Preprocessing-and-Filtering

2024-06-01

Overview

SLIDE uses the feature-feature correlation structure in the data to create latent factors. Input data does not need to be autoscaled (meaning each feature has `mean = 0` and `std dev = 1`), as this is done when calculating the latent factors. However, there are a few important considerations:

- **The goal is to get down to approximately $\leq$ `5000` features for SLIDE to run smoothly**
- **Be sure to remove features with `std deviation = 0` before starting - these will cause err e(uses)-**

The sections below outline

Load Sample data
emove features with `std dev = 0`
3. Handling missing values

```
unfiltered_x = as.matrix(read.csv("SLIDE/Data_Scripts/CD4_Expansion/x.csv",
    row.names = 1
```

Remove features with `standard dev = 0`

First things first: remove all features that have a `std dev = 0` because these are uninformative and will cause errors in calculating the correlation matrix in SLIDE.

```
# find columns with sd = 0
zero_sd_features = which(apply(unfiltered_x, 2, sd) == 0)

filtered_x = unfiltered_x[, -zero_sd_features]
```

Handling Missing Values

SLIDE requires that the input sample-feature data be a numeric matrix and contain no characters or `NA/NAN/Inf` values.

Data repair - Find `NA/NAN/Inf` values

```
# this will give you a vector with positions in the flattened matrix
bad_values = which(is.na(unfiltered_x) | is.nan(unfiltered_x) | is.infinite(unfiltered_x))

# you can replace the bad values with zeros or see below for
# imputation
repaired_x = unfiltered_x[bad_values] = 0
```

Imputation

If you wish to impute values (for example, by using the mean), you can do so similarly

```
# same as above
bad_values = which(is.na(unfiltered_x) | is.nan(unfiltered_x) | is.infinite(unfiltered_x))

# go through each column and replace bad values with the mean of that
# column
imputed_x = apply(unfiltered_x, MARGIN = 2, function(x) x[which(is.na(x) |
    is.nan(x) | is.infinite(x))] = mean(x, na.rm = TRUE))
```

Filtering

- **Sparsity**: both in features and samples will create problems. Consider removing features, then samples based on sparsity (generally, any sample filtering should come after feature filtering).
  - A good initial check is to see how many features have `median = 0` (50% of feature are zeros) - if there are a significant number of these features, instead of removing all of them consider removing features that have more than 50% zeros (e.g. remove features that are 90% zeros or 75% zeros). Afterwards, do the same for samples (although we are generally more conservative and only remove samples that have a significant proportion of zeros).
- **Variance**: Low variance features are less informative, so Consider filtering these out.

  - Remove features with `std dev = 0`, because these are uninformative

- **Coefficient of Variation**: Ratio of feature `std dev / mean`. Features with high coe□cient of variation may be noise and features with low coe□cient of variation may be uninformative.

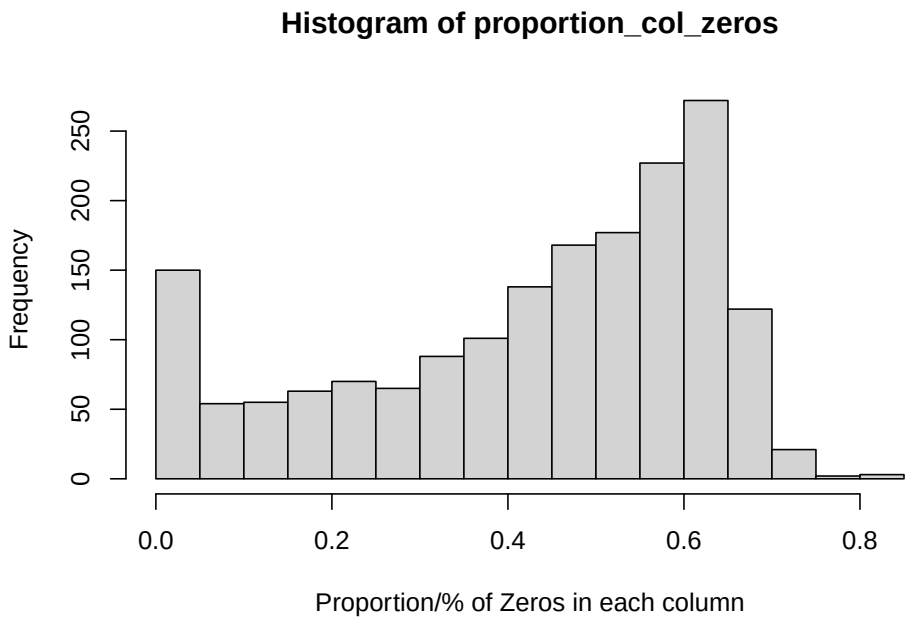
Sparsity Filtering

Filtering features (columns)

You can measure the proportion of zeros in each row/column. Start by filtering out features, then filter samples

```
proportion_col_zeros = apply(unfiltered_x, MARGIN = 2, function(x) length(which(x ==
    0))/length(x))

hist(proportion_col_zeros, xlab = "Proportion/% of Zeros in each column")
```



We can now remove any column that has more than `0.9` (90%) zeros.

```
filtered_x = unfiltered_x[, -which(proportion_col_zeros > 0.9)]
```

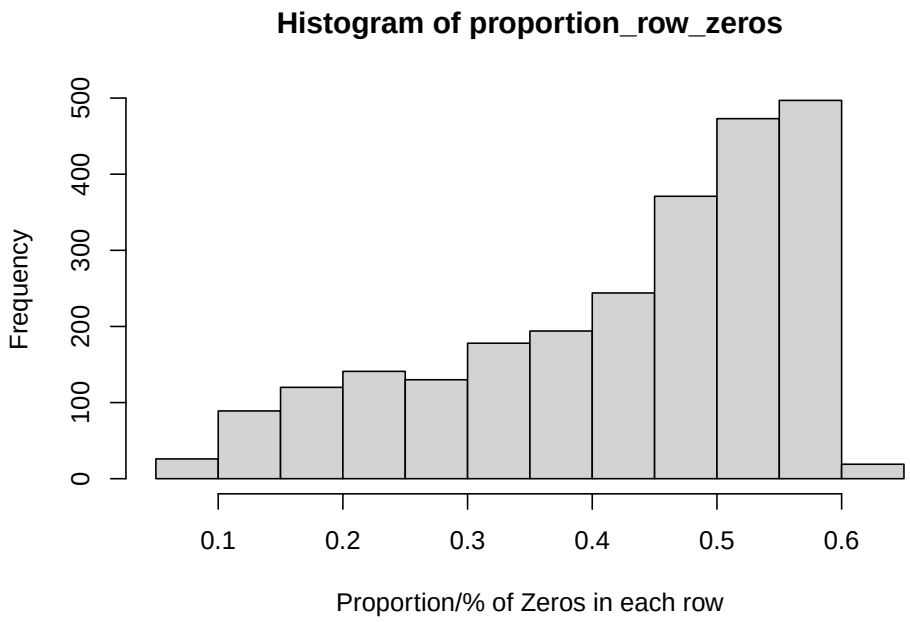
Note: we can use these vectors with the quantile function to set a threshold for percentage of zeros - note: you can also use the quantile function.

Filtering samples (rows)

We can do the same for rows (**remember to remove the corresponding rows from your response vector as well!**)

```r
# Get indices for rows that are the top 10% sparse
proportion_row_zeros = apply(unfiltered_x, MARGIN = 1, function(x) length(which(x ==
    0))/length(x))

hist(proportion_row_zeros, xlab = "Proportion/% of Zeros in each row")
```



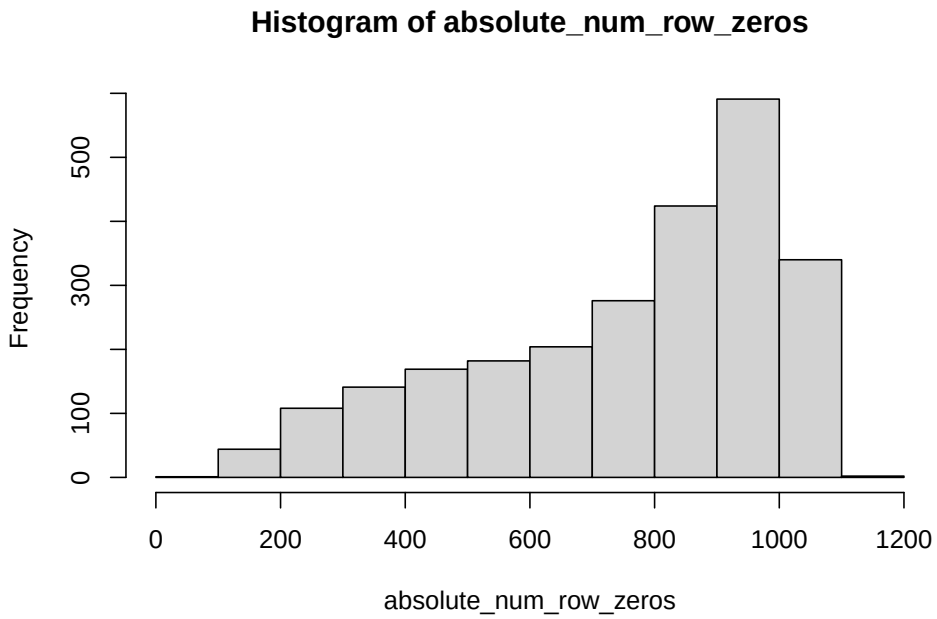

```r
rows_above_zero_threshold = which(proportion_row_zeros > 0.9)

filtered_x = unfiltered_x[-rows_above_zero_threshold, ]
filtered_y = unfiltered_y[-rows_above_zero_threshold]
```

Note: you may want to filter out zeros based on absolute number (versus just filtering out by quantile); in this case, your apply function will not normalize for length:

```r
absolute_num_col_zeros = apply(unfiltered_x, MARGIN = 2, function(x) length(which(x ==
    0)))
absolute_num_row_zeros = apply(unfiltered_x, MARGIN = 1, function(x) length(which(x ==
    0)))

# should look the same as hist for row % above
hist(absolute_num_row_zeros)
```



Alternatively, you can use the `zeroFiltering` function in the SLIDE package: - Features with more than `col_thresh` zeros will be filtered out - Samples with more than `row_thresh`‘ zeros will be filtered out

```
# this will remove all features and samples that have more than 50%
# zeros (any feature or column with median = 0)
col_thresh = nrow(unfiltered_x)/2
row_thresh = ncol(unfiltered_x)/2

# columns with more than col_thresh number of zeros will be filtered
# rows with more than row_thresh number of zeros will be filtered
filtered_mats = SLIDE::zeroFiltering(unfiltered_x, unfiltered_y, col_thresh = nrow(unfiltered_x)/2,
    row_thresh = ncol(unfiltered_x)/2)
#> Original dataframe dimension is  2482  by  1776
#> Filtered dataframe dimension is  1493  by  952
filtered_x = filtered_mats$filtered_x
filtered_y = filtered_mats$filtered_y
```

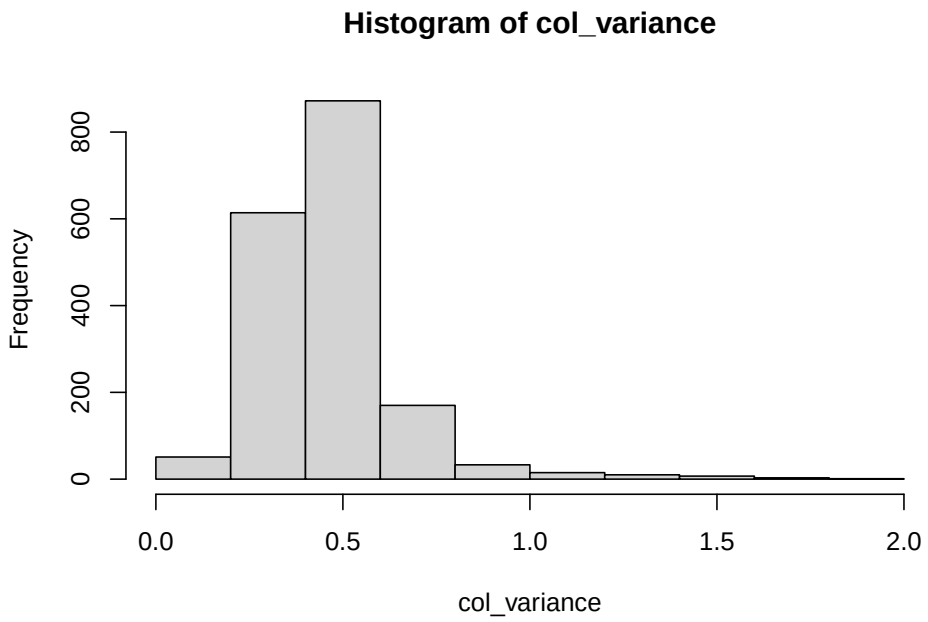
Variance Filtering

We use similar functions to filter by variance; generally speaking, you will want to look at the variance of mean-centered data (although your input data does not need to be centered or scaled, as that is done in SLIDE)

```
# mean center each column
scaled_x = apply(unfiltered_x, MARGIN = 2, function(x) scale(x, center = TRUE,
    scale = FALSE))

# get the variance of each column
col_variance = apply(scaled_x, MARGIN = 2, var)
```

```
hist(col_variance)
```



Next, using this histogram, we want to filter out features with low variance. It looks like `0.20` is a good choice to remove the lowest variance features

```
# we can filter out features with very low variance (typically we use
# somewhere between the 25th-45th percentiles). We use 25th
# percentile below
low_var_cols = which(col_variance < 0.2)

filtered_x = unfiltered_x[, -low_var_cols]
```

Alternatively, we filter out the bottom 25th percentile by variance. We can also use the `quantile` function to control the number of features we filter, and so that we don't need to explicitly pick a variance threshold (e.g. filtering out the 25th percentile removes the lowest 25% of features by variance)

Note, you can repeat this same process for rows by changing the `MARGIN` argument in the apply function - e.g. for columns, `MARGIN = 2` `apply(unfiltered_x, MARGIN = 2, var)` and for rows, `MARGIN = 1` `apply(unfiltered_x, MARGIN = 1, var)`

```
# we can filter out features with very low variance (typically we use
# somewhere between the 25th-45th percentiles). We use 25th
# percentile below
low_var_cols = which(col_variance < quantile(col_variance, 0.25))

filtered_x = unfiltered_x[, -low_var_cols]
```