Moreen:Kaggle competition #2

2023-03-03

Reading in the data sets.

The first step was to read in the data sets into *R-studio*.The data sets were the **test_set_rna**, **training_set_rna** and **training_set_adt**

```
test_rna<-as.matrix(read.csv("test_set_rna.csv",row.names = 1))
train_rna<-as.matrix(read.csv("training_set_rna.csv",row.names = 1))
train_adt<-as.matrix(read.csv("training_set_adt.csv",row.names = 1))
```

Transforming and scaling data.

There was no need to scale the data since according to the professor both RNA and ADT data has been standard log-normalized. Highly variable genes were selected for analysis, and non-variable ADT proteins were removed.

Multivariate regression function.

In coding the multivariate function i used the approach suggested by the professor where i Considered AX=Y where A = train_rna, Y = train_adt, and then solving for X.Then i considered A2X = Y2 where A2 = test_RNA and Y2 = test_ADT (what i want to predict).

The function takes in four parameters **A_train** which is the predictor matrix,**Y_train** which is the response/dependent variable matrix,**A_test** which is the predictor matrix for the response matrix we are trying to predict and **k** which is parameter standing for which method we want use to solve the multivariate system with.

Step 1 The function starts by defining 3 nested functions that are the 3 different algorithms that can be used to solve the multivariate system.An explanation of the three different functions is as given below:

- **Gaussian_solver**: The function is defined with these parameters: A a matrix which equals (AˆTA) and b a vector which equals (AˆTY)

1) First, the function checks if the matrix A is square and if the vector b has the same number of rows as A.
2) Then, the function uses Gaussian elimination to reduce the augmented matrix [A|b] to row echelon form. This involves finding the pivot elements (non-zero elements on the diagonal) and using them to eliminate the elements below each pivot element in the same column. This is done using a nested loop structure that iterates over the rows and columns of the matrix.
3) After the matrix is in row echelon form, the function checks that the last pivot element is non-zero. If it is zero, then the system has no unique solution, and the function returns an error.
4) Finally, the function performs back-substitution to find the solution vector x. This involves starting at the last row of the matrix and solving for the corresponding element of x, then substituting that value into the previous row and solving for the next element of x, and so on until all elements of x have been found.

- **NNLS_solve**:The function is defined with these parameters: *a* a matrix which equals (AˆTA) and *b* a vector which equals (AˆTY), and optional parameters *max_iter* for the maximum number of iterations to perform and *tol_cutoff* for the tolerance cutoff value for convergence. The output is the vector of unknowns X (represented by the variable x).

1) The function iteratively updates the values of X until the solution converges. At each iteration, the function calculates the residuals b - aX and updates the unknowns in X by adding a portion of the residual to each unknown. If adding the full portion of the residual would result in a negative value for any unknown, that unknown is set to zero instead.

2) The function stops iterating when the maximum number of iterations is reached or the change in X between iterations is below the tolerance cutoff value. The function then returns the final vector of unknowns X

- **Coordinate_descent**:The function is defined with these parameters: *A* a matrix which equals (AˆTA) and *b* a vector which equals (AˆTY) in an equation AX=Y,and optional parameters *max_iter* for the maximum number of iterations to perform and *tol_cutoff* for the tolerance cutoff value for convergence. The output is the vector of unknowns X (represented by the variable x).

1) This function implements the coordinate descent algorithm for solving a linear system of equations of the form Ax = b, where A is a square matrix and b is a vector. The algorithm starts with an initial guess for x, and at each iteration, it updates one coordinate (i.e., one element) of x, while keeping the others fixed. Specifically, for each coordinate i, it solves for xi using the other variables, updates xi in the solution vector, and repeats the process for the next coordinate. The algorithm stops when the residual error (i.e., the difference between Ax and b) falls below a given tolerance level or when the maximum number of iterations is reached. The function returns the solution vector x.

Step2 After defining the 3 functions listed above the function computes the *ATA* matrix by taking the tcrossprod of (A_train) and also calculates the *ATy* matrix by computing tcrossprod(A_train, Y_train).An *X* matrix is also initialized to store the solutions with rows equal to the columns in *ATA* and columns equal to number of columns in ATy

Step3 The function then uses *If-else* statements to allow the user to specify which algorithm they would like to use to solve for *X*,Where K == *'G'* would mean Gaussian_solver, K == *'N'* would mean NNLS_solve and K== *'C'* would mean Coordinate_descent

Step4 Finally it calculates the predicted values by taking the crossproduct of X and the A_test matrix and returns a list with the predicted values matrix *pred* and the *X* matrix.

```
predictor <- function(A_train, Y_train, A_test, K) {

    Gaussian_Solver <- function(A, b) {
        # Check that A is a square matrix
        if (dim(A)[1] != dim(A)[2]) {
            stop("Error: A must be a square matrix")
        }

        # Check that b is a vector with the same number of
        # rows as A
        if (length(b) != dim(A)[1]) {
            stop("Error: b must be a vector with the same number of rows as A")
        }

        # Solve the system of linear equations using
```

```r
# Gaussian elimination
n <- dim(A)[1]
Ab <- cbind(A, b)

# For each row i from 1 to n-1, where n is the
# number of rows in the matrix, the function finds
# the pivot element (i.e., the element in column i
# and row i), and checks that it is not equal to
# zero. If the pivot element is zero, the function
# stops and returns an error message.It then
# normalizes the pivot row by dividing it by the
# pivot element

for (i in 1:(n - 1)) {
    # Find the pivot element
    pivot <- Ab[i, i]
    if (pivot == 0) {
        stop("Error: pivot element is zero")
    }

    # Normalize the pivot row
    Ab[i, ] <- Ab[i, ]/pivot

    # Subtract the pivot row from the remaining
    # rows: The function then subtracts the pivot
    # row multiplied by a factor from each of the
    # remaining rows to eliminate the entries below
    # the pivot element in column i. The factor is
    # the element in column i and row j, where j is
    # the row number of the row being eliminated.
    for (j in (i + 1):n) {
        factor <- Ab[j, i]
        Ab[j, ] <- Ab[j, ] - factor * Ab[i, ]
    }
}

# Check that the last pivot element is not zero
if (Ab[n, n] == 0) {
    stop("Error: last pivot element is zero")
}

# Back-substitute to find the solution vector

# initializes x as a vector of length n, where n is
# the number of rows in the matrix A.
x <- numeric(n)
# computes the last entry of x by dividing the last
# element of the last row of Ab (which represents
# the last equation in the system) by the last
# pivot element.
x[n] <- Ab[n, n + 1]/Ab[n, n]

# The function then iterates over the remaining
```

```
    # rows from n-1 to 1 and computes the remaining
    # entries of x using the formula: x[i] = (Ab[i,
    # n+1] - sum(Ab[i, (i+1):n] * x[(i+1):n])) / Ab[i,
    # i]. This formula involves subtracting the sum of
    # the products of the remaining entries of row i
    # and the corresponding entries of x from the
    # right-hand side of equation i, and then dividing
    # by the pivot element in row i.
    for (i in (n - 1):1) {
        x[i] <- (Ab[i, n + 1] - sum(Ab[i, (i + 1):n] * x[(i +
            1):n]))/Ab[i, i]
    }

    return(x)
}

NNLS_solve <- function(a, b, max_iter = 10000, tol_cutoff = 1e-09) {
    # Initializing the unknown vector x to all zeros.
    x <- rep(0, length(b))
    for (iter in 1:max_iter) {
        tol <- 0
        for (i in 1:length(b)) {
            # For each entry in the unknown vector x,
            # calculate the increment
            z <- b[[i]]/a[i, i]

            # If the increment z would result in a
            # negative value for the unknown x_i, set
            # x_i to zero instead and adjust the
            # increment z accordingly.
            if ((x[[i]] + z) < 0) {
              z <- -x[[i]]
              x[[i]] <- 0
            } else {
              x[[i]] <- x[[i]] + z
            }
            # Update the residuals by subtracting z
            # times the corresponding column of the
            # matrix a from b
            b <- b - z * a[, i]

            # Calculate the tolerance as the sum of the
            # absolute values of z / (x_i + 1e-15) over
            # all entries in the unknown vector x.
            tol <- tol + abs(z/(x[[i]] + 1e-15))
        }
        if (iter%%100 == 0)
            cat(" iter: ", iter, ", tol: ", tol, "\n")
        if (tol < tol_cutoff)
            break
    }
    # Return the final vector of unknowns x.
    return(x)
```

```
}

Coordinate_descent <- function(A, b, max_iter = 1000, tol = 1e-06) {
    # Check that A is a square matrix
    if (dim(A)[1] != dim(A)[2]) {
        stop("Error: a must be a square matrix")
    }

    # Check that b is a vector with the same number of
    # rows as A
    if (length(b) != dim(A)[1]) {
        stop("Error: b must be a vector with the same number of rows as A")
    }

    # Initialize solution vector x
    n <- dim(A)[1]
    x <- rep(0, n)

    # Perform coordinate descent
    for (iter in 1:max_iter) {
        for (i in 1:n) {
            # Solve for xi using the other variables
            # For a given coordinate i, the function
            # solves for xi using the other variables
            # by subtracting the contribution of xi to
            # the left-hand side of the equation Ax=b,
            # which is A[i,] %% x - A[i,i] * x[i]. The
            # term A[i,] %% x is the sum of the
            # products of the ith row of A and the
            # corresponding values of x, and A[i,i] *
            # x[i] is the contribution of xi to this
            # sum. This gives us an expression for xi
            # in terms of the other variables.
            Aixi <- A[i, ] %*% x - A[i, i] * x[i]
            xi <- (b[i] - Aixi)/A[i, i]

            # Update xi in the solution vector
            x[i] <- xi
        }

        # Check for convergence The function checks if
        # the residual error (i.e., the difference
        # between Ax and b) falls below a given
        # tolerance level tol (default 1e-6). If the
        # residual error is below the tolerance level,
        # the loop is terminated.
        residual <- norm(A %*% x - b, type = "2")
        if (residual < tol) {
            break
        }
    }

    # Return solution vector x
```

```
        return(x)
    }

    # Here matrix ATA and ATy are computed from the inputs
    # of the function
    ATA <- tcrossprod(A_train)
    ATy <- tcrossprod(A_train, Y_train)
    n <- ncol(ATy)  # number of columns in ATy
    X <- matrix(0, nrow = ncol(ATA), ncol = n)  # create  an matrix to store solutions
    for (i in 1:n) {
        # Choosing which function to use to solve for X
        if (K == "G") {
            X[, i] <- Gaussian_Solver(ATA, ATy[, i])  # Gaussian elimination
        }
        if (K == "N") {
            X[, i] <- nnls_solve(ATA, ATy[, i])  # Non Negative least square/gradient descent
        }
        if (K == "C") {
            X[, i] <- coordinate_descent(ATA, ATy[, i])  # Coordinate_descent
        }
    }

    # Computing the predicted values using the final value
    # of the X matrix
    pred <- crossprod(X, A_test)
    return(list(pred = pred, X = X))
}
```

The code below shows how the predictor function defined above is used on our data sets.We pass the *train_rna*, *train_adt* ,*test_rna* and *'G'* indicating Gaussian_solver as our arguments.

NB:The Gaussian method produced our teams highest score of 0.8022.

```
predictor_model= predictor(train_rna, train_adt, test_rna,'G')
```

To get the predicted values from our model we use the dollar sign between the name of our model and the output we want which is the predicted values in this case *pred* as shown below.

```
predicted_testadt<- predictor_model$pred
```

We then use the process provided by the professor to format the out put in readiness for submission.First we use the melt function to reshape it from wide to long format, assigning values to the appropriate columns and then saving the data as a csv document as shown below

```
sample_submission <- reshape2::melt(predicted_testadt)
head(sample_submission)
```

```
##   Var1          Var2     value
## 1    1 test_sample_1 1.3478993
## 2    2 test_sample_1 2.6607981
## 3    3 test_sample_1 1.1291860
## 4    4 test_sample_1 0.6005354
## 5    5 test_sample_1 2.6112450
## 6    6 test_sample_1 1.4908048
```

```r
# Then assign the required values to the designated columns *ID* and *Expected* as shown below

sample_submission <- data.frame(
  "ID" = paste0("ID_", 1:nrow(sample_submission)),
  "Expected" = sample_submission$value)
head(sample_submission)
```

```
##     ID  Expected
## 1 ID_1 1.3478993
## 2 ID_2 2.6607981
## 3 ID_3 1.1291860
## 4 ID_4 0.6005354
## 5 ID_5 2.6112450
## 6 ID_6 1.4908048
```

```r
# Lastly we save our submissions as a csv document.

write.csv(sample_submission, "my_submissionGuassian.csv", row.names = FALSE)
```

Results from the Multivariate function

Out of the three methods methods in the function the Gaussian produced the best score of 0.8022 followed by the Coordinate descent method that had a score of 0.80127.The NNLS method took quite a bit of time to run.

contribution to team

My contribution to the team was creating the Gaussian_Solver function which helped us get our highest score.

Other methods tried

I also tried using NMF to predict the train_adt values.

The function takes four arguments,A: the matrix to be factorized,K: the rank of the factorization (number of columns in W and rows in H),*mask*: a binary matrix that specifies which entries of A should be used in the factorization (0 = use entry, 1 = ignore entry),*max_iter*: the maximum number of iterations to perform and *eps*: a small positive value used to avoid division by zero

The function initializes matrices W and H randomly, then iteratively updates the matrices until convergence or the maximum number of iterations is reached.It uses alternating least squares to update W and H.

In the update of H It updates the jth column of H by multiplying it with a positive factor that minimizes the squared error between the masked elements of A[,j] and the corresponding elements in W %*% H[,j] while keeping all the other columns of H fixed. The mask matrix is used to exclude entries from the calculation that should not be used for the factorization.

To update W the function updates the zth row of W by multiplying it with a positive factor that minimizes the squared error between the masked elements of A[z,] and the corresponding elements in W[z,] %*% H. Again, the mask matrix is used to exclude entries from the calculation.

After each iteration, the function checks for convergence by calculating the correlation between the predicted matrix W %*% H and the original matrix A. If the correlation exceeds a pre-defined threshold (stop_threshold = 0.85), the function stops iterating and returns the final values of W, H, and the predicted

matrix. If the correlation is below the threshold, the function continues iterating until the maximum number of iterations is reached.

At the end of the function, the predicted matrix is calculated using the final values of W and H, and the function returns a list containing the final values of W, H, and the predicted matrix *pred*

```
nmf_ALS3 <- function(A, K, mask, max_iter = 100, eps = 1e-06,
    verbose = TRUE) {
    # Initializing W and H with random numbers
    W <- matrix(runif(nrow(A) * K), nrow(A), K)
    H <- matrix(runif(K * ncol(A)), K, ncol(A))

    # Initialize variables for storing previous iterations'
    # values
    W_prev <- W
    H_prev <- H
    prev_corr <- cor(A, W %*% H)
    stop_threshold <- 0.85  # Stopping criteria

    # Perform ALS updates
    for (i in 1:max_iter) {
        # Update H
        for (j in 1:ncol(A)) {
            H[, j] <- H[, j] * (t(W[mask[, j] == 0, ]) %*% A[mask[,
                j] == 0, j])/(t(W[mask[, j] == 0, ]) %*% W[mask[,
                j] == 0, ] %*% H[, j] + eps)
        }

        # Update W
        for (z in 1:nrow(A)) {
            W[z, ] <- W[z, ] * (A[z, which(mask[z, ] == 0)] %*%
                t(H[, which(mask[z, ] == 0)]))/(W[z, ] %*% H[,
                which(mask[z, ] == 0)] %*% t(H[, which(mask[z,
                ] == 0)]) + eps)
        }

        # Check convergence Calculate predicted matrix
        pred <- W %*% H

        # Calculate correlation between predicted and
        # actual matrix
        curr_corr <- cor(as.vector(pred[, 1:3000]), as.vector(A[,
            1:3000]))
        print(curr_corr)

        # Check convergence
        if (curr_corr >= stop_threshold) {
            break
        }
        prev_corr <- curr_corr

        # Update previous iterations' values
        W_prev <- W
        H_prev <- H
```

```
    }

    # Calculate predicted matrix
    pred <- W %*% H

    # Return NMF model
    list(W = W, H = H, pred = pred)
}
```

In the code below i bind the matrices to get matrix *A_bind* that has all the data including the a portion of unknown values the test_adt which we want to predict.

```
A_train1 <- rbind(train_adt, train_rna)
B <- rbind(matrix(0, 25, 1000), test_rna)
A_bind <- cbind(A_train1, B)
```

In the code chunk below we create the masked matrix with 1's representing the unknown values which we need to predict and 0's with the values which we already know to help in knowing which values to use when updating H and W matrices in the NMF model .

```
mask <- matrix(0, nrow(A_bind), ncol(A_bind))
mask[1:25, 4001:5000] <- 1
```

Then i use the nmf model defined above to predict the matrix A_bind including the unknown values.

```
modelnmf=nmf_ALS3(A_bind, 13, mask, max_iter=100, eps=1e-6, verbose=FALSE)
```

```
## [1] 0.7816916
## [1] 0.7879489
## [1] 0.7885812
## [1] 0.78917
## [1] 0.7897411
## [1] 0.7903156
## [1] 0.7909177
## [1] 0.7915763
## [1] 0.792327
## [1] 0.7932142
## [1] 0.7942926
## [1] 0.7956267
## [1] 0.7972865
## [1] 0.7993339
## [1] 0.8017957
## [1] 0.8046248
## [1] 0.8076716
## [1] 0.8107055
## [1] 0.8134968
## [1] 0.8159004
## [1] 0.8178783
## [1] 0.8194689
## [1] 0.8207431
## [1] 0.8217744
## [1] 0.8226265
```

```
## [1] 0.8233498
## [1] 0.8239827
## [1] 0.8245539
## [1] 0.8250841
## [1] 0.8255881
## [1] 0.8260763
## [1] 0.8265559
## [1] 0.8270314
## [1] 0.8275056
## [1] 0.82798
## [1] 0.8284559
## [1] 0.8289338
## [1] 0.8294141
## [1] 0.829897
## [1] 0.8303824
## [1] 0.8308698
## [1] 0.8313584
## [1] 0.8318474
## [1] 0.8323356
## [1] 0.8328217
## [1] 0.8333041
## [1] 0.8337813
## [1] 0.8342514
## [1] 0.8347127
## [1] 0.8351636
## [1] 0.8356024
## [1] 0.8360278
## [1] 0.8364387
## [1] 0.8368342
## [1] 0.8372136
## [1] 0.8375765
## [1] 0.8379228
## [1] 0.8382525
## [1] 0.8385656
## [1] 0.8388627
## [1] 0.839144
## [1] 0.8394102
## [1] 0.839662
## [1] 0.8399
## [1] 0.840125
## [1] 0.8403379
## [1] 0.8405394
## [1] 0.8407304
## [1] 0.8409116
## [1] 0.8410837
## [1] 0.8412476
## [1] 0.8414037
## [1] 0.8415528
## [1] 0.8416954
## [1] 0.841832
## [1] 0.8419631
## [1] 0.842089
## [1] 0.8422101
## [1] 0.8423268
```

```
## [1] 0.8424394
## [1] 0.8425481
## [1] 0.8426532
## [1] 0.842755
## [1] 0.8428537
## [1] 0.8429493
## [1] 0.8430422
## [1] 0.8431325
## [1] 0.8432204
## [1] 0.8433058
## [1] 0.8433891
## [1] 0.8434702
## [1] 0.8435494
## [1] 0.8436266
## [1] 0.843702
## [1] 0.8437756
## [1] 0.8438476
## [1] 0.8439179
## [1] 0.8439867
## [1] 0.8440541
## [1] 0.8441199
```

I then extract the predicted values from the model and again extract only the part of the matrix that had the unknown values as shown in the code below using their indices.

```
predicted_values <- modelnmf$pred
test_adt2 <-predicted_values[1:25, 4001:5000]
```

Finally i follow the same format as previously described above and have the data ready for submission on kaggle.

```
sample_submission2 <- reshape2::melt(test_adt2)
head(sample_submission2)
```

```
##   Var1 Var2     value
## 1    1    1 1.5074507
## 2    2    1 2.4827405
## 3    3    1 1.2810555
## 4    4    1 0.4259287
## 5    5    1 2.6956821
## 6    6    1 0.8914635
```

```
sample_submission2 <- data.frame(
  "ID" = paste0("ID_", 1:nrow(sample_submission2)),
  "Expected" = sample_submission2$value)
head(sample_submission2)
```

```
##     ID  Expected
## 1 ID_1 1.5074507
## 2 ID_2 2.4827405
## 3 ID_3 1.2810555
## 4 ID_4 0.4259287
## 5 ID_5 2.6956821
## 6 ID_6 0.8914635
```

```
write.csv(sample_submission2, "my_submissionnmf.csv", row.names = FALSE)
```

Results from the NMF model

The results from the NMF model were not as great as the results from the Multivariate models.The highest score i got from my model was 0.7016 but i think with modifications it would have performed better than this.