STstuff package vignette

David Venet

June 23, 2022

1 Introduction

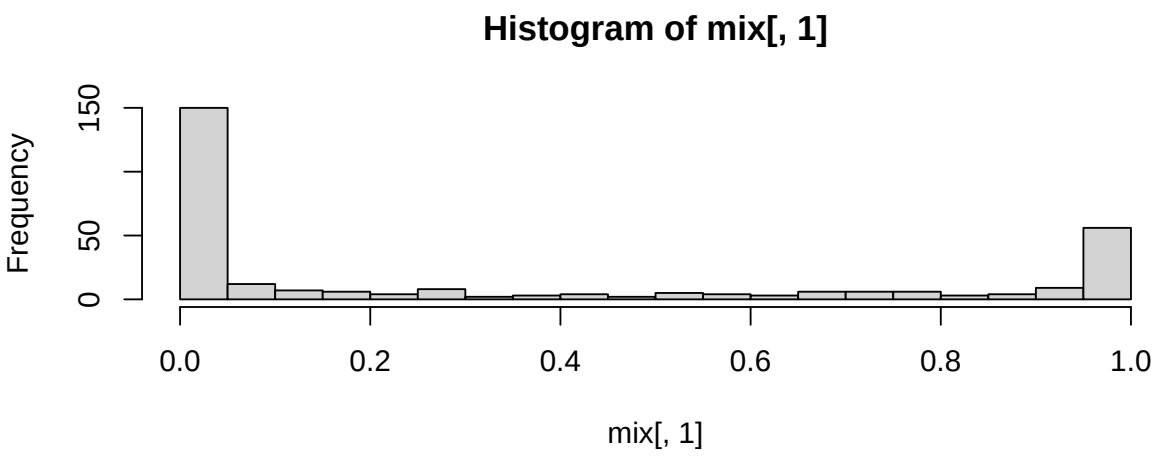
2 Simulation

We simulate data, where each spot is a mixture of 3 basic vectors. Those vectors are chosen to be correlated, as gene expression is. The parameters of the mixture are taken from a Dirichlet distribution, so that they are often close to 1 but not always. Then we take the mixture from each spot as the mean of a negative binomial and draw random values from there.

```
library(STstuff);

library(dirmult); # for the Dirichlet (in the simulation)

Nprot = 3; Nspot = 100*Nprot;
Ngene = 500;
gb = rexp(Ngene)^2+1; # Gene specific values
prot = matrix(rexp(Ngene*Nprot), ncol=Nprot) * gb; # prototypes
protl = log(1+1e4*t(prot)/colSums(prot))
mix = rdirichlet(Nspot, alpha=c(1,1,1)*.1) # Mixture from all 3 classes for each spot
base = mix %*% t(prot)
X = matrix(rnbinom(base, mu=base, size=1), ncol=ncol(base));
```

The distribution of the mix is often close to 0 or 1, as wanted.

```
hist(mix[,1], breaks=30)
```

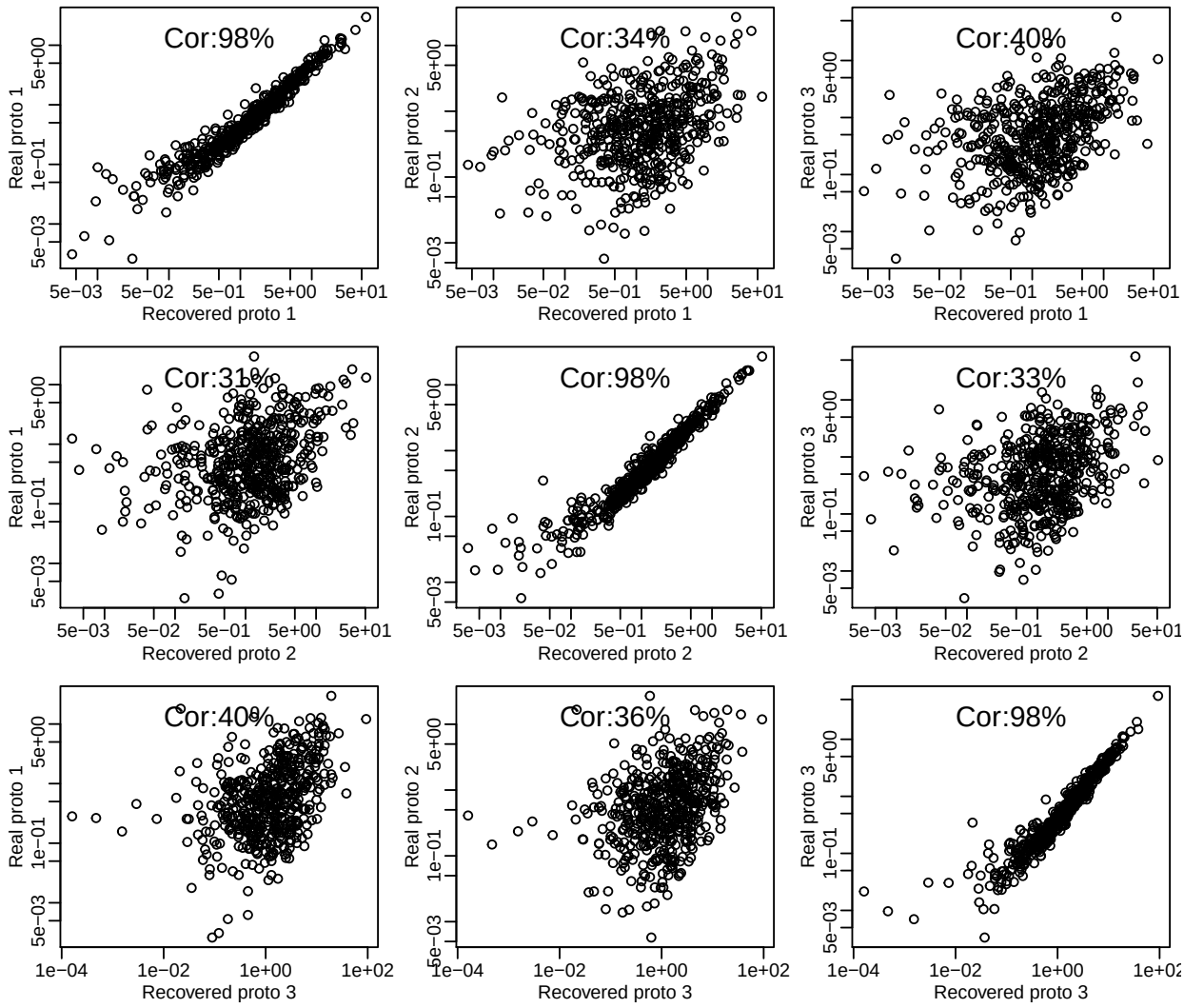


From here, we first do a kmeans to get the clusters. We start from the real prototype to ensure that we get the right ordering.

```
y = log(1+1e4*X/rowSums(X))
km = kmeans(y, prot1);
```

We can see that the cluster centers, although correlated to the correct prototypes, are also correlated to each other.

```
pr = t(km$centers)
pr0 = (exp(pr)-1)*colSums(prot)/1e4;

par(mfrow=c(3,3), mar=c(3,3,.5,.5), mgp=c(1.5,.5,0))
for (i in 1:3)
{ for (j in 1:3)
  { plot(prot[,i], pr0[,j], log='xy', xlab=paste("Recovered proto", i),
      ylab=paste("Real proto", j)); #abline(0,1)
    mtext(paste0("Cor:", round(cor(prot[,i], pr[,j], method='s')*100), "%"), side=3, line=-2)
  }
}

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

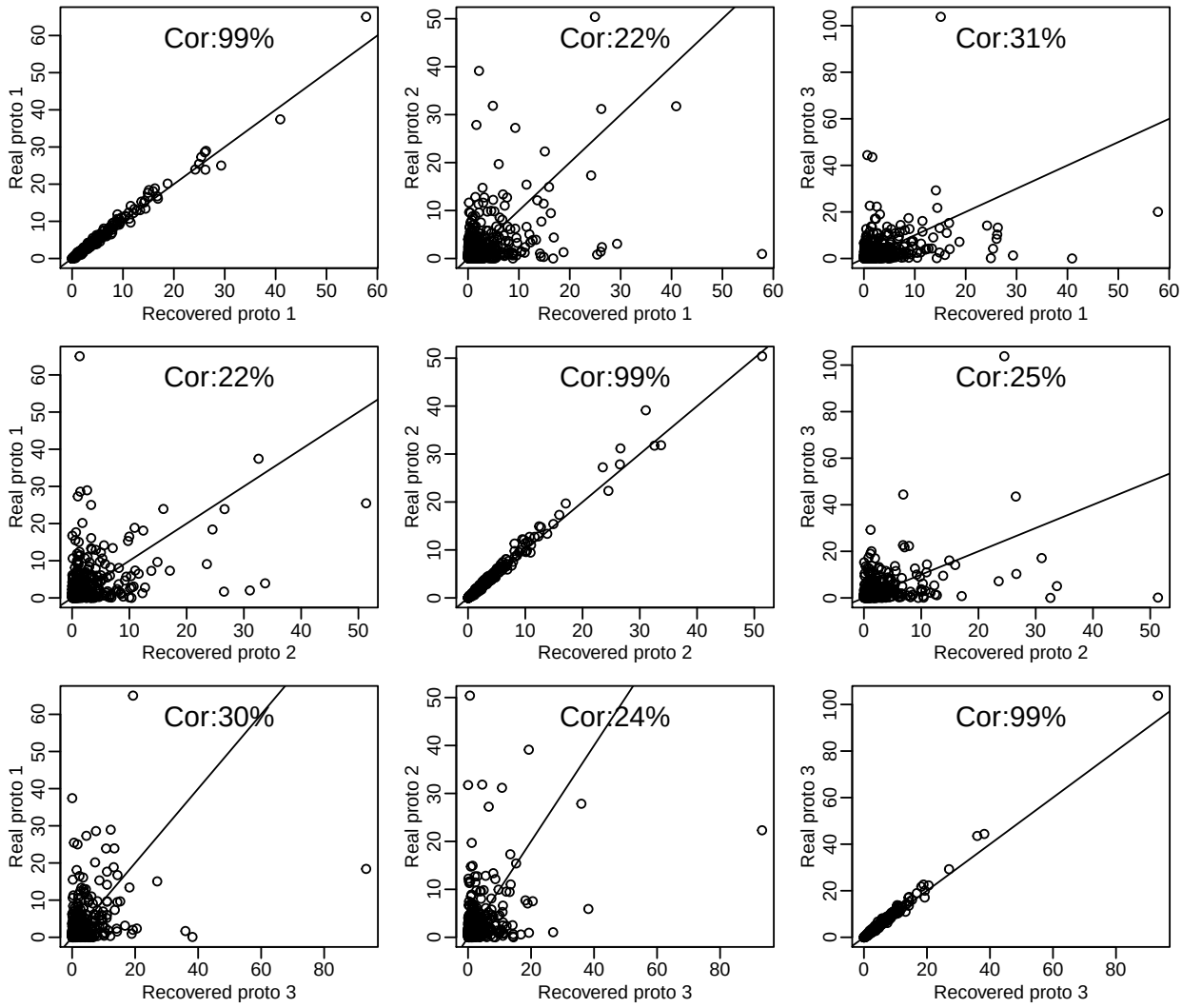


We can deconvolute the cluster.

```
deconv = NNMFfromKM(X, km$cluster, Niter=10, mccores=7)

##       Iter:1
##       Iter:2
##       Iter:3
##       Iter:4
##       Iter:5
##       Iter:6
##       Iter:7
##       Iter:8
##       Iter:9
##       Iter:10

pr2 = deconv$proto; # pr2 are the new prototypes
```

The new prototypes are still well correlated with the real prototypes, but they are also less correlated to each other. Note that some correlation is still expected because the original prototypes are also correlated.

```
par(mfrow=c(3,3), mar=c(3,3,.5,.5), mgp=c(1.5,.5,0))
for (i in 1:3)
{ for (j in 1:3)
  { plot(prot[,i], pr2[,j], xlab=paste("Recovered proto", i),
      ylab=paste("Real proto", j)); abline(0,1)
    mtext(paste0("Cor:", round(cor(prot[,i], pr2[,j], method='s')*100), "%"), side=3, line=-2)
  }
}
```



We do not typically know the number of factors in the mixture. The number of clusters will typically be larger than this. To simulate this, we use a kmeans with 4 groups.

```
km = kmeans(y, 4, nstart=100);
```

From this, we can use `deconvoluteClusters` function to get the best mixture. In this case the `rmCutOff` parameter was increased to ensure that the bad cluster is removed, but that's not recommended in practice.

```
dec2 = deconvoluteClusters(X, km$cluster, rmCutOff=.6, mccores=7, clean=FALSE)

##   Kill:
##      Iter:1
##      Iter:2
##      Iter:3
##      Iter:4
##      Iter:5
##      Iter:6
##      Iter:7
##      Iter:8
##      Iter:9
##      Iter:10
##   Full prototypes
```

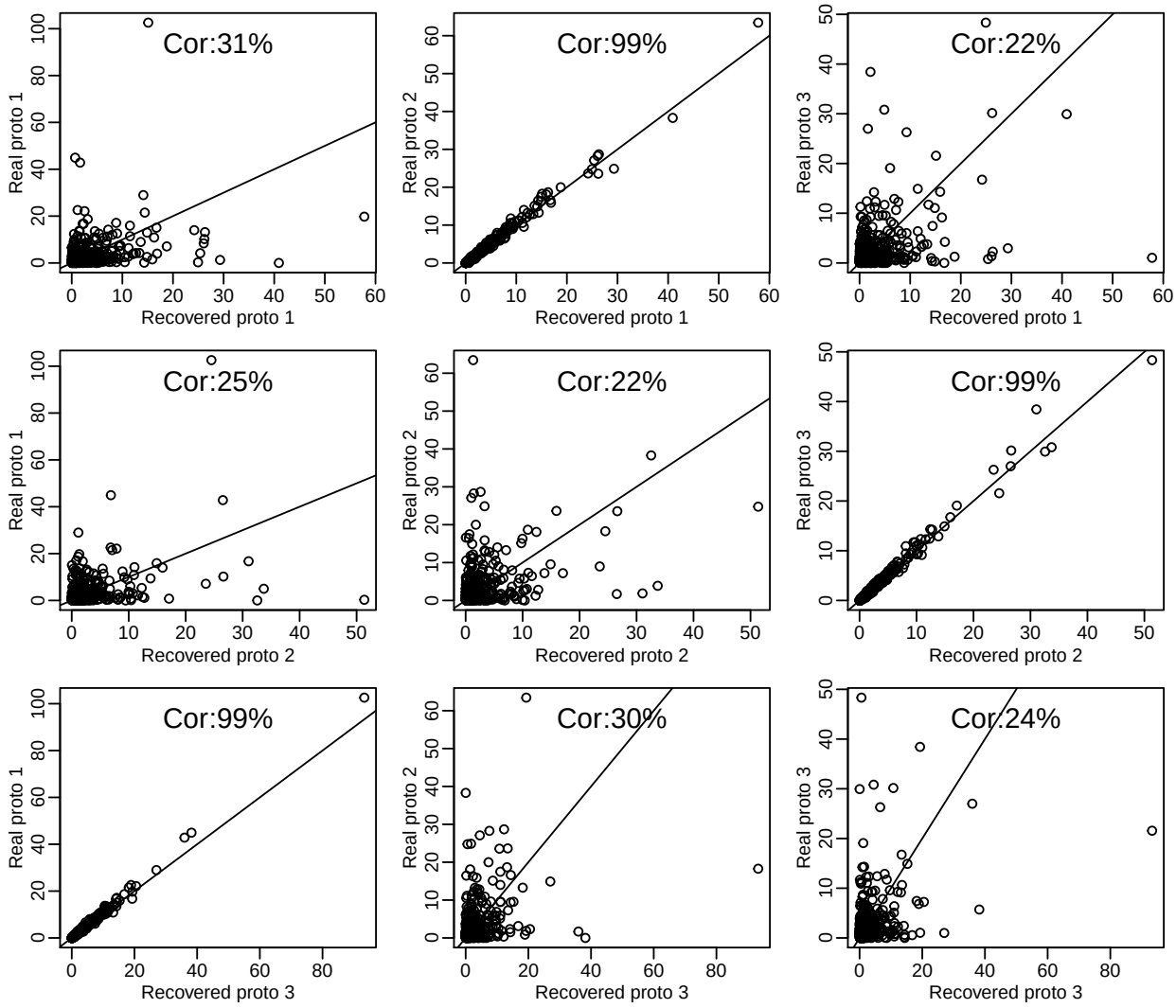
The `Kill` shown is list of cluters that are removed. Note the calculation is done on a subset of 500 genes that fit the clustering well, then the prototypes are calculated using `NNMFproto` on all genes.

The clusters recovered can be extracted, their name corresponding to the indices of the original kmeans. The prototypes are a list with two items, `proto` indicating the protoype values and `sigma` giving the extra variance from the negative binomial.

```
pr3 = dec2$proto$proto
show(colnames(pr3))

## [1] "1" "2" "3" "4"

show(dim(pr3))

## [1] 500   4
```

We can check that the prototypes still fit well. Of course the ordering is random.

```
par(mfrow=c(3,3), mar=c(3,3,.5,.5), mgp=c(1.5,.5,0))
for (i in 1:3)
{ for (j in 1:3)
  { plot(prot[,i], pr3[,j], xlab=paste("Recovered proto", i),
      ylab=paste("Real proto", j)); abline(0,1)
    mtext(paste0("Cor:", round(cor(prot[,i], pr3[,j], method='s')*100), "%"), side=3, line=-2)
  }
}
```



3 Recovery from another dataset

This is done via the `getClustFrom` function.

4 NNMF in other dataset

This is quite similar, except it uses NNMF instead of negative binomial models. This is done by using the `NNMFother` function.

4.1 Direct example

As an example, imagine we have some sort of prototypes (4 prototypes, 100 genes) and mixing matrix (50 samples):

```
prot = matrix(rgamma(4*100, 3, .01), ncol=4); rownames(prot) = 1:100;
mix = matrix(runif(4*50)^2, ncol=4); mix = mix/rowSums(mix)
```

The expected values are then just the product of the two

```
e = prot %*% t(mix)
```

The observed values would then be taken from a Poisson distribution with those means

```
obs = matrix(rpois(length(e), e), ncol=ncol(e))
```

From `obs` and `prot`, we can easily recover the original mixture (correlations are very close to 1):

```
W = t(nnlm(prot, obs, loss='mkl')$coefficients);
show(diag(cor(W, mix)))

## [1] 0.9987497 0.9986669 0.9991443 0.9989789
```

4.2 In another study

Now, the tricky part is that the genes in the new study are not on the same scale. To simulate that, we add a scaling factor (here more or less between .1 and 10)

```
scale = exp(rnorm(100));
```

This will give use new expected and observed values, as well as new recovered mixing coefficients (worse than previously):

```
e2 = e * scale
obs2 = matrix(rpois(length(e2), e2), ncol=ncol(e2)); rownames(obs2) = rownames(e);
W = t(nnlm(prot, obs2, loss='mkl')$coefficients);
show(diag(cor(W, mix)))

## [1] 0.9400875 0.7480119 0.9664416 0.9904923
```

So this is where the `NNMFother` function kicks in.

```
fit = NNMFother(obs2, prot)
show(diag(cor(fit$W, mix)))

## [1] 0.9989843 0.9988874 0.9990185 0.9992698

show(cor(fit$sc, scale, method='s'))

## [1] 0.9989559
```

And what if some genes (1 to 20) were just garbage?

```
obs3 = obs2;
for (i in 1:20) { obs3[i,] = sample(obs3[i,], replace=TRUE) }
fit = NNMFother(obs3, prot, maxIter=20)
show(diag(cor(fit$W, mix)))
```

```
## [1] 0.9894250 0.9868811 0.9618847 0.9834111

show(cor(fit$sc, scale, method='s'))

## [1] 0.9987279
```

Results are substantially worse. It is however possible to judge how well each gene fits the NMF.

```
r = (prot*fit$sc) %*% t(fit$W)
cc = sapply(1:nrow(r), function(i) cor(r[i,], obs3[i,], method='s'))
show(fivenum(cc[1:20]))

## [1] -0.17499576 -0.02385935  0.07495179  0.13437735  0.33434267

show(fivenum(cc[21:100]))

## [1] 0.1067448 0.8006673 0.9046835 0.9463207 0.9879229
```

The correlation of OK genes is much higher. If we redo the analysis on only the genes with correlation of at least `0.7`

```
w = cc>.7;
fit = NNMFother(obs3[w,], prot[w,], maxIter=20)
show(diag(cor(fit$W, mix)))

## [1] 0.9989343 0.9985187 0.9988844 0.9992216
```

Of course here the 0.7 cutoff is arbitrary and somewhat optimized.

5 Conclusion