MLSeq: Machine Learning Interface to RNA-Seq Data

***Dincer Goksuluk*[*1], *Gokmen Zararsiz*[2], *Selcuk Korkmaz*[3], *Vahap Eldem*[4], *Bernd Klaus*[5], *Ahmet Ozturk*[2] and *Ahmet Ergun Karaagaoglu*[1]**

[1] Hacettepe University, Faculty of Medicine, Department of Biostatistics, Ankara, TURKEY

[2] Erciyes University, Faculty of Medicine, Department of Biostatistics, Kayseri, TURKEY

[3] Trakya University, Faculty of Medicine, Department of Biostatistics, Edirne, TURKEY

[4] Istanbul University, Faculty of Science, Department of Biology, Istanbul, TURKEY

[5] EMBL Heidelberg, Heidelberg, Germany

*dincer.goksuluk@hacettepe.edu.tr

May 19, 2021

NOTE: *MLSeq* has major changes from version **1.20.1** and this will bump following versions to **2.y.z** in the next release of Bioconductor (ver. 3.8). Most of the functions from previous versions were changed and new functions are included. Please see Beginner's Guide before continue with the analysis.

Abstract

MLSeq is a comprehensive package for application of machine-learning algorithms in classification of next-generation RNA-Sequencing (RNA-Seq) data. Researchers have appealed to *MLSeq* for various purposes, which include prediction of disease outcomes, identification of best subset of features (genes, transcripts, other isoforms), and sorting the features based on their predictive importance. Using this package, researchers can upload their raw RNA-seq count data, preprocess their data and perform a wide range of machine-learning algorithms. Preprocessing approaches include deseq median ratio and trimmed mean of M means (TMM) normalization methods, as well as the logarithm of counts per million reads (log-cpm), variance stabilizing transformation (vst), regularized logarithmic transformation (rlog) and variance modeling at observational level (voom) transformation approaches. Normalization approaches can be used to correct systematic variations. Transformation approaches can be used to bring discrete RNA-seq data hierarchically closer to microarrays and conduct microarray-based classification algorithms. Currently, *MLSeq* package contains 90 microarray-based classifiers including the recently developed voom-based discriminant analysis classifiers. Besides these classifiers, *MLSeq* package also includes discrete-based classifiers, such as Poisson linear discriminant analysis (PLDA) and negative binomial linear discriminant analysis (NBLDA). Over the preprocessed data, researchers can build classification models, apply parameter optimization on these models, evaluate the model performances and compare the performances of different classification models. Moreover, the class labels of test samples can be predicted with the built models. *MLSeq* is a user friendly, simple and currently the most comprehensive package developed in the literature for RNA-Seq classification. To start using this package, users need to upload their count data, which contains the number of reads mapped to each transcript for each sample. This kind of count data can be obtained from

RNA-Seq experiments, also from other sequencing experiments such as ChIP-sequencing or metagenome sequencing. This vignette is presented to guide researchers how to use this package.

MLSeq version: 2.10.0

Contents

1 Introduction

With the recent developments in molecular biology, it is feasible to measure the expression levels of thousands of genes simultaneously. Using this information, one major task is the gene-expression based classification. With the use of microarray data, numerous classification algorithms are developed and adapted for this type of classification. RNA-Seq is a recent technology, which uses the capabilities of next-generation sequencing (NGS) technologies. It has some major advantages over microarrays such as providing less noisy data and detecting novel transcripts and isoforms. These advantages can also affect the performance of classification algorithms. Working with less noisy data can improve the predictive performance of classification algorithms. Further, novel transcripts may be a biomarker in related disease or phenotype. *MLSeq* package includes several classification algorithms, also normalization and transformation approaches for RNA-Seq classification.

In this vignette, you will learn how to build machine-learning models from raw RNA-Seq count data. *MLSeq* package can be loaded as below:

```
library(MLSeq)
```

2 Preparing the input data

MLSeq package expects a count matrix that contains the number of reads mapped to each transcript for each sample and class label information of samples in an S4 class *DESeqDataSet*.

After mapping the RNA-Seq reads to a reference genome or transcriptome, number of reads mapped to the reference genome can be counted to measure the transcript abundance. It is very important that the count values must be raw sequencing read counts to implement the methods given in *MLSeq*. There are a number of functions in Bioconductor packages which summarizes mapped reads to a count data format. These tools include `featureCounts` in *Rsubread* [1], `summarizeOverlaps` in *GenomicRanges* [2] and *easyRNASeq* [3]. It is also possible to access this type of count data from Linux-based softwares as *htseq-count* function in *HTSeq* [4] and *multicov* function in *bedtools* [5] softwares. In this vignette, we will work with the cervical count data. Cervical data is from an experiment that measures the expression levels of 714 miRNAs of human samples [6]. There are 29 tumor and 29 non-tumor cervical samples and these two groups can be treated as two separate classes for classification purpose. We can define the file path with using `system.file`:

```
filepath <- system.file("extdata/cervical.txt", package = "MLSeq")
```

Next, we can load the data using `read.table`:

```
cervical <- read.table(filepath, header=TRUE)
```

After loading the data, one can check the counts as follows. These counts are the number of mapped miRNA reads to each transcript.

```
head(cervical[ ,1:10]) # Mapped counts for first 6 features of 10 subjects.
```

```
##          N1   N2   N3    N4   N5  N6 N7   N8   N9   N10
## let-7a  865  810 5505  6692 1456 588  9 4513 1962 10167
## let-7a*   3   12   30    73    6   2  0  199   10   173
## let-7b  975 2790 4912 24286 1759 508 33 6162 1455 18110
## let-7b*  15   18   27   119   11   3  0  116   17   233
## let-7c  828 1251 2973  6413  713 339 23 2002  476  3294
## let-7c*   0    0    0     1    0   0  0    3    0     3
```

Cervical data is a *data.frame* containing 714 miRNA mapped counts given in rows, belonging to 58 samples given in columns. First 29 columns of the data contain the miRNA mapped counts of non-tumor samples, while the last 29 columns contain the count information of tumor samples. We need to create a class label information in order to apply classification models. The class labels are stored in a *DataFrame* object generated using `DataFrame` from *S4Vectors*. Although the formal object returned from `data.frame` can be imported into *DESeqDataSet*, we suggest using *DataFrame* in order to prevent possible warnings/errors during downstream analyses.

```
class <- DataFrame(condition = factor(rep(c("N","T"), c(29, 29))))
class
```

```
## DataFrame with 58 rows and 1 column
##     condition
##      <factor>
## 1           N
## 2           N
## 3           N
## 4           N
## 5           N
## ...       ...
## 54          T
## 55          T
## 56          T
## 57          T
## 58          T
```

3 Splitting the data

We can split the data into two parts as training and test sets. Training set can be used to build classification models, and test set can be used to assess the performance of each model. The ratio of splitting data into two parts depends on total sample size. In most studies, the amount of training set is taken as 70% and the remaining part is used as test set. However, when the number of samples is relatively small, the split ratio can be decreased towards 50%. Similarly, if the total number of samples are large enough (e.g 200, 500 etc.), this ratio might be increased towards 80% or 90%. The basic idea of defining optimum splitting ratio can be expressed as: 'define such a value for splitting ratio where we have enough samples in the training and test set in order to get a reliable fitted model and test predictions.' For our example, cervical data, there are 58 samples. One may select 90% of the samples (approx. 52 subjects) for training set. The fitted model is evantually reliable, however, test accuracies

are very sensitive to unit misclassifications. Since there are only 6 observations in the test set, misclassifying a single subject would decrease test set accuracy approximately 16.6%. Hence, we should carefully define the splitting ratio before continue with the classification models.

```
library(DESeq2)

set.seed(2128)

# We do not perform a differential expression analysis to select differentially
# expressed genes. However, in practice, DE analysis might be performed before
# fitting classifiers. Here, we selected top 100 features having the highest
# gene-wise variances in order to decrease computational cost.
vars <- sort(apply(cervical, 1, var, na.rm = TRUE), decreasing = TRUE)
data <- cervical[names(vars)[1:100], ]
nTest <- ceiling(ncol(data) * 0.3)
ind <- sample(ncol(data), nTest, FALSE)

# Minimum count is set to 1 in order to prevent 0 division problem within
# classification models.
data.train <- as.matrix(data[ ,-ind] + 1)
data.test <- as.matrix(data[ ,ind] + 1)
classtr <- DataFrame(condition = class[-ind, ])
classts <- DataFrame(condition = class[ind, ])
```

Now, we have 40 samples which will be used to train the classification models and have remaining 18 samples to be used to test the model performances. The training and test sets are stored in a *DESeqDataSet* using related functions from *DESeq2* [7]. This object is then used as input for *MLSeq*.

```
data.trainS4 = DESeqDataSetFromMatrix(countData = data.train, colData = classtr,
                                      design = formula(~condition))
data.testS4 = DESeqDataSetFromMatrix(countData = data.test, colData = classts,
                                     design = formula(~condition))
```

4 Available machine-learning models

MLSeq contains more than 90 algorithms for the classification of RNA-Seq data. These algorithms include both microarray-based conventional classifiers and novel methods specifically designed for RNA-Seq data. These novel algorithms include voom-based classifiers [8], Poisson linear discriminant analysis (PLDA) [9] and Negative-Binomial linear discriminant analysis (NBLDA) [10]. Run `availableMethods` for a list of supported classification algorithm in *MLSeq*.

5 Normalization and transformation

Normalization is a crucial step of RNA-Seq data analysis. It can be defined as the determination and correction of the systematic variations to enable samples to be analyzed in the same scale. These systematic variations may arise from both between-sample variations including library size (sequencing depth) and the presence of majority fragments; and within-sample variations including gene length and sequence composition (GC content). In *MLSeq*, two effective normalization methods are available. First one is the "deseq median ratio normalization", which estimates the size factors by dividing each sample by the geometric means of the transcript counts [7]. Median statistic is a widely used statistics as a size factor for each sample. Another normalization method is "trimmed mean of M values (TMM)". TMM first trims the data in both lower and upper side by log-fold changes (default 30%) to minimize the log-fold changes between the samples and by absolute intensity (default 5%). After trimming, TMM calculates a normalization factor using the weighted mean of data. These weights are calculated based on the inverse approximate asymptotic variances using the delta method [11]. Raw counts might be normalized using either `deseq`-median ratio or `TMM` methods.

After the normalization process, it is possible to directly use the discrete classifiers, e.g. PLDA and NBLDA. In addition, it is possible to apply an appropriate transformation on raw counts and bring the data hierarchically closer to microarrays. In this case, we can transform the data and apply a large number of classifiers, e.g. nearest shrunken centroids, penalized discriminant analysis, support vector machines, etc. One simple approach is the logarithm of counts per million reads (log-cpm) method, which transforms the data from the logarithm of the division of the counts by the library sizes and multiplication by one million (Equation 1). This transformation is simply an extension of the shifted-log transformation $z_{ij} = \log_2 x_{ij} + 1$.

$$z_{ij} = \log_2 \left(\frac{x_{ij} + 0.5}{X_{.j} + 1} \times 10^6 \right) \quad (1)$$

Although log-cpm transformation provides less-skewed distribution, the gene-wise variances are still unequal and possibly related with the distribution mean. Hence, one may wish to transform data into continuous scale while controlling the gene-wise variances. Anders and Huber [12] presented variance stabilizing transformation (vst) which provides variance independent from mean. Love et al. [7] presented regularized logarithmic (rlog) transformation. This method uses a shrinkage approach as used in *DESeq2* paper. Rlog transformed values are similar with vst or shifted-log transformed values for genes with higher counts, while shrunken together for genes with lower counts. *MLSeq* allows researchers perform one of transformations `log-cpm`, `vst` and `rlog`. The possible `normalization-transformation` combinations are:

- `deseq-vst`: Normalization is applied with deseq median ratio method. Variance stabilizing transformation is applied to the normalized data
- `deseq-rlog`: Normalization is applied with deseq median ratio method. Regularized logarithmic transformation is applied to the normalized data
- `deseq-logcpm`: Normalization is applied with deseq median ratio method. Log of counts-per-million transformation is applied to the normalized data
- `tmm-logcpm`: Normalization is applied with trimmed mean of M values (TMM) method. Log of counts-per-million transformation is applied to the normalized data.

The normalization-transformation combinations are controlled by `preProcessing` argument in `classify`. For example, we may apply rlog transformation on deseq normalized counts by setting `preProcessing = "deseq-rlog"`. See below code chunk for a minimal working example.

```
# Support Vector Machines with Radial Kernel
fit <- classify(data = data.trainS4, method = "svmRadial",
                preProcessing = "deseq-rlog", ref = "T",
                control = trainControl(method = "repeatedcv", number = 2,
                                       repeats = 2, classProbs = TRUE))
show(fit)
```

Furthermore, Zararsiz et al. [8] presented voomNSC classifier, which integrates voom transformation [13] and NSC method [14, 15] into a single and powerful classifier. This classifier extends voom method for RNA-Seq based classification studies. VoomNSC also makes NSC algorithm available for RNA-Seq technology. The authors also presented the extensions of diagonal discriminant classifiers [16], i.e. voom-based diagonal linear discriminant analysis (voomDLDA) and voom based diagonal quadratic discriminant analysis (voomDQDA) classifiers. All three classifiers are able to work with high-dimensional ($n < p$) RNA-Seq counts. VoomDLDA and voomDQDA approaches are non-sparse and use all features to classify the data, while voomNSC is sparse and uses a subset of features for classification. Note that the argument `preProcessing` has no effect on voom-based classifiers since voom transformation is performed within classifier. However, we may define normalization method for voom-based classifiers using `normalize` arguement. As an example, consider fitting a voomNSC model on deseq normalized counts:

```
set.seed(2128)

# Voom based Nearest Shrunken Centroids.
fit <- classify(data = data.trainS4, method = "voomNSC",
                normalize = "deseq", ref = "T",
                control = voomControl(tuneLength = 20))

trained(fit)  ## Trained model summary
```

We will cover trained model in section Optimizing model parameters.

6 Model building

The *MLSeq* has a single function `classify` for the model building and evaluation process. This function can be used to evaluate selected classifier using a set of values for model parameter (aka *tuning parameter*) and return the optimal model. The overall model performances for training set are also returned.

6.1 Optimizing model parameters

MLSeq evaluates k-fold repeated cross-validation on training set for selecting the optimal value of tuning parameter. The number of parameters to be optimized depends on the selected classifier. Some classifiers have two or more tuning parameter, while some have no

tuning parameter. Suppose we want to fit RNA-Seq counts to Support Vector Machines with Radial Basis Function Kernel (svmRadial) using deseq normalization and vst transformation,

```
set.seed(2128)

# Support vector machines with radial basis function kernel
fit.svm <- classify(data = data.trainS4, method = "svmRadial",
                    preProcessing = "deseq-vst", ref = "T", tuneLength = 10,
                    control = trainControl(method = "repeatedcv", number = 5,
                                           repeats = 10, classProbs = TRUE))

show(fit.svm)
```

```
##
##   An object of class "MLSeq"
##   Model Description: Support Vector Machines with Radial Basis Function Kernel (svmRadial)
##
##             Method   :  svmRadial
##
##        Accuracy(%)   :  95
##     Sensitivity(%)   :  94.12
##     Specificity(%)   :  95.65
##
##   Reference Class    :  T
```

The model were trained using 5-fold cross validation repeated 10 times. The number of levels for tuning parameter is set to 10. The length of tuning parameter space, `tuneLength`, may be increased to be more sensitive while searchin optimal value of the parameters. However, this may drastically increase the total computation time. The tuning results are obtained using setter function `trained` as,

```
trained(fit.svm)
```

```
## Support Vector Machines with Radial Basis Function Kernel
##
##  40 samples
## 100 predictors
##   2 classes: 'N', 'T'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 10 times)
## Summary of sample sizes: 31, 33, 32, 32, 32, 32, ...
## Resampling results across tuning parameters:
##
##   C       Accuracy   Kappa
##     0.25  0.8424603  0.6937976
##     0.50  0.9103571  0.8204990
##     1.00  0.9329365  0.8639560
##     2.00  0.9486508  0.8933900
##     4.00  0.9433730  0.8801816
##     8.00  0.9505952  0.8975496
```

```
##    16.00  0.9461508  0.8889782
##    32.00  0.9458730  0.8870002
##    64.00  0.9360714  0.8668239
##   128.00  0.9405159  0.8725831
##
## Tuning parameter 'sigma' was held constant at a value of 0.006054987
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were sigma = 0.006054987 and C = 8.
```

The optimal values for tuning parameters were sigma $= 0.00605$ and C $= 8$. The effect of tuning parameters on model accuracies can be graphically seen in Figure 1.

```
plot(fit.svm)
```

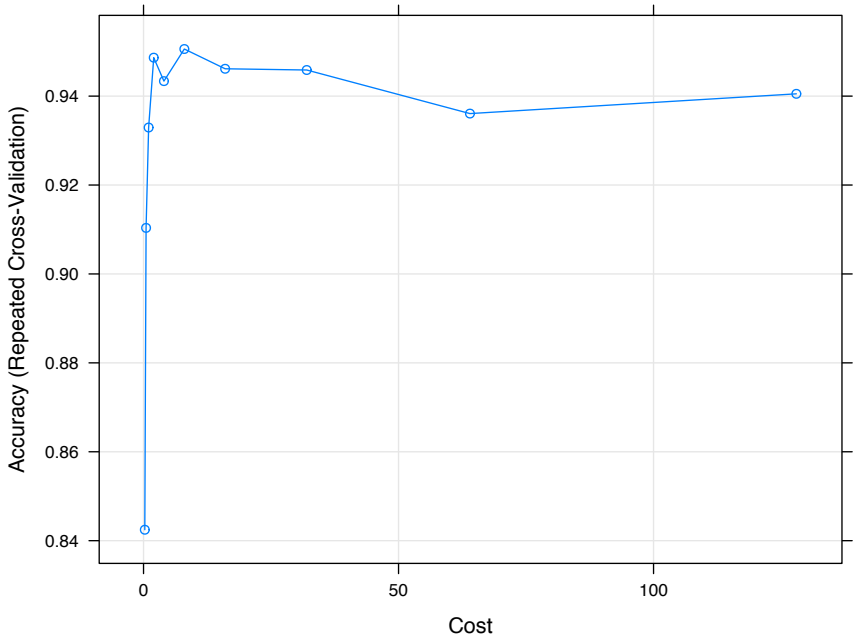

Figure 1: Tuning results for fitted model (svmRadial)

6.2 Defining control list for selected classifier

For each classifier, it is possible to define how model should be created using control lists. We may categorize available classifiers into 3 partitions, i.e *continuous*, *discrete* and *voom-based* classifiers. Continuous classifiers are based on *caret*'s library while discrete and voom-based classifiers use functions from *MLSeq*'s library. Since each classifier category has different control parameters to be used while building model, we should use corresponding control function for selected classifiers. We provide three different control functions, i.e (i) `trainControl` for continuous, (ii) `discreteControl` for discrete and (iii) `voomControl` for voom-based classifiers as summarized in Table 1.

Table 1: Control functions for classifiers

Function	Classifier
`discreteControl`	PLDA, PLDA2, NBLDA
`voomControl`	voomDLDA, voomDQDA, voomNSC
`trainControl`	All others.

Now, we fit *svmRadial*, *voomDLDA* and *PLDA* classifiers to RNA-seq data and find the optimal value of tuning parameters, if available, using 5-fold cross validation without repeats. We may control model building process using related function for the selected classifier (Table 1).

```
# Define control list
ctrl.svm <- trainControl(method = "repeatedcv", number = 5, repeats = 1)
ctrl.plda <- discreteControl(method = "repeatedcv", number = 5, repeats = 1,
                             tuneLength = 10)
ctrl.voomDLDA <- voomControl(method = "repeatedcv", number = 5, repeats = 1,
                             tuneLength = 10)

# Support vector machines with radial basis function kernel
fit.svm <- classify(data = data.trainS4, method = "svmRadial",
                preProcessing = "deseq-vst", ref = "T", tuneLength = 10,
                control = ctrl.svm)

# Poisson linear discriminant analysis
fit.plda <- classify(data = data.trainS4, method = "PLDA", normalize = "deseq",
                     ref = "T", control = ctrl.plda)

# Voom-based diagonal linear discriminant analysis
fit.voomDLDA <- classify(data = data.trainS4, method = "voomDLDA",
                         normalize = "deseq", ref = "T", control = ctrl.voomDLDA)
```

The fitted model for *voomDLDA*, for example, is obtained using folowing codes. Since *voomDLDA* has no tuning parameters, the training set accuracy is given over cross-validated folds.

```
trained(fit.voomDLDA)
```

```
##
## Voom-based Diagonal Linear Discriminant Analysis (voomDLDA)
##
##   40 samples
##  100 predictors
##    2 classes: 'N', 'T'  (Reference category: 'T')
##
## Normalization: DESeq median ratio.
## Resampling: Cross-Validated (5 fold, repeated 1 times)
## Summary of sample sizes:  32, 32, 32, 32, 32
## Summary of selected features:  All features are selected.
##
##      Model   Accuracy
```

```
##   voomDLDA   0.9250000
##
## There is no tuning parameter for selected method.
## Cross-validated model accuracy is given.
```

7 Predicting the class labels of test samples

Class labels of the test cases are predicted based on the model characteristics of the trained model, e.g. discriminating function of the trained model in discriminant-based classifiers. However, an important point here is that the test set must have passed the same steps with the training set. This is especially true for the normalization and transformation stages for RNA-Seq based classification studies. Same preprocessing parameters should be used for both training and test sets to affirm that both sets are on the same scale and homoscedastic each other. If we use deseq median ratio normalization method, then the size factor of a test case will be estimated using gene-wise geometric means, m_j, from training set as follows:

$$\hat{s}^* = \frac{m^*}{\sum_{j=1}^{n} m_j}, \quad m^* = \text{median}_i \left\{ \frac{x_i^*}{(\prod_{j=1}^{n} x_{ij})^{1/n}} \right\} \qquad (2)$$

A similar procedure is applied for the transformation of test data. If vst is selected as the transformation method, then the test set will be transformed based on the dispersion function of the training data. Otherwise, if rlog is selected as the transformation method, then the test set will be transformed based on the dispersion function, beta prior variance and the intercept of the training data.

MLSeq predicts test samples using training set parameters. There are two functions in *MLSeq* to be used for predictions, `predict` and `predictClassify`. The latter function is an alias for the generic function `predict` and was used as default method in *MLSeq* up to package version 1.14.z. Default function for predicting new observations replaced with `predict` from version 1.16.z and later. Hence, both can be used for same purpose.

Likely training set, test set should be given in *DESeqDataSet* class. The predictions can be done using following codes,

```
#Predicted class labels
pred.svm <- predict(fit.svm, data.testS4)
pred.svm
```

```
##  [1] T T N T N N T T T T T T N N N N T T
## Levels: N T
```

Finally, the model performance for the prediction is summarized as below using `confusionMatrix` from *caret*.

```
pred.svm <- relevel(pred.svm, ref = "T")
actual <- relevel(classts$condition, ref = "T")

tbl <- table(Predicted = pred.svm, Actual = actual)
confusionMatrix(tbl, positive = "T")
```

```
## Confusion Matrix and Statistics
##
##          Actual
## Predicted  T  N
##         T 11  0
##         N  1  6
##
##                Accuracy : 0.9444
##                  95% CI : (0.7271, 0.9986)
##     No Information Rate : 0.6667
##     P-Value [Acc > NIR] : 0.006766
##
##                   Kappa : 0.88
##
##  Mcnemar's Test P-Value : 1.000000
##
##             Sensitivity : 0.9167
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 0.8571
##              Prevalence : 0.6667
##          Detection Rate : 0.6111
##    Detection Prevalence : 0.6111
##       Balanced Accuracy : 0.9583
##
##        'Positive' Class : T
##
```

8 Comparing the performance of classifiers

In this section, we discuss and compare the performance of the fitted models in details. Before we fit the classifiers, a random seed is set for reproducibility as `set.seed(2128)`. Several measures, such as overall accuracy, sensitivity, specificity, etc., can be considered for comparing the model performances. We compared fitted models using overall accuracy and sparsity measures since the prevalence of positive and negative classes are equal. Sparsity is used as the measure of proportion of features used in the trained model. As sparsity goes to 0, less features are used in the classifier. Hence, the aim might be selecting a classifier which is sparser and better in predicting test samples, i.e higher in overall accuracy.

We selected SVM, voomDLDA and NBLDA as non-sparse classifiers and PLDA with power transformation, voomNSC and NSC as sparse classifiers for the comparison of fitted models. Raw counts are normalized using *deseq* method and *vst* transformation is used for continuous classifiers (NSC and SVM).

```
set.seed(2128)

# Define control lists.
ctrl.continuous <- trainControl(method = "repeatedcv", number = 5, repeats = 10)
ctrl.discrete <- discreteControl(method = "repeatedcv", number = 5, repeats = 10,
```

```
                                tuneLength = 10)
ctrl.voom <- voomControl(method = "repeatedcv", number = 5, repeats = 10,
                                tuneLength = 10)

# 1. Continuous classifiers, SVM and NSC
fit.svm <- classify(data = data.trainS4, method = "svmRadial",
                    preProcessing = "deseq-vst", ref = "T", tuneLength = 10,
                    control = ctrl.continuous)

fit.NSC <- classify(data = data.trainS4, method = "pam",
                    preProcessing = "deseq-vst", ref = "T", tuneLength = 10,
                    control = ctrl.continuous)

# 2. Discrete classifiers
fit.plda <- classify(data = data.trainS4, method = "PLDA", normalize = "deseq",
                     ref = "T", control = ctrl.discrete)

fit.plda2 <- classify(data = data.trainS4, method = "PLDA2", normalize = "deseq",
                     ref = "T", control = ctrl.discrete)

fit.nblda <- classify(data = data.trainS4, method = "NBLDA", normalize = "deseq",
                     ref = "T", control = ctrl.discrete)

# 3. voom-based classifiers
fit.voomDLDA <- classify(data = data.trainS4, method = "voomDLDA",
                         normalize = "deseq", ref = "T", control = ctrl.voom)

fit.voomNSC <- classify(data = data.trainS4, method = "voomNSC",
                         normalize = "deseq", ref = "T", control = ctrl.voom)

# 4. Predictions
pred.svm <- predict(fit.svm, data.testS4)
pred.NSC <- predict(fit.NSC, data.testS4)
# ... truncated
```

Table 2: Classification results for cervical data

Classifier	Accuracy	Sparsity
SVM	0.944	
NSC	0.889	0.910
PLDA (Transformed)	0.889	1.000
NBLDA	0.833	
voomDLDA	0.889	
voomNSC	0.722	0.020

Among selected predictors, we can select one of them by considering overall accuracy and sparsity at the same time. Table 2 showed that **SVM** has the highest classification accuracy. Similarly, **voomNSC** gives the lowest sparsity measure comparing to other classifiers. Using the performance measures from Table 2, one may decide the best classifier to be used in classification task.

In this tutorial, we compared only few classifiers and showed how to train models and predict new samples. We should note that the model performances depends on several criterias, e.g normalization and transformation methods, gene-wise overdispersions, number of classes etc. Hence, the model accuracies given in this tutorial should not be considered as a generalization to any RNA-Seq data. However, generalized results might be considered using a simulation study under different scenarios. A comprehensive comparison of several classifiers on RNA-Seq data can be accessed from Zararsiz et al. [17].

9 Determining possible biomarkers using sparse classifiers

In an RNA-Seq study, hundreds or thousands of features are able to be sequenced for a specific disease or condition. However, not all features but usually a small subset of sequenced features might be differentially expressed among classes and contribute to discrimination function. Hence, determining differentially expressed (DE) features are one of main purposes in an RNA-Seq study. It is possible to select DE features using sparse algorithm in *MLSeq* such as NSC, PLDA and voomNSC. Sparse models are able to select significant features which mostly contributes to the discrimination function by using built-in variable selection criterias. If a selected classifier is sparse, one may return selected features using getter function `selectedGenes`. For example, voomNSC selected 2% of all features. The selected features can be extracted as below:

```
selectedGenes(fit.voomNSC)

## [1] "miR-143"  "miR-125b"
```

We showed selected features from sparse classifiers on a venn-diagram in Figure 2. Some of the features are common between sparse classifiers. voomNSC, PLDA, PLDA2 (Power transformed) and NSC, for example, commonly discover 2 features as possible biomarkers.

```
## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"
```

```
## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"
```

```
## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Roman" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Bold" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Bold" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Bold" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Bold" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Bold" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Bold" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Bold" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Bold" could not be found for family "serif"

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
font "Times-Bold" could not be found for family "serif"
```

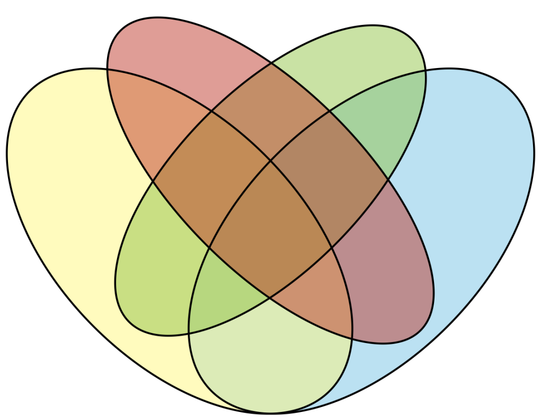

Figure 2: Venn-diagram of selected features from sparse classifiers

10 Updating an MLSeq object using `update`

MLSeq is developed using S4 system in order to make it compatible with most of the BIOCONDUCTOR packages. We provide setter/getter functions to get or replace the contents of an S4 object returned from functions in *MLSeq*. Setter functions are useful when one wishes to change components of an S4 object and carry out its effect on the remaining components. For example, a setter function `method<-` can be used to change the classification method of a given *MLSeq* object. See following code chunks for an example.

```
set.seed(2128)

ctrl <- discreteControl(method = "repeatedcv", number = 5, repeats = 2,
                        tuneLength = 10)

# PLDA without power transformation
fit <- classify(data = data.trainS4, method = "PLDA", normalize = "deseq",
                ref = "T", control = ctrl)
show(fit)
```

```
##
##   An object of class "MLSeq"
##   Model Description: Poisson Linear Discriminant Analysis (PLDA)
##
##             Method   :  PLDA
##
##        Accuracy(%)   :  92.68
##     Sensitivity(%)   :  94.12
##     Specificity(%)   :  91.67
##
##   Reference Class    :  T
```

Now, we may wish to see the results from PLDA classifier with power transformation. We can either change the corresponding arguement as `method = "PLDA2"` and run above codes or simply use the generic function `update` after related replacement method `method<-`. Once the method has been changed, a note is returned with *MLSeq* object.

```
method(fit) <- "PLDA2"
show(fit)
```

```
##
##   An object of class "MLSeq"
##   Model Description: Poisson Linear Discriminant Analysis with Power Transformation (PLDA2)
##
##   NOTE: MLSeq object is modified but not updated.
##         Update 'MLSeq' object to get true classification accuracies.
##
##             Method   :  PLDA2
##
##        Accuracy(%)   :  92.68
##     Sensitivity(%)   :  94.12
##     Specificity(%)   :  91.67
```

```
##
##   Reference Class   :  T
```

It is also possible to change multiple arguments at the same time using related setter functions. In such cases, one may run `metaData(...)` for a detailed information on fitted object.

```
ref(fit) <- "N"
normalization(fit) <- "TMM"
metaData(fit)
```

```
## class:  MLSeqMetaData, in S4 class
## Updated:  NO
## Modified:  YES
## Modified Elements (3):  method, ref, normalization
## Initial Data:  A DESeqDataSet object
```

It can bee seen from `metaData(fit)` that several modifications have been requested for fitted model but it is not updated. We should run `update` to carry over the effect of modified object into *MLSeq* object. One should note that the updated object should be assigned to the same or different object since `update` does not overwrite fitted model.

```
fit <- update(fit)
```

```
##
##
##  Update is successfull...
```

```
show(fit)
```

```
##
##   An object of class "MLSeq"
##   Model Description: Poisson Linear Discriminant Analysis with Power Transformation (PLDA2)
##
##            Method   :  PLDA2
##
##       Accuracy(%)   :  95
##     Sensitivity(%)  :  95.65
##     Specificity(%)  :  94.12
##
##   Reference Class   :  N
```

10.1 Transitions between continuous, discrete and voom-based classifiers

The control lists and some of the arguments in `classify` need to be specified depending on the selected classifier. This constraint should be carefully taken into account while updating an *MLSeq* object. We may wish to move from continuous based classifier to discrete or voom-based classifier, and vice versa. Consider we want to change classifier to "rpart" for `fit`.

```
method(fit) <- "rpart"
update(fit)
```

```
## Warning in stop(warning("Incorrect elements in 'control' argument.  It should
be defined using 'trainControl(...)'  function.")):  Incorrect elements in 'control'
argument.  It should be defined using 'trainControl(...)'  function.
```

```
## Error in .local(object, ...) :
##   Incorrect elements in 'control' argument. It should be defined using 'trainControl(...)' function.
```

Since the control list for continuous and discrete classifiers should be specified using related control function, the update process will end up with an error unless the control list is also modified. First, we specify appropriate control list and then change the classifier. Next, we may update fitted object as given below:

```
control(fit) <- trainControl(method = "repeatedcv", number = 5, repeats = 2)

# 'normalize' is not valid for continuous classifiers. We use 'preProcessing'
# rather than 'normalize'.
preProcessing(fit) <- "tmm-logcpm"

fit <- update(fit)
```

```
##
##
##  Update is successfull...
```

```
show(fit)
```

```
##
##   An object of class "MLSeq"
##   Model Description: Classification and Regression Tree (CART) (rpart)
##
##             Method   :  rpart
##
##        Accuracy(%)   :  85.37
##     Sensitivity(%)   :  83.33
##     Specificity(%)   :  88.24
##
##   Reference Class    :  N
```

Similar transitions can be done for voom-based classifiers. For a complete list of package functions, please see package manuals.

11 Session info

```
sessionInfo()
```

```
## R version 4.1.0 RC (2021-05-10 r80283)
## Platform: x86_64-apple-darwin17.0 (64-bit)
## Running under: macOS Mojave 10.14.6
```

```
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.1/Resources/lib/libRblas.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.1/Resources/lib/libRlapack.dylib
##
## locale:
## [1] C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
##  [1] grid      parallel  stats4    stats     graphics  grDevices utils
##  [8] datasets  methods   base
##
## other attached packages:
##  [1] xtable_1.8-4               pamr_1.56.1
##  [3] survival_3.2-11            cluster_2.1.2
##  [5] VennDiagram_1.6.20         futile.logger_1.4.3
##  [7] edgeR_3.34.0               limma_3.48.0
##  [9] DESeq2_1.32.0              SummarizedExperiment_1.22.0
## [11] Biobase_2.52.0             MatrixGenerics_1.4.0
## [13] matrixStats_0.58.0         GenomicRanges_1.44.0
## [15] GenomeInfoDb_1.28.0        IRanges_2.26.0
## [17] S4Vectors_0.30.0           BiocGenerics_0.38.0
## [19] MLSeq_2.10.0               caret_6.0-88
## [21] ggplot2_3.3.3              lattice_0.20-44
## [23] knitr_1.33
##
## loaded via a namespace (and not attached):
##  [1] colorspace_2.0-1       ellipsis_0.3.2         class_7.3-19
##  [4] XVector_0.32.0         proxy_0.4-25           bit64_4.0.5
##  [7] AnnotationDbi_1.54.0   prodlim_2019.11.13     fansi_0.4.2
## [10] lubridate_1.7.10       codetools_0.2-18       splines_4.1.0
## [13] cachem_1.0.5           geneplotter_1.70.0     pROC_1.17.0.1
## [16] annotate_1.70.0        kernlab_0.9-29         png_0.1-7
## [19] BiocManager_1.30.15    compiler_4.1.0         httr_1.4.2
## [22] assertthat_0.2.1       Matrix_1.3-3           fastmap_1.1.0
## [25] formatR_1.9            htmltools_0.5.1.1      tools_4.1.0
## [28] gtable_0.3.0           glue_1.4.2             GenomeInfoDbData_1.2.6
## [31] reshape2_1.4.4         dplyr_1.0.6            Rcpp_1.0.6
## [34] vctrs_0.3.8            Biostrings_2.60.0      nlme_3.1-152
## [37] iterators_1.0.13       timeDate_3043.102      gower_0.2.2
## [40] xfun_0.23              stringr_1.4.0          lifecycle_1.0.0
## [43] XML_3.99-0.6           zlibbioc_1.38.0        MASS_7.3-54
## [46] scales_1.1.1           ipred_0.9-11           BiocStyle_2.20.0
## [49] lambda.r_1.2.4         RColorBrewer_1.1-2     yaml_2.2.1
## [52] memoise_2.0.0          rpart_4.1-15           stringi_1.6.2
## [55] RSQLite_2.2.7          highr_0.9              genefilter_1.74.0
## [58] foreach_1.5.1          e1071_1.7-6            BiocParallel_1.26.0
## [61] lava_1.6.9             rlang_0.4.11           pkgconfig_2.0.3
## [64] bitops_1.0-7           evaluate_0.14          purrr_0.3.4
## [67] recipes_0.1.16         bit_4.0.4              tidyselect_1.1.1
## [70] sSeq_1.30.0            plyr_1.8.6             magrittr_2.0.1
```

```
## [73] R6_2.5.0              generics_0.1.0        DelayedArray_0.18.0
## [76] DBI_1.1.1             pillar_1.6.1          withr_2.4.2
## [79] KEGGREST_1.32.0       RCurl_1.98-1.3        nnet_7.3-16
## [82] tibble_3.1.2          crayon_1.4.1          futile.options_1.0.1
## [85] utf8_1.2.1            rmarkdown_2.8         locfit_1.5-9.4
## [88] data.table_1.14.0     blob_1.2.1            ModelMetrics_1.2.2.2
## [91] digest_0.6.27         munsell_0.5.0
```

References

[1] Yang Liao, Gordon K Smyth, and Wei Shi. featurecounts: an efficient general purpose program for assigning sequence reads to genomic features. *Bioinformatics*, 30(7): 923–930, 2014. URL https://doi.org/10.1093/bioinformatics/btt656.

[2] Michael Lawrence, Wolfgang Huber, Hervé Pages, Patrick Aboyoun, Marc Carlson, Robert Gentleman, Martin T Morgan, and Vincent J Carey. Software for computing and annotating genomic ranges. *PLoS Comput Biol*, 9(8):e1003118, 2013. URL https://doi.org/10.1371/journal.pcbi.1003118.

[3] Nicolas Delhomme, Ismaël Padioleau, Eileen E Furlong, and Lars M Steinmetz. easyRNASeq: a bioconductor package for processing RNA-seq data. *Bioinformatics*, 28(19):2532–2533, 2012. URL https://doi.org/10.1093/bioinformatics/bts477.

[4] Simon Anders, Paul Theodor Pyl, and Wolfgang Huber. HTSeq–a Python framework to work with high-throughput sequencing data. *Bioinformatics*, 31(2):166–169, Jan 2015. URL https://doi.org/10.1093/bioinformatics/btu638.

[5] Aaron R Quinlan and Ira M Hall. BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*, 26(6):841–842, 2010. URL https://doi.org/10.1093/bioinformatics/btq033.

[6] Daniela Witten, Robert Tibshirani, Sam Guoping Gu, Andrew Fire, and Weng-Onn Lui. Ultra-high throughput sequencing-based small RNA discovery and discrete statistical biomarker analysis in a collection of cervical tumours and matched controls. *BMC Biology*, 8(1):1, 2010. URL https://doi.org/10.1186/1741-7007-8-58.

[7] Michael I Love, Wolfgang Huber, and Simon Anders. Moderated estimation of fold change and dispersion for RNA-seq data with DESeq2. *Genome Biology*, 15(12):1, 2014. URL https://doi.org/10.1186/s13059-014-0550-8.

[8] Gokmen Zararsiz, Dincer Goksuluk, Bernd Klaus, Selcuk Korkmaz, Vahap Eldem, Erdem Karabulut, and Ahmet Ozturk. voomDDA: discovery of diagnostic biomarkers and classification of RNA-seq data. *PeerJ*, 5:e3890, 2017. URL https://doi.org/10.7717/peerj.3890.

[9] Daniela M. Witten. Classification and clustering of sequencing data using a poisson model. *The Annals of Applied Statistics*, 5(4):2493–2518, 2011. URL https://doi.org/10.1214/11-AOAS493.

[10] Kai Dong, Hongyu Zhao, Tiejun Tong, and Xiang Wan. NBLDA: negative binomial linear discriminant analysis for RNA-seq data. *BMC Bioinformatics*, 17(1):369, Sep 2016. URL https://doi.org/10.1186/s12859-016-1208-1.

[11] Mark D Robinson and Alicia Oshlack. A scaling normalization method for differential expression analysis of RNA-seq data. *Genome Biology*, 11(3):1, 2010. URL https://doi.org/10.1186/gb-2010-11-3-r25.

[12] Simon Anders and Wolfgang Huber. Differential expression analysis for sequence count data. *Genome Biol*, 11(10):R106, 2010. URL https://doi.org/10.1186/gb-2010-11-10-r106.

[13] Charity W Law, Yunshun Chen, Wei Shi, and Gordon K Smyth. Voom: precision weights unlock linear model analysis tools for RNA-seq read counts. *Genome Biology*, 15(2):1, 2014. URL https://doi.org/10.1186/gb-2014-15-2-r29.

[14] Robert Tibshirani, Trevor Hastie, Balasubramanian Narasimhan, and Gilbert Chu. Class prediction by nearest shrunken centroids, with applications to DNA microarrays. *Statistical Science*, 18(1):104–117, 2003. URL https://doi.org/10.1214/ss/1056397488.

[15] Robert Tibshirani, Trevor Hastie, Balasubramanian Narasimhan, and Gilbert Chu. Diagnosis of multiple cancer types by shrunken centroids of gene expression. *Proc Natl Acad Sci U S A*, 99(10):6567–72, May 2002. URL https://doi.org/10.1073/pnas.082099299.

[16] Sandrine Dudoit, Jane Fridlyand, and Terence P. Speed. Comparison of discrimination methods for the classification of tumors using gene expression data. *Journal of the American Statistical Association*, 97(457):77–87, 2002. URL https://doi.org/10.1198/016214502753479248.

[17] Gokmen Zararsiz, Dincer Goksuluk, Selcuk Korkmaz, Vahap Eldem, Gozde Erturk Zararsiz, Izzet Parug Duru, and Ahmet Ozturk. A comprehensive simulation study on classification of RNA-seq data. *PLoS One*, 12(8):e0182507, 2017. URL https://doi.org/10.1371/journal.pone.0182507.