There are three essential inputs in order to run $DataRemix()$.

- **svdres**: This stands for the SVD decomposition output of the gene expression profile $svd(\mathbf{matrix})$. If the matrix is large, SVD decomposition doesn't need to be full-rank $min(nrow(\mathbf{matrix}), ncol(\mathbf{matrix}))$ which is computationally intensive.
- **matrix**: This stands for the gene expression profile with the dimension $gene-by-sample$. If **svdres** is not full-rank, matrix needs to be included in order to calculate the residual. Generally including **matrix** makes the residual computation more efficient.
- **objective function**: Users have to specify a objective of interest. The objective function would use remixed data based on **svdres** and **matrix** as input. It's natural to include more parameters in the objective function. The following two examples will demonstrate how to include objective-specific parameters into the $DataRemix()$ function.

Here we list two examples to illustrate how to run $DataRemix()$ function. The first example is to optimize known pathway recovery based on the GTEx gene expression profile. The second case is a toy example where we know the ground truth.

1 GTEx Correlation Network

In this section, we define the objective to be optimizing the known pathway recovery based on the GTEx gene expression data. We formally define the objective as the average AUC across pathways and we also keep track of the average AUPR value. $corMatToAUC()$ is the main objective function with two inputs: $data$ and GS. $data$ is the GTEx gene correlation matrix and GS stands for the pathway matrix. You can refer to the $corMatToAUC()$ documentation for more information.

```
> library(DataRemix)
```

We first load the data. $GTex\_cc$ stands for the GTEx gene correlation matrix with dimension 7294-by-7294 and *canonical* represents the canonical mSigDB pathways with dimension 7294-by-1330. $GTex\_cc$ and *canonical* correspond to $data$ and GS as input to $corMatToAUC()$. In this case, we directly remix the correlation matrix $GTex\_cc$. The other way is to remix the gene expression profile first and then calculate the correlation matrix where $GTex\_cc$ is remixed in an indirect way. First we need to perform SVD decomposition of $GTex\_cc$. Since it takes long to decompose $GTex\_cc$, we pre-compute the SVD decomposition of $GTex\_cc$ and load it as $GTex\_svdres$.

```
> load(url("https://www.dropbox.com/s/o949wkg76k0ccaw/GTex_cc.rdata?dl=1"))
> load(url("https://www.dropbox.com/s/wsuze8w2rp0syqg/GTex_svdres.rdata?dl=1"))
> load(url("https://github.com/wgmao/DataRemix/blob/master/inst/extdata/canonical.rdata?raw=
> #svdres <- svd(GTex_cc)
```

We first run $corMatToAUC()$ on the un-remixed correlation matrix $GTex\_cc$ to show what $corMatToAUC()$ outputs.

```
> GTex_default <- corMatToAUC(GTex_cc, canonical, objective = "mean.AUC")
> GTex_default
```

```
[1] 0.04512071 0.72433810
```

The first value corresponds to the average AUPR across all pathways and the second value corresponds to the average AUC across all pathways. This is the default behavior of $corMatToAUC()$. We now try to infer the optimal combinations of k, p and μ using $DataRemix()$. In this case GS is the additional input required by $corMatToAUC()$ function. Users just need to include any additional parameter like GS required by the objective at the end of function input.

```
> rownames(GTex_svdres$u) <- rownames(GTex_cc)
> rownames(GTex_svdres$v) <- colnames(GTex_cc)
> DataRemix.res <- DataRemix(GTex_svdres, GTex_cc, corMatToAUC,
+                            k_limits = c(1, length(GTex_svdres$d)%/%2),
+                            p_limits = c(-1,1), mu_limits = c(1e-12,1),
+                            num_of_initialization = 5, num_of_thompson = 150,
+                            basis = "omega", basis_size = 2000, verbose = F,
+                            GS = canonical)
```

It is highly recommended to assign *rownames* and *colnames* to **svdres**. Other parameters are explained as follows.

- **k_limits = c(1, length(GTex_svdres$d)/2)**: The upper limit of possible k is half of the rank which is 3,647 in this case.
- **p_limits = c(-1,1)**: This is the default range for p
- **mu_limits = c(1e-12,1)**: The is the default range for μ
- **num_of_initialization = 5**: Number of initialization steps before Thompson Sampling starts. It doesn't need to be a large number and 5 is the default option.
- **num_of_thompson = 150**: Number of Thompson Sampling steps. Generally the performance of the objective will be improved as sampling steps increase.
- **basis = "omega"**: The default option is to use the exponential kernel. There are also Gaussian kernel and Laplacian kernel as available options.
- **basis_size = 2000**: As **base_size** increases, the approximation of kernel will be more accurate. 2,000 is a good trade-off in general.
- **verbose = F**: If the computation takes long time to finish, it's helpful to print out intermediate results by setting **verbose** to be True.

We can convert the output from *DataRemix*() into a ranking table and we can easily tell the best combinations of parameters by looking at this ranking table. Here are the explanations of the *DataRemix_display*() parameters.

- *DataRemix.res*: This is the output in the last step.
- *col.names* = *c*(”*Rank*”,”*k*”,”*p*”,”*mu*”,”*meanAUPR*”,”*meanAUC*”): The first four values (”Rank”, ”k”, ”p”, ”mu”) are fixed. Two additional values (”mean AUPR”, ”mean AUC”) correspond to the output values of the objective function *corMatToAUC*(). These additional vaules need to be customized based on the objective function in use.
- *top.rank* = 15: We want to see the top 15 best-performing combinations of parameters.

```
> DataRemix_display(DataRemix.res, col.names = c("Rank", "k", "p", "mu",
+                    "mean AUPR", "mean AUC"), top.rank = 15)
```

```
|Rank |k    |p         |mu        |mean AUPR |mean AUC  |
|:----|:----|:---------|:---------|:---------|:---------|
|1    |1332 |0.3116507 |1.0000000 |0.1054801 |0.7763531 |
|2    |2138 |0.3145490 |1.0000000 |0.1074130 |0.7762209 |
|3    |712  |0.3182285 |0.0132889 |0.0991840 |0.7761954 |
|4    |818  |0.3189755 |0.0000003 |0.1002526 |0.7761044 |
|5    |2419 |0.3304548 |0.0003722 |0.1063780 |0.7760946 |
|6    |2175 |0.3554004 |0.0000069 |0.1037526 |0.7760602 |
|7    |814  |0.3159315 |0.0000305 |0.1003838 |0.7760317 |
|8    |539  |0.3058044 |0.0000000 |0.0973729 |0.7759846 |
|9    |1522 |0.3113343 |0.0000000 |0.1057226 |0.7759222 |
|10   |2333 |0.3267357 |0.0000008 |0.1065736 |0.7759112 |
|11   |3432 |0.3305944 |0.0008201 |0.1063380 |0.7758878 |
|12   |2578 |0.3410727 |0.0000000 |0.1055098 |0.7758741 |
|13   |2074 |0.3156959 |0.0000000 |0.1071175 |0.7758633 |
|14   |3123 |0.3449386 |0.0000000 |0.1049870 |0.7758340 |
|15   |1021 |0.3170331 |0.0005042 |0.1023577 |0.7758292 |
```

2 A Toy Example

In this section, we define a simple objective function called *eval*() which calculates the sum of a penalty term and the squared error between the DataRemix reconstruction and the original input matrix. The input matrix is a 100-by-9 matrix with random values. In this case, we know that when (k=9, p=1) or ($\mu = 1$, p=1), DataRemix reconstruction is the same as the original matrix and the objective function achieves the minimal value which is equal to the penalty term we add.

```
> library(DataRemix)
> eval <- function(X_reconstruct, X, penalty){
+   return(-sum((X-X_reconstruct)^2)+penalty)
+ }#eval
```

First we generate a random matrix with dimension 100-by-9 and perform the SVD decomposition.

```
> set.seed(1)
> num_of_row <- 100
> num_of_col <- 9
> X <- matrix(rnorm(num_of_row*num_of_col), nrow = num_of_row, ncol = num_of_col)
> svdres <- svd(X)
```

Here X and *penalty* are additional inputs for the *eval*() function. If we have the full SVD decomposition, we can leave matrix as NULL. For some large-scale matrices, if the SVD computation is time intensive, we don't need to finish the full SVD. Instead we can just compute the SVD decomposition up to a sufficient rank and include the original gene expression profile to calculate the residual.

```
> DataRemix.res <- DataRemix(svdres, matrix = NULL, eval,
+                  k_limits = c(1, length(svdres$d)), p_limits = c(-1,1),
+                  mu_limits = c(1e-12,1), num_of_initialization = 5,
+                  num_of_thompson = 50, basis = "omega", basis_size = 2000,
+                  xi = 0.1, full = T, verbose = F, X = X, penalty = 100)
```

We can convert the output from DataRemix into a ranking table with the help of *DataRemix_display*(). Here we want to check the performance of all sampling steps including initialization steps and Thompson Sampling steps.

```
> DataRemix_display(DataRemix.res, col.names = c("Rank", "k", "p", "mu", "Eval")
+                   , top.rank = 55)
```

Rank	k	p	mu	Eval
1	9	1.0000000	0.5346220	100.000000
2	9	1.0000000	0.0000002	100.000000
3	9	1.0000000	0.0020029	100.000000
4	9	1.0000000	0.0000000	100.000000
5	8	1.0000000	1.0000000	100.000000
6	9	1.0000000	0.0000000	100.000000
7	9	0.9885065	0.2289662	99.311898
8	1	0.9529448	1.0000000	97.975183
9	1	0.9519021	0.9843942	97.695083
10	9	0.9724742	0.0050690	96.199642
11	8	1.0000000	0.5972939	90.131563
12	4	0.9195328	1.0000000	82.287338
13	9	0.9098824	0.0000000	64.766607

14	9	0.8858570	0.0000000	46.484357
15	7	1.0000000	0.3397068	40.339607
16	8	1.0000000	0.0000000	39.148404
17	8	0.9829893	0.0063752	38.505809
18	5	0.8468720	1.0000000	35.804011
19	7	0.9071131	0.4959941	32.402425
20	8	0.9560001	0.0008490	30.362741
21	8	0.9334786	0.0000000	19.872703
22	8	0.8955563	0.0000023	-4.386826
23	4	0.9761921	0.4788741	-13.743072
24	7	0.7920235	1.0000000	-26.907903
25	7	1.0000000	0.0008924	-36.595679
26	7	1.0000000	0.0000078	-36.837663
27	7	1.0000000	0.0000000	-36.839786
28	1	-0.3328697	1.0000000	-49.196025
29	8	0.8238091	0.0721614	-57.771414
30	9	0.7668662	0.0000008	-71.658327
31	7	0.8649459	0.0000000	-99.852010
32	6	0.9774939	0.0000001	-119.276886
33	9	0.7085628	0.0000000	-136.993698
34	1	-1.0000000	0.6418311	-160.524211
35	3	0.3084525	1.0000000	-188.915207
36	2	-0.5627978	1.0000000	-192.583489
37	9	0.6405562	0.0003280	-213.240183
38	5	0.8670688	0.0586511	-218.104981
39	8	0.5435504	1.0000000	-293.878895
40	3	-1.0000000	1.0000000	-323.198408
41	9	0.5303805	0.0000000	-329.654878
42	5	0.3357020	0.8206325	-333.830467
43	4	0.7414971	0.0000011	-433.784003
44	1	0.9830502	0.1402641	-491.662872
45	4	-0.3851272	0.5282913	-503.773466
46	5	-0.8533292	0.8497418	-550.780064
47	2	1.0000000	0.0001793	-555.267264
48	8	0.1224019	0.0000000	-649.549622
49	1	0.9899227	0.0000000	-700.187003
50	1	-0.2663078	0.0739005	-733.472061
51	4	0.0626453	0.0007248	-755.796228
52	3	0.0283578	0.0000000	-786.519047
53	1	0.2086664	0.0000142	-819.885212
54	9	-0.8925854	0.0000000	-837.037311
55	9	-0.9629881	0.1921438	-840.465127