
134 result_NDCG = np.empty((4, n_groups))
135 for i in range(n_groups):
136     mean = np.mean([np.mean(NDCG[i, :, 1]), np.mean(NDCG[i, :, 2]), np.
137                     mean(NDCG[i, :, 3]), np.mean(NDCG[i, :, 4])])
138     std = np.std([NDCG[i, :, 1], NDCG[i, :, 2], NDCG[i, :, 3], NDCG[i, :, 4]])
139     result_NDCG[0, i] = (np.mean(NDCG[i, :, 1]) - mean) / std
140     result_NDCG[1, i] = (np.mean(NDCG[i, :, 2]) - mean) / std
141     result_NDCG[2, i] = (np.mean(NDCG[i, :, 3]) - mean) / std
142     result_NDCG[3, i] = (np.mean(NDCG[i, :, 4]) - mean) / std
143 score_NDCG = np.empty(4)
144 for i in range(4):
145     score_NDCG[i] = np.mean(result_NDCG[i, :])
146
147 simlearn = (score_prec100[0], score_acc[0], score_NDCG[0])
148 PPR = (score_prec100[1], score_acc[1], score_NDCG[1])
149 Exp = (score_prec100[2], score_acc[2], score_NDCG[2])
150 LLGC = (score_prec100[3], score_acc[3], score_NDCG[3])
151 index = np.arange(3)
152 bar_width = 0.2
153
154 fig, ax = plt.subplots()
155 plt.xlim(-0.7, 3)
156 rec1 = plt.bar(index - bar_width, simlearn, width=bar_width, bottom=
157     None, color = 'darkorange', label = 'SimLearn',
158     edgecolor = 'black', linewidth = 1, linestyle = '-')
159 rec2 = plt.bar(index, PPR, width=bar_width, bottom=None, color = '
160     darkblue', label = 'PPR',
161     edgecolor = 'black', linewidth = 1, linestyle = '-')
162 rec3 = plt.bar(index + bar_width, Exp, width=bar_width, bottom=None,
163     color = 'darkgreen', label = 'Exp',
164     edgecolor = 'black', linewidth = 1, linestyle = '-')

```



```
160 rec4 = plt.bar(index+2*bar_width, LLGC, width=bar_width, bottom=None,  
    color = 'darkred', label = 'LLGC',  
161 edgecolor = 'black', linewidth = 1, linestyle = '-')  
162 plt.xticks(index + bar_width/2, ('prec100', 'acc', 'NDCG'))  
163 plt.ylabel('Std. dev. from average results')  
164 plt.grid(axis = 'y', linestyle = '—')  
165 plt.legend(loc = 'bottom left')  
166 plt.show()
```