Large-Scale Spectral Clustering on Graphs

Gautier Appert
Master 2 M.V.A.
E.N.S. Cachan
gappert@ens-cachan.fr

Guillaume Salha
Master 2 M.V.A.
E.N.S. Cachan
gsalha@ens-cachan.fr

Abstract

Spectral Clustering has become a widely used technique to detect complex shapes and manifold structure on graph data. In the past few years, many previous works on this method reported good experimental results on some challenging clustering problems. However, as data grows in scale, which is common in today's era of Big Data, the classical Spectral Clustering algorithm becomes almost impossible to apply because of its computational cost. In this paper, we provide a review of some of the main methods to tackle this issue. We focus on the two costly aspects of the algorithm: the construction of the graph itself from vectorial data and the eigendecomposition of the corresponding Laplacian matrix. Then, we propose an implementation of large-scale Spectral Clustering, both on artificial and real data.

1 Introduction

Graphs are useful models whenever we deal with relations between objects. A lot of real-life situations can be represented as vertices linked by edges, from - social, biological, computer - networks to flat image data. In this paper, we work on weighted undirected graphs, where weights on edges are interpreted as similarity measures between vertices.

Finding good clusters between these vertices has been the objective of considerable research in machine learning. Applications include image segmentation, community detection in social networks and web pages categorization. Spectral Clustering [Shi and Malik, 2000; Ng *et al.*, 2001] has become a popular method to perform such clustering, providing results that often outperform traditional algorithms such as k-means [Ng *et al.*, 2001; Luxburg, 2007], while still being quite simple to implement. This is not surprising since Spectral Clustering takes into account the graph structure and connectivity of edges, whereas k-means only attempts to determine compact clusters. The first approach is more relevant to detect complex shapes and manifold structure on graph data. This phenomenon is once again highlighted in Section 2 of this paper.

The main drawback of this method is its computational cost, which makes it almost impossible to apply directly when data becomes too large. Two aspects of the algorithm are costly. Firstly, we need to construct the graph itself, and classical methods have quadratic complexities. Moreover, as we explain in the following section, we need to extract some eigenvectors from the so-called Laplacian matrix of the graph to perform Spectral Clustering. This eigendecomposition step has cubic complexity in the worst case. This is a real problem since large graphs are quite common nowadays. Graphs modeling people and their interactions from social networks such as Facebook, Twitter or LinkedIn are good examples of situations where the number of vertices explodes.

In the remaining of this paper, we provide an overview of some of the main methods to tackle this issue. General ideas behind Spectral Clustering and its complexity are developed in Section 2. In Section 3 we explain how to construct large graphs, while Section 4 is dedicated to the eigendecomposition of the Laplacian. Finally, we provide implementations in Section 5, on artificial data but also on real data from Reuters, and we conclude in Section 6.

2 Spectral Clustering

In this section, we briefly introduce the Spectral Clustering algorithm [Shi and Malik, 2000; Ng *et al.*, 2001] and explain why we face problems in high dimension.

2.1 Graph construction

We assume that we work with vectorial data i.e. that the graph itself has to be constructed. A prerequisite to build a similarity graph is to define a similarity function to score the distance between the n vertices in a graph. The following inverse exponential function is a common measure [Luxburg, 2007], controlled by the Euclidean distance:

$$w_{i,j} = exp\Big\{ -\frac{||x_i - x_j||^2}{2\sigma^2} \Big\}$$

x_i and x_j denote vertices i and j, while σ^2 controls the bandwidth of the similarity and has to be determined. The weighted adjacency matrix of the graph is $W = (w_{i,j})_{i,j=1,...,n}$. If $w_{ij} = 0$ vertices i and j are not connected in the graph. Moreover, W is symmetric because the graph is undirected.

Several representations of similarity between vertices are possible. We could simply work with the previous matrix W. However, it is often more convenient to set some weights to zero and only keep local similarities. From a graphical point of view, we significantly decrease the number of edges in the graph. The two main methods are:

- k-nearest neighbors graphs: the weight between vertices i and j is $w_{i,j}$ if i is one of the k nearest vertices of j in the sense of the similarity measure, or if j is one of the k nearest vertices of i, and 0 otherwise ;
- ε-neighborhood graphs: we connect all vertices whose pairwise similarity is greater than ε (i.e. whose Euclidean distance is lower than some fixed threshold) with weight $w_{i,j}$, and we set remaining similarities to 0.

We clearly observe that these two approaches are quite different. For instance, the (mutual) k-nn method is more adapted to construct a connected graph when vertices are in different regions of the space, since it can connect points on different scales. More generally, constructing a "good" similarity graph is not a trivial task. More theoretical details on how to choose the similarity function itself, but also the construction method and parameters σ^2, k or ε are presented in [Luxburg, 2007].

The interesting aspect for our work is that we could not scale these classical graph construction algorithms to large graphs. Building a k-nn or a ε graph makes no difference, since both algorithms have quadratic complexities. It indeed takes $O(n^2p)$ time to compute the k-nn graph or ε graph (i.e. the weighted adjacency matrix) for n data points in the p dimensional Euclidean space [Liu *et al.*, 2013]. Actually, we could have used the fact that the similarity matrix is symmetric to reduce the total number of operations. Nevertheless, the complexity would still be have been quadratic and would still have led to issues for large graphs.

2.2 Laplacian matrix and eigenvectors

The degree matrix D is defined as the $n \times n$ diagonal matrix with degrees $D_{i,i} = \sum_{j=1}^{n} w_{i,j}$ for $i = 1, ..., n$. From this matrix, we define the unormalized graph Laplacian matrix as:

$$L := D - W.$$

A review of its properties is given in [Mohar, 1997]. Among them, L is symmetric and positive semi-definite. If the graph is connected, the smallest eigenvalue of L is 0, associated to the constant one eigenvector. Otherwise, the multiplicity of the eigenvalue 0 is equal to the number of connected components in the graph. Its eigenspace is spanned by indicator vectors of these components.

In machine learning community, Spectral Clustering has been made popular by [Shi and Malik, 2000], [Meila and Shi, 2001] and [Ng *et al.*, 2001], even if we retrieve some key ideas in older works such as [Donath and Hoffman, 1973]. We state here the most common Spectral Clustering algorithm on L, as presented in [Luxburg, 2007]. The algorithm proceeds as follows :

- the first step is to construct the graph, as before, and derive the corresponding unormalized Laplacian L ;
- then, we compute the k smallest eigenvectors $u_1, ..., u_k$ of L, where k denotes the number of clusters ;
- if $U \in \mathbb{R}^{n \times k}$ is the matrix containing eigenvectors as columns, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the i-th row of U, for $i = 1, ..., n$. We cluster $(y_i)_{i=1,...,n}$ with the k-means algorithm, into k clusters C_1,...,C_k ;
- we finally obtain clusters A_1,...,A_k with $A_i = \{j | y_j \in C_i\}$.

Some versions of the algorithm replace L by so-called normalized Laplacians. [Ng *et al.*, 2001] compute the k first eigenvectors of the symmetric Laplacian L_{sym}, while [Shi and Malik, 2000] work with the random-walk Laplacian L_{rw} (which is equivalent to compute eigenvectors of the generalized eigenproblem $Lu = \lambda Du$), with:

$$L_{sym} := D^{-1/2} L D^{-1/2} \qquad L_{rw} := D^{-1} L$$

We could wonder which of the three Laplacians should be used to compute eigenvectors. [Luxburg, 2007] advices to look at the degree distribution of the similarity graph. If the graph seems regular, all Laplacians should work equally well for clustering. In the opposite case, the article provides several arguments in favor of normalized Laplacians rather than unormalized, and in favor of L_{rw} rather than L_{sym}.

One main argument comes from the graph partitioning point of view. Indeed, normalized and unormalized Spectral Clustering can be seen as solutions of the relaxation of two NP hard optimisation problems [Wagner and Wagner, 1993] of graph partitioning, respectively called RatioCut and NCut. While both problems seek to minimize the between-cluster similarities, only NCut seeks to maximize at the same time the within-cluster similarities. Since unormalized spectral clustering only implements the first objective, it might be less relevant for some complex shapes in graph data. A detailed discussion on these optimisation problems and on Laplacian matrix choice is provided in [Luxburg, 2007], but also in [Shi and Malik, 2000; Ng *et al.*, 2001; Bach and Jordan, 2006].

In both cases, computing the eigendecomposition of the Laplacian is a real bottleneck from a computational point of view. It requires $O(n^3)$ time to extract eigenvectors, in the naive way [Liu *et al.*, 2013]. Therefore our main problem in this paper is to find methods to accelerate this eigendecomposition step, in order to be able to perform large-scale Spectral Clustering. It comes in addition to the previous problem of large graph construction.

2.3 Example on Two-Moons graphs

We quickly provide an illustration of Spectral Clustering on an interesting graph structure, that we also use in Section 5. We simulated on MATLAB 200 two-dimensional points. The two true clusters correspond to the two "moons". We observe in Figure 1 assignments from Spectral Clustering and from the k-means algorithm. We used the k-nn method to connect the graph with an inverse exponential similarity, with $k = 10$ and $\sigma^2 = 0.1$. The graph is connected.

We show that - unormalized - Spectral Clustering provides a perfect clustering, while k-means fails to identify the structure of vertices, that are not "compact". Spectral Clustering is therefore more appropriate than k-means in this fashion where it is important to study the connectivity of vertices.

We computed the clustering assigments from the two first eigenvectors. In fact, only the second eigenvector is useful, since the first one - associated with the eigenvalue 0 - is simply the normalized constant one eigenvector. This second eigenvector can be seen as a kind of indicator function, that will take "high" values when a vertex comes from one of the two clusters, and "low" values otherwise.

Results were almost immediate on our laptop. However, we will saw in Section 5 that we need around 5 minutes to obtain the same result with $n = 10000$.

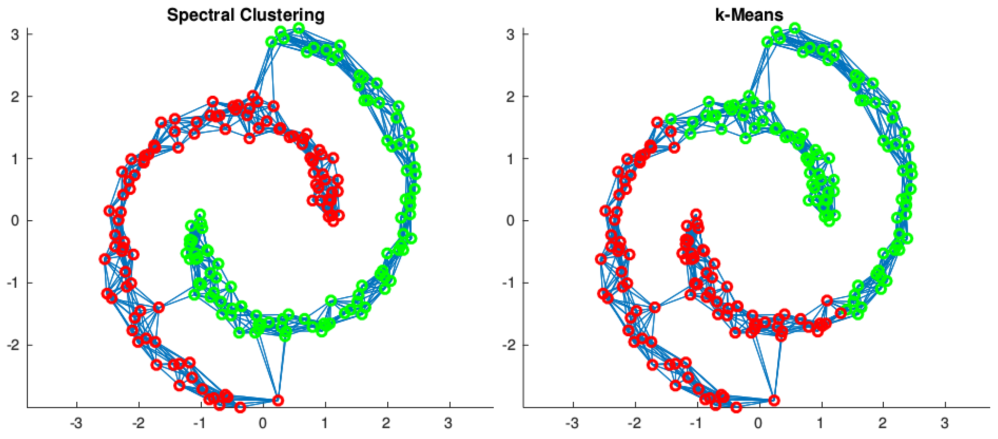

Figure 1: Spectral Clustering vs k-Means

3 Large Graph Construction

We now provide an overview of some popular methods to perform large graph construction. We chose to focus on computational cost reduction even if, as we explain, memory cost could also be a real issue.

3.1 Approximate Nearest Neighbors

Substantial efforts have been made in the litterature to decrease the complexity of k-nn graph construction, maybe the most frequently used algorithm in practice, through approximations. [Chen *et al.*, 2009] present a divide-and-conquer algorithm to construct approximate k-nn graph. Their idea is to recursively divide vertices into subsets - with overlapping - and to build k-nn graphs on each subset. Then, the graph is constructed by joining subsets using overlapping parts. In their paper, [Chen *et al.*, 2009] report an empirical time complexity of $O(pn^{1.22})$, which is sub-quadratic, while obtaining satisfying approximations. The divide step is based on a spectral bisection of data which is not detailed here. In a more recent paper, [Wang *et al.*, 2012] propose an alternative algorithm using the same ideas, except that there is no overlapping parts and that the division step is repeated several times. A drawback of this approach is that it can only be applied in Euclidean space.

Several other approches have been developed. For instance, [Dong *et al.*, 2011] describe a fast k-nn construction method which is based on local search. Their idea is that the neighbor of a neighbor is likely to also be a neighbor. Their algorithm initializes each vertex with a random set of neighbors, and iteratively improves vertices' neighborhoods. Even if, as it is explained in [Zhang *et al.*, 2013], there is no guarantee on the complexity of this algorithm, [Dong *et al.*, 2011] reported an empirical cost of $O(pn^{1.14})$.

Since it is impossible to give formal details on each algorithm in this paper, we chose to simply give a quick overview of these methods. The key point to remember is that real efforts have been made in the litterature to try to build - approximate - large k-nn graphs. We now present in a more detailed way another approximate nearest neighbors method, using Locality Sensitive Hashing.

3.2 ANN through Locality Sensitive Hashing

One of the most popular methods to perform approximate nearest neighbors is to use Locality Sensitive Hashing (LSH). The objective of LSH is to hash vertices in a similarity preserving way i.e. such that similar vertices map to the same "bucket" with high probability and keeping dissimilar vertices in different buckets. As explained in [Zhang *et al.*, 2013], LSH method for k-nn has two steps: training and querying. In the training step, LSH learns a so-called hash function $h(x) = \{h_1(x), h_2(x), ..., h_m(x)\}$ with m the code length. [Zhang *et al.*, 2013] take the example of binary coding using linear projection, where LSH is used with a hash function of the form $h_i(x) = sign(w_i^T x + b_i)$ with $\{w_i, b_i\}_{i=1,...,m}$ parameters to be learned. Each point of the dataset is represented by the hash mapping $h(x)$, and a hash table is constructed by hashing points into buckets according to their codes. In the querying step, [Zhang *et al.*, 2013] also explain that LSH first determines the hash code of the query and then returns its neighbors from the same bucket. LSH has theoretical performance gurantees and is often constant or sub-linear in search time [Inkyk and Motwani, 1998] which is of course an important result in our framework.

In a graph setting, our objective is to hash vertices in small groups, and then find each vertex's neighbors inside its group. This is an approximate method since it is almost impossible to really insure that true neighbors are always gathered. Since k-nn constructions are performed in small groups - and assuming all groups have the same size N - constructing the whole graph has $O(pNn)$ complexity. Therefore, to obtain relevant and fast results:

- similar vertices should be gathered in the same buckets to obtain a good approximation of the true k-nn graph, therefore N should not be too small ;
- to reduce complexity, we should insure that $N << n$.

This is of course a contradiction, and the optimal N is therefore a trade-off between the two objectives. Moreover, we should insure equal size buckets, since previous works on highly uneven hash tables led to poor empirical accuracies [Dong *et al.*, 2011]. In their work, [Zhang *et al.*, 2013] propose an efficient way to obtain equal size groups. Given a hash code matrix $Y \in \{0,1\}^{n\times m}$ where row $y_i \in \{0,1\}^m$ is the hash code of vertex x_i, they project hash codes onto a random direction $w \in \mathbb{R}^m$ and obtain $p = Yw$. Then they sort vertices by projection values to get the sequence $\{x_{\pi_1}, ..., x_{\pi_n}\}$ where $\{p_{\pi_1} \leq p_{\pi_2} ... \leq p_{\pi_n}\}$ and derive n/N groups of equal size following the ordering. Since vertices in the same bucket will have the same projection values, they have high probability to stay in the same group.

Nevertheless, the final result may lead to poor accuracy, because it is simply the union of n/N small graphs. Therefore, [Zhang *et al.*, 2013] propose to repeat the procedure l times with different hash functions and combine the l resulting graphs. A given vertex having at most $k \times l$ potential neighbors, we only keep the k closest according to the similarity measure. They also propose a final task called one step neighbor propagation, which objective is to add the neighbors of its own neighbors to the - at most - $k \times l$ potential neighbors. This full procedure, summarized in Algorithm 1, provided good empirical results of real datasets, both in term of accuracy and running time. Authors proved that the complexity of the whole algorithm is $O(nl(mp+pN+log(n)+k)+npk^2)$. Since k, m and N are small in practice, the actual complexity is $O(nl(p+log(n)))$.

```
Data: $X, k, l, m, N$
Result: Final approximate $k$-nn graph $G$.
begin
    for $i = 1, ..., l$ do
        $Y_i = LSH_i(X, m)$
        Project $Y_i$ onto random direction $w_i$:
          $p = Y_i w_i$
        Sort $p$ values and get $\{x_{\pi_1}, ..., x_{\pi_n}\}$
        for $j = 1, .., n/N$ do
            $S_j = \{x_{\pi_{(j-1)\times N+1}}, ..., x_{\pi_{j\times N}}\}$
            $g_j = k$-nn($S_j$,$k$)
        end
        $G_i = \cup_j \{g_j\}$
    end
    Combine $\{G_1, ..., G_l\}$ to get $G$
    Refine $G$ by one-step-neighbor propagation
end
```

Algorithm 1: A.N.N. [Zhang *et al.*, 2013]

3.3 Parallel computing

An alternative way to improve graph construction algorithms is to resort to the MapReduce paradigm to perform parallel computing on a cluster. A MapReduce program consists of a pair of so-called map and reduce functions. The objective of the map procedure is to perform filtering and sorting of data, while the reduce procedure performs summary operations. [Li *et al.*, 2012] explain that a typical MapReduce cluster consists of many slaves machines, responsible for map and reduce tasks, and a master machine that supervises the execution of the MapReduce program. Data are generally stored in a distributed file system (DFS) splitting them in equal size parts. Data are then distributed - and sometimes replicated - to the machines. A popular open-source implementation of MapReduce programming mode, with support for distributed shuffles, is part of Apache Hadoop. Parallel computing on graph construction algorithms could therefore - when it is possible and under some conditions - be an interesting option to deal with large datasets causing memory issues and to improve time performance of algorithms. Most of the following algorithms assume that it is possible to replicate some data to all slaves running tasks. In Hadoop for instance, this is possible through submission of files to the master for placement in the distributed cache for a job.

As explained in [Zhang *et al.*, 2013], the previous approximate k-nn algorithm with LSH is easy to parallelize. Indeed the k-nn steps in each of the n/N groups - containing different vertices - can be proceeded at the same time on n/N different machines. Moreover, each of the l steps from LSH to graph combining could also be done in parallel. Avoiding considerations on communication inside the cluster, this approach divides computation time. Coming back to the exact graph construction method of Section 2, we observe that distribution of computations is also possible. For instance in a ε-graph, distance computations for each vertex could be done in parallel.

Some papers study technical details of parallel algorithms for graph constructions. [Wang *et al.*, 2011] propose in their work another parallel algorithm for approximate k-nn graph construction, with faster results from their MPI/OpenMP mixed mode codes. [Li *et al.*, 2012] also study efficient parallel k-nn joins in MapReduce. Exact (H-BRJ) and approximate algorithms (H-zkNNJ) are proposed. They demonstrate the scalability of their ideas using Hadoop, with experiments in large real and synthetic datasets, with millions of points. [Plaku and Kavraki, 2009] provide alternative distributed algorithms, emphasizing on efficient data preprocessing for optimal distribution of computation for k-nn graphs on clusters, taking full advantages of available ressources. They also emphasize on efficient communication schemes, in order to reduce communication costs inside a cluster.

An interesting alternative approach for parallel approximate k-nn graph construction with Euclidean distance is provided by [Connor and Kumar, 2008]. The authors use Morton ordering - also called Z ordering - to sort vertices. Morton ordering is a space filling curve which maps multidimensional data to one dimension while preserving locality of the data points. For each vertex, we compute its k approximate neighbors by scanning k vertices to its left and right. [Connor and Kumar, 2008] explain how to refine and easily parallelize this algorithm in a MapReduce way, with almost linear computation time.

3.4 Sparsity

We decided to focus on the computational cost of Spectral Clustering. However, memory cost could also quickly become an issue even on a descent machine. For instance in image analysis, segmenting a full HD picture of 1920×1080 would require to store in memory a graph with over millions of vertices/pixels, which would require about 1Tbyte of memory. The Hadoop framework - and its storage part called HDFS - is a way to store Tbytes of data. As suggested in [Samet *et al.*, 2007], another technical way to handle this problem is to make disk-based data structures.

Another solution is to notice that most elements of adjacency and Laplacian matrices are 0. Therefore, it is in general possible to store them in sparse format. Such matrices are typically stored as a two-dimensional array with a lower memory cost, so the representation is relevant for large-scale graphs. For instance, the 200×200 Laplacian matrix associated to the Two-Moons graphs - definitely not a large graph - of Section 2 used 320 000 bytes in full matrix format, and only 40 456 bytes in sparse format through the MATLAB `sparse` function. Another advantage of this representation, as we explain in next section, is that there exists eigensolvers for sparse matrices. Finally, if the Laplacian matrix is dense - for instance, if we directly work with a given graph with a lot of edges - it is possible to remove most of the weights by adding a k-nn or ε-neighborhood construction step.

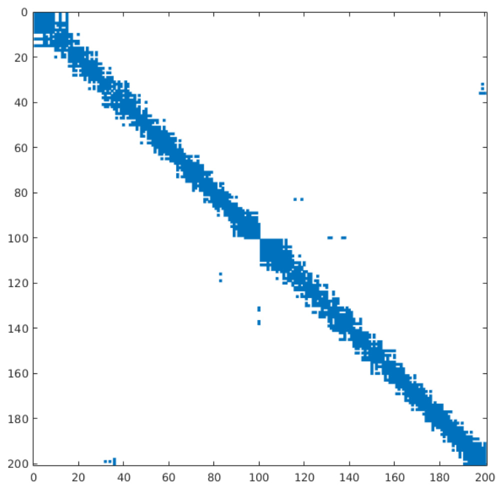

Figure 2: Non-null elements of L in Two-Moons

4 Eigendecomposition of the Laplacian

This section is dedicated to some of the main methods to avoid the cubic computational cost of Laplacian's eigendecomposition in Spectral Clustering.

4.1 Sampling methods and KASP algorithm

A first natural idea to create scalable algorithms is to use sampling techniques. Several methods adopting this fashion have been proposed, the objective being to use preprocessing to reduce the size of data. In other words, these methods try to apply the classical Spectral Clustering only on a subset of vertices and then to extend results to the whole graph. We develop here the KASP algorithm of [Yan, Huang and Jordan, 2009], for *k-means based approximate Spectral Clustering*. The authors suggest to proceed as follows:

- perform k-means on the entire dataset, with a large number of clusters $k = n/\gamma$;
- perform Spectral Clustering on the n/γ centers;
- extend results by assigning to vertices the value of the closest center.

In the article, they take $k \approx n/8$ for medium-size datasets - roughly 2 000 to 10 000 points and 10 to 166 features - and $k \approx n/20$ to $k \approx n/3000$ for large-size datasets - roughly 60 000 to 1 000 000 points and 10 to 42 features. They obtain satisfying accuracies, especially on medium-size datasets. In particular, results are comparable to the Nyström method developed in the next subsection. At the price of a little loss in precision, they significantly reduce time computation (we also illustrate this phenomenom in Section 5). More formally, [Yan, Huang and Jordan, 2009] show that the overall complexity of KASP algorithm is $O(k^3 + knt)$ where t is the number of iterations of the k-means step.

Alternative ways to reduce data size also exist in the litterature. [Shinnou and Sasaki, 2008] adopt the following quite close approach. They once again suggest to perform k-means on the entire dataset, with a large number of clusters k. Then, they remove vertices close to centers according to a pre-defined distance threshold. Spectral Clustering is applied on the remaining vertices and on the centers, and removed vertices get the assigment of their center. Such an alternative approach is sometimes called *Committees-based Spectral Clustering (CSC)*, e.g. in [Chen and Cai, 2011].

4.2 Nyström method

Among the other solutions to approximate the - infeasible - computation of Laplacian eigenvectors, [Fowlkes *et al.*, 2004] adopt the Nyström method which is an older technique for finding a numerical solution of integral equations. [Fowlkes *et al.*, 2004] use the Nyström method to find the "not too far" eigendecomposition of a reduced matrix. To be more precise, we first have to fix an integer $m << n$ and to sample a random subset $M \subset \{1, ..., n\}$ with $|M| = m$, to create a $m \times m$ matrix L_{MM} from the Laplacian. Then the objective is to derive an orthonormal matrix $U^{(m)}$ and a diagonal matrix $\Lambda^{(m)}$ such that:

$$L_{MM} U^{(m)} = U^{(m)} \Lambda^{(m)}$$

From $U^{(m)}$ and $\Lambda^{(m)}$ we derive an approximation of eigenvalues and eigenvectors of L, using the procedure of Algorithm 2. The pseudo-code is described as in [Homrighausen and McDonald, 2011], and assumes that eigenvalues are in an increasing order.

```
Data: $L, m < n, k$
Result: Approximation of the $k$ smallest
        eigenvalues ($\hat{\lambda}_i$) and eigenvectors ($\hat{u}_i$)
        of Laplacian $L$
begin
    Sample a random subset $M \subset \{1, ..., n\}$
      with $|M| = m$
    Derive the sub-Laplacian $L_{MM}$
    Compute $U^{(m)}$ and $\Lambda^{(m)}$ s.t.
        $L_{MM} U^{(m)} = U^{(m)} \Lambda^{(m)}$
    for $i = 2, ..., k$ do
        $\hat{\lambda}_i = \frac{n}{m} \Lambda_{ii}^{(m)}$
        $\hat{u}_i = \sqrt{\frac{m}{n}} \frac{1}{\Lambda_{ii}^{(m)}} W_{\{1,...,n\},M} U_i^{(m)}$
        with $U_i^{(m)}$ the $i$-th column of $U^{(m)}$ and
          only if $\Lambda_{ii}^{(m)} > 0$
    end
end
```

Algorithm 2: Nyström approximation

Two important points of this algorithm need to be discussed. First, there exists efficient ways to derive $U^{(m)}$ and $\Lambda^{(m)}$. Computations are provided in [Fowlkes *et al.*, 2004; Chen *et al.*, 2011]. There are quite long and technical so we put them beyond the scope of this paper. The conclusion is that the Nyström approximation can be derived in $O((n-m)m^2) + O(m^3) + O(mnk)$ [Chen *et al.*, 2011].

Moreover, without surprise, choosing M is important. The most common strategy is simply to choose M by sampling m times without replacement from $\{1, ..., n\}$, leading to the so-called "uniform Nyström" method. However, as explained in [Homrighausen and McDonald, 2011], work has been made on data-dependent choices. For instance, we could draw from $\{1, ..., n\}$ in proportion to the size of the diagonal elements of L. This stategy refers to the "weighted Nyström" method.

Finally, there also exists alternatives to the Nyström approximation, such as the Gaussian projection algorithm [Halko *et al.*, 2009]. This algorithm produces, for a matrix A, a subspace of the column space of A through the action of A on a random set of linearly independent vectors. There is a difference with the Nyström method, since important features of the column space of A can be missed by simply subsampling columns. [Halko *et al.*, 2009] provide a detailed explanation of the procedure, and [Homrighausen and McDonald, 2011] report theoretical - and generally not comparable - inequalities for these two methods.

4.3 Arnoldi-Lanczos eigensolver

Once we obtain a sparse Laplacian matrix L, we can use sparse eigensolvers. In particular, it might be interesting to implement algorithms that only compute the k first eigenvectors of L, and not the whole eigendecomposition (which is useless and very costly).

Most of these algorithms are variants of the Lanczos-Arnoldi factorization. Lanczos algorithm [Lanczos, 1950] is one of the most popular large-scale eigensolver, designed for symmetric matrices - to be exact, Hermitian matrices - such as L. Matrices are only involved in matrix-vector products which preserve sparsity. The main objective of the Lanczos algorithm is to derive an orthogonal transformation of L into a tridiagonal matrix T. Because of the orthogonality of this transformation, T and L are similar - this means for instance that they have the same eigenvalues - but it is easier to compute eigenvectors from T using its tridiagonal form. More precisely, the objective of the algorithm is to compute at each of m steps (with a user-defined $m << N$ in general) the $m \times m$ tridiagonal matrix $T_{mm} = V_m^T L V_m$.

```
**Data:** $L, m$
**Result:** $m \times m$ approx. similar matrix $T_{mm}$.
**begin**
    $v_1 =$ random vector with norm 1
    $v_0 = 0$, $\beta_1 = 0$
    **for** $i = 1, ..., m-1$ **do**
        $w'_i = Lv_i$
        $\alpha_i = (w'_i)^T v_i$
        $w_i = w_i^T - \alpha_i v_i - \beta_i v_{i-1}$
        $\beta_{i+1} = ||w_i||$
        $v_{i+1} = w_i / \beta_{i+1}$
    **end**
    $w_m = Lv_m$
    $\alpha_m = w'_m v_m$
**end**
```

Algorithm 3: Lanczos procedure

We denote diagonal elements α_j and off-diagonal elements β_j, i.e. T_{mm} is such that:

$$T_{mm} = \begin{pmatrix} \alpha_1 & \beta_2 & 0 & \dots & 0 \\ \beta_2 & \alpha_2 & \beta_3 & & \\ 0 & \beta_3 & \alpha_3 & \ddots & \\ \vdots & & \ddots & \ddots & \beta_m \\ 0 & & 0 & \beta_m & \alpha_m \end{pmatrix}$$

Elements of the T_{mm} matrices are computed according to Algorithm 3, reported above. From vectors v_j we construct the $m \times m$ transformation matrices:

$$V_m = (v_1, ..., v_m)$$

In practice, if we only need to compute the k first eigenvectors of L, only $v_1, ..., v_k$ will be saved in memory at each step. Using the fact that T_{mm} is tridiagonal we can efficiently compute its eigenvalues - which we can proved to be approximations of the eigenvalues of L - and its eigenvectors, denoted $u_i^{(m)}$, using for instance the QR algorithm. Finally, an approximation of the $n \times 1$ i-th eigenvector of L, denoted y_i, is computed as follows:

$$y_i \approx V_m u_i^{(m)}$$

To be precise we notice that, in practice, more technical consideration must be applied from Algorithm 2 to obtain stable results and correct numerical errors, which are not detailed here. The Arnoldi algorithm generalizes this procedure for non-Hermitian matrices. Some variations of Lanczos and Arnoldi algorithms are provided in the litterature, to perform faster computations and more stable approximations. The implicitly restarted Lanczos and Arnoldi algorithms are popular improvements.

They are implemented in the ARPACK package, a collection of Fortran77 subroutines designed to solve large scale eigenvalue problems, that we use in Section 5 for our applications. Implementation of ARPACK has been directly made e.g. in Python through SciPy and in MATLAB through the `eigs` command. From [Tsironis, Sozio and Vazirgiannis, 2013], the overall complexity of ARPACK is $O(m^3) + (O(nm) + O(nk)) \times O(m - c) \times (\#\text{restarted Arnoldi})$ in computational time and $O(nk) + O(nm)$ in memory requirement, with k the number of neighbors and c the number of desired clusters.

4.4 Parallel computing

As for the previous graph construction step, an interesting complementary way to reduce time complexity is to resort to the MapReduce paradigm to perform parallel computing on a cluster.

In their paper, [Tsironis, Sozio and Vazirgiannis, 2013] investigate variants of Spectral Clustering that can be parallelized in MapReduce in an efficient way. They use HEIGEN, a parallel eigensolver designed to be accurate and able to run in MapReduce, which is part of the peta-scale graph library PEGASUS [Kang *et al.*, 2009]. It implements a distributed version of the Lanczos eigensolver for symmetric matrices, previously presented. They obtain a final procedure which is scalable, with applications on social network data (roughly 1 000 to 20 000 vertices).

[Song *et al.*, 2008] also present parallel Spectral Clustering algorithms, studying both memory use and computation on a cluster. The authors work with sparse matrices, following the approach presented in Section 3. They use the ARPACK eigensolver, and a parallel version of this implementation called Parallel ARPACK (PARPACK). We recall that ARPACK, and by extension PARPACK, both provide an implementation of the implicitly restarted Arnoldi method. [Song *et al.*, 2008] highlight the scalability of this approach through an application on a dataset of roughly 200 000 vertices and a large photo of roughly 600 000 vertices.

Several other implementations of parallel Spectral Clustering in distributed systems are available in the litterature. For instance, [Chen *et al.*, 2008] use once again PARPACK and sparse matrices, but also parallel k-means (see below) and Nyström approximation. They also study potentials ways to reduce disk I/O.

4.5 Improving the k-means step

Finally, the very last step of Spectral Clustering - k-means on smallest eigenvectors - can also be improved. As explained in several works, k-means is an algorithm that can be easily parallelized in MapReduce, which makes it easily scalable to large datasets [Zhao *et al.*, 2009; Tsironis, Sozio and Vazirgiannis, 2013]. Moreover, since k-means is sensitive to initialization, [Tsironis, Sozio and Vazirgiannis, 2013] also recommend to implement the k-means++ algorithm, proposed by [Arthur and Vassilvitskii, 2007]. Its objective is to select initial centroids in a relevant - and probabilistic - way, such that final results are more likely to be accurate and such that the algorithm converges in a small number of iterations.

5 Applications

In this last section, we provide some concrete applications of previous algorithms. We both work on artificial data, from the Two Moons structure, and on real data from the RCV1 database of [Chen *et al.*, 2011]. We used MATLAB for implementations. The code has been tested under 64-bit Linux environment using MATLAB R2015b, on a laptop with 4 Go of RAM and Core i5 processor. When it was possible, we used the already implemented tools of MATLAB, e.g. the ARPACK implementation for - implicitly restarted - Arnoldi algorithm.

5.1 Two Moons

Let us start with artificial data. We once again simulate data points from the Two Moons structure with two ground truth clusters. Now, we work on two datasets of respectively 1 000 and 10 000 vertices, i.e. on graphs that are respectively five times and twenty times larger that the one in Section 2. Such graphs could be considerated as medium-size graphs. They are interesting because it is still computationally possible to perform the standard Spectral Clustering algorithm, and therefore we can directly compare the performance of different methods with respect to the standard one. Due to space and time constraints, we only focus on methods presented in Section 4. So, we consider here that we already constructed the Laplacian matrix from vectorial data. However, it is not difficult to extend our analysis to methods of Section 3, since we can find a lot of already implemented tools in main softwares. During this course for instance, we used the GraphLab library in Python to perform approximate nearest neighbors construction through LSH, in a very easy and high-level way. Moreover, we do not implement parallel computing in this paper.

The Two Moons structure is interesting because it is a quite challenging clustering problem that really needs methods such as Spectral Clustering to be solved. In other words, a k-means algorithm fails to identify clusters as shown before. At the same time, with a precise tuning of parameters for Spectral Clustering - similar to the tuning of Section 2 - it is possible to retrieve a perfect clustering. On our two graphs with respectively 1 000 and 10 000 vertices, we compare the Spectral Clustering method with KASP algorithm for different values of γ. We recall that the number of clusters in the first step of KASP algorithm is $k = n/\gamma$. We also compare with the - uniform - Nyström approximation, the Arnoldi procedure with a dense Laplacian and the Arnoldi procedure with a sparse Laplacian. Comparisons are made in terms of time improvement w.r.t. the standard method, and in terms of clustering accuracy, measured by the formula $Accuracy := \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}(\hat{y}_i = y_i)$.

Results for both graphs are reported in Tables 1 and 2. For methods including random sampling - KASP and Nyström - we report the average running times and accuracies, along with standard deviations, based on 50 trials of algorithms.

Concerning KASP algorithm, we observe that the parameter γ plays an important role. When γ increases, i.e. when Spectral Clustering is performed on a smaller subset of $k = n/\gamma$ vertices, running time decreases in a roughly quadratic way (reflecting the quadratic complexity of Laplacian construction on the k points). But, at the same time, accuracy also decreases. This result was expected, since we have to extrapolate clustering assignments on an increasing number of vertices, based on a smaller number of vertices. Standard deviations also increase. Therefore, there is an important trade-off on the choice of γ. This is illustrated in Figure 3, for the graph with $n = 1000$. As reported in tables, we obtained interesting results on the two graphs for, respectively, $\gamma = 10$ and $\gamma = 20$. Indeed, accuracies are almost perfect and time improvement is reasonable ($\times 20.64$ for the largest graph).

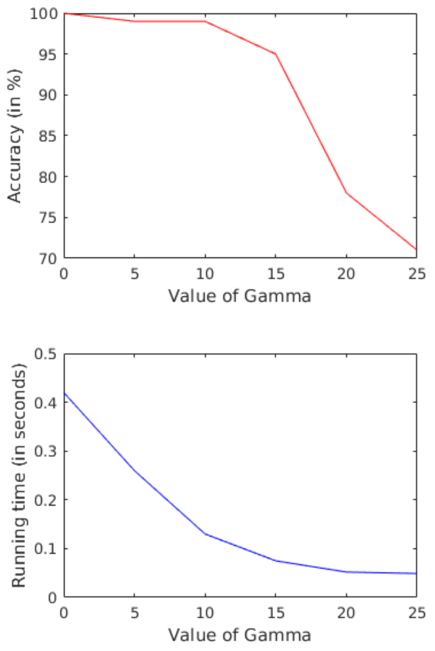

Figure 3: Accuracies and running times in KASP algorithm, for different values of γ

n = 1 000	Running time in seconds	Time improvement w.r.t. Standard Spect. Clust.	Accuracy in %	Standard Deviation within the 50 trials
Standard Spectral Clustering	0.42	–	100	–
KASP with $\gamma = 5$	0.26	$\times 1.62$	99.8	< 0.01
KASP with $\gamma = 10$	0.13	$\times 3.23$	99.7	0.01
KASP with $\gamma = 20$	0.06	$\times 7$	78.2	0.09
Uniform Nyström	0.15	$\times 2.8$	99.7	< 0.01
Arnoldi (ARPACK)	0.14	$\times 3$	100	–
Arnoldi + Sparse matrix	0.05	$\times 8.4$	100	–

Table 1: Comparison of different methods for Two Moons dataset, with $n = 1000$

n = 10 000	Running time in seconds	Time improvement w.r.t. Standard Spect. Clust.	Accuracy in %	Standard Deviation within the 50 trials
Standard Spectral Clustering	276.55	–	100	–
KASP with $\gamma = 5$	134.42	$\times 2.06$	99.9	< 0.01
KASP with $\gamma = 20$	13.40	$\times 20.64$	99.7	< 0.01
KASP with $\gamma = 100$	1.69	$\times 163.64$	83.2	0.12
Uniform Nyström	28.41	$\times 9.74$	99.6	< 0.01
Arnoldi (ARPACK)	24.25	$\times 11.40$	100	–
Arnoldi + Sparse matrix	0.38	$\times 727.76$	100	–

Table 2: Comparison of different methods for Two Moons dataset, with $n = 10000$

Clearly, running time is not a problem when $n = 1000$, even if we show that alternative methods provide in this framework a significant speed up while still being really accurate. However, results for $n = 10000$, reported in Table 2, illustrate the cubic complexity of the standard algorithm. Indeed, running time goes from 0.42 seconds to 276.55 seconds. At the same time, it goes from 0.05 seconds to 0.38 seconds for Arnoldi method with sparse matrix, our fastest result in the table.

Nyström method (with $m = 100$ which led to the best result in our case ; see the next application for more details about choosing this parameter in Nyström method) and Arnoldi method on dense Laplacian matrix provide comparable results for both graphs. There is a little advantage for Arnoldi, which still returns a perfect clustering i.e. there is no loss with respect to standard Spectral Clustering. Moreover, Arnoldi method exploiting the sparsity of L outperforms these two previous approaches. It seems that this is the most appropriate way to proceed in this Two Moons framework. Indeed, we obtain once again a perfect clustering and a very satisfying time improvement with respect to standard Spectral Clustering: results are 727.76 times faster.

A way to improve our analysis on the Two Moons structure could be to try to implement even more eigendecomposition methods, such as weighted Nyström, CSC and Gaussian projection briefly evoked in Section 4. Including parallel computing could also be an interesting approach to obtain even faster results and create larger graphs without memory issues: our previous graph with $n = 10000$ was e.g. already associated to a Laplacian matrix with $10000 \times 10000 = 100$ millions elements.

5.2 RCV1 database

We now focus on a real database used in [Chen *et al.*, 2011]. Their RCV1 database is an archive of 193 844 manually categorized newswire stories from Reuters. These documents are gathered in 103 categories, which are our ground truth clusters in this application. Therefore, the clustering problem is more challenging than the previous one. As explained in [Chen *et al.*, 2011], documents are represented by a cosine normalization of a log transformed TD-IDF feature vector. We retrieved the database on the website of Chen, along with his MATLAB code.

It was not possible to directly store similarities and Laplacian matrices with our implementations, due to out-of-memory issues. So, we decided to use most of their code for RCV1 construction because it was designed to handle large graphs with around 200 000 vertices on a laptop like ours. [Chen *et al.*, 2011] used a divide-and-conquer strategy for nearest neighbors search in order to directly store L in sparse format. They also rewrited a k-means algorithm that is supposed to be more efficient that the one implemented in MATLAB (that we used for Two Moons).

We reproduce here the approach of their article, i.e. we compare two main methods: uniform Nyström and Arnoldi (ARPACK). In the article, our Nyström method is denoted "Nyström with orthogonalization" because they also consider a version of the algorithm without orthogonalizing eigenvectors. Since this method led to less accurate results in the article, it is not discussed here. Moreover, in the article, our Arnoldi method - on sparse matrix - is denoted "Fixed-σ Spectral Clustering" because the parameter σ of the similarity measure is fixed to 2. They also tried to implement a "selftune Spectral Clustering" to adaptatively assign σ. It is not discussed here because, once again, it provided less accurate results.

We focus our analysis on clustering accuracy in this quite difficult problem with 103 clusters. For Nyström approach, we highlight in Figure 4 that the final result is sensitive to the choice of the number of random samples m. For Arnoldi approach, where we constructed the weighted adjacency matrix through the - efficiently programmed by [Chen *et al.*, 2011] - k-nn method, we also observe that the number of neighbors k has an impact on accuracy. In this application, we retrieve the result of the article i.e. we choose to work with $m = 1000$ and $k = 75$. Moreover, as in the article, the Arnoldi space dimension was set to be two times the number of clusters.

For each parameter, accuracies reported in Figure 4 are averages on 20 trials. So, standard deviations are still quite large and we could nuance our conclusions because differences are not always statistically significants. However, we still maintain our parameter choices because [Chen *et al.*, 2011] obtain curves with same shapes with lower standard deviations for accuracies estimations.

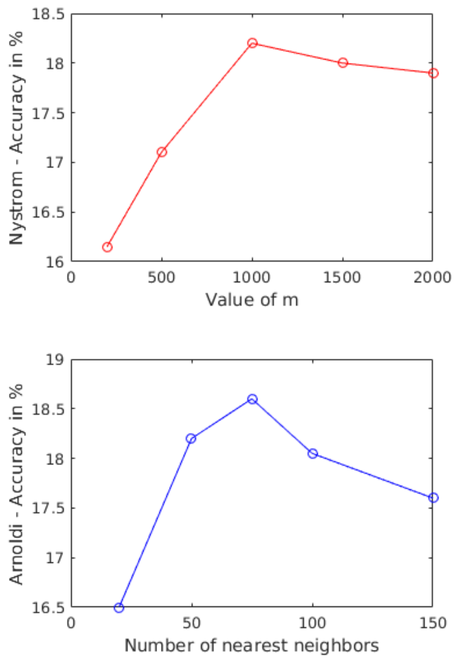

Figure 4: Accuracies on RCV1 for Nyström and Arnoldi with different parameters

We now report information about results of both methods with, respectively, $m = 1000$ and $k = 75$. We obtain a clustering accuracy of 18.5% for Arnoldi method. This result is better than our average clustering accuracy for Nyström method, which is 18.1%. Our standard deviation for Nyström, around 0.5 percentage points, is too large to allow us to conclude that Arnoldi is more appropriate than Nyström in this framework. However, we are consistent with the results of the article. Indeed, [Chen *et al.*, 2011] return an average accuracy of 18.5% for Arnoldi and 18.3% for Nyström.

At first glance, these accuracies may seem low and disappointing, especially in comparison with the almost perfect clustering assigments on Two Moons. However, we must remember that the problem is now way more difficult. Indeed, we have 103 ground truth clusters for RCV1 whereas there was only 2 clusters for Two Moons. In other words, a "random predictor" could obtain a 50% accuracy on Two Moons on average, but only a $\frac{1}{103} \approx 0.9\%$ accuracy on RCV1. Therefore, the RCV1 problem is more challenging, and accuracies around 18% should not be seen as bad results.

6 Conclusion

Spectral Clustering has become a popular technique in machine learning because of its interesting theoretical and experimental results. As we explained, it is often a revelant approach to detect complex shapes and manifold structure on graph data. In this paper, we proposed an overview of some methods to perform Spectral Clustering on large graphs, with applications on artificial and real data. We focused on the two costly aspects of the algorithm: the construction of the graph and the eigendecomposition of the Laplacian matrix. This overview is of course not exhaustive. Actually, we do not pretend to be exhaustive, but rather to present some of the most popular methods used in the litterature to tackle issues. Several other approaches could have been developed, including some very recent research such as [Ramasamy et Madhow, 2015] paper in NIPS 2015, trying to perform compressive spectral embedding by sidestepping the SVD.

Acknowledgments

This work is our final project for Pr. Michal Valko's course on *Graphs in Machine Learning*, at Ecole Normale Superieure de Cachan (fall 2015). We would like to thank him for this great semester, as well as T.A. Daniele Calandriello. We also would like to sincerely thank Dr. Romaric Gaudel from SequeL team at INRIA Lille - Nord Europe. He kindly accepted to supervise our project and gave us good directions and very relevant advices to work on this topic.

References

Bach, F. & Jordan, M. I. (2006), Learning spectral clustering, with application to speech separation. *Journal of Machine Learning Research*, 7, 1963-2001.

Cai, D. (2015), Compressed spectral regression for efficient nonlinear dimensionality reduction. *Proceedings of the 24th International Joint Conference on Artificial Intelligence*, IJCAI 2015, p. 3359-3365.

Chen J., Fang, H. & Saad, Y. (2009), Fast Approximate kNN Graph Construction for High Dimensional Data via Recursive Lanczos Bisection. *The Journal of Machine Learning Research*, archive 10: 1989-2012.

Chen, W.-Y., Song, Y., Bai, H., Lin, C.-J., & Chang, E. Y. (2011), Parallel spectral clustering in distributed systems. *Pattern Analysis and Machine Intelligence, IEEE Transactions*, 33(3):568-586.

Chen, X. & Cai, D. (2011), Large Scale Spectral Clustering with Landmark-Based Representation. *AAAI*.

Choromanska, A., Jebara, T., Kim, H., Mohan, M. & Monteleoni, C. (2013), Fast spectral clustering via the Nystrom method. *Algorithmic Learning Theory*, Springer Berlin Heidelberg, pp. 367-381.

Connor, M. & Kumar, P. (2008), Parallel Construction of k-Nearest Neighbor Graphs for Point Clouds. *Proceedings of the Eurographics / IEEE VGTC Workshop on Volume Graphics*.

Donath, W. E. & Hoffman, A. J. (1973). Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 17(5), 420-425.

Fowlkes, C., Belongie, S., Chung, F. & Malik, J. (2004), Spectral grouping using the Nystrom method. *Pattern Analysis and Machine Intelligence, IEEE Transactions* 26(2):214-225.

Hefeeda, M., Gao, F. & Abd-Almageed, W. (2012), Distributed approximate spectral clustering for large-scale datasets. *HPDC*, 223-234.

Homrighausen, D. & McDonald, D. J. (2011), Spectral approximations in machine learning. *arXiv preprint*, arXiv:1107.4340.

Kang, U., Chau, D.H., Faloutsos, C. (2012), Pegasus: Mining billion-scale graphs in the cloud. *ICASSP*, 5341-5344.

Khoa N. & Chawla S. (2012), Large-scale spectral clustering using resistance distance and Spielman-Teng solvers. *Proc. of 2012 Int. Conf. on Discovery Science*, 7-21.

Lanczos, C. (1950), An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *J. Res. Nat. Bureau Standards*, Sec. B, 45, pp. 255-282.

Li, F., Zhang, C. & Jestes, J. (2012), Efficient parallel kNN joins for large data in MapReduce. *Proceedings of the 15th International Conference on Extending Database Technology*, ACM, pp. 38-49.

Liu, W., He, J., & Chang, S.-F. (2010), Large graph construction for scalable semi-supervised learning. *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 679-686.

Liu, J., Wang, C., Danilevsky, M. & Han, J. (2013), Large-scale spectral clustering on graphs. *In Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, AAAI press, pp. 1486-1492.

Luxburg, U. (2007), A Tutorial on Spectral Clustering. *Statistics and Computing*, 17(4):395-416.

Maschhoff, K. & Sorensen, D. (1996), A portable implementation of ARPACK for distributed memory parallel architectures. *Proceedings of Copper Mountain Conference on Iterative Methods*.

Miao G., Song Y., Zhang D. & Bai H. (2008), Parallel spectral clustering algorithm for large-scale community data mining. *17th WWW workshop on Social Web Search and Mining (SWSM)*.

Ng, A. Y., Jordan, M. I. & Weiss, Y. (2001), On spectral clustering: Analysis and an algorithm. *Advances in neural information processing system (NIPS)*, 849-856.

Plaku, E., & Kavraki, L. (2007), Distributed Computation of the knn Graph for Large High-Dimensional Point Sets. *Journal of Parallel and Distributed Computing*, 67(3): 346-359.

Ramasamy, D. & Madhow, U. (2015), Compressive spectral embedding: sidestepping the SVD. *Advances in Neural Information Processing Systems*, 28.

Sakai T. & Imiya. A. (2009), Fast spectral clustering with random projection and sampling. *Proc. of the 6th Int. Conf. on Machine Learning and Data Mining in Pattern Recognition (MLDM)*, 372-384.

Sanakaranarayanan, J., Samet, H. & Varshney, A. (2007), A fast all nearest neighbor algorithm for applications involving large point-clouds. *Comput. Graph. 31*, 2, 157-174.

Shi J. & Malik J. (1997), Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 22(8):888-905.

Shinnou H. & Sasaki M. (2008), Spectral clustering for a large data set by reducing the similarity matrix size. *Proc. of the 6th Int. Conf. on Language Resources and Evaluation (LREC)*.

Song, Y., Chen, W., Bai, H., Lin, C. & Chang, E.Y. (2008), Parallel Spectral Clustering. *ECML/PKDD* (2):374-389.

Tsironis, S., Sozio, M. & Vazirgiannis, M. (2013), Accurate spectral clustering for community detection in MapReduce. *Frontiers of network analysis: methods, models, and applications*, Lake Tahoe, NIPS workshop.

Wang, D., Zheng, Y. & Cao, J. (2012), Parallel construction of approximate kNN graph. *Distributed Computing and Applications to Business, Engineering & Science (DCABES), 2012 11th International Symposium*, pp. 22-26.

Yan D., Huang L. & Jordan M. I. (2009), Fast approximate spectral clustering. *Proc. of the 7th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, 907-916.

Zhang, Y. M., Huang, K., Geng, G. & Liu, C. L. (2013). Fast kNN graph construction with locality sensitive hashing. *Machine Learning and Knowledge Discovery in Databases*, pp. 660-674, Springer Berlin Heidelberg.

Zeng, Z., Zhu, M., Yu, H. & Ma, H. (2014), Minimum Similarity Sampling Scheme for Nystrom Based Spectral Clustering on Large Scale High-Dimensional Data. *Modern Advances in Applied Intelligence*, pp. 260-269, Springer International Publishing.

Zhao, W., Ma, H. & He, Q. (2009), Parallel k-means clustering based on MapReduce. *CloudCom*, 674-679.