Preliminary Work

Bill

June 2022

1 - Context

We adapt the approach of Chanpuriya et al. in paper [1], and try to derive a single low dimensional embedding of a high triangle count network which recovers the triangle structure. Paper [1] factorized the network adjacency matrix into distinct low rank embeddings X and Y using a method similar to Spotify's awesome 'Logistic Matrix Factorization' described in [2]. Explicitly, matrices X and Y were derived by minimising the loss function:

$$L = \sum_{i,j} -\log l(\tilde{A}_{i,j}[XY^T]_{i,j})$$

Where $l(x)$ is the function $(1 + e^{-x})^{-1}$ and $\tilde{A}$ is the adjacency matrix in which zeroes have been replaced by negative ones.

We think the approach of [1] is dodgy, when considering the matrix of probabilities of there being an edge between nodes using YX^T should be just as valid a choice as taking XY^T but these matrices are not equal since the embeddings X and Y differ. This lack of transitivity may indicate that the embeddings fail to capture any latent structure of the network, indeed the authors of [1] state in their conclusion that the 'embeddings do not give good performance on downstream classification tasks'.

Instead we derive a single low rank embedding of A call it X with underlying statistical model that node i is connected to node j with probability $l(X_i X_j^T)$. We derive this embedding by minimising over a near identical loss to the above. Namely:

$$L = \sum_{i,j} -\log l(\tilde{A}_{i,j}[XX^T]_{i,j})$$

Interestingly, paper [3] shows that network triangle structure is not recoverable from a low rank SVD embedding, or a fitted logistic function of the SVD embedding. This had me wincing for a couple of seconds but we note that our embedding is not a fitted logistic of SVD, and our conclusions not forgone. So, will our embedding also recover the triangle structure of our network, like in paper [1], or will our findings add weight to the unnatural suggestion that each node requires two different embeddings?

The remainder of this document is structured as follows:

1. We provide code for generating the toy network used in [1].
2. We attempt to replicate Chanpuriya's results on the toy network.
3. We carry out the adapted embedding on the toy network and compare our results with those of the previous section.

Let's begin.

2 - Toy Network Generation

The toy network used by Chanpuriya et al. in [1] is a cycle of triangles. We provide functions for constructing this same network for our own analysis. To start we store the adjacency matrix of a triangle as a variable called `tri`.

```
tri <- c(0, 1, 1, 1, 0, 1, 1, 1, 0)
dim(tri) <- c(3,3)
tri
```

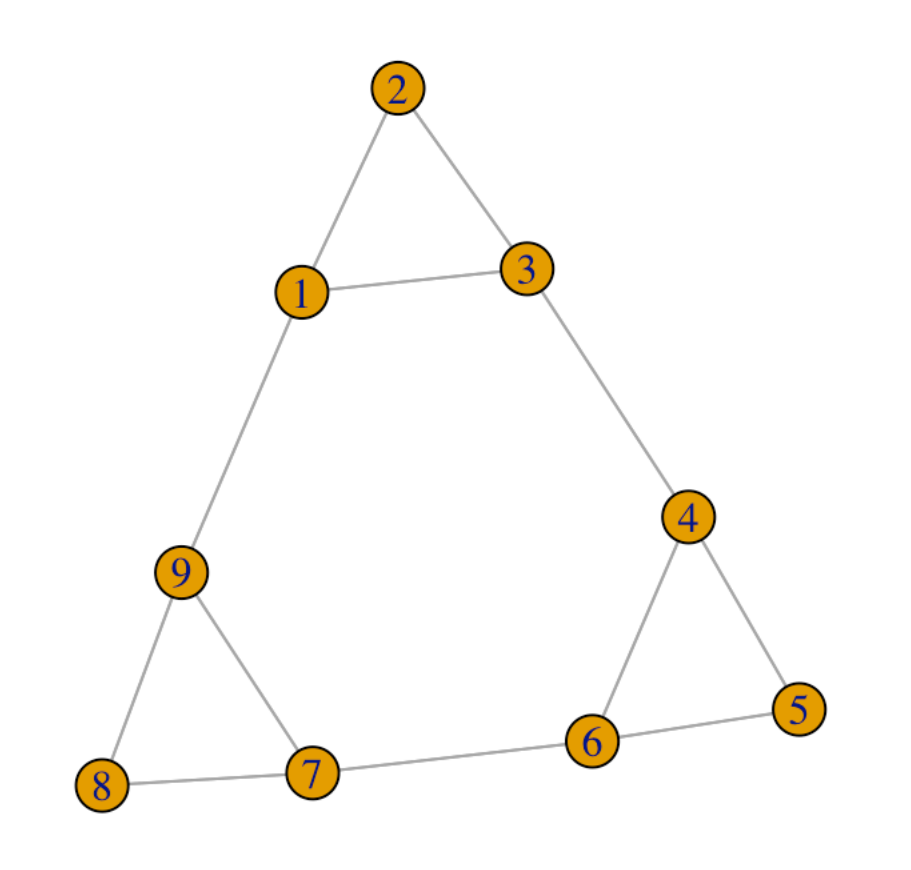
```
##      [,1] [,2] [,3]
## [1,]    0    1    1
## [2,]    1    0    1
## [3,]    1    1    0
```

From here we write a function which produces a cycle of `tri_count` triangles.

```
tri_cycle <- function(tri_count){
  # Kronecker product to get union of tri_count disjoint triangles
  A <- diag(tri_count) %x% tri
  # Link them up
  for(j in 1:tri_count){
    A[3*j, (3*j + 1) %% (3*tri_count)] = 1
    A[(3*j + 1) %% (3*tri_count), 3*j] = 1
  }
  return(A)
}
```

To be super clear about what the output of `tri_cycle` we've plotted the output network with `tricount` equal to three below.

```
G <- graph_from_adjacency_matrix(tri_cycle(3))
# Stops the graph from being plotted with arrows.
E(G)$arrow.mode="-"
plot(G)
```



The adjacency matrix of the toy network in paper [1] is precisely the output of `tri_cycle(100)`. We therefore create the variable `toy` for later.

```
toy <- tri_cycle(100)
```

3 - Replication of Chanpuriya's Result

We now turn to replicating the results of paper [1]. In the paper the loss function was optimised using `scipy`'s inbuilt optimiser, the resulting code obscures what's actually going on and so we re-write the optimisation explaining each step clearly. We minimise the loss function L by using gradient descent and therefore start by differentiating L with respect to each row X_i of matrix X, and each row Y_j of matrix Y. Doing this yields:

$$\frac{\partial L}{\partial X_i} = \sum_j \tilde{A}_{i,j}[l(\tilde{A}_{i,j}X_iY_j^T) - 1]Y_j$$

$$\frac{\partial L}{\partial Y_j} = \sum_i \tilde{A}_{i,j}[l(\tilde{A}_{i,j}X_iY_j^T) - 1]X_i$$

I've checked and double checked these derivatives but they don't seem to be the derivatives used in Chanpuriya's code, we proceed under the assumption that they are equivalent. We note that these are just linear combinations of the rows of Y_j and X_i respectively and can thus express $\frac{\partial L}{\partial X}$ and $\frac{\partial L}{\partial Y}$ as matrix products. Where by $\frac{\partial L}{\partial X}$ we mean the matrix which is the stacking of the row vectors $\frac{\partial L}{\partial X_i}$. For notational convenience we let matrix M be defined by $M_{i,j} = \tilde{A}_{i,j}[l(\tilde{A}_{i,j}X_iY_j^T) - 1]$ and then we have:

$$\frac{\partial L}{\partial X} = MY$$

$$\frac{\partial L}{\partial Y} = M^TX$$

This compact form will be used when we write our gradient descent algorithm. Let us begin this descent by writing a function which converts the usual adjacency matrix to the adjacency matrix with zeroes replaced by negative ones.

```
tilde <- function(A){
  A <- 2 * A - 1
  return(A)
}
tilde(tri)
```

```
##      [,1] [,2] [,3]
## [1,]   -1    1    1
## [2,]    1   -1    1
## [3,]    1    1   -1
```

Excellent, next we write a function to produce a randomised n by e embedding initialisation where n is the number of vertices e is the number of dimensions of our embedding. Each entry of the initialised embedding is a uniform $[-1, 1]$ random variable chosen independently.

```
initial <- function(n,e){
  init <- runif(n * e, -1, 1)
  dim(init) <- c(n,e)
  return(init)
}
```

We now note that the expression $l(\tilde{A}_{i,j}[XY^T]_{i,j})$ is required to calculate both the loss and the gradient functions. We therefore implement this computation independently. It's sensible to store all values of $l(\tilde{A}_{i,j}[XY^T]_{i,j})$ in an n by n matrix.

```
l <- function(x){
  return(1 / (1 + exp(-x)))
}

l_matrix <- function(X, Y, A){
  L_mat <- A * (X %*% t(Y))
  L_mat <- apply(L_mat, c(1,2), l)
  return(L_mat)
}
```

We are now ready to write a functions which takes matrices X, Y and $\tilde{A}$ and returns the (scalar) loss L and gradient matrices.

```
loss_and_gradients <- function(X, Y, A){
  # First find the useful l_matrix
  L_mat <- l_matrix(X, Y, A)
  # Find the loss
  loss_matrix <- -1 * log(L_mat)
  loss <- sum(loss_matrix)
  # Now find the gradient matrices
  M_mat <- A * (L_mat + (-1))
  X_grad <- M_mat %*% Y
  Y_grad <- t(M_mat) %*% X
  return(list(loss, X_grad, Y_grad))
}
```

The optimisation is then carried out with vanilla gradient descent, simply for ease of implementation. This is extremely basic, and we will replace this with AdaGrad [4], as suggested in [3], or ADAM at some point (A job for a better coder than me: Adam).

```
# We test first on a small network
A <- tri_cycle(2)
A <- tilde(A)

# Initialise with three dimensional embeddings
X <- initial(6, 3)
Y <- initial(6, 3)

# Find the initial loss
loss_list <- loss_and_gradients(X, Y, A)
print(paste("Initial Loss: ", loss_list[[1]]))
```

```
## [1] "Initial Loss:  27.6547266949167"
```

```
# Carry out 200 steps of gradient descent
for(i in 1:200){
  loss_list <- loss_and_gradients(X, Y, A)
  X <- X - (0.1 * loss_list[[2]])
  Y <- Y - (0.1 * loss_list[[3]])
}

# Find the final loss
print(paste("Final Loss: ", loss_list[[1]]))
```

```
## [1] "Final Loss:  0.173392261605741"
```

Great! This has appeared to work. We now check to see how well the adjacency matrix has been recovered by our embedding:

```
max(tri_cycle(2) - apply(X %*% t(Y), c(1,2), l))
```

```
## [1] 0.008527713
```

```
min(tri_cycle(2) - apply(X %*% t(Y), c(1,2), l))
```

```
## [1] -0.0201801
```

Each entry of the adjacency matrix has been approximated to within 0.1, pretty good going. We now try to replicate Chanpuriya's result on the `toy` network we set at the beginning.

```
# We now test on the toy network
A <- tilde(toy)

# Initialise with five dimensional embeddings, like in [1].
X <- initial(300, 5)
Y <- initial(300, 5)

# Find the initial loss
loss_list <- loss_and_gradients(X, Y, A)
print(paste("Initial Loss: ", loss_list[[1]]))
```

```
## [1] "Initial Loss:  68239.1096666189"
```

```
# Carry out 10000 steps of gradient descent
for(i in 1:10000){
  loss_list <- loss_and_gradients(X, Y, A)
  X <- X - (0.01 * loss_list[[2]])
  Y <- Y - (0.01 * loss_list[[3]])

  if(i %% 1000 == 0){
    print(paste("Loss: ", loss_list[[1]]))
  }
}
```

```
## [1] "Loss:  995.034434772156"
## [1] "Loss:  578.665602448572"
## [1] "Loss:  393.001347283079"
## [1] "Loss:  293.017310115291"
## [1] "Loss:  231.731768729357"
## [1] "Loss:  197.295277250274"
## [1] "Loss:  172.520848108909"
## [1] "Loss:  155.348984954037"
## [1] "Loss:  138.548599191537"
## [1] "Loss:  127.601779664171"
```

```
# Return the final loss
print(paste("Final Loss: ", loss_list[[1]]))
```

```
## [1] "Final Loss:  127.601779664171"
```

And again we observe the error compared to the adjacency matrix:

```
differences <- toy - apply(X %*% t(Y), c(1,2), l)
rowSums(differences > 0.1)
```

```
##   [1] 0 0 0 0 0 0 1 1 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [38] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 3 3 2 3 1 0 0 0 1 2 2 1 1 0 0
##  [75] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
## [112] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [149] 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 2 3 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [186] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [223] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [260] 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 3 2 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [297] 0 0 0 0
```

```
rowSums(differences < -0.1)
```

```
##   [1] 0 0 0 0 0 1 0 0 2 6 1 1 1 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [38] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 3 2 2 0 0 0 0 2 3 4 2 2 0 0
##  [75] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0
## [112] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [149] 0 0 0 0 0 0 0 0 0 0 0 0 0 1 7 3 3 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [186] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [223] 0 0 0 2 0 1 2 0 1 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [260] 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 2 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [297] 0 0 0 0
```

This isn't bad going at all, only a small percentage of rows of $l(XY^T)$ have entries which differ from the adjacency matrix by more than 0.1! We regard this as a fair replication of Chanpuriya's findings, especially given we've only used a vanilla gradient descent and the embeddings generated are just approaching some random local minimum (I'm not sure sure whether the loss function is convex to be fair.)

4 - The Adapted Embedding

Our adapted loss function is fit in a nearly identical manner but the gradient function may need to be slightly modified. Recall that our modified loss is given by $L = \sum_{i,j} -\log l(\tilde{A}_{i,j}[XX^T]_{i,j})$. Differentiating L with respect to row X_i yields:

$$\frac{\partial L}{\partial X_i} = \sum_j 2\tilde{A}_{i,j}[l(\tilde{A}_{i,j}X_iX_j^T) - 1]X_j$$

And so defining matrix $M_{i,j} = 2\tilde{A}_{i,j}[l(\tilde{A}_{i,j}X_iX_j^T) - 1]$ yields the compact form $\frac{\partial L}{\partial X} = MY$. We can ignore the constant 2 as this will be eaten up by our learning rate parameter and our method of gradient descent can be replicated from the previous section nearly identically.

```
# We now test on the toy network
A <- tilde(toy)

# Initialise with a five dimensional embedding
X <- initial(300, 5)

# Find the initial loss
loss_list <- loss_and_gradients(X, X, A)
print(paste("Initial Loss: ", loss_list[[1]]))
```

```
## [1] "Initial Loss:  68381.0073039382"
```

```
# Carry out 10000 steps of gradient descent
for(i in 1:10000){
  loss_list <- loss_and_gradients(X, X, A)
  X <- X - (0.01 * loss_list[[2]])

  if(i %% 1000 == 0){
    print(paste("Loss: ", loss_list[[1]]))
  }
}
```

```
## [1] "Loss:  62310.0262890582"
## [1] "Loss:  62309.3529462174"
## [1] "Loss:  62309.1433141287"
## [1] "Loss:  62309.0201534319"
## [1] "Loss:  62308.9378292805"
## [1] "Loss:  62308.882155523"
## [1] "Loss:  62308.8437174782"
## [1] "Loss:  62308.8158883851"
## [1] "Loss:  62308.7944906057"
## [1] "Loss:  62308.7771769739"
```

```
# Find the final loss
print(paste("Final Loss: ", loss_list[[1]]))
```

```
## [1] "Final Loss:  62308.7771769739"
```

```
differences <- toy - apply(X %*% t(X), c(1,2), l)
rowSums(differences > 0.1)
```

```
##   [1] 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3
##  [38] 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2
##  [75] 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3
## [112] 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3
## [149] 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2
## [186] 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3
## [223] 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3
## [260] 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2 3 3 2
## [297] 3 3 2 3
```

```
rowSums(differences < -0.1)
```

```
##   [1] 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297
##  [19] 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297
##  [37] 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297
##  [55] 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297
##  [73] 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297
##  [91] 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297
## [109] 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297
## [127] 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297
## [145] 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297
## [163] 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297
## [181] 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297
## [199] 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297
## [217] 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297
## [235] 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297
## [253] 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297
## [271] 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297 297 298 297
## [289] 297 298 297 297 298 297 297 298 297 297 298 297
```

Interesting, the single embedding is not even close to recovering the adjacency matrix. We don't yet know whether this is due to the gradient descent method used or if there is something deeper going on. We are left with two important to consider going forward:

1. Will improving the gradient descent method allow the single embedding to recover the adjacency matrix? What about adapting the loss function?
2. If not, can we generalise Seshadri's result in [3] and prove that no single low dimensional embedding can recover the adjacency matrix?

Sources

[1] Chanpuriya et al. Node embeddings and exact low-rank representations of complex networks. NeurIPS

[2] Johnson. Logistic matrix factorization for implicit feedback data. NeurIPS

[3] Seshadri et al. The impossibility of low-rank representations for triangle-rich complex networks. PNAS

[4] Duchi et al. Adaptive subgradient methods for online learning and stochastic optimization. JMLR.