ΧΑΡΟΚΟΠΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΜΑΤΙΚΗΣ

Αναφορά Εργασίας
Factorization

Ομάδα 13:

Καζάκος Χρήστος, it22033

Κωνσταντίνος Κατσάρας, it22045

Μανούσος Λιναρδάκης, it22064

1. Υπολογισμός Πίνακα Z:

Το άρθρο “Distributed Large-scale Natural Graph Factorization” από τους Ahmed et al. [1] περιγράφει ένα σειριακό αλγόριθμο για factorization γράφων. Συγκεκριμένα, ο αλγόριθμος είναι ο παρακάτω:

Algorithm 1 Sequential stochastic gradient descent

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

Ο αλγόριθμος υπολογίζει για κάθε κόμβο ένα διάνυσμα Zi μέσω των οποίων μπορεί κανείς να αποφασίσει την ύπαρξη μιας ακμής (i, j) μεταξύ των κόμβων i, j. Στην πιο απλή μορφή του μπορεί κανείς να χρησιμοποιήσει ένα απλό μοντέλο εσωτερικού γινομένου όπου η πληροφορία της ύπαρξης μιας ακμής (i, j) μπορεί να περιγραφεί αποτελεσματικά με το εσωτερικό γινόμενο ⟨Zi , Zj⟩.

Κάποια ακόμη notations από το άρθρο [1] για καλύτερη κατανόηση του αλγορίθμου:

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	$\{v \in V \mid v \notin O_k, \exists u \in O_k, (u, v) \in E\}$
V_k	$O_k \cup B_k$

Είναι αξιοσημείωτο ότι ο πίνακας γειτνίασης Y του γράφου είναι **συμμετρικός** σύμφωνα με το άρθρο [1]. Επιπρόσθετα το r είναι το rank ή αλλιώς οι στήλες του πίνακα Z.

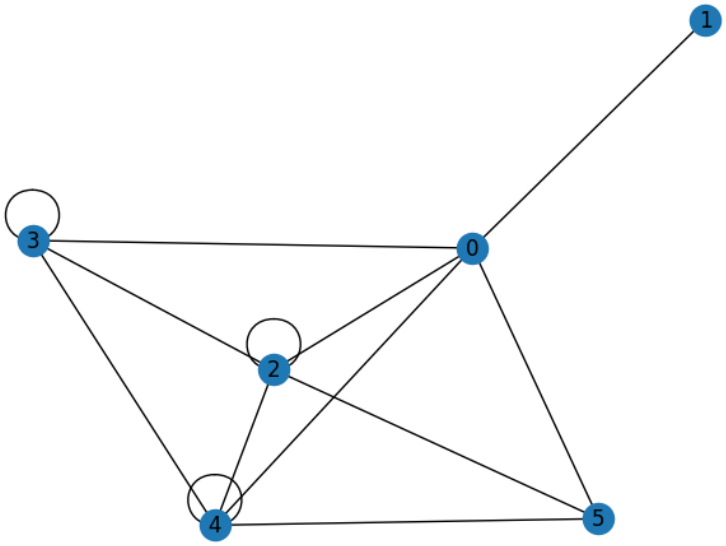
Υλοποιήσαμε τον αλγόριθμο 1 στη python:

```
def seqgd(Y, r: int, e: float, l: float=0.1):
    """
    The sequential gradient descent algorithm.
    Parameters:
      Y: the adjancency 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) for idx in indices]

    while True:
        Z_prev = Z.copy()
        for i, j in graph_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, 'fro') ** 2
        print(diff)
        if diff <= e:
            return Z
```

Έπειτα δοκιμάσαμε με δικά μας δεδομένα και με δεδομένα από το https://snap.stanford.edu/data/index.html τον αλγόριθμο.

Για παράδειγμα, κατασκευάσαμε (τυχαία) τον εξής γράφο:



Που προέκυψε από τον εξής (τυχαίο) πίνακα γειτνίασης Y:

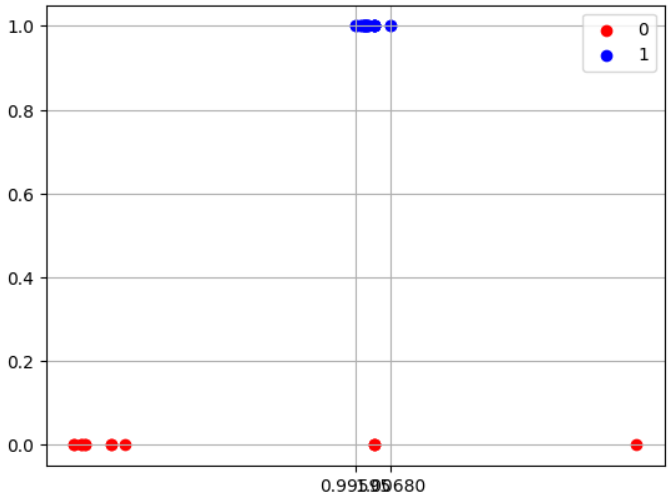
```
[[0. 1. 1. 1. 1. 1.]
 [1. 0. 0. 0. 0. 0.]
 [1. 0. 1. 1. 1. 1.]
 [1. 0. 1. 1. 1. 0.]
 [1. 0. 1. 1. 1. 1.]
 [1. 0. 1. 0. 1. 0.]]
```

Βάζοντας τον πίνακα Y στον αλγόριθμο 1, προκύπτει ο πίνακας Z. Στο παράδειγμα, ορίσαμε αριθμό στηλών του Z (rank) ίσο με 2 και accuracy ίσο με 1e-10, με λ = 1e-5 . Υπολογίζοντας ύστερα το εσωτερικό γινόμενο ⟨Zi , Zj⟩ ή ισοδύναμα το γινόμενο ZZ^T (το οποίο το αποθηκεύουμε σε έναν πίνακα και το εκτυπώνουμε), προκύπτει το εξής αποτέλεσμα:

```
[[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]]
```

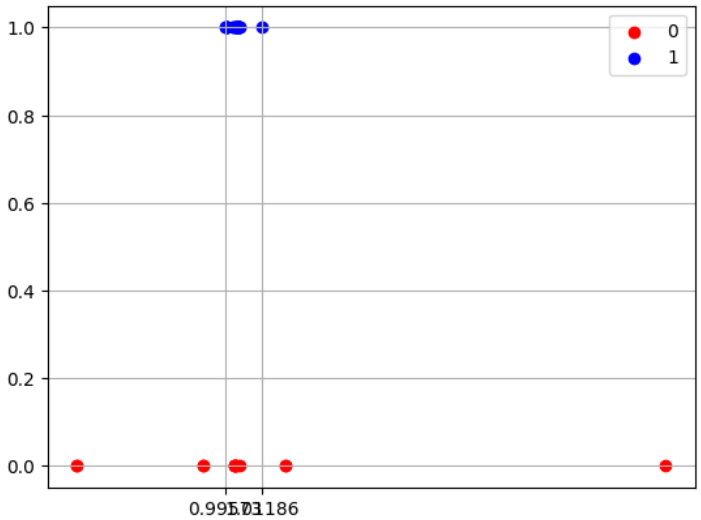
Παρατηρήστε ότι στη θέση (0, 1) έχουμε μία τιμή πολύ κοντά στο 1, η οποία δείχνει ότι είναι πολύ πιθανή η ύπαρξη ακμής μεταξύ 0 και 1.

Σύμφωνα και με το άρθρο, στόχος είναι η κατασκευή ενός πίνακα Z τέτοιου ώστε το γινόμενο ZZ^T να είναι πολύ “κοντά” στο Y στα κελιά που δεν είναι 0 (non-zero entries). Και όντως όπως βλέπουμε και από το παραπάνω παράδειγμα, αυτό ισχύει!
Συγκεκριμένα, μπορούμε να πούμε ότι όσα νούμερα είναι από το 0.996 ως το 1.007 στο ZZ^T είναι 1 στο Y, ενώ οι υπόλοιποι 0. Το range αυτό το ορίσαμε από το ακόλουθο σχήμα που προέκυψε, κάνοντας plot τις τιμές του ZZ^T και βάζοντας χρώματα (κόκκινο = 0, μπλε = 1) αν στην θέση i, j του ZZ^T είναι 0 ή 1 στο Y. Οι τιμές “1” στο ZZ^T βρίσκονται στο διάστημα [0.996, 1.007]:



Με τη χρήση του “φίλτρου” αυτού (μετατροπή σε άσους όσα στοιχεία του ZZ^T βρίσκονται στο [0.996, 1.007] και τα υπόλοιπα 0) καταφέραμε να ξαναφτιάξουμε (σχεδόν) τον αρχικό πίνακα Y, με “λάθος” **8.33%**.

Υπάρχει μόνο μία μικρή “απώλεια" σε ένα μηδενικό. Για να το βελτιώσουμε αυτό μπορούμε να ξανατρέξουμε τον αλγόριθμο, και βγαίνει αυτό το αποτέλεσμα:



Επειδή το Z αρχικοποιείται τυχαία κάθε φορά, το αποτέλεσμα είναι διαφορετικό, αλλά ακόμα οι τιμές του γινομένου ZZ^T είναι πολύ “κοντά” στο Y στα κελιά που δεν είναι 0 (non-zero entries). Μπορούμε να το χρησιμοποιήσουμε αυτό για να προσεγγίσουμε το αρχικό Y ακόμα καλύτερα.

Για αυτό, σκεφτήκαμε την εξής συνάρτηση:

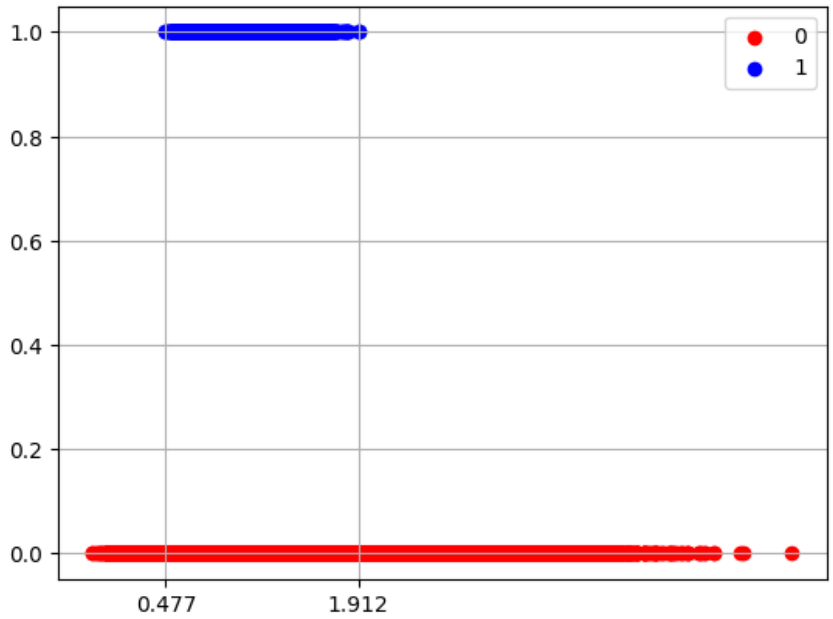
```
def calc_manyZ(num_exp=3, rank=2, acc=1e-10, lamd=1e-5):
    l = []
    for i in range(num_exp):
        Z = seqgd(Y, rank, acc, lamd)
        Z_test = filter_convert(Y, Z)
        l.append(Z_test)
    avg = sum(l) / len(l)
    avg = np.where((avg < 1), 0, 1)  # filter
    print(avg)
    return l,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

Η συνάρτηση αυτή υπολογίζει πολλές φορές (εδώ είναι 3) το Z (όπου κάθε φορά είναι διαφορετικό, αλλά πάντα ισχύει αυτό με τα non-zero entries) και κρατάει στο τέλος ένα M.O. Αν κάποια από τα στοιχεία του M.O είναι μικρότερα από 1, τότε αυτό σημαίνει ότι δεν είναι πάντα στο range των άσων, οπότε τα θεωρούμε 0. Με αυτό το τρόπο, ο τελικός πίνακας avg έχει **0%** διαφορά από τον αρχικό Y.

Συνεπώς, ο πίνακας Z μπορεί να χρησιμοποιηθεί για την “συμπίεση” του πίνακα Y, κρατώντας σημαντικό μέρος της αρχικής πληροφορίας/”συμπεριφοράς”. Αυτή η ιδιότητα είναι πολύ χρήσιμη όταν έχουμε μεγάλους γράφους λόγω εξοικονόμησης χώρου και χρόνου (όταν φορτώνουμε τα δεδομένα).

Επίσης δοκιμάσαμε και για το κοινωνικό δίκτυο του Facebook (συγκεκριμένα από αυτό το αρχείο) που περιέχει . Είναι αξιοσημείωτο ότι ο πίνακας Y για τους φίλους στο Facebook είναι πάντα συμμετρικός (άρα πληρεί τις προϋποθέσεις του αλγορίθμου 1), καθώς για να γίνεις φίλος με κάποιον στο facebook πρέπει και αυτός να γίνει φίλος με εσένα (όπως επισημαίνεται και στο άρθρο εδώ). Για να είμαστε σίγουροι, ελέγξαμε για την συμμετρία του γράφου αυτού, συγκρίνοντας το Y με το Y^T, και ήταν ίσα άρα ο πίνακας είναι συμμετρικός.

Το αποτέλεσμα του αλγορίθμου 1 είναι ο αντίστοιχος πίνακας Z (με rank 10, accuracy 1e-5 και λ=1e-5) διαστάσεων 4039x10. Το σχήμα των 0 και 1 για ένα τρέξιμο του αλγορίθμου είναι αυτό:



Παρατηρήστε ότι πάλι οι άσοι είναι μαζεμένοι στο διάστημα [0.477, 1.912] και τα περισσότερα 0 είναι εκτός αυτού του διαστήματος. Οπότε κάποιος θα μπορούσε να χρησιμοποιήσει ένα πιο εξειδικευμένο μηχανισμό φιλτραρίσματος ή να κρατήσει των Μ.Ο περισσότερων Z (όπως κάναμε παραπάνω) για να ξαναφτιάξει τον πίνακα Y. Έτσι η πληροφορία που θα αποθηκεύσει θα είναι μικρότερη από το να αποθήκευε όλο το πίνακα Y μεγέθους 4039x4039.

2. Εφαρμογές:

Ο πίνακας Z παρέχει μια περιορισμένη, αλλά ενδεικτική αναπαράσταση του αρχικού γράφου με λιγότερες διαστάσεις, κάτι που είναι χρήσιμο σε πολλές εφαρμογές που έχουν να κάνουν με μεγάλα datasets και μπορούν να αναπαρασταθούν με πίνακες γειτνίασης.

Για παράδειγμα (όπως επισημαίνεται και στο άρθρο [1]) μία εφαρμογή είναι το collaborative filtering. Συγκεκριμένα, οι γραμμές του adjacency matrix μπορεί να είναι οι διαφορετικοί χρήστες και οι στήλες οι οντότητες που έχουν (1) ή δεν έχουν (0) οι χρήστες. Με την χρήση του πίνακα Z μπορούμε να επεξεργαστούμε και να προτείνουμε γρήγορα νέες οντότητες σε χρήστες.

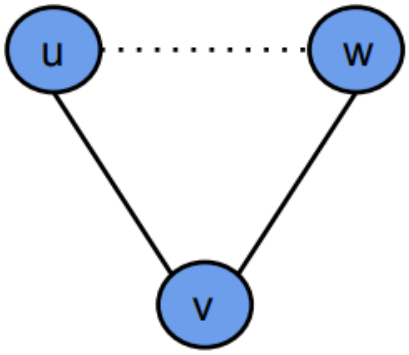
Μία άλλη εφαρμογή είναι το graph partitioning, στο οποίο ο στόχος είναι να χωρίσουμε τα nodes του γράφου σε ομάδες, ελαχιστοποιώντας τον όγκο των συνδέσεων μεταξύ των ομάδων. Για παράδειγμα ο πίνακας γειτνίασης μπορεί να έχει γραμμές που αντιστοιχούν σε χρήστες, ενώ οι στήλες του σε άλλους (χρήστες). Η ύπαρξη σχέσης μεταξύ χρηστών συμβολίζεται με 1 στο γράφο ενώ η μη ύπαρξη με 0. Με τη χρήση του πίνακα Z, μπορούμε να επεξεργαστούμε και να χωρίσουμε το γράφο σε ομάδες χρηστών πιο γρήγορα.

Εμείς εφαρμόσαμε στο dataset με τους φίλους στο Facebook (που χρησιμοποιήσαμε και πριν) ένα σύστημα προτάσεων για νέους φίλους.

Ο αλγόριθμος που χρησιμοποιούμε για την πρόταση φίλων είναι ο παρακάτω:

```
def recommend_close_friends(A):
   rec = np.zeros(A.shape)
   for i in range(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
```

Ουσιαστικά αυτή η απλή συνάρτηση κάνει recommend στο χρήστη φίλους των φίλων του (αν δεν τους έχει ήδη). Με γραφική αναπαράσταση (ο κόμβος u είναι ο χρήστης, ο κόμβος v είναι ο φίλος του χρήστη και ο κόμβος w είναι ο προτεινόμενος φίλος του χρήστη):



Τρέχοντας τον αλγόριθμο αυτό δύο φορές, μία με το πίνακα γειτνίασης Y και μία με το ZZ^T (με rank ίσο με 10, accuracy 1e-5 και λ=1e-5) και βάζοντας ένα φιλτράρισμα έτσι ώστε όσες μεταβλητές του ZZ^T είναι μεταξύ του διαστήματος των άσων (σύμφωνα με τον Y) να παίρνουν 1 και οι υπόλοιποι 0, οι πίνακες rec που δημιουργήθηκαν διέφεραν κατά **18%**. Το ποσοστό αυτό μπορεί να μειωθεί και άλλο αν χρησιμοποιούσαμε μία πιο εξειδικευμένη τεχνική φιλτραρίσματος ή κάνοντας περισσότερα πειράματα.