HAROKOPIO UNIVERSITY
DEPARTMENT OF INFORMATION & TELEMATICS

Job Report
Factorization

Group 13:

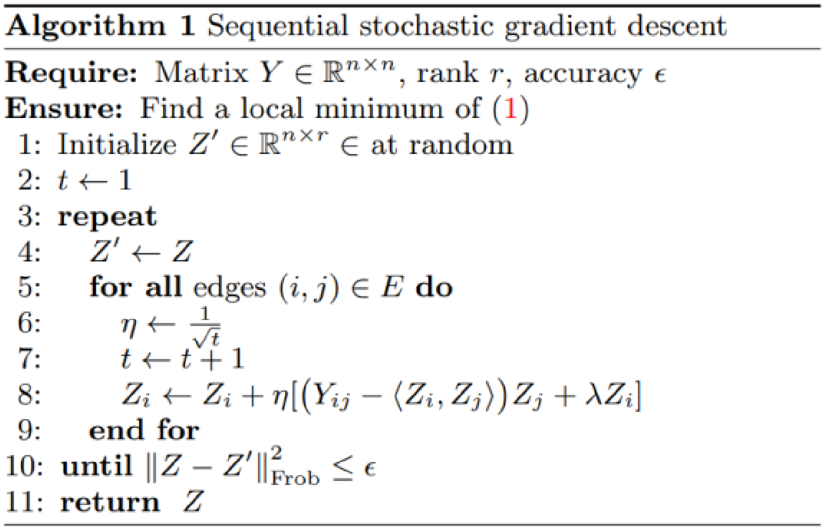
Kazakos Christos, it22033

Konstantinos Katsaras, it22045

Manousos Linardakis, it22064

1. Calculating Z Table:

The article “Distributed Large-scale Natural Graph Factorization” by Ahmed et al. [1] describes a serial algorithm for graph factorization. Specifically, the algorithm is as follows:

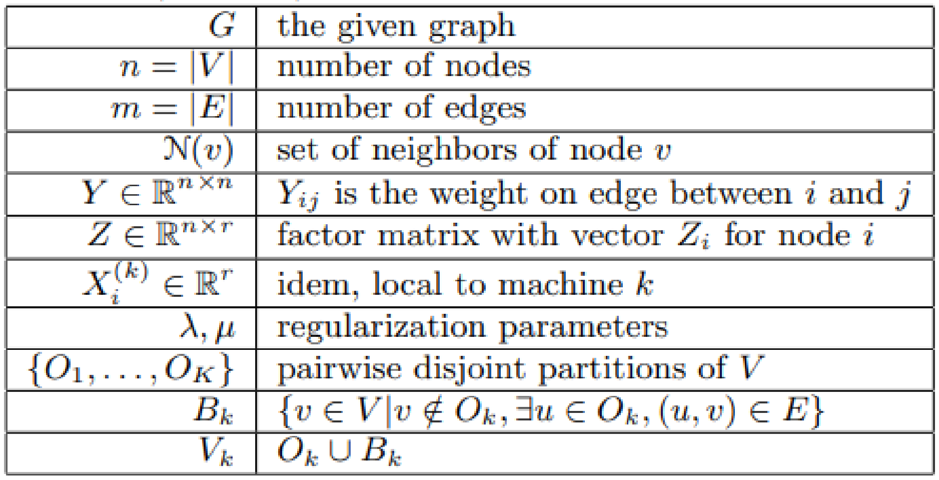


Algorithm 1 Sequential stochastic gradient descent

```
Require: Matrix $Y \in \mathbb{R}^{n \times n}$, rank $r$, accuracy $\epsilon$
Ensure: Find a local minimum of (1)
 1: Initialize $Z' \in \mathbb{R}^{n \times r} \in$ at random
 2: $t \leftarrow 1$
 3: repeat
 4:     $Z' \leftarrow Z$
 5:     for all edges $(i, j) \in E$ do
 6:         $\eta \leftarrow \frac{1}{\sqrt{t}}$
 7:         $t \leftarrow t + 1$
 8:         $Z_i \leftarrow Z_i + \eta[(Y_{ij} - \langle Z_i, Z_j \rangle) Z_j + \lambda Z_i]$
 9:     end for
10: until $\|Z - Z'\|^2_{\text{Frob}} \leq \epsilon$
11: return $Z$
```

The algorithm computes for each node a vector Zi through which one can decide the existence of an edge (i, j) between nodes i, j. In its simplest form one can use a simple inner product model where the information of the existence of an edge (i, j) can be efficiently described by the inner product ⟨Zi, Zj⟩.

Some more notations from the article [1] for a better understanding of the algorithm:



G	the given graph
$n = \|V\|$	number of nodes
$m = \|E\|$	number of edges
$\mathcal{N}(v)$	set of neighbors of node v
$Y \in \mathbb{R}^{n \times n}$	Y_{ij} is the weight on edge between i and j
$Z \in \mathbb{R}^{n \times r}$	factor matrix with vector Z_i for node i
$X_i^{(k)} \in \mathbb{R}^r$	idem, local to machine k
λ, μ	regularization parameters
$\{O_1, \ldots, O_K\}$	pairwise disjoint partitions of V
B_k	$\{v \in V \| v \notin O_k, \exists u \in O_k, (u, v) \in E\}$
V_k	$O_k \cup B_k$

It is noteworthy that the adjacency matrix Y of the graph is**symmetrical**according to the article [1]. In addition r is the rank or otherwise the columns of table Z.

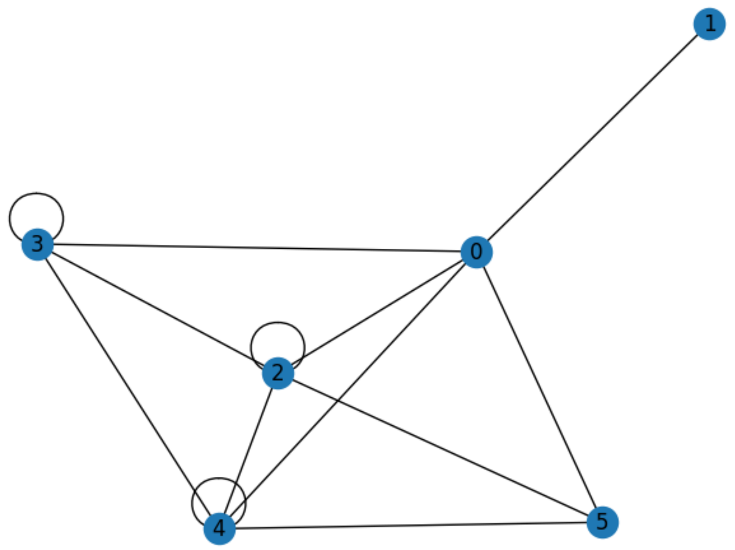
We implemented Algorithm 1 in python:

```
defseqgd(Y, r: int, e: float, l: float=0.1):
    """
    The sequential gradient descent algorithm.
    Parameters:
      Y: the adjacency matrix.
      r: the rank.
      e: the accuracy.
      l: lambda value.
    Returns: Z
    """
    n = Y.shape[0]
    Z = np.random.rand(n, r)
    t =1
    indices = np.transpose(np.where(Y !=0))
    graph_edges = [tuple(idx)foridxinindices]

    while True:
        Z_prev = Z.copy()
        fori, jingraph_edges:
            h =1/ np.sqrt(t)
            t +=1
            Z[i] += h * (((Y[i, j] - np.dot(Z[i], Z[j])) * Z[j]) + (l *
Z[i]))
        diff = np.linalg.norm(Z - Z_prev,'from') **2
        print(diff)
        ifdiff <= e:
            returnZ
```

Then we tested with our own data and with data from https://snap.stanford.edu/data/index.html the algorithm.

For example,



Derived from the following (random) adjacency matrix Y:

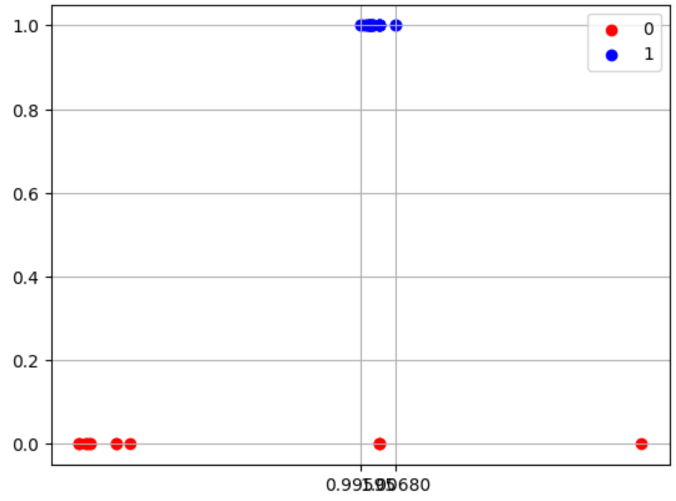
```
[[0. 1. 1. 1. 1. 1.] [1. 0. 0. 0.
  0. 0.] [1. 0. 1. 1. 1. 1.] [1.
  0. 1. 1. 1. 0.] [1. 0. 1. 1. 1.
  1.] [1. 0. 1. 0. 1. 0.]]
```

Putting array Y into algorithm 1 yields array Z. In the example, we set number of columns of Z (rank) equal to 2 and accuracy equal to 1e-10, with λ = 1e-5 . Then calculating the inner product⟨Zi, Zj⟩ or equivalently the productZZ_T(which we store in a table and print), the following result is obtained:

```
[[1.08426117 0.99991388 0.99595329 0.99800602 1.0067994 0.99936239] [
  0.99991388 0.92353995 0.90738347 0.9095419 0.91909806 0.91094639] [
  0.99595329 0.90738347 1.00198844 1.00178895 0.9984984                 1.00182771]
  [0.99800602 0.9095419 1.00178895 1.00164322 0.9986413                 1.00171343]
  [1.0067994 0.91909806 0.9984984 0.9986413 0.99719347 0.99887996] [0.99936239
  0.91094639 1.00182771 1.00171343 0.99887996 1.00180206]]
```

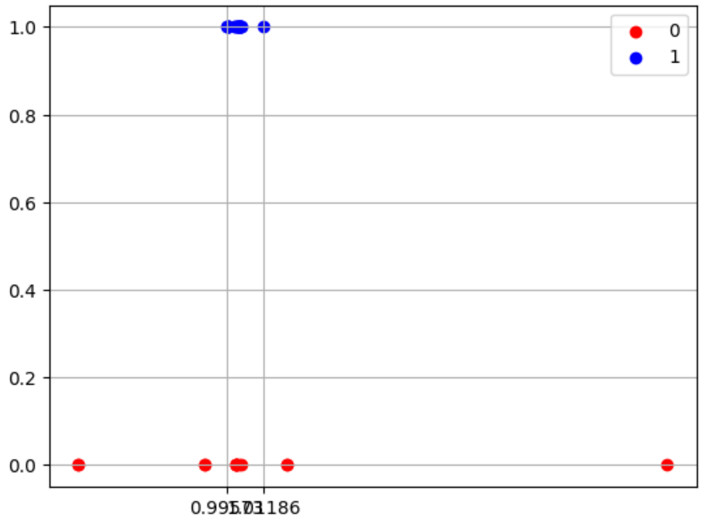
Notice that at position (0, 1) we have a value very close to 1, indicating that an edge between 0 and 1 is very likely.

According to the article, the goal is to construct a matrix Z such that the product $ZZ_T$to be very "close" to Y in cells that are not 0 (non-zero entries). And indeed, as we can see from the example above, this is true!
Specifically, we can say that the numbers from 0.996 to 1.007 in ZZ_T
is 1 in Y, while the rest are 0. We defined this range from the following figure that emerged, plotting the values of $ZZ_T$and putting colors (red = 0, blue = 1) if in position i, j of $ZZ_T$is 0 or 1 in Y. Values "1" in $ZZ_T$are in the interval [0.996, 1.007]:



By using this "filter" (turning into aces as many elements of $ZZ_T$are at [0.996, 1.007] and the rest 0) we managed to rebuild (almost) the original Y matrix, with "wrong"**8.33%**.

There is only a small "loss" to a zero. To improve this we can run the algorithm again, and we get this result:



Because Z is initialized randomly each time, the result is different, but still the product values $ZZ_T$is very "close" to Y in cells that are not 0 (non-zero entries). We can use this to approximate the original Y even better.

For this, we thought of the following function:

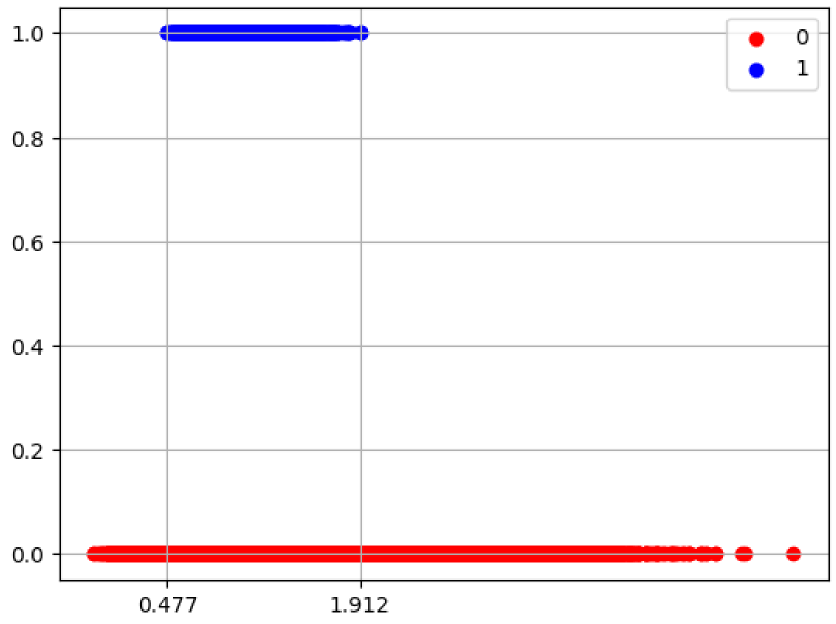
```
def calc_manyZ(num_exp=3,rank=2,acc=1e-10,lamd=1e-5):
    l = []
    foriinrange(num_exp):
        Z = seqgd(Y, rank, acc, lamd)
        Z_test = filter_convert(Y, Z)
        l.append(Z_test)
    avg = sum(l) / len(l)
    avg = np.where((avg < 1), 0, 1) #filter
    print(avg)
    return l, avg
```

This function calculates Z several times (here it is 3) (where it is different each time, but this is always the case with non-zero entries) and at the end keeps an M.O. If any of the elements of M.O are less than 1, then this means that they are not always in the range of aces, so we consider them as 0. In this way, the final avg matrix has**0%** difference from the original Y.

Therefore, the Z matrix can be used to "compress" the Y matrix, keeping a significant part of the original information/"behavior". This property is very useful when we have large graphs due to saving space and time (when loading the data).

We also tested for the social network Facebook (specifically from this file) containing . It is noteworthy that the Y-matrix for Facebook friends is always symmetric (so it satisfies the conditions of Algorithm 1), since in order to be friends with someone on Facebook, they must also be friends with you (as also pointed out in the articlehere). To be sure, we checked for symmetry of this graph by comparing Y to Y_T, and were equal so the matrix is symmetric.

The result of algorithm 1 is the corresponding matrix Z (with rank 10, accuracy 1e-5 and λ=1e-5) of dimensions 4039x10. The pattern of 0s and 1s for one run of the algorithm is this:



Notice that again the aces are clustered in the interval [0.477, 1.912] and most 0s are outside this interval. So one could use a more specialized filtering mechanism or keep M.O more Z's (as we did above) to rebuild the Y array. So the information one would store would be less than if one stored the entire Y array of size 4039x4039.

2. Applications:

The Z-matrix provides a limited but indicative representation of the original graph with fewer dimensions, which is useful in many applications dealing with large datasets that can be represented by adjacency matrices.

For example (as pointed out in the article [1]) one application is collaborative filtering. In particular, the rows of the adjacency matrix can be the different users and the columns the entities that the users have (1) or do not have (0). By using the Z table we can quickly edit and recommend new entities to users.

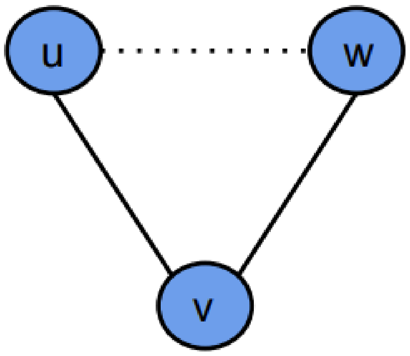
Another application is graph partitioning, in which the goal is to divide the nodes of the graph into groups, minimizing the amount of connections between the groups. For example the adjacency table can have rows corresponding to users, while its columns correspond to others (users). The existence of a relationship between users is denoted by 1 in the graph and the non-existence by 0. By using the Z table, we can process and divide the graph into groups of users faster.

We applied to the Facebook friends dataset (which we have used before) a new friend recommendation system.

The algorithm we use for suggesting friends is as follows:

```
def recommend_close_friends(A):
    rec= np.zeros(A.shape)
    for iinrange(A.shape[0]):
        friends = np.where(A[i] != 0)[0]            # friends of user i
        rec[i] = np.sum(A[friends], axis=0)
        rec[i, np.append(friends, i)] =0
    rec= np.where(rec>1,1,rec)
    return rec
```

Essentially, this simple function recommends to the user friends of his friends (if he doesn't already have them). With writing server, node v is host):
friend of the user and the com



Running the algorithm au of Y and one with ZZ_T(with rank equal to 10, accuracy 1e-5 and λ=1e-5) and putting a filtering so that as many variables of $ZZ_T$is between the interval of aces (according to Y) taking 1 and the rest 0, the rec tables generated differed by**18%**. This percentage can be reduced even more if we used a more specialized filtering technique or by doing more experiments.