Classification models

Tanay

2024-12-09

I am using the Khan dataset from the ISLR2 Package for our project. Here is the basic structure of the project:

1. Loading the necessary libraries
2. Learning about the dataset (and some exploratory analysis)
3. Analysis
4. Summary

1. Loading the necessary libraries

```
options(repos = c(CRAN = "https://cran.rstudio.com"))
library(ISLR2)
library(e1071)
library(rpart)
library(randomForest)
library(ipred)
library(class)
library(MASS)
library(glmnet)
library(pROC)
library(caret)
library(rpart.plot)
```

2. Learning about the data-set

```
#?Khan
#names(Khan)
summary(Khan)
```

```
##        Length Class  Mode
## xtrain 145404 -none- numeric
## xtest   46160 -none- numeric
## ytrain     63 -none- numeric
## ytest      20 -none- numeric
```

```
table(Khan$ytrain)
```

```
##
##  1  2  3  4
##  8 23 12 20
```

```
table(Khan$ytest)
```

```
##
## 1 2 3 4
## 3 6 6 5
```

The Khan dataset contains gene expression measurements for 2308 genes across tissue samples from 63 subjects with four types of small round blue cell tumors, used for tumor classification models.

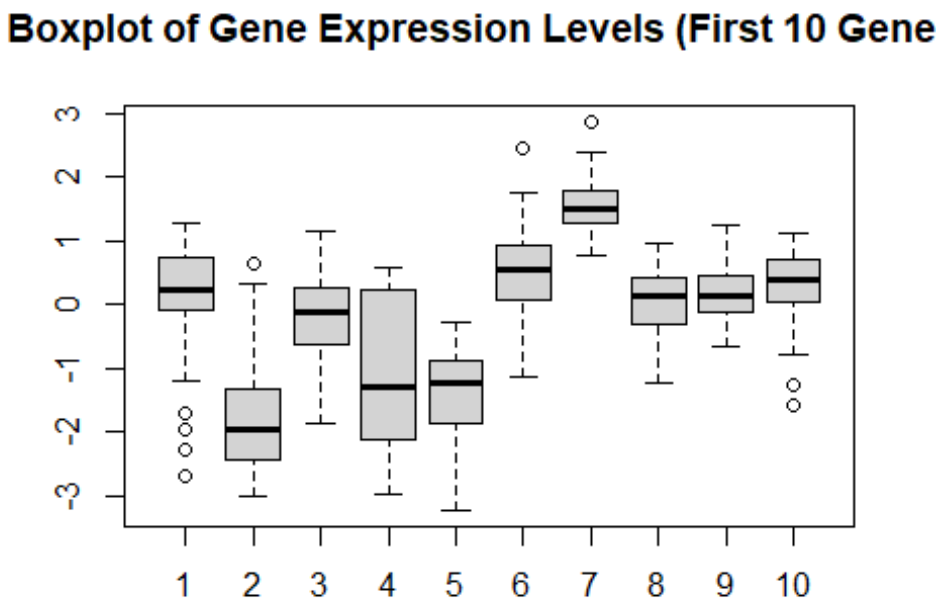
```
anyNA(Khan$xtrain)
```

```
## [1] FALSE
```

```
anyNA(Khan$xtest)
```

```
## [1] FALSE
```

```
boxplot(Khan$xtrain[, 1:10], main = "Boxplot of Gene Expression Levels (First
10 Genes)")
```




```
summary(Khan$xtrain[, 1:10])
```

```
##        V1                  V2                  V3                  V4
##  Min.   :-2.68385   Min.   :-3.0078   Min.   :-1.8515   Min.   :-2.9565
##  1st Qu.:-0.08132   1st Qu.:-2.4271   1st Qu.:-0.6342   1st Qu.:-2.1215
##  Median : 0.24420   Median :-1.9498   Median :-0.1136   Median :-1.2744
##  Mean   : 0.14693   Mean   :-1.7390   Mean   :-0.2487   Mean   :-1.0781
##  3rd Qu.: 0.73539   3rd Qu.:-1.3187   3rd Qu.: 0.2530   3rd Qu.: 0.2355
##  Max.   : 1.28551   Max.   : 0.6548   Max.   : 1.1607   Max.   : 0.5838
```

```
##        V5                  V6                  V7                 V8          
##  Min.   :-3.2164   Min.   :-1.11810   Min.   :0.7761   Min.   :-1.21807  
##  1st Qu.:-1.8602   1st Qu.: 0.08685   1st Qu.:1.2884   1st Qu.:-0.32172  
##  Median :-1.2117   Median : 0.54227   Median :1.5102   Median : 0.13732  
##  Mean   :-1.3857   Mean   : 0.51729   Mean   :1.5522   Mean   : 0.09513  
##  3rd Qu.:-0.8824   3rd Qu.: 0.94408   3rd Qu.:1.7915   3rd Qu.: 0.43787  
##  Max.   :-0.2647   Max.   : 2.45273   Max.   :2.8641   Max.   : 0.95663  
##        V9                 V10          
##  Min.   :-0.6392   Min.   :-1.57214  
##  1st Qu.:-0.1236   1st Qu.: 0.05232  
##  Median : 0.1336   Median : 0.38655  
##  Mean   : 0.1620   Mean   : 0.34515  
##  3rd Qu.: 0.4557   3rd Qu.: 0.69340  
##  Max.   : 1.2558   Max.   : 1.12249
```

There are potential outliers in the data, as seen from the boxplot. In gene expression data, outliers may represent biologically significant phenomena, such as rare mutations or unique gene activations, and removing them could lose valuable information. Moreover, many modern algorithms, such as regularized models, are robust to outliers, reducing the need for explicit handling.

Additionally, since the Khan dataset has a predefined train-test split, altering the data could disrupt the representativeness of the test set. Retaining outliers preserves the dataset's integrity .

Segregating the data-set

```
xtrain = Khan$xtrain
xtest = Khan$xtest
ytrain = Khan$ytrain
ytest = Khan$ytest
```

3. Data Analysis

Firstly, use Principal Component Analysis (PCA) for dimension reduction.

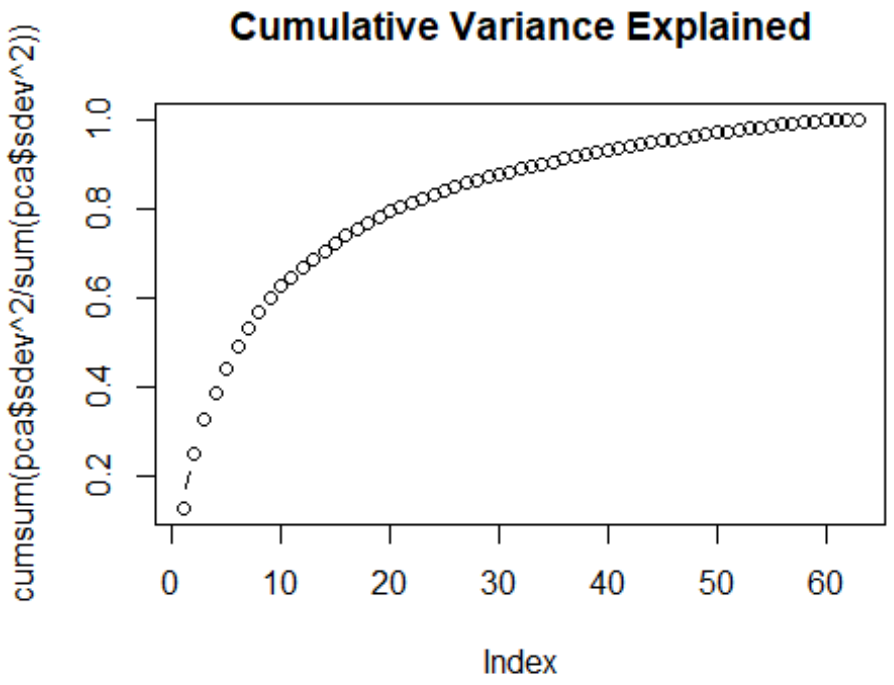
```
pca = prcomp(xtrain, scale. = T)
summary(pca)
```

```
## Importance of components:
##                            PC1     PC2      PC3      PC4      PC5      PC6
## Standard deviation     17.256 16.8607 13.27696 11.64472 11.14247 10.76815
## Proportion of Variance  0.129  0.1232  0.07638  0.05875  0.05379  0.05024
## Cumulative Proportion   0.129  0.2522  0.32856  0.38731  0.44111  0.49134
##                            PC7     PC8     PC9    PC10    PC11    PC12
PC13
## Standard deviation     9.86418 9.00332 8.38429 7.87027 7.11430 6.91896
6.64849
## Proportion of Variance 0.04216 0.03512 0.03046 0.02684 0.02193 0.02074
0.01915
```

```
## Cumulative Proportion  0.53350 0.56862 0.59908 0.62592 0.64785 0.66859
0.68774
##                            PC14   PC15    PC16    PC17    PC18    PC19
PC20
## Standard deviation     6.54870 6.3366 6.15465 6.00711 5.63487 5.25819
5.21728
## Proportion of Variance 0.01858 0.0174 0.01641 0.01563 0.01376 0.01198
0.01179
## Cumulative Proportion  0.70632 0.7237 0.74013 0.75577 0.76953 0.78151
0.79330
##                            PC21    PC22    PC23    PC24    PC25    PC26
PC27
## Standard deviation     5.07561 4.80838 4.61082 4.48687 4.44156 4.37914
4.16441
## Proportion of Variance 0.01116 0.01002 0.00921 0.00872 0.00855 0.00831
0.00751
## Cumulative Proportion  0.80446 0.81448 0.82369 0.83241 0.84096 0.84927
0.85678
##                            PC28   PC29    PC30    PC31   PC32    PC33
PC34
## Standard deviation     4.00187 3.9624 3.92077 3.79589 3.7520 3.65065
3.61299
## Proportion of Variance 0.00694 0.0068 0.00666 0.00624 0.0061 0.00577
0.00566
## Cumulative Proportion  0.86372 0.8705 0.87719 0.88343 0.8895 0.89530
0.90096
##                            PC35    PC36    PC37    PC38    PC39    PC40
PC41
## Standard deviation     3.55415 3.50073 3.41091 3.35278 3.31193 3.27819
3.21950
## Proportion of Variance 0.00547 0.00531 0.00504 0.00487 0.00475 0.00466
0.00449
## Cumulative Proportion  0.90643 0.91174 0.91678 0.92165 0.92640 0.93106
0.93555
##                            PC42    PC43    PC44    PC45    PC46    PC47
PC48
## Standard deviation     3.16274 3.11082 3.09288 3.04167 3.00256 2.94952
2.85728
## Proportion of Variance 0.00433 0.00419 0.00414 0.00401 0.00391 0.00377
0.00354
## Cumulative Proportion  0.93989 0.94408 0.94822 0.95223 0.95614 0.95991
0.96344
##                            PC49    PC50    PC51    PC52    PC53    PC54
PC55
## Standard deviation     2.83613 2.77984 2.74167 2.69686 2.67729 2.66268
2.55836
## Proportion of Variance 0.00349 0.00335 0.00326 0.00315 0.00311 0.00307
0.00284
## Cumulative Proportion  0.96693 0.97028 0.97353 0.97669 0.97979 0.98286
0.98570
```

```
##                            PC56    PC57   PC58    PC59    PC60    PC61
PC62
## Standard deviation     2.50983 2.42584 2.4010 2.34567 2.24771 2.04810
0.55462
## Proportion of Variance 0.00273 0.00255 0.0025 0.00238 0.00219 0.00182
0.00013
## Cumulative Proportion  0.98843 0.99098 0.9935 0.99586 0.99805 0.99987
1.00000
##                             PC63
## Standard deviation     1.042e-14
## Proportion of Variance 0.000e+00
## Cumulative Proportion  1.000e+00
```

```r
plot(cumsum(pca$sdev^2 / sum(pca$sdev^2)), type = "b", main = "Cumulative
Variance Explained")
```




```r
num_components <- which(cumsum(pca$sdev^2 / sum(pca$sdev^2)) > 0.95)[1]
pca_xtrain <- pca$x[, 1:num_components]
pca_xtest <- predict(pca, newdata = xtest)[, 1:num_components]
```

In this case PCA is highly effective for gene clustering due to its ability to handle the high dimensionality typical of gene expression data. By reducing tons of genes to a few principal components that explain most of the variance, PCA simplifies the data while preserving its essential structure. This reduction minimizes noise and improves the signal-to-noise ratio, making clustering algorithms more robust and computationally efficient.Additionally, PCA captures correlations between genes, which is vital for grouping co-regulated or

functionally related genes into clusters. By focusing on the most meaningful patterns in the data, PCA enhances clustering performance and provides biologically relevant insights.

I first try using KNN

```
set.seed(1)
pred = knn(pca_xtrain, pca_xtest, ytrain, k = 3)
table(Predicted = pred, Actual = ytest)
```

```
##           Actual
## Predicted 1 2 3 4
##         1 3 0 0 0
##         2 0 5 1 1
##         3 0 0 3 0
##         4 0 1 2 4
```

```
mean(pred != ytest)
```

```
## [1] 0.25
```

```
pred = knn(pca_xtrain, pca_xtest, ytrain, k = 3)
```

The Knn results are summarized in the above confusion matrix Predicted Class 1 matched all three Actual Class 1,Predicted Class 2 had five correct predictions for Actual Class 2 but misclassified one instance of Actual Class 3 and one of Actual Class 4. Predicted Class 3 accurately identified all three instances of Actual Class 3.Predicted Class 4 had four correct matches for Actual Class 4, while misclassifying one instance of Actual Class 2 and two of Actual Class 3.The overall classification error rate was 0.25, indicating that 25% of the predictions were incorrect.

I then use classification trees.

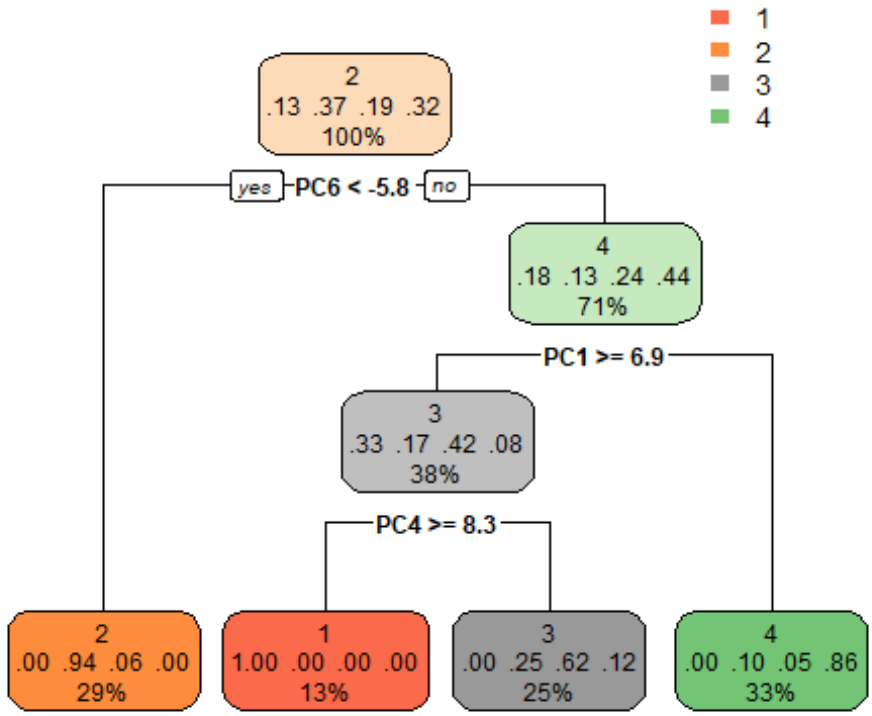
A. Without the ensemble methods:

```
#Here, the data needs to be converted to data.frame
xtrain_df = as.data.frame(pca_xtrain)
xtest_df = as.data.frame(pca_xtest)


model_t = rpart(as.factor(ytrain) ~ ., data = data.frame(xtrain_df, ytrain =
as.factor(ytrain)))


pred1 = predict(model_t, xtest_df, type = "class")


rpart.plot(model_t)
```




```
table(Predicted = pred1, Actual = ytest)
```

```
##           Actual
## Predicted 1 2 3 4
##         1 3 0 0 0
##         2 0 1 0 0
##         3 0 1 2 0
##         4 0 4 4 5
```

```
mean(pred1 != ytest)
```

```
## [1] 0.45
```

The classification tree without ensemble gives us the above confusion matrix: Predicted Class 1 correctly identified all three instances of Actual Class 1. Predicted Class 2 made one correct prediction for Actual Class 2 but failed to classify other instances accurately. Predicted Class 3 correctly identified two instances of Actual Class 3 but misclassified one instance of Actual Class 2. Predicted Class 4 had five correct matches for Actual Class 4 but misclassified four instances of Actual Class 2 and four instances of Actual Class 3. The overall classification error rate was 0.45, indicating that 45% of the predictions were incorrect.

B. Next, I use the ensemble methods.

B.1 Bagging

```
set.seed(2)
model_b = bagging(as.factor(ytrain) ~ ., data = data.frame(xtrain_df,
ytrain))
pred2 = predict(model_b, xtest_df)
table(Predicted = pred2, Actual = ytest)
```

```
##          Actual
## Predicted 1 2 3 4
##         1 3 0 0 0
##         2 0 4 0 0
##         3 0 0 1 0
##         4 0 2 5 5
```

```
mean(pred2 != ytest)
```

```
## [1] 0.35
```

The bagging methods gives the above confusion matrix: Predicted Class 1 correctly identified all three instances of Actual Class 1. Predicted Class 2 made four correct predictions for Actual Class 2, with no misclassifications. Predicted Class 3 accurately classified one instance of Actual Class 3 but did not classify other instances. Predicted Class 4 had five correct matches for Actual Class 4, but it misclassified two instances of Actual Class 2 and five instances of Actual Class 3. The overall classification error rate was 0.35, indicating that 35% of the predictions were incorrect.

B.2 Random forests:

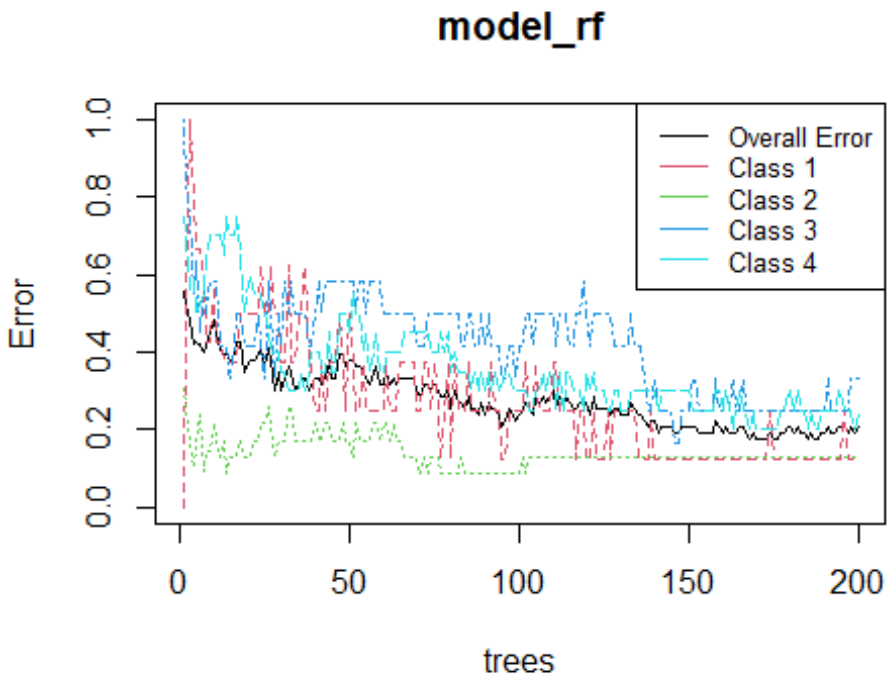
Here, I am setting number of trees to 200.

```
set.seed(4)
model_rf = randomForest(as.factor(ytrain) ~ ., data = data.frame(xtrain_df,
ytrain = as.factor(ytrain)), ntree =200)
pred3 = predict(model_rf, xtest_df)

plot(model_rf)
levels(as.factor(ytrain))
```

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

```
legend("topright", legend = c("Overall Error", "Class 1", "Class 2", "Class
3", "Class 4"),col = c(1, 2, 3, 4, 5), lty = 1, cex = 0.8)
```




```
table(Predicted = pred3, Actual = ytest)
```

```
##           Actual
## Predicted 1 2 3 4
##         1 2 0 0 0
##         2 0 4 1 0
##         3 0 0 0 0
##         4 1 2 5 5
```

```
mean(pred3 != ytest)
```

```
## [1] 0.45
```

The random forest of 200 is as follows: Predicted Class 1 correctly identified two instances of Actual Class 1, while misclassifying one instance as Predicted Class 4. Predicted Class 2 made four correct predictions for Actual Class 2 but misclassified one instance of Actual Class 3. Predicted Class 3 did not correctly classify any instances of Actual Class 3. Predicted Class 4 had five correct matches for Actual Class 4, but it misclassified one instance of Actual Class 1, two instances of Actual Class 2, and five instances of Actual Class 3. The overall classification error rate was 0.45, indicating that 45% of the predictions were incorrect.

Next, I use SVM

```
model_svm = svm(pca_xtrain, as.factor(ytrain), kernel = "linear", probability
=T)
```

```
pred4 = predict(model_svm, pca_xtest)
table(Prediction = pred4, Actual = ytest)
```

```
##           Actual
## Prediction 1 2 3 4
##          1 3 0 0 0
##          2 0 6 0 0
##          3 0 0 6 0
##          4 0 0 0 5
```

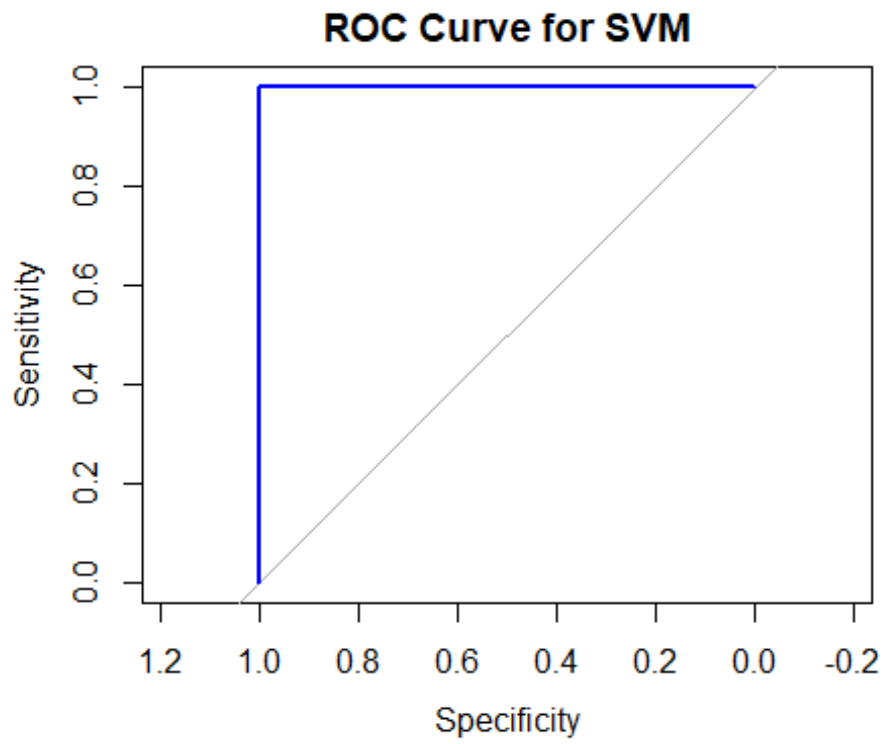
```
mean(pred4 != ytest)
```

```
## [1] 0
```

```
ytest1 <- as.numeric(ytest) - 1
pred_prob = attr(predict(model_svm, pca_xtest, probability = TRUE),
"probabilities")[, 2]
head(pred_prob)
```

```
##         V1         V2         V4         V6         V7         V8
## 0.05300245 0.13348541 0.97639395 0.11022086 0.01560644 0.01460979
```

```
roc1 = roc(ytest, pred_prob)
plot(roc1, main = "ROC Curve for SVM", col="blue", lwd=2)
```



The SVM method predicts all the cases correctly, and give us a 0 test error.

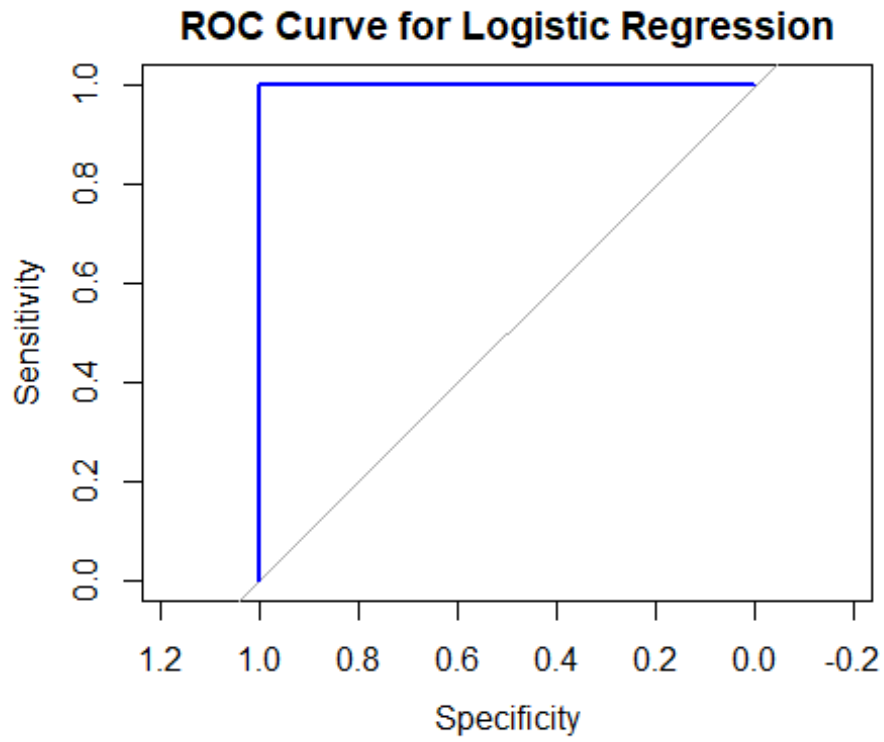
I use logistic regression next

```r
khan_train = data.frame(pca_xtrain, y = as.factor(ytrain))
glm_model = glm(y ~ ., data = khan_train, family = "binomial")
pred5 = predict(glm_model, newdata = xtest_df, type = "response")

table(pred5 > 0.5, ytest)
```

```
##        ytest
##         1 2 3 4
##   FALSE 3 0 0 0
##   TRUE  0 6 6 5
```

```r
roc2 = roc(ytest, pred5)
plot(roc2, main = "ROC Curve for Logistic Regression", col="blue", lwd=2)
```



I performed the logistic regression, and as we can see the model missed all the class one, and the test error is 0.15.

Lasso

```r
cv_model = cv.glmnet(
  pca_xtrain, ytrain,
  alpha = 1,
  nfolds = 10
)

lambda_opt = cv_model$lambda.min
cat("Optimal lambda:", lambda_opt, "\n")
```

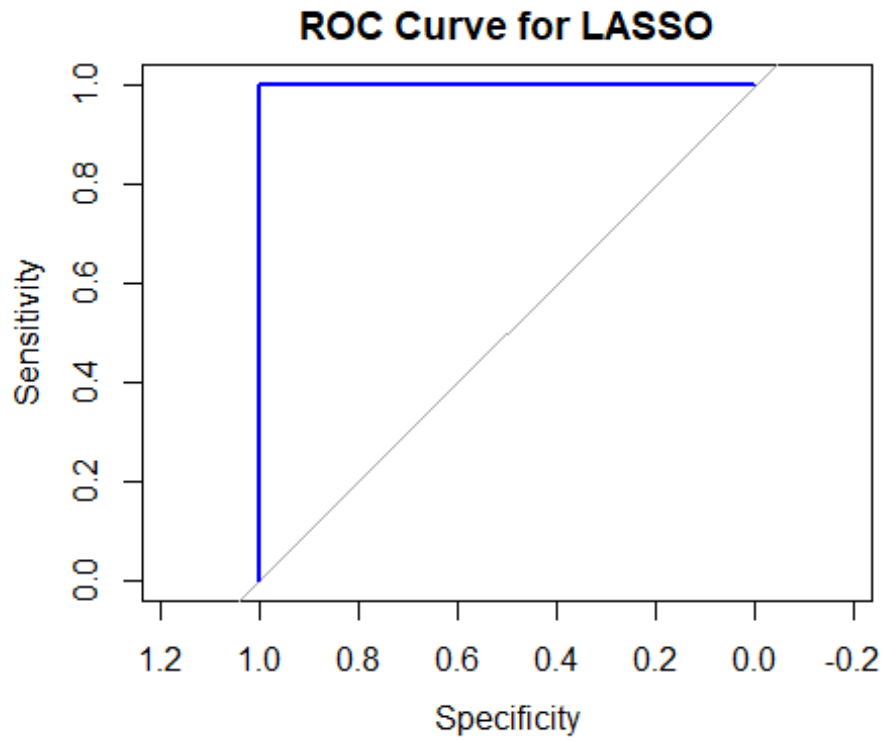
```
## Optimal lambda: 0.0111245

pred7 = predict(cv_model, newx = pca_xtest, s = lambda_opt,type = "response")
table(pred5 > 0.5, ytest)

##        ytest
##         1 2 3 4
##   FALSE 3 0 0 0
##   TRUE  0 6 6 5

roc_obj = roc(ytest, as.numeric(pred7))

plot(roc_obj, main = "ROC Curve for LASSO", col="blue", lwd=2)
```



I performed the Lasso regression with cross-validation to find the optimal lambda, and we used the optimal lambda to predict the test set and calculate the test error of 0.15.

LDA

```
model_lda =  lda(y ~ ., data = khan_train)
pred6 <- predict(model_lda, newdata = data.frame(pca_xtest))$class
table(pred6, ytest)

##      ytest
## pred6 1 2 3 4
##     1 3 0 0 0
##     2 0 6 0 0
```

```
##     3 0 0 6 0
##     4 0 0 0 5
```

```
mean(pred6 !=ytest)
```

```
## [1] 0
```

In the LDA model, it predicts all the cases correctly, and give us a 0 test error.

Deep learning methods

```
library(h2o)
h2o.init()
```

```
##
## H2O is not running yet, starting it now...
##
## Note:  In case of errors look at the following log files:
##
C:\Users\HP\AppData\Local\Temp\Rtmp2xRXLD\file79c22ab3f78/h2o_HP_started_from
_r.out
##
C:\Users\HP\AppData\Local\Temp\Rtmp2xRXLD\file79c42067a6b/h2o_HP_started_from
_r.err
##
##
## Starting H2O JVM and connecting:  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         3 seconds 60 milliseconds
##     H2O cluster timezone:       America/New_York
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.44.0.3
##     H2O cluster version age:    1 year, 5 months and 4 days
##     H2O cluster name:           H2O_started_from_R_HP_oid149
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   1.91 GB
##     H2O cluster total cores:    8
##     H2O cluster allowed cores:  8
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
##     H2O Internal Security:      FALSE
##     R Version:                  R version 4.3.1 (2023-06-16 ucrt)
```

```
train_h2o <- as.h2o(pca_xtrain)
```

```
##   |
|                                                                      |   0%
```

```
|
|======================================================================| 100%

test_h2o <- as.h2o(pca_xtest)

##   |
|                                                                      |   0%
|
|======================================================================| 100%

ytrain <- as.h2o(as.factor(Khan$ytrain))

##   |
|                                                                      |   0%
|
|======================================================================| 100%

ytest <- as.h2o(as.factor(Khan$ytest))

##   |
|                                                                      |   0%
|
|======================================================================| 100%

train_data <- h2o.cbind(train_h2o, ytrain)
test_data <- h2o.cbind(test_h2o, ytest)
colnames(train_data)[ncol(train_data)] <-"target"
colnames(test_data)[ncol(test_data)] <-"target"
target <-"target"
predictors <- setdiff(names(train_data), target)
dl_model <- h2o.deeplearning( x = predictors, y = target, training_frame =
train_data, validation_frame = test_data, hidden = c(100,100), epochs =20,
activation ="Rectifier", loss ="CrossEntropy",rate = 0.01,seed =1)

##   |
|                                                                      |   0%
|
|======================================================================| 100%

performance <- h2o.performance(dl_model, newdata = test_data)
classification_error <- h2o.confusionMatrix(performance)$Error[3]
print(paste("Classification Error Rate :", classification_error))

## [1] "Classification Error Rate : 0.166666666666667"
```

In the deep learning model, the classification error rate is 0.166666666666667

4. Summary

The Khan dataset, with its predefined train-test split, presents unique challenges for model evaluation. The small sample size (63 observations) and high-dimensional nature of the data make it prone to overfitting, which limits the effectiveness of cross-validation (CV).

Even with pre-separated data, CV may lead to high variance in performance due to the small test sample sizes, and dependencies between samples could cause leakage. Therefore, alternative techniques, such as directly applying models to the pre-separated data or using feature selection, may be more appropriate.

Regarding model performance, K-Nearest Neighbors (KNN) has the highest test error at 0.25, while Decision Tree and Random Forest both show test errors of 0.45, indicating they perform poorly. Bagging slightly improves the error to 0.35. On the other hand, Support Vector Machine (SVM) and Linear Discriminant Analysis (LDA) achieve perfect test errors of 0, making them the best performers. Logistic Regression and Lasso show test errors of 0.15, effectively capturing underlying patterns with simpler models. Deep Learning has a test error of 0.16667, slightly higher than Logistic Regression and Lasso, suggesting minor overfitting or suboptimal tuning.

Outliers in gene expression data, while potentially indicative of biologically significant phenomena like rare mutations, may be retained since modern algorithms, including regularized models, are robust to them. Altering the data could disrupt the representativeness of the test set. Additionally, Principal Component Analysis (PCA) is highly effective for gene clustering, as it reduces the high-dimensional data to a few principal components, enhancing the signal-to-noise ratio and improving clustering efficiency. PCA also captures gene correlations, which is critical for grouping co-regulated or functionally related genes, providing biologically meaningful insights.

In conclusion, Logistic Regression, Lasso, and SVM perform the best, while more complex models like Decision Tree, Random Forest, and Bagging are less effective. Retaining outliers and using PCA for dimensionality reduction ensures that biologically relevant patterns are preserved in the analysis.