SA07 - Drop-Seq NMF Analysis

Jack Cazet

October 15, 2018, updated on July 8, 2019

Preparing Single Cell Data for NMF

To perform NMF, we need to generate a matrix of log-normalized single cell read counts. This can be readily extracted from a Seurat object's data slot; however, we do not include the full dataset because the NMF computation becomes prohibitively slow when dealing with very large matrices. In addition, there are many genes that are not biologically informative (due to high levels of noise or low variability), which could obfuscate meaningful signal in the data. Thus, we limit the analysis to a set of variable genes selected using Seurat (typically between ~1500-2000). After subsetting, the expression matrix is saved as a .csv file with genes as rows and cells as columns.

```r
# read in the pre-made Seurat object
ds.ds <- readRDS("objects/ds.en.rds")

# check the length of the variable gene list if the number is
# too large it could introduce unwanted noise and
# dramatically extend computation time
cat("Number of variable genes  \n", length(ds.ds@var.genes),
    "  \n")
```

Number of variable genes
3987

```r
# If there are too many variable genes you can tweak the
# criteria for variable genes to lower that number
ds.ds <- FindVariableGenes(object = ds.ds, mean.function = ExpMean,
    dispersion.function = LogVMR, x.low.cutoff = 0.21, x.high.cutoff = 4,
    y.cutoff = 0.18, display.progress = F, do.plot = F)

cat("Number of variable genes  \n", length(ds.ds@var.genes),
    "  \n")
```

Number of variable genes
1769

```r
# isolate the normalized read data for the variable genes and
# save as a csv
ds.df <- ds.ds@data
ds.df <- ds.df[rownames(ds.df) %in% ds.ds@var.genes, ]
ds.df <- as.data.frame(as.matrix(ds.df))

write.csv(ds.df, file = "NMF_Matrix.csv")

kable(ds.df[1:10, 1:5], format = "latex", caption = "Normalized Expression Data Format Example",
    digits = 3) %>% kable_styling(latex_options = c("hold_position",
    "scale_down", font_size = 7), full_width = F)
```

Table 1: Normalized Expression Data Format Example

	01-D1_ACGTACACACGA	01-P2_AATAGCCGGTGA	02-CO_GCATCAAAGTGN	02-CO_GCAGTCAAAGTG	02-P1_CCGGTCTATTAG
t34456aep|TAF3_CHICK	0.000	0.000	0.000	0.000	1.134
t32664aep|AQP9_HUMAN	1.227	1.364	0.000	1.258	0.000
t38118aep|AGRIN_MOUSE	0.000	0.000	0.000	0.000	0.000
t33743aep|EXOC7_RAT	0.000	1.079	0.000	1.258	0.000
t32686aep|MRP1_CHICK	0.000	1.079	1.286	1.258	0.000
t35860aep|KIF3A_MACFA	0.590	0.000	0.000	0.000	0.000
t32310aep|LSM1_HUMAN	0.000	0.678	0.000	1.258	0.000
t35979aep|ECI1_RAT	0.590	0.000	0.000	1.258	0.000
t35994aep|MLRP2_ACRMI	0.590	1.079	0.000	1.258	1.134
t32481aep|GUAA_RAT	0.590	0.000	1.286	0.000	0.000

NMF analysis

We used the run_nmf.py function from a previously published NMF analysis (Farrell et al., 2018) to analyze the data (https://github.com/YiqunW/NMF/; commit 6fbb023) using the following parameters: -rep 20 -scl "false" -miter 10000 -perm True -run_perm True -tol 1e-7 -a 2 -init "random" -analyze True. Each NMF analysis was repeated 20 times using different randomly initialized conditions, enabling us to evaluate reproducibility. Additionally, the analysis was initially performed using a broad range of K values. The results were then compiled using the integrate_and_output.sh script, and evaluated to identify the range of K values that gave the best results (see below). This then guided a second round of analyses using a narrower set of K values (ten total K values by steps of one), from which a final set of NMF results were selected.

NMF Metagene Quality assessment (estimating the factorization rank)

Because the optimal number of NMF metagenes that can describe a dataset cannot be determined *a priori*, this parameter (referred to as K) needs to be determined empirically. To guide our selection of an optimal K value, we identified the point at which increasing K no longer increased the number of meaningful metagenes. We considered a metagene to be meaningful if it reproducibly described the covariation of multiple genes. Using a K value at the point where the number of informative metagenes becomes saturated allows us to maximize the resolution at which we analyze gene co-expression.

Because metagenes that recur across replicates for a particular K will not necessarily be perfectly identical, we developed an approach to link metagenes from different replicates so that we could evaluate reproducibility on a metagene-by-metagene basis. The rationale for our approach was that reproducible and robust metagenes should reliably cluster together similar groups of cells cells in independent replicate analyses. First, we clustered cells based on their highest scoring metagene (after normalizing all metagene cell scores to have a maximum value of 1). Then, metagenes were linked from one replicate to another by identifying the metagene in the second replicate that had the highest number of cells in common with the metagene in the first replicate.

To estimate a metagene's reproducibility, we created a metric (cluster reproducibility score) to evaluate the robustness of the metagene-based clustering. Specifically, we calculated the average proportion of cells that were clustered together in one replicate that were also clustered together in all other replicates. For example, a poorly reproducible metagene might group together 100 cells in one replicate, but in another replicate only cluster together 20 of the original 100 cells, giving the metagene a cluster reproducibility score of 0.2. In contrast, a highly reproducible metagene would have a score close to 1.

To estimate the approximate number of genes that drive a particular metagene, we determined the average number of genes whose score was within one order of magnitude of the top scoring gene across replicates. Our criteria for an informative metagene was that it should have, on average, more than 10 genes, and should have a cluster reproducibility score above 0.6. Metagenes that did not fulfill these criteria were disregarded after final K selection.

NMF Over Broad K Range

This example below shows the implementation of the factorization rank selection scheme described above for the subset of endodermal epithelial cells. Initially, we performed NMF for K values ranging from 10 to 100 by steps of 5. The python NMF function generates a R object containing all cell and gene scores for all replicates from all K values. We import this object, determine the number of informative metagenes for each K, and plot the results to identify the point of metagene saturation.

```r
# import robj from python script
load("nmf/en_K40/Endo_Broad_NMF.Robj")

# rename object
res <- result_obj

rm(result_obj)


# get list of k values that were used
getKs <- names(res)

getKs <- strsplit(getKs, "=")

getKs <- as.numeric(vapply(getKs, function(x) x[2], ""))


# this function takes as an argument an index referring to a
# position in the list of K values within the results and
# determines both the average number of genes driving each
# metagene and the metagene's cluster consistency score
findConsistMeta <- function(q) {

    # look at a single K value
    kval <- res[[q]]

    # initialize objects into which we will put the average
    # number of genes/consistency scores per metagene
    consistMeta <- NULL
    goodMeta <- NULL

    # this for loop moves through each metagene
    for (j in 1:length(colnames(kval[[1]][["G"]]))) {

        # get metagenes from the first replicate
        runRep <- kval[[1]]

        # first we'll identify the cells that are clustered into that
        # metagene

        # get Cell scores
        cellScores <- runRep[["C"]]

        # normalize scores so that the top cell has a score of one
        cellScores <- apply(cellScores, 1, function(x) x/max(x))
        # assign cluster ID based on top metagene score
        MaxClust <- apply(cellScores, 1, function(x) colnames(cellScores)[which.max(x)])

        # look at all cells belonging to the current metagene being
        # considered
        clustCells <- names(MaxClust[MaxClust == colnames(cellScores)[j]])

        # next we'll count the number of genes within one order of
        # magnitude of the top scoring gene

        # pull gene scores
        geneScores <- runRep[["G"]]

        # determine how many genes are within an order of magnitude
        # of the top gene
        geneCount <- geneScores[, paste0("X", colnames(cellScores)[j])]
        geneCount <- length(geneCount[geneCount >= (max(geneCount)/10)])

        # look at these clustered cells in the other replicates

        # initialize the objects into which we will put the results
        # for each run (within one K, within one metagene)
        metaConsistency <- NULL

        metaGeneCount <- geneCount

        # disregard the for loop if there are no cells assigned to
        # the cluster in question
        if (length(clustCells) != 0) {

            # this for loop moves through each replicate (excluding the
            # first, which we already looked at)
            for (i in 2:length(kval)) {

                # get cell scores
                otherRep <- kval[[i]][["C"]]

                # normalize so that the top gene has a score of 1
                otherRep <- apply(otherRep, 1, function(x) x/max(x))

                # assign cells to a cluster
                otherRep <- apply(otherRep, 1, function(x) colnames(otherRep)[which.max(x)])

                # isolate the cells that were grouped together into a cluster
                # in the first replicate
                otherRep <- otherRep[names(otherRep) %in% clustCells]

                # ask how many of those cells belong to the same cluster
                otherRep <- table(otherRep)

                # add it to the rep list
                metaConsistency <- c(metaConsistency, max(otherRep)/length(clustCells))
                # pull the metagene name of the best hit (determined by cell
                # overlap) and determine how many genes are within an order
                # of magnitude of the top gene

                # pull the name of the metagene that best matches the query
                # metagene
                otherRepMetaHit <- names(otherRep[order(-otherRep)])[1]

                # get gene scores for the rep in question
                otherRep.genes <- kval[[i]][["G"]]

                otherRep.genes <- otherRep.genes[, paste0("X",
                  otherRepMetaHit)]

                # how many genes are within an order of magnitude of the top
                # gene?
                otherRep.genes <- length(otherRep.genes[otherRep.genes >=
                  (max(otherRep.genes)/10)])

                # add results to the rep list
                metaGeneCount <- c(metaGeneCount, otherRep.genes)
            }
        } else {
            metaConsistency <- 0
            metaGeneCount <- 0
        }

        # compile the results per metagene
        consistMeta <- c(consistMeta, mean(metaConsistency))
        goodMeta <- c(goodMeta, mean(metaGeneCount))
    }

    # compile results into a data frame for each K value
    goodMetaCount <- data.frame(metaGeneID = colnames(kval[[1]][["G"]]),
        consistency_score = consistMeta, gene_count = goodMeta)
    return(goodMetaCount)
}

# because running findConsistMeta can be time-consuming, we
# can use a parallel for loop
library(doParallel)
cl <- makeCluster(6)
registerDoParallel(cl)

# run findConsistMeta across all K values
allKconsistMeta <- foreach(i = 1:length(res)) %dopar% findConsistMeta(i)

# Exclude those metagenes that don't meet QC thresholds
allKconsistMeta.IDs <- lapply(allKconsistMeta, function(x) as.character(x[,
    1][x[, 2] > 0.6 & x[, 3] > 10]))

# give each DF it's correct name
names(allKconsistMeta.IDs) <- getKs
```

```r
# get number of informative metagenes for each K value
numMetaKeep <- vapply(allKconsistMeta.IDs, length, numeric(1))

numMetaKeep <- data.frame(kval = getKs, MK = numMetaKeep)

kable(allKconsistMeta[[5]], format = "latex", caption = "Example Results for a Single K Value",
    digits = 3) %>% kable_styling(latex_options = c("hold_position"),
    full_width = F)
```

Table 2: Example Results for a Single K Value

metaGeneID	consistency_score	gene_count
X0	0.845	12.75
X1	0.847	68.75
X2	0.833	130.45
X3	0.852	290.90
X4	0.908	289.85
X5	0.834	123.50
X6	0.715	174.45
X7	0.821	213.75
X8	0.891	279.10
X9	0.848	129.15
X10	0.866	196.10
X11	0.925	11.90
X12	0.809	5.65
X13	0.937	172.55
X14	0.519	326.00
X15	0.944	86.35
X16	0.835	243.00
X17	0.894	9.15
X18	0.924	856.95
X19	0.940	235.95
X20	0.973	363.15
X21	0.760	148.25
X22	0.922	211.80
X23	0.881	33.20
X24	0.864	159.05

```r
# plot number of informative metagenes as a function of K
# (the diagonal is just there to show what we would see if
# each K increase yielded exclusively informative metagenes)
gg <- ggplot(data = numMetaKeep, aes(x = kval, y = MK)) + geom_line(colour = "blue")
gg <- gg + geom_abline(slope = 1)
gg <- gg + theme_gray()
gg <- gg + scale_x_continuous(limits = c(10, 100))
gg <- gg + scale_y_continuous(limits = c(10, 100))
gg <- gg + labs(x = "K value used", y = "Number of Informative Metagenes")
gg
```

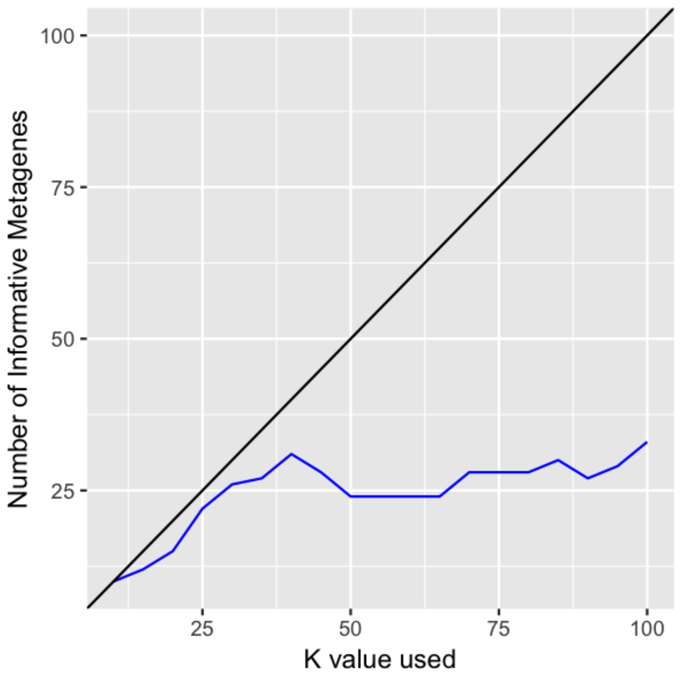

Figure 1: Broad K NMF

The plot above shows the number of informative metagenes as a function of K. From this plot we can determine that the increase in informative metagenes ceases around K=40.

Final K Selection

A second round of NMF was performed using K values from 35 to 45 by steps of 1 to identify the K within that range with the highest number of informative metagenes.

```
# import robj from python script
load("nmf/en_K40/Endo_Narrow_NMF.Robj")

# rename object
res <- result_obj

rm(result_obj)


# get list of k values that were used
getKs <- names(res)

getKs <- strsplit(getKs, "=")

getKs <- as.numeric(vapply(getKs, function(x) x[2], ""))

# because running findConsistMeta can be time-consuming, we
# can use a parallel for loop
library(doParallel)
cl <- makeCluster(6)
```

```r
registerDoParallel(cl)

# run findConsistMeta across all K values
allKconsistMeta <- foreach(i = 1:length(res)) %dopar% findConsistMeta(i)

# Exclude those metagenes that don't meet QC thresholds
allKconsistMeta.IDs <- lapply(allKconsistMeta, function(x) as.character(x[,
    1][x[, 2] > 0.6 & x[, 3] > 10]))

# give each DF it's correct name
names(allKconsistMeta.IDs) <- getKs

# get number of informative metagenes for each K value
numMetaKeep <- vapply(allKconsistMeta.IDs, length, numeric(1))

numMetaKeep <- data.frame(kval = getKs, MK = numMetaKeep)
```

```r
# plot number of informative metagenes as a function of K
# (the diagonal is just there to show what we would see if
# each K increase yielded exclusively informative metagenes)
gg <- ggplot(data = numMetaKeep, aes(x = kval, y = MK)) + geom_line(colour = "blue")
gg <- gg + geom_abline(slope = 1)
gg <- gg + theme_gray()
gg <- gg + scale_x_continuous(limits = c(35, 45))
gg <- gg + scale_y_continuous(limits = c(20, 50))
gg <- gg + labs(x = "K value used", y = "Number of Informative Metagenes")
gg
```

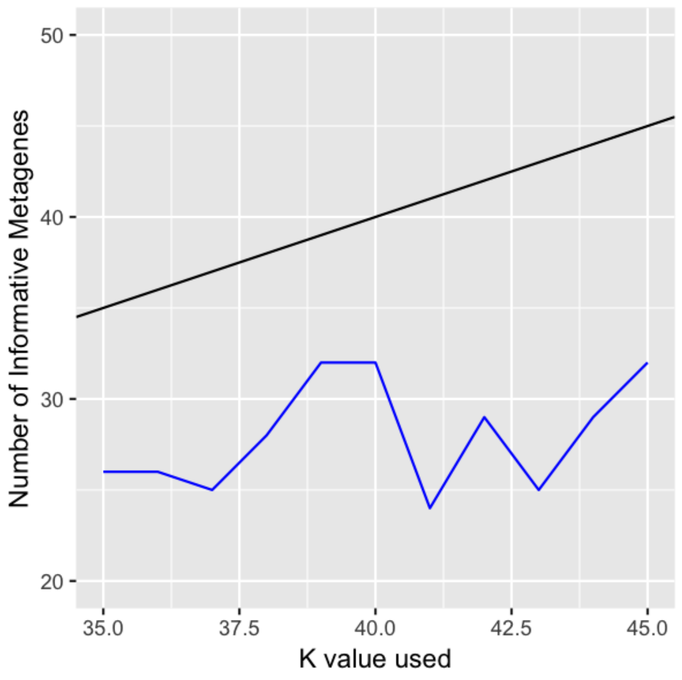

Figure 2: Narrow K NMF

From the above plot, we see that 39,40, and 45 all have the maximum number of informative metagenes and are thus equivalent. We used K=40 for our final analysis.

Post-NMF Analysis

After the optimal K has been selected, we subset the gene and cell scores for the desired K value to exclude uninformative metagenes from downstream analysis (although we save the uninformative metagene results separately in case they need to be consulted later for any reason).

```r
# specify the K value we want to use
x <- "K=40"

# isolate the results for desired K value
kval <- res[[x]]

# pull the results for the first replicate
runRep <- kval[[1]]

# get the cell scores
cellScores <- runRep[["C"]]

# get the list of good metagene indexes for that kvalue
metaIndex <- allKconsistMeta.IDs[[gsub("K=", "", x)]]

# because the gene number per metagene is an average of all
# replicates, there may be some metagenes in this particular
# replicate that are on the 'good' list, even if the gene in
# this replicate does not pass our threshold This just
# catches those exceptions and excludes them

# get gene scores
geneScores <- runRep[["G"]]

# find metagenes that don't pass the gene number threshold
extraFilter <- apply(geneScores, 2, function(x) x[x > (max(x)/10)])

list.condition <- sapply(extraFilter, function(x) length(x) >
    10)
extraFilter <- names(extraFilter[list.condition])

# exclude metagenes from this replicate that don't pass the
# threshold
metaIndex <- metaIndex[metaIndex %in% extraFilter]

# convert to an index (instead of a column name)
metaIndex <- as.numeric(gsub("X", "", metaIndex))

metaIndex <- metaIndex + 1

# get the index of 'bad' metagenes in case they need to be
# review them later
badMetaIndex <- 1:length(rownames(cellScores))
badMetaIndex <- badMetaIndex[!(badMetaIndex %in% metaIndex)]

# exclude bad metagenes
goodCellScores <- cellScores[metaIndex, ]

goodGeneScores <- geneScores[, metaIndex]

# exclude good metagenes
badCellScores <- cellScores[badMetaIndex, ]

badGeneScores <- geneScores[, badMetaIndex]
```

Plotting Metagenes

One of the easiest and often most informative ways of evaluating metagenes is to visualize the cell scores on a tSNE plot of the single cell data. This can often be used to infer the underlying biological process(es) behind each metagene. We did this by simply adding cell scores as new metadata columns to the Seurat object and plotting them using FeaturePlot.

```r
# add cell scores to metadata in the seurat object

ds.ds <- readRDS("objects/Hydra_Seurat_Endo_lineage_plot.rds")

# invert cellscores
t.goodCellScores <- as.data.frame(t(goodCellScores))

# fix the cellID formating
rownames(t.goodCellScores) <- substring(rownames(t.goodCellScores),
    2)

rownames(t.goodCellScores) <- gsub("[.]", "-", rownames(t.goodCellScores))

# add cell scores as metagene columns to seurat object
meta <- ds.ds@meta.data

meta$ID <- rownames(meta)

t.goodCellScores$ID <- rownames(t.goodCellScores)

meta <- merge(meta, t.goodCellScores, by = "ID", all.x = T)

rownames(meta) <- meta$ID

meta$ID <- NULL

ds.ds@meta.data <- meta

rm(meta)

t.goodCellScores$ID <- NULL

# spot check first nine metagenes
FeaturePlot(ds.ds, colnames(t.goodCellScores)[1:9], cols.use = c("grey",
    "blue"))
```

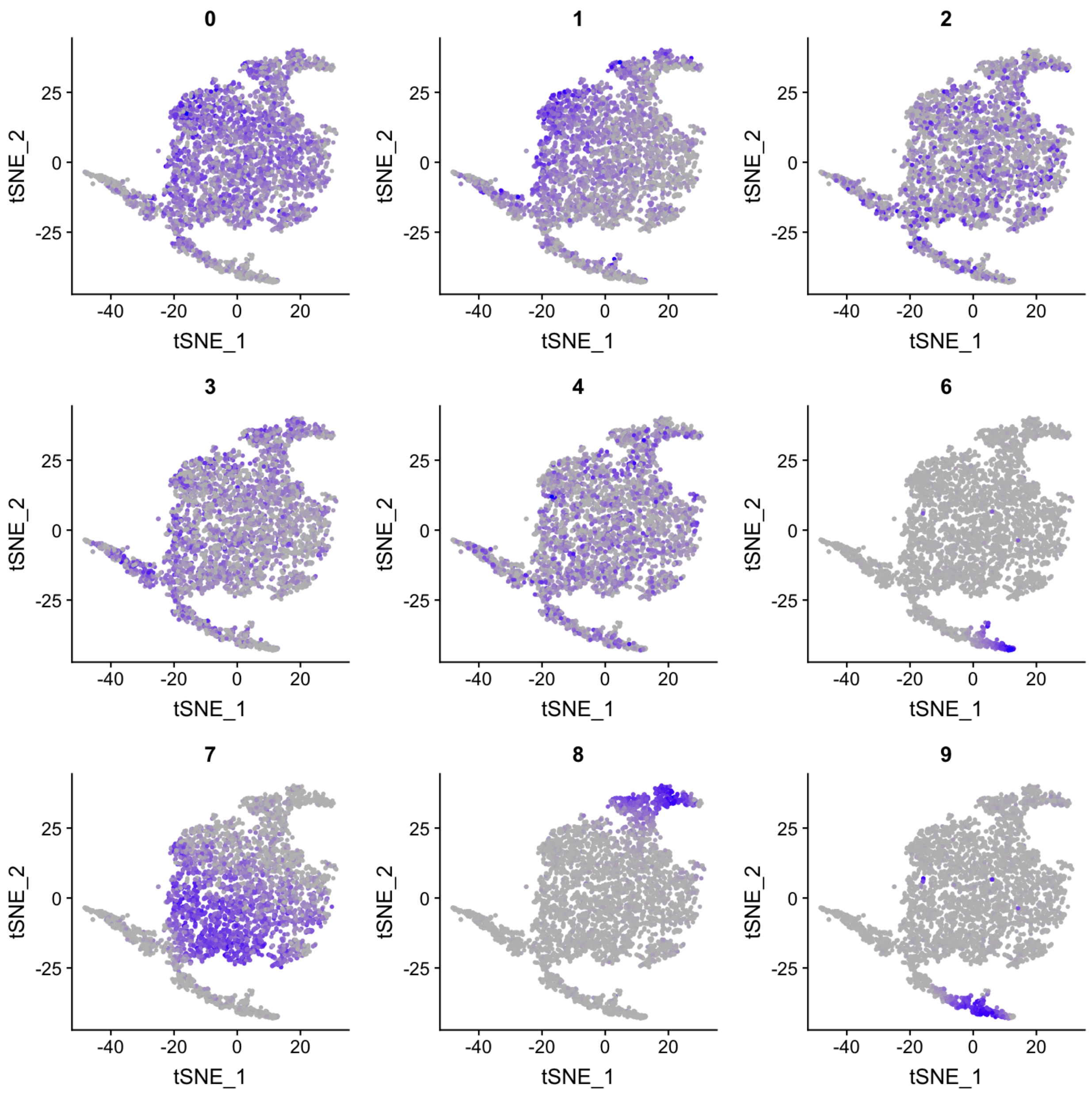

Figure 3: The first nine metagenes

Exporting Results

After the analysis has been finalized we can save the results. We save the gene and cell scores after separating them into good and bad metagenes. We also export the tSNE plots of the cell scores along with IDs and plots for the top 30 genes for each metagene.

```
# tables of cell and gene scores
write.csv(goodCellScores, file = "GoodMeta_CellScores.csv")
```

```r
write.csv(goodGeneScores, file = "GoodMeta_GeneScores.csv")

write.csv(badCellScores, file = "BadMeta_CellScores.csv")

write.csv(badGeneScores, file = "BadMeta_GeneScores.csv")

# read in the top 30 gene tables generated by the python
# script
top30 <- runRep$top30genes

# partition into good and bad metagenes and write results
goodTop30 <- rownames(cellScores)[metaIndex]

goodTop30 <- c(paste0("Module.", goodTop30), paste0("Weights.",
    goodTop30))

badTop30 <- top30[, !(colnames(top30) %in% goodTop30)]

goodTop30 <- top30[, colnames(top30) %in% goodTop30]

write.csv(goodTop30, file = "GoodMeta_Top30.csv")

write.csv(badTop30, file = "BadMeta_Top30.csv")

# plot the top 30 genes for each metagene

# this is a plotting function to generate tSNE plots using a
# nested directory structure
printGenetSNE <- function(w, y, x, z) {
    pdf(paste0(z, "/", w, "_", x, ".pdf", sep = ""), width = 10,
        height = 10)
    FeaturePlot(y, x, cols.use = c("grey", "blue"), pt.size = 2)
    dev.off()
}

# get good metagene gene lists
goodTop30.names <- goodTop30[, grep("Module", colnames(goodTop30))]

# create directory in which to place metagene subdirectories
dir.create("GoodMetagene_Plots", showWarnings = F)

# move through metagenes and create plots
for (i in 1:length(colnames(goodTop30.names))) {

    # create directory to place metagene genes
    directory <- paste0("GoodMetagene_Plots/", gsub("Module[.]",
        "", colnames(goodTop30.names)[i]))
    dir.create(directory, showWarnings = F)

    for (j in 1:30) {
        printGenetSNE(j, ds.ds, goodTop30.names[j, i], directory)
    }
}
```

```r
# get bad metagene gene lists
badTop30.names <- badTop30[, grep("Module", colnames(badTop30))]

# create directory in which to place metagene subdirectories
dir.create("badMetagene_Plots", showWarnings = F)

# move through metagenes and create plots
for (i in 1:length(colnames(badTop30.names))) {

    # create directory to place metagene genes
    directory <- paste0("badMetagene_Plots/", gsub("Module[.]",
        "", colnames(badTop30.names)[i]))
    dir.create(directory, showWarnings = F)

    for (j in 1:30) {
        printGenetSNE(j, ds.ds, badTop30.names[j, i], directory)
    }
}

printGenetSNE <- function(x, y, z) {
    pdf(paste0(z, x, ".pdf", sep = ""), width = 10, height = 10)
    print(FeaturePlot(y, x, cols.use = c("grey", "blue"), pt.size = 2))
    dev.off()
}

# export good and bad metagene cell score tSNEs
directory <- paste0("GoodMetagene_Plots/")
lapply(colnames(t.goodCellScores), function(x) printGenetSNE(x,
    ds.ds, directory))

# we need to reload the seurat object to add the bad
# metagenes
ds.ds <- readRDS("objects/ds.en.s1.orig.ident.pc10_rm19_rm36g_cl10.rds")

# invert cellscores
t.badCellScores <- as.data.frame(t(badCellScores))

# fix the cellID formating
rownames(t.badCellScores) <- substring(rownames(t.badCellScores),
    2)

rownames(t.badCellScores) <- gsub("[.]", "-", rownames(t.badCellScores))

# add cell scores as metagene columns to seurat object

meta <- ds.ds@meta.data

meta$ID <- rownames(meta)

t.badCellScores$ID <- rownames(t.badCellScores)

meta <- merge(meta, t.badCellScores, by = "ID", all.x = T)
```

```
rownames(meta) <- meta$ID

meta$ID <- NULL

ds.ds@meta.data <- meta

rm(meta)

t.badCellScores$ID <- NULL

directory <- paste0("badMetagene_Plots/")
lapply(colnames(t.badCellScores), function(x) printGenetSNE(x,
    ds.ds, directory))
```

Software versions

This document was computed on Mon Jul 08 12:43:54 2019 with the following R package versions.

```
R version 3.6.0 (2019-04-26)
Platform: x86_64-apple-darwin15.6.0 (64-bit)
Running under: macOS High Sierra 10.13.4

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/3.6/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/3.6/Resources/lib/libRlapack.dylib

Random number generation:
 RNG:     Mersenne-Twister
 Normal:  Inversion
 Sample:  Rounding

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] kableExtra_1.1.0 xtable_1.8-4     knitr_1.23       magick_2.0
[5] Seurat_2.3.3     Matrix_1.2-17    cowplot_0.9.4    ggplot2_3.1.1

loaded via a namespace (and not attached):
  [1] diffusionMap_1.1-0.1 Rtsne_0.15           colorspace_1.4-1
  [4] class_7.3-15         modeltools_0.2-22    ggridges_0.5.1
  [7] mclust_5.4.3         htmlTable_1.13.1     base64enc_0.1-3
 [10] rstudioapi_0.10      proxy_0.4-23         npsurv_0.4-0
 [13] flexmix_2.3-15       bit64_0.9-7          mvtnorm_1.0-10
 [16] xml2_1.2.0           codetools_0.2-16     splines_3.6.0
 [19] R.methodsS3_1.7.1    lsei_1.2-0           robustbase_0.93-5
 [22] jsonlite_1.6         Formula_1.2-3        ica_1.0-2
 [25] cluster_2.0.9        kernlab_0.9-27       png_0.1-7
 [28] R.oo_1.22.0          httr_1.4.0           readr_1.3.1
 [31] compiler_3.6.0       backports_1.1.4      assertthat_0.2.1
 [34] lazyeval_0.2.2       formatR_1.6          lars_1.2
 [37] acepack_1.4.1        htmltools_0.3.6      tools_3.6.0
 [40] igraph_1.2.4.1       gtable_0.3.0         glue_1.3.1
 [43] reshape2_1.4.3       RANN_2.6.1           dplyr_0.8.1
 [46] Rcpp_1.0.1           gdata_2.18.0         ape_5.3
 [49] nlme_3.1-140         iterators_1.0.10     fpc_2.2-1
 [52] gbRd_0.4-11          lmtest_0.9-37        xfun_0.7
 [55] stringr_1.4.0        rvest_0.3.4          irlba_2.3.3
 [58] gtools_3.8.1         DEoptimR_1.0-8       MASS_7.3-51.4
 [61] zoo_1.8-6            scales_1.0.0         hms_0.4.2
 [64] doSNOW_1.0.16        parallel_3.6.0       RColorBrewer_1.1-2
 [67] yaml_2.2.0           reticulate_1.12      pbapply_1.4-0
 [70] gridExtra_2.3        rpart_4.1-15         segmented_0.5-4.0
 [73] latticeExtra_0.6-28  stringi_1.4.3        foreach_1.4.4
 [76] checkmate_1.9.3      caTools_1.17.1.2     bibtex_0.4.2
 [79] Rdpack_0.11-0        SDMTools_1.1-221.1   rlang_0.3.4
 [82] pkgconfig_2.0.2      dtw_1.20-1           prabclus_2.3-1
 [85] bitops_1.0-6         evaluate_0.14        lattice_0.20-38
 [88] ROCR_1.0-7           purrr_0.3.2          labeling_0.3
 [91] htmlwidgets_1.3      bit_1.1-14           tidyselect_0.2.5
 [94] plyr_1.8.4           magrittr_1.5         R6_2.4.0
 [97] snow_0.4-3           gplots_3.0.1.1       Hmisc_4.2-0
[100] pillar_1.4.1         foreign_0.8-71       withr_2.1.2
[103] fitdistrplus_1.0-14  mixtools_1.1.0       survival_2.44-1.1
[106] scatterplot3d_0.3-41 nnet_7.3-12          tsne_0.1-3
[109] tibble_2.1.2         crayon_1.3.4         hdf5r_1.2.0
[112] KernSmooth_2.23-15   rmarkdown_1.13       grid_3.6.0
[115] data.table_1.12.2    webshot_0.5.1        metap_1.1
[118] digest_0.6.19        diptest_0.75-7       tidyr_0.8.3
[121] R.utils_2.8.0        stats4_3.6.0         munsell_0.5.0
[124] viridisLite_0.3.0
```