A quick tour of SOUP

Lingxue Zhu

2018-03-19

This vignette walks through an example of fetal brain data and illustrates the basic usage of `SOUP`.

Installation

`SOUP` can be installed directly from GitHub with `devtools`:

```
library(devtools)
devtools::install_github("lingxuez/SOUP")
```

Now we can load `SOUP`. We also load the `ggplot2` package for visualization purpose.

```
library(SOUP)
library(ggplot2)
```

Fetal Brain Data

First, we illustrate the usage of SOUP on a fetal brain single-cell dataset (*Camp et al. (2015)*). The Camp data is distributed together with the SOUP package, with 220 cells and 12,694 genes, which can be directly loaded:

```
cell.info = camp$cell.info
head(cell.info)
#>              cell.id cell.type
#> 1 A5_fetal_12wpc_c1        N3
#> 2 D7_fetal_12wpc_c1       BP1
#> 3 E5_fetal_12wpc_c1        N3
#> 4 G6_fetal_12wpc_c1        N2
#> 5 H2_fetal_12wpc_c1       BP2
#> 6 B7_fetal_12wpc_c2       AP2

counts = camp$counts
dim(counts)
#> [1]   220 12694
```

To see more details of Camp data:

```
help(camp)
```

1. Data pre-processing

In practice, we find it always beneficial to pre-process single-cell RNA-seq dataset, including:

1. Normalize such that each cell has total sum of $1e6$, which is equivalent to a TPM normalization.
2. Log transformation.

The following line performs these steps:

```
log.expr = log2(scaleRowSums(counts)*(10^6) + 1)
dim(log.expr)
#> [1]   220 12694
```

2. Gene selection

We propose a selection procedure to identify the set of important genes for clustering. This procedure includes two parts:

1. DESCEND algorithm (*Wang et al. (2017)*), which works on the count data directly using a Poisson model;
2. SPCA algorithm (*Witten et al. (2009)*), which works better on the log-transformed data.

Because these two algorithms work in different spaces, we ask users to provide the raw counts when possible, and the function `selectGenes` will use the correct scale for both algorithms. Note that this procedure can be slow. One can parallelize the DESCEND algorithm by specifying the number of cores in `n.cores`. For the purpose of this tutorial, please feel free to skip this step, and directly use the pre-computed results we provide.

```
#> NOT RUN
select.out = selectGenes(counts, type="count", n.cores=10)
select.genes = select.out$select.genes
#> NOT RUN
```

We provide the selection results of 430 genes in `camp$select.genes`, which can be directly loaded:

```
select.genes = camp$select.genes
log.select.expr = log.expr[, colnames(log.expr) %in% select.genes]
dim(log.select.expr)
#> [1] 220 430
```

3. SOUP clustering

SOUP clustering is then applied to the set of selected genes. It is highly recommended to use the log-scaled data. We can run `SOUP` over a sequence of K's, the number of clusters. Here we present the results over $K \in \{2, ..., 5\}$:

```
system.time({
  soup.out = SOUP(log.select.expr, Ks=c(2:5), type="log")
})
#>    user  system elapsed
#>   1.110   0.079   1.259
```

The output of function `SOUP` includes a sequence of membership matrices,as well as a sequence of cluster centers, one per given K:

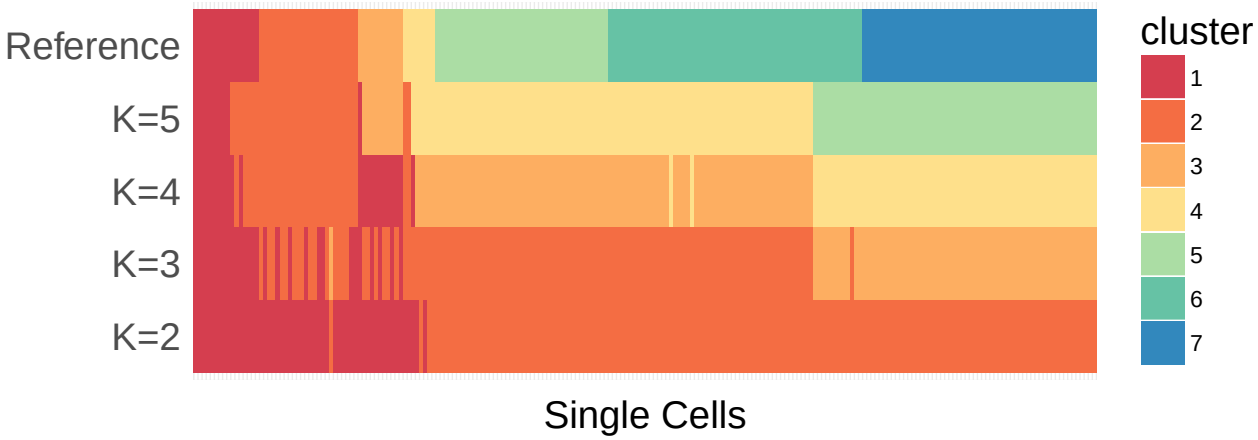
```
length(soup.out$memberships)
#> [1] 4
length(soup.out$centers)
#> [1] 4
```

Hard clustering. SOUP can be treated as hard clustering, by assigning each cell to its major type with the largest membership. The hard assignments are stored in `soup.out$major.labels`, and we present the results of $K = 5$ below as an example:

```
table(soup.out$major.labels[[4]])
#>
#>  1  2  3  4  5
#> 33 10 69 10 98
```

The function `heatmapKseq` provides a visualization of the hard clustering results of different K's, compared to the reference cell type.

```
g.kseq = heatmapKseq(memberships=soup.out$memberships,
                     Ks=soup.out$Ks,
                     cell.type=cell.info$cell.type)
g.kseq
```



Soft membership and trajectory. On the other hand, using SOUP soft memberships, one can obtain an ordering of cells corresponding to their developmental trajectory. In particular, we estimate a timepoint for each cell by

1. Order the clusters so that they represent developmental order. In this dataset, we first identify the ending point by using the cluster with the largest group of `N3` cells, which is the group of most matured neurons. Then we consecutively find the next previous cluster that has the highest cluster center correlation.
2. The timepoint of cell i is estimated by $t_i = \sum_k k\hat{\theta}_{ik}$, where $\hat{\theta}_{i\cdot}$ is the estimated SOUP membership.

The function `getTimeline` implements these steps, and returns the vector $(t_1, ..., t_n)$. Here, we present the results with $K = 5$:

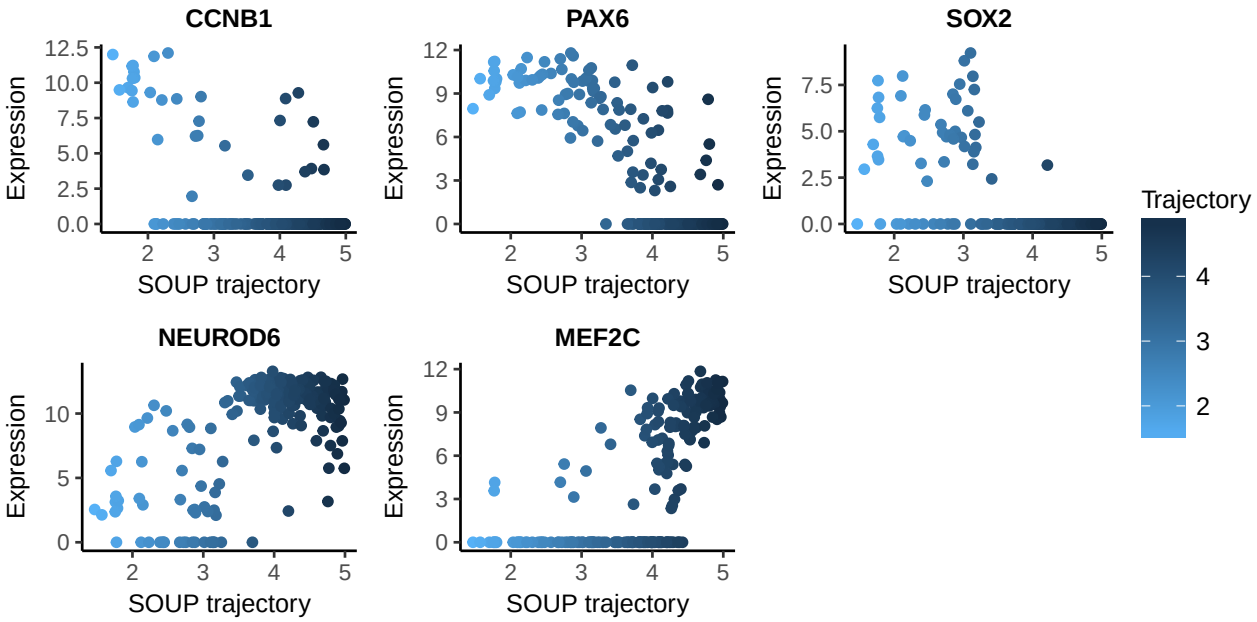
```
K.use = 5
i.K = match(K.use, soup.out$Ks)

#> pick the ending point: with the most N3 cells
soup.label = soup.out$major.labels[[i.K]]
k.end = which.max(table(soup.label, cell.info$cell.type)[, "N3"])

#> estimate timepoints
soup.timeline = getTimeline(membership = soup.out$memberships[[i.K]],
                            centers = soup.out$centers[[i.K]],
                            k.end=k.end)
```

To examine the estimated trajectory, we visualize the expression levels of 5 marker genes along the estimated SOUP trajectory, using function `plotMultipleGeneTimeline`:

```
genelist = c("CCNB1", "PAX6", "SOX2", "NEUROD6", "MEF2C")
g.timeline = plotMultipleGeneTimeline(expr=log.select.expr,
                                      genelist=genelist,
                                      timeline=soup.timeline,
                                      nrow=2, ncol=3)
g.timeline
```



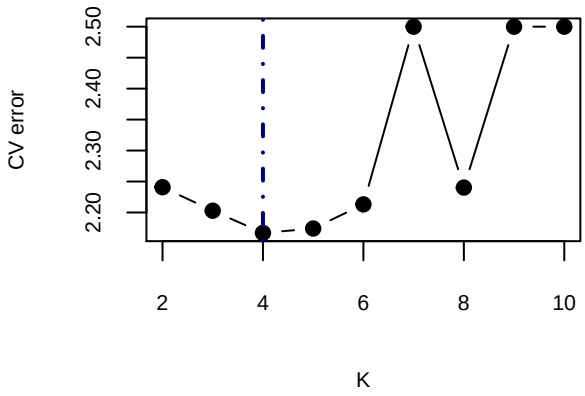
4. Cross Validation

Finally, we present a cross validation procedure to select the optimal K, number of clusters. In particular, we search over $K \in \{2, .., 10\}$ using 10-fold cross validation, and the errors are averaged over `nCV` repetitions. For illustration purpose, here we conduct only 2 repetitions. A parallelization option has been implemented, and the number of cores to be used can be specified via `mc.cores`. Note that the computation is parallelized across folds, so there is no need to use more than `nfold` of cores. For reproducibility, we set the seeds below:

```
Ks=c(2:10)
system.time({
  cv.soup.out = cvSOUP(log.select.expr, Ks=Ks, type="log",
                       nfold=10, nCV=2, mc.cores = 2,
                       seeds=c(42, 142), verbose=FALSE)
})
#>    user  system elapsed
#>  32.991   2.088  25.115
```

The optimal K is the one that achieves the lowest cross validation error, given by `cv.soup.out$K.cv`. We visualize the cross validation error, in the log scale, over different K's, and the optimal K is 4. For better visualization, values are capped at 2.5:

```
trunc.log.cvm = pmin(log(cv.soup.out$cvm), 2.5)
plot(Ks, trunc.log.cvm, type="b", pch=19,
     xlab="K", ylab="CV error", cex.lab=0.7, cex.axis=0.7)
abline(v=cv.soup.out$K.cv, lty=4, lwd=2, col="darkblue")
```



References

1. Camp *et al.* (2015) Human cerebral organoids recapitulate gene expression programs of fetal neocortex development. *PNAS.*
2. Wang J *et al.* (2017) Gene expression distribution deconvolution in single cell rna sequencing. *bioRxiv 227033.*
3. Witten DM, Tibshirani R, Hastie T (2009) A penalized matrix decomposition, with applications to sparse principal components and canonical correlation analysis. *Biostatistics.*