Introducing biClassify

Alexander F. Lapanowski and Irina Gaynanova

Contents

Introduction

`biClassify` is a package for adapting Linear Discriminant Analysis (LDA), Quadratic Discriminant Analysis (QDA), and Kernel Discriminant Analysis to a variety of situations where the conventional methods may not work. In particular, this package has methodology for the following problems:

1. Linear and Quadratic classification in the large-sample case with small-to-medium sized number of features. The available compressed LDA and QDA methods provide alternatives to random sub-sampling which are shown to produce lower mean misclassification error rates and lower standard error in the misclassification error rates [Lapanowski and Gaynanova] (preprint, 2020).
2. Kernel classification where the data has a medium-to-large number of features. In this case, one would like to learn a non-linear decision boundary and have simultaneous sparse feature selection. The sparse kernel discriminant analysis method provided, Sparse Kernel Optimal Scoring, is presented in Lapanowski and Gaynanova, 2019.

Quick Start

The purpose of this section is to give the user a quick overview of the package and the types of problems it can be used to solve. Accordingly, we implement only the basic versions of the available methods, and more detailed presentations are given in later sections.

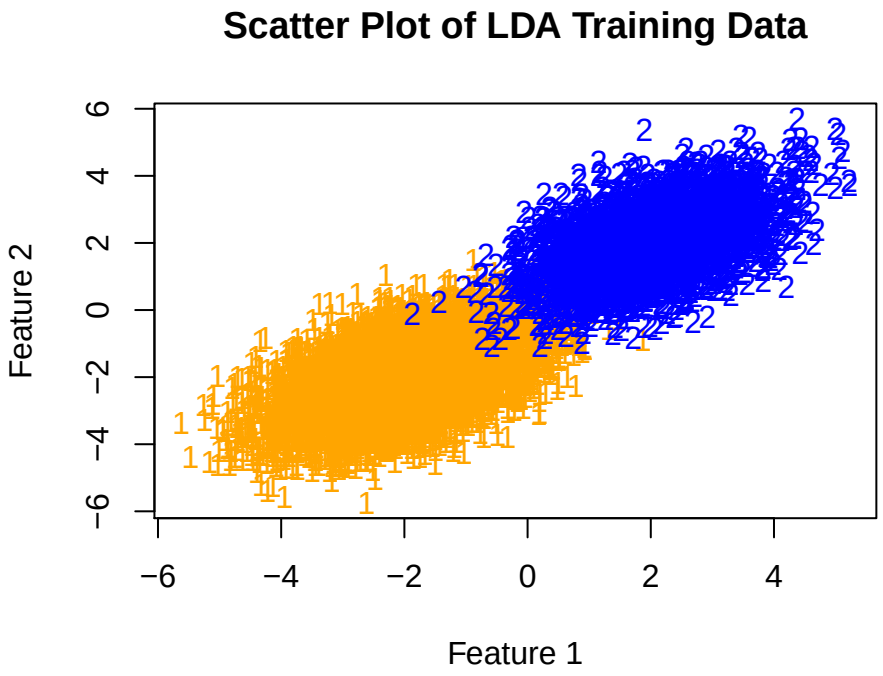
We first load the package

```
library(biClassify)
```

Quick LDA Example

Our first example illustrates the compressed LDA function on data well-suited for LDA. The first two features of the training data in \texttt{LDA_Data} are plotted below:

```
data(LDA_Data)
```



This data set has $n = 10,000$ training samples with $p = 10$ features. It is normally distributed, and the two classes have equal covariance matrices. The test data was independently generated from the same distribution, but it has only $n = 1,000$ samples.

Let us use compressed LDA to predict the test data labels.

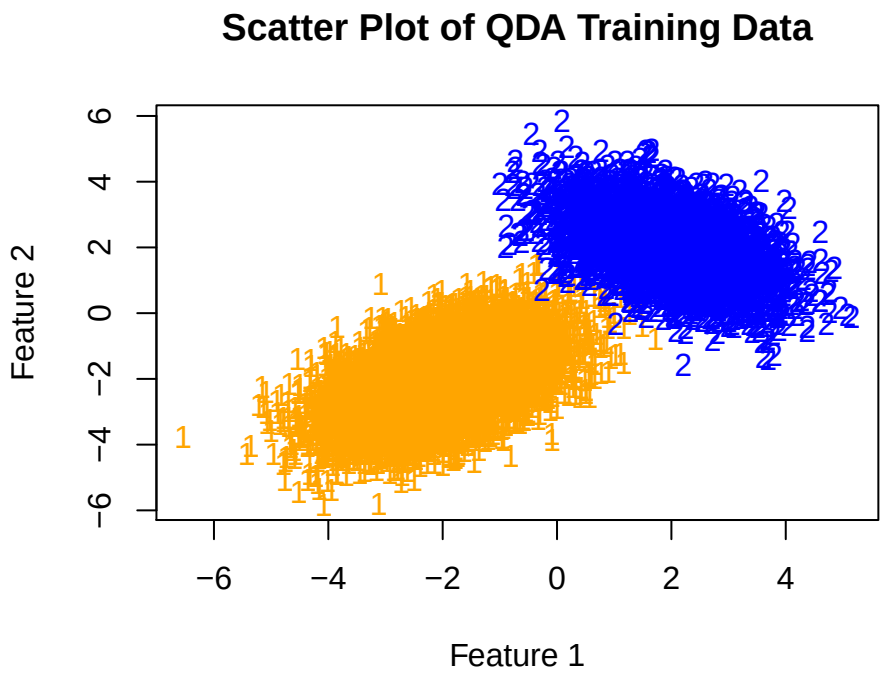
```
test_pred <- lda(TrainData = LDA_Data$TrainData,
                 TrainCat = LDA_Data$TrainCat,
                 TestData = LDA_Data$TestData,
                 Method = "Compressed")
mean(test_pred != LDA_Data$TestCat)
#> [1] 0
```

The automatic impementation of compressed LDA predicted the Test labels perfectly! However, this is due, in part, to the classes being well-separated and having the same covariance structure. Let us now consider an example of where LDA will not perform well.

Quick QDA Example

Our next example illustrates the compressed QDA function on data well-suited for QDA. The first two features of the training data in \texttt{QDA_Data} are plotted below:

```
data(QDA_Data)
```



A modification of Quadratic Discriminant Analysis is well-suited to such data. The package comes with a function `qda` for such purposes.

```
test_pred <- qda(TrainData = QDA_Data$TrainData,
                 TrainCat = QDA_Data$TrainCat,
                 TestData = QDA_Data$TestData,
                 Method = "Compressed")
mean(test_pred != QDA_Data$TestCat)
#> [1] 0
```

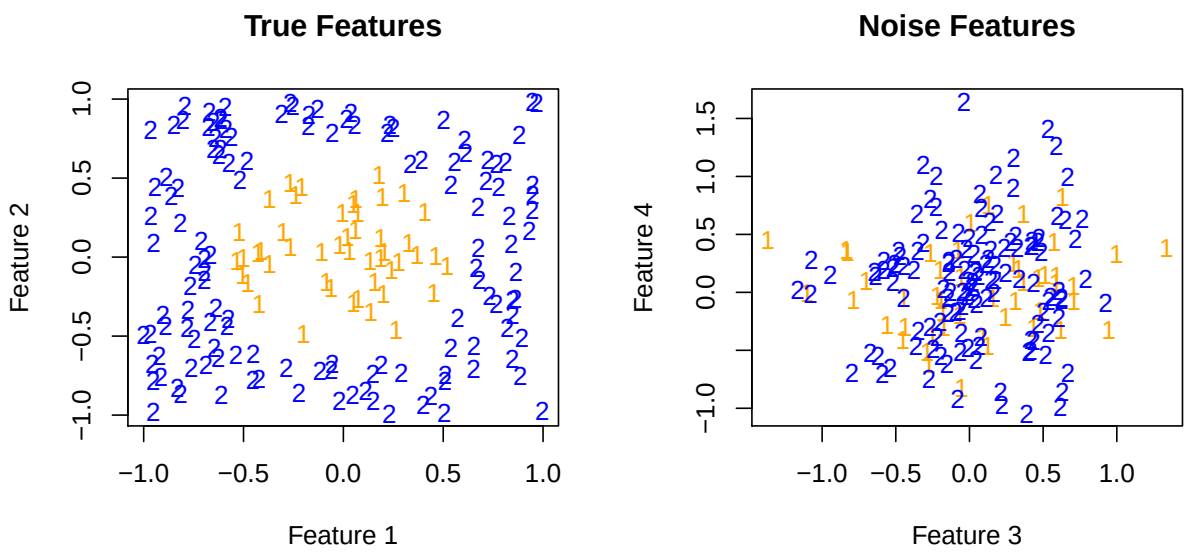
Compressed QDA gives perfect class prediction

Quick Sparse Kernel Optimal Scoring Example

What happens if the data is not well-suited to either Linear or Quadratic Discriminant Analysis? Moreover, what happens if, in addtion to a non-linear decision boundary between classes, there also appear to be variables which do not contribute to group separation?

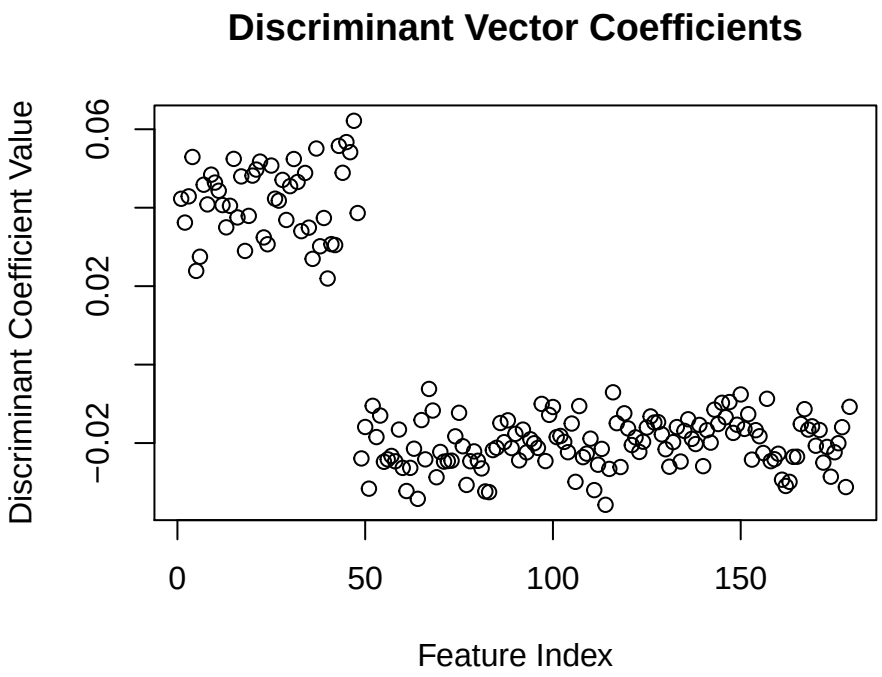
For example, consider the \texttt{KOS_Data} shown below.

```
data(KOS_Data)
```



For this data set, neither lda or qda would suffice. The function `KOS` is the sparse kernel optimal scoring algorithm presented in Lapanowski, and Gaynanova (2019). It is particularly well-suited to such problems, as can be seen from the following.

```
output <- KOS(TrainData = KOS_Data$TrainData,
              TrainCat = KOS_Data$TrainCat,
              TestData = KOS_Data$TestData)
print(output$Weight)
#> [1] 1 1 0 0
mean(output$Predictions != KOS_Data$TestCat)
#> [1] 0
plot(output$Dvec,
     main = "Discriminant Vector Coefficients",
     xlab = "Feature Index",
     ylab="Discriminant Coefficient Value")
```



`Weight` in the output is how much weight the kernel classifier gives to each feature. The weight values lie in $[-1, 1]$, and zero weight means that the feature does not contribute to computing the discriminant function. The KOS function correctly identifies that the first two features are important for class separation, and gives them full weight. It also correctly identifies Features 3 and 4 as being "noise", and it gives them zero weight.

`Predictions` are the predicted class labels for the test data. As we can see, `KOS` has perfect classification.

`Dvec` are the coefficients of the kernel classifier.

Compressed Linear Discriminant Analysis

This section provides a more in-depth treatment to the Linear Discriminant methods available in `biClassify`.

There are five seperate linear discriminant methods avilable through the `lda` wrapper function:

1. `Full` Linear Discriminant Analsysis, which is LDA trained on the full data [Cite Mardia or Elements of Stat. Learning].
2. `Compressed` Linear Discriminant Analysis [Lapanowski, Gaynanova] (2020, preprint).
3. `Projected` LDA [Lapanowski, Gaynanova] (2020, preprint)
4. `Subsampled` LDA, where LDA is trained on data which is sub-sampled uniformly from both classes.
5. `FashRandomFisher` Discriminant Analysis as in [Ye, et. al.] (2017).

The individual methods are invoked by setting the `Method` argument. Let us first load the data for notational convenience.

```
TrainData <- LDA_Data$TrainData
TrainCat <- LDA_Data$TrainCat
TestData <- LDA_Data$TestData
TestCat <- LDA_Data$TestCat
```

Full LDA

This method is the result of setting `Method` equal to `"Full"`. This method is traditional Linear Discriminant Analysis, as presented in [Citation]. No additional parameters need to be supplied, and the code will run as stated.

```
test_pred <- lda(TrainData, TrainCat, TestData)
table(test_pred)
#> test_pred
#>   1   2
#> 700 300
mean(test_pred != TestCat)
#> [1] 0
```

which produces a list containing a vector of predicted class labels for `TestData` and the discriminant vector used in LDA.

Compressed LDA

Compressed LDA seeks to solve the LDA problem on reduced-size data. It first compressed the groups of centered data $(X^g - \overline{X}_g)$ via a compression matrix Q^g. The entries $Q^g_{i,j}$ are i.i.d. sparse radamacher random variables with distribution

$$\mathbb{P}(Q^g_{i,j} = 1) = \mathbb{P}(Q^g_{i,j} = -1) = \frac{p}{2} \text{ and } \mathbb{P}(Q^g_{i,j} = 0) = 1 - p.$$

This method is the result of setting `Method` equal to `"Compressed"`. It is compressed LDA, as presented in Lapanowski, Gaynanova (2020), preprint. Compressed LDA reduces the group sample amounts from n_1 and n_2 to m_1 and m_2 respectively.

Compressed LDA requires the parameters `m1`, `m2`, `s`.

The easiest way to run Compressed LDA is to set `Mode` to `Automatic` and not worry about supplying additional parameters.

```
test_pred <- lda(TrainData, TrainCat, TestData, Method = "Compressed", Mode = "Automatic")
table(test_pred)
#> test_pred
#>   1   2
#> 700 300
mean(test_pred != TestCat)
#> [1] 0
```

`Automatic` is the default value for `Mode`, and so one could simply run

```
test_pred <- lda(TrainData, TrainCat, TestData, Method = "Compressed")
table(test_pred)
#> test_pred
#>   1   2
#> 700 300
mean(test_pred != TestCat)
#> [1] 0
```

and obtain the same output.

When `Mode` is set to `Interactive`, prompts will appear asking for the compression amounts m_1, m_2, and sparsity level s to be used in compression. The user will type in the amounts:

```
output <- lda(TrainData, TrainCat, TestData, Method = "Compressed", Mode = "Interactive")
"Please enter the number m1 of group 1 compression samples: "700
"Please enter the number m2 of group 2 compression samples: "300
"Please enter sparsity level s used in compression: "0.01
```

and the output is produced.

If the user is interested in running simulation studies or has mastery over the functionality, they may wish to give the `lda` function all parameters.

```
test_pred <- lda(TrainData, TrainCat, TestData, Method = "Compressed",
                 Mode = "Research", m1 = 700, m2 = 300, s = 0.01)

table(test_pred)
#> test_pred
#>   1   2
#> 700 300
mean(test_pred != TestCat)
#> [1] 0
```

WARNING: The argument `Mode` will override any supplied parameters if its value is `Automatic` or `Research`.

Sub-Sampled LDA

Sub-sampled LDA is just LDA trained on data sub-sampled uniformly from both classes.

To run sub-sampled LDA, set `Method` equal to `Subsampled`. It requires the additional parameters `m1` and `m2`.

The easiest way to run Compressed LDA is to set `Mode` to `Automatic` and not worry about supplying additional parameters.

```
test_pred <- lda(TrainData, TrainCat, TestData, Method = "Subsampled", Mode = "Automatic")
table(test_pred)
#> test_pred
#>   1   2
#> 700 300
```

`Automatic` is the default value for `Mode`, and so one could simply run

```
test_pred <- lda(TrainData, TrainCat, TestData, Method = "Subsampled")
table(test_pred)
#> test_pred
#>   1   2
#> 700 300
```

and obtain the same output.

When `Mode` is set to `Interactive`, prompts will appear asking for the sub-sample amounts m_1, m_2 for each group to be used. The user will type in the amounts:

```
test_pred <- lda(TrainData, TrainCat, TestData, Method = "Subsampled", Mode = "Interactive")
"Please enter the number m1 of group 1 sub-samples: "700
"Please enter the number m2 of group 2 sub-samples: "300
```

and the output is produced.

If the user is interested in running simulation studies or has mastery over the functionality, they may wish to give the `lda` function all parameters.

```
output <- lda(TrainData, TrainCat, TestData, Method = "Subsampled",
              Mode = "Research", m1 = 700, m2 = 300)

table(output)
#> output
#>   1   2
#> 700 300
mean(output != TestCat)
#> [1] 0
```

WARNING: The argument `Mode` will override any supplied parameters if its value is `Automatic` or `Research`.

Projected LDA

This method is the result of setting `Method` equal to `"PRojected"`. It is Projected LDA, as presented in Lapanowski, Gaynanova (2020), preprint. Projected LDA creates the discriminant vector on compressed data and then projects the full training data onto the discriminant vector.

Projected LDA requires the parameters `m1`, `m2`, `s`.

The easiest way to run Projected LDA is to set `Mode` to `Automatic` and not worry about supplying additional parameters.

```
output <- lda(TrainData, TrainCat, TestData, Method = "Projected", Mode = "Automatic")
table(output)
#> output
#>   1   2
#> 700 300
mean(output != TestCat)
#> [1] 0
```

`Automatic` is the default value for `Mode`, and so one could simply run

```
output <- lda(TrainData, TrainCat, TestData, Method = "Projected")
table(output)
#> output
#>   1   2
#> 700 300
mean(output != TestCat)
#> [1] 0
```

and obtain the same output.

When `Mode` is set to `Interactive`, prompts will appear asking for the compression amounts m_1, m_2, and sparsity level s to be used in compression. The user will type in the amounts:

```
output <- lda(TrainData, TrainCat, TestData, Method = "Projected", Mode = "Interactive")
"Please enter the number m1 of group 1 compression samples: "700
"Please enter the number m2 of group 2 compression samples: "300
"Please enter sparsity level s used in compression: "0.01
```

and the output is produced.

If the user is interested in running simulation studies or has mastery over the functionality, they may wish to give the `lda` function all parameters.

```
test_pred <- lda(TrainData, TrainCat, TestData, Method = "Projected",
                 Mode = "Research", m1 = 700, m2 = 300, s = 0.01)

table(test_pred)
#> test_pred
#>   1   2
#> 700 300
mean(output != TestCat)
#> [1] 0
```

WARNING: The argument `Mode` will override any supplied parameters if its value is `Automatic` or `Research`.

Fast Random Fisher Discriminant Analysis

This method is the result of setting `Method` equal to `"fastRandomFisher"`. It is the Fast Random Fisher Discriminant Analysis algorithm, as presented in Ye, et. al. (2017). Fast Random fisher creates the discriminant vector on reduced sample amounts m, and then projects the full training data onto the learned discriminant vector. The difference between Fast Random Fisher Discriminant Analysis and Projected LDA is that Fast Random Fisher mixes the groups together when forming the discriminant vector, but Projected LDA does not.

Fast Random Fisher requires the parameters `m`, and `s`.

The easiest way to run Fast Random Fisher is to set `Mode` to `Automatic` and not worry about supplying additional parameters.

```
test_pred <- lda(TrainData, TrainCat, TestData, Method = "fastRandomFisher", Mode = "Automatic")
table(test_pred)
#> test_pred
#>   1   2
#> 700 300
mean(test_pred != TestCat)
#> [1] 0
```

`Automatic` is the default value for `Mode`, and so one could simply run

```
test_pred <- lda(TrainData, TrainCat, TestData, Method = "fastRandomFisher")
table(test_pred)
#> test_pred
#>   1   2
#> 700 300
mean(test_pred != TestCat)
#> [1] 0
```

and obtain the same output.

When `Mode` is set to `Interactive`, prompts will appear asking for the total amount of compressed samples m and sparsity level s to be used in compression. The user will type in the amounts:

```
output <- lda(TrainData, TrainCat, TestData, Method = "fastRandomFisher", Mode = "Interactive")
"Please enter the number m of total compressed samples: "1000
"Please enter sparsity level s used in compression: "0.01
```

and the output is produced.

If the user is interested in running simulation studies or has mastery over the functionality, they may wish to give the `lda` function all parameters.

```
test_pred <- lda(TrainData, TrainCat, TestData, Method = "fastRandomFisher",
                 Mode = "Research", m = 1000, s = 0.01)

table(test_pred)
#> test_pred
#>   1   2
#> 700 300
mean(test_pred != TestCat)
#> [1] 0
```

WARNING: The argument `Mode` will override any supplied parameters if its value is `Automatic` or `Research`.

Quadratic Discriminant Analysis

This section provides a more in-depth treatment to the Linear Discriminant methods available in `biClassify`.

There are three seperate quadratic discriminant methods avilable through the `qda` wrapper function:

1. `Full` Quadratic Discriminant Analsysis, which is QDA trained on the full data [Cite Mardia or Elements of Stat. Learning].
2. `Compressed` Qinear Discriminant Analysis [Lapanowski, Gaynanova] (2020, preprint).
3. `Subsampled` QDA, where QDA is trained on data which is sub-sampled uniformly from both classes.

The individual methods are invoked by setting the `Method` argument. Let us first load the data for notational convenience.

```
TrainData <- QDA_Data$TrainData
TrainCat <- QDA_Data$TrainCat
TestData <- QDA_Data$TestData
TestCat <- QDA_Data$TestCat
```

Full QDA

This method is the result of setting `Method` equal to `"Full"`. This method is traditional Quadratic Discriminant Analysis, as presented in [Citation]. No additional parameters need to be supplied, and the code will run as stated. Unlike the `lda` function, only the class predictions are produced:

```
Predictions <- qda(TrainData, TrainCat, TestData, Method = "Full")
table(Predictions)
#> Predictions
#>   1   2
#> 700 300
```

Compressed QDA

This method is the result of setting `Method` equal to `"Compressed"`. It is compressed QDA, as presented in Lapanowski, Gaynanova (2020), preprint. Compressed QDA reduces the group sample amounts from n_1 and n_2 to m_1 and m_2 respectively via compression and trains QDA on the reduced samples.

Compressed QDA requires the parameters `m1`, `m2`, `s`.

The easiest way to run Compressed QDA is to set `Mode` to `Automatic` and not worry about supplying additional parameters.

```
output <- qda(TrainData, TrainCat, TestData, Method = "Compressed", Mode = "Automatic")
table(output)
#> output
#>   1   2
#> 700 300
```

`Automatic` is the default value for `Mode`, and so one could simply run

```
output <- qda(TrainData, TrainCat, TestData, Method = "Compressed")
table(output)
#> output
#>   1   2
#> 700 300
```

and obtain the same output.

When `Mode` is set to `Interactive`, prompts will appear asking for the compression amounts m_1, m_2, and sparsity level s to be used in compression. The user will type in the amounts:

```
output <- qda(TrainData, TrainCat, TestData, Method = "Compressed", Mode = "Interactive")
"Please enter the number m1 of group 1 compression samples: "700
"Please enter the number m2 of group 2 compression samples: "300
"Please enter sparsity level s used in compression: "0.01

table(output)
```

and the output is produced.

If the user is interested in running simulation studies or has mastery over the functionality, they may wish to give the `qda` function all parameters.

```
output <- qda(TrainData, TrainCat, TestData, Method = "Compressed",
              Mode = "Research", m1 = 700, m2 = 300, s = 0.01)

summary(output)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>     1.0     1.0     1.0     1.3     2.0     2.0
```

Sub-Sampled QDA

Sub-sampled QDA is just QDA trained on data sub-sampled uniformly from both classes. To run sub-sampled QDA, set `Method` equal to `Subsampled`.

It requires the additional parameters `m1` and `m2`.

The easiest way to run sub-sampled QDA is to set `Mode` to `Automatic` and not worry about supplying additional parameters.

```
output <- qda(TrainData, TrainCat, TestData, Method = "Subsampled", Mode = "Automatic")
table(output)
#> output
#>   1   2
#> 700 300
```

`Automatic` is the default value for `Mode`, and so one could simply run

```
output <- qda(TrainData, TrainCat, TestData, Method = "Subsampled")
summary(output)
```

```
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>     1.0     1.0     1.0     1.3     2.0     2.0
```

and obtain the same output.

When `Mode` is set to `Interactive`, prompts will appear asking for the sub-sample amounts m_1, m_2 for each group to be used. The user will type in the amounts:

```
output <- qda(TrainData, TrainCat, TestData, Method = "Subsampled", Mode = "Interactive")
"Please enter the number m1 of group 1 sub-samples: "700
"Please enter the number m2 of group 2 sub-samples: "300

summary(output)
```

and the output is produced.

If the user is interested in running simulation studies or has mastery over the functionality, they may wish to give the `qda` function all parameters.

```
output <- qda(TrainData, TrainCat, TestData, Method = "Subsampled",
              Mode = "Research", m1 = 700, m2 = 300)

summary(output)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>     1.0     1.0     1.0     1.3     2.0     2.0
```

WARNING: The argument `Mode` will override any supplied parameters if its value is `Automatic` or `Research`.

Sparse Kernel Discriminant Analysis

This section presents the kernel optimal scoring method available in the `biClassify` package. Kernel optimal scoring is presented in [Lapanowski and Gaynanova] Artificial Intelligence and Statistics, 2019.

Kernel optimal scoring finds the kernel discriminant coefficients $\alpha \in \mathbb{R}^n$ by solving a kernelized form of the optimal scoring problem

$$\min_{f \in \mathcal{H}} \left\{ \frac{1}{n} \sum_{i=1}^{n} |y_i \widehat{\theta} - \left\langle \Phi(x_i) - \overline{\Phi}\,,\, f \right\rangle_{\mathcal{H}}|^2 + \gamma \|f\|_{\mathcal{H}}^2 \right\} = \min_{\alpha \in \mathbb{R}^n} \left\{ \frac{1}{n} \|Y\widehat{\theta} - C\mathbf{K}C\alpha\|_2^2 + \gamma \alpha^\top \mathbf{K} \alpha \right\}$$

It is equivalent to kernel discriminant analysis.

We include simultaneous sparse feature selection by weighting the features using $w \in [-1, 1]^n$, so that the weighted samples are

$$wx = (w_1 x_1, \ldots, w_p x_p)^\top.$$

The weighted kernel matrix $\mathbf{K}_w$ is defined by $(\mathbf{K}_w)_{i,j} := k(wx_i, wx_j)$. To perform sparse feature selection, we add a sparsity penalty on the weight vector $\lambda \|w\|_1$ and minimize

$$\min_{\alpha \in \mathbb{R}^n\,,\, w \in [-1,1]^p} \left\{ \frac{1}{n} \|Y\widehat{\theta} - C\mathbf{K}_w C\alpha\|_2^2 + \lambda \|w\|_1 + \gamma \alpha^\top \mathbf{K}_w \alpha \right\}.$$

Let us load the data set used in kernel optimal scoring

```
TrainData <- KOS_Data$TrainData
TrainCat <- KOS_Data$TrainCat
TestData <- KOS_Data$TestData
TestCat <- KOS_Data$TestCat
```

Parameter Selection

This subsection details how `KOS` selects the parameters σ^2, γ, and λ.

To select the gaussian kernel parameter σ^2,is selected based on the $\{.05, .1, .2, .3, .5\}$ quantiles of the set of squared distances between the classes

$$\{\|x_{i_1} - x_{i_2}\|_2^2 : x_{i_1} \in C_1,\, x_{i_2} \in C_2\}.$$

The ridge parameter γ is selected by adapting a kernel matrix shrinkage technique [Lancewicki, 2017] to the setting of ridge regression. For more details, see [Lapanowski and Gaynanova, 2019].

The sparsity parameter λ is selected using 5-fold cross-validation to minimize the error rate over a grid of 20 equally-spaced values.

The function `SelectParams` implements these methods automatically. For more details, see [Lapanowski and Gaynanova, 2019].

```
SelectParams(TrainData, TrainCat)
#> $Sigma
#> [1] 0.7390306
#>
#> $Gamma
#> [1] 0.137591
#>
#> $Lambda
#> [1] 0.02902946
```

If parameters are not supplied to `KOS`, the function first invokes `SelectParams` to generate any missing parameters.

Hierarchical Parameters

Sparse kernel optimal scoring has three parameters: a Gaussian kernel parameter `Sigma`, a ridge parameter `Gamma`, and a sparsity parameter `Lambda`. They have a hierarchical dependency, in that `Sigma` influences `Gamma`, and both influence `Lambda`. The ordering is

Top `Sigma`

Middle `Gamma`

Bottom `Lambda`

When using either of the functions, the user is only allowed to specify parameter combinations which adhere to the hierarchical ordering above. That is, they can only input parameters which go from Top to Bottom. For example, they could specify both `Sigma` and `Gamma`, but leave `Lambda` as the default `NULL` value. On the other hand, the user would not be allowed to specify only `Lambda` while leaving `Sigma` and `Gamma` as their default `NULL` values.

```
SelectParams(TrainData, TrainCat, Sigma = 1, Gamma = 0.1)
#> $Sigma
#> [1] 1
#>
#> $Gamma
#> [1] 0.1
#>
#> $Lambda
#> [1] 0.01078724
```

If the user supplies parameter values which violate the hierarchical ordering, the error message `Hierarchical order of parameters violated.` will be returned.

```
SelectParams(TrainData, TrainCat, Gamma = 0.1)
#> Error in SelectParams(TrainData, TrainCat, Gamma = 0.1): Hierarchical order of parameters violated.
```

KOS

This package comes with an all-purpose function for running kernel optimal scoring.

```
Sigma <- 1.325386
Gamma <- 0.07531579
Lambda <- 0.002855275

output <- KOS(TestData, TrainData, TrainCat, Sigma = Sigma,
              Gamma = Gamma, Lambda = Lambda)
print(output$Weight)
#> [1] 1 1 0 0
table(output$Predictions)
#>
#>  1  2
#> 26 68
summary(output$Dvec)
#>        V1
#>  Min.   :-0.06434
#>  1st Qu.:-0.04449
#>  Median :-0.02783
#>  Mean   :-0.00616
#>  3rd Qu.: 0.04377
#>  Max.   : 0.10005
```