# Triangle Recovery

Bill

July 2022

## 1. Attaining the Cora Adjacency Matrix

```
cora <- read.csv("/Users/billnunn/Desktop/Project/cora/cora.cites",
                 sep = '\t', header = FALSE)
head(cora)
```

```
##   V1      V2
## 1 35    1033
## 2 35  103482
## 3 35  103515
## 4 35 1050679
## 5 35 1103960
## 6 35 1103985
```

We find a list of the node names and find the number of vertices, and add a dictionary of the new names which are chosen to be consecutive integers.

```
node_names <- union(cora[,1], cora[,2])
length(node_names)
```

```
## [1] 2708
```

```
name_dict <- data.frame(new_names = 1:length(node_names),
                        row.names = node_names)
```

We can now parse through the edge list and replace the old names with the new ones using the dictionary of names.

```
for(i in 1:5429){
  cora[i, 1] = name_dict[as.character(cora[i, 1]),1]
  cora[i, 2] = name_dict[as.character(cora[i, 2]),1]
}
```

And to check this has worked we print the union of entries in the cora data frame.

```
head(union(cora[,1], cora[,2]))
```

```
## [1] 1 2 3 4 5 6
```

Great, the nodes have been renamed to consecutive integers. The function `graph_from_edgelist` in `igraph` now outputs the correct graph.

```
G <- graph_from_edgelist(as.matrix(cora), directed = F)
a <- get.adjacency(G)
```

And as a final check we observe the dimension of the adjacency matrix is what we expected.

```
dim(a)
```

```
## [1] 2708 2708
```

## 2. GRDP Embedding

We produce a $d$ dimensional embedding of the adjacency matrix of the Cora network using SVD and generate new adjacency matrices via the generalised random dot product model described by Rubin-Delanchy et al. We start by deriving the $d$ largest eigenvalues by magnitude.

```
e <- eigen(a)

largest_eigenvalue_indices <- c()
eigenvalues <- c()
for(i in 1:2708){
  if(sign(e$values[i]) * e$values[i] > 4.5){
    largest_eigenvalue_indices <- c(largest_eigenvalue_indices,
                                    i)
    eigenvalues <- c(eigenvalues, e$values[i])
  }
}
```

The columns of `e$vectors` are the eigenvectors, we restrict to the ones found above and produce the embedding.

```
dot <- diag(sign(eigenvalues))
U <- e$vectors[,largest_eigenvalue_indices]
X <- U %*% diag(sqrt(abs(eigenvalues)))
```

```
A_SVD <- X %*% dot %*% t(X)
```

## 3. Symmetric Logistic Embedding

We embed the adjacency matrix of the Cora network using gradient descent and generate new graphs according to symmetric logistic model.

We add the code running the symmetric logistic embedding.

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

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

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)
}

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))
}

# ADAM optimiser.
initAdam = function(epsilon = 0.01, beta1 = 0.9, beta2 = 0.999, delta = 1E-8){
  epsilon <<- epsilon
  beta1 <<- beta1
  beta2 <<- beta2
  delta<<-delta
  r_x<<-0
  s_x<<-0
  t_x<<-0
  r_y<<-0
  s_y<<-0
  t_y<<-0
}
stepx = function(gradientEst) {
  t_x <<- t_x + 1
  s_x <<- beta1*s_x + (1-beta1)*c(gradientEst)
  r_x <<- beta2*r_x + (1-beta2)*(c(gradientEst)^2)
  s_hat = s_x / (1-beta1^t_x)
  r_hat = r_x / (1-beta2^t_x)
  inc = - epsilon * s_hat / (sqrt(r_hat) + delta)
  return(inc)
}
stepy = function(gradientEst) {
  t_y <<- t_y + 1
  s_y <<- beta1*s_y + (1-beta1)*c(gradientEst)
  r_y <<- beta2*r_y + (1-beta2)*(c(gradientEst)^2)
  s_hat = s_y / (1-beta1^t_y)
  r_hat = r_y / (1-beta2^t_y)
  inc = - epsilon * s_hat / (sqrt(r_hat) + delta)
  return(inc)
}
```

We first adapt the adjacency matrix ready for the gradient descent.

```
Modified_a <- as.matrix(tilde(a))
```

We initialise the embedding with 16 dimensions and find the initial loss.

```
X <- initial(2708, 16)
dot <- diag(x = c(-1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1),
            16, 16)
Y <- X %*% dot

loss_list <- loss_and_gradients(X, Y, Modified_a)
print(paste("Initial Loss: ", loss_list[[1]]))
```

We first descent rapidly for 50 steps.

```
initAdam(0.1, 0.9, 0.99, 1E-7)

for (i in 1:50) {
  loss_list = loss_and_gradients(X,Y,Modified_a)
  X <- X + stepx(loss_list[[2]])
  Y <- X %*% dot
  if(i %% 5 == 0){
    print(i)
    print(paste("Loss: ", loss_list[[1]]))
  }
}
```

And now to descend in finer detail for 500 steps.

```
initAdam(0.01, 0.9, 0.99, 1E-7)

for (i in 1:500) {
  loss_list = loss_and_gradients(X,Y,Modified_a)
  X <- X + stepx(loss_list[[2]])
  Y <- X %*% dot
  if(i %% 50 == 0){
    print(i)
    print(paste("Loss: ", loss_list[[1]]))
  }
}
```

```
A_SL <- X %*% dot %*% t(X)
```

## 4. Recovery of Low Degree Structure

We now replicate the plots produced in Seshadri's PNAS paper for the SVD and Symmetric Logistic cases.

```
Simulated_Adjacency <- function(A){
  Simulated_A <- A
  for(i in 1:2708){
    for(j in i:2708){
      if(A[i,j] > 1){
        Simulated_A[i,j] = 1
        Simulated_A[j,i] = 1
      }
      if(A[i,j] < 0){
        Simulated_A[i,j] = 0
        Simulated_A[j,i] = 0
      }
      else{
        edge <- as.integer(rbernoulli(1, A[i,j]))
        Simulated_A[i,j] = edge
        Simulated_A[j,i] = edge
      }
    }
  }
  for(i in 1:2708){
    Simulated_A[i,i] = 0
  }
  return(Simulated_A)
}
```

Adams final blocks of code should now be appropriate after constructing the graphs with the following simulated adjacency matrices. And in true Blue Peter fashion we read first read precomputed versions of the embedding matrices in.

```
A_SVD <- read.csv("/Users/billnunn/Desktop/A_SVD.csv")[,2:2709]
A_SVD <- as.matrix(A_SVD)

A_SL <- read.csv("/Users/billnunn/Desktop/A_SL.csv")[,2:2709]
A_SL <- as.matrix(A_SL)

Simulated_SVD <- Simulated_Adjacency(A_SVD)
Simulated_SL <- Simulated_Adjacency(A_SL)
```

Adams code with some of the variables renamed for consistency:

```
G_SVD = graph_from_adjacency_matrix(Simulated_SVD, mode="undirected")
G_SL = graph_from_adjacency_matrix(Simulated_SL, mode="undirected")

delta = rep(0, max(degree(G)))
delta_SVD = rep(0, max(degree(G_SVD)))
delta_SL = rep(0, max(degree(G_SL)))

combined = union(cora[,1], cora[,2])
r = length(combined)

for (i in 1:max(degree(G))){
  too_big = c()
  for (j in 1:2708){
    if (degree(G,j) > i){
      too_big[length(too_big) + 1] = combined[j]
    }
  }
  G1 = delete_vertices(G,too_big)
  delta[i] = sum(count_triangles(G1)) / vcount(G)
}

for (i in 1:max(degree(G_SVD))){
  too_big = c()
  for (j in 1:r){
    if (degree(G_SVD,j) > i){
      too_big[length(too_big) + 1] = combined[j]
    }
  }
  G1 = delete_vertices(G_SVD,too_big)
  delta_SVD[i] = sum(count_triangles(G1)) / vcount(G_SVD)
}

for (i in 1:max(degree(G_SL))){
  too_big = c()
  for (j in 1:r){
    if (degree(G_SL,j) > i){
      too_big[length(too_big) + 1] = combined[j]
    }
  }
  G1 = delete_vertices(G_SL,too_big)
  delta_SL[i] = sum(count_triangles(G1)) / vcount(G_SL)
}
```

And finally we can replicate Seshadri's plot.

```
plot(c(1:max(degree(G))),delta,log='xy',type='l')
```

```
## Warning in xy.coords(x, y, xlabel, ylabel, log): 1 y value <= 0 omitted from
## logarithmic plot
```

```
lines(c(1:max(degree(G_SVD))),delta_SVD, col='red')
lines(c(1:max(degree(G_SL))),delta_SL, col='blue')
```