Statistical Learning, Homework #3

Elisa Paolazzi [MAT. 224424]

20/05/2022

R Setup

```
library(tidyverse)
library(e1071) # for SVM
library(ggplot2) # for nice plots
library(dplyr) # for data manipulation
library(rsample) # for train-test split
library(caret) # for K-fold CV
library(ggpubr) # for plots' grid
library(ROCR) # for ROC curves

# choose a seed number
seed = 42
```

Data exploration

The first step of the analysis is to load the data (`gene_expr.tsv`) and and conduct a brief data exploration:

```
# load the dataset
gene_df = read.csv("gene_expr.tsv", sep = "\t")
# dataset dimensions
dim(gene_df)
```

```
## [1]   79 2002
```

```
# show an extract of the dataset
knitr::kable(head(gene_df[,1:8]))
```

sampleID	X354_s_at	X37387_r_at	X510_g_at	X32274_r_at	X41129_at	X32896_at	X37035_at
1005	3.191091	6.569342	5.795215	8.577962	6.074154	2.663559	8.202604
1010	3.014668	6.515097	4.236670	8.298168	4.844150	2.780224	7.208483
3002	3.064113	6.608859	5.318619	7.942474	6.395045	2.823451	8.137416
4007	3.340386	6.383065	5.906107	7.888818	6.146822	2.520204	8.459870
4008	3.616074	6.333751	6.043892	8.327389	5.240616	2.694700	7.740742
4010	3.171587	6.633316	4.435124	8.548463	4.778371	2.853498	6.935658

As specified in the homework assignment, we are dealing with 79 observations (patients with leukemia). In particular we have 2002 columns: 2000 of them contain expression for different genes, while the remaining two are the `sampleID` column, which is useful to identify the patients, and the response variable (`y`) column, which categorizes the patients into two groups.

Given the huge number of features (2000), which far exceeds the number of observations (79), we are in the case of high-dimensional data. This kind of context could imply the so-called *curse of dimensionality*, which refers to the increased computational efforts required for the data processing and/or analysis of such big data. This is in fact a very common situation in computational biology, which is frequently dealing with high-dimensional data.

Back to the data exploration, we can immediately check for missing values:

```
# check number of NAs
sum(is.na(gene_df))
```

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

Fortunately, we do not detect any NA.

Focusing now on the response variable `y`, we know that it divides the patients into two groups: patients with a chromosomal translocation (labeled as "1") and patients cytogenetically normal (labeled as "−1"). We are in fact dealing with a categorical outcome variable, and therefore ours will be a classification task.

For this reason we convert the `y` column into factors:

```
# convert the column as factor
gene_df$y = as.factor(gene_df$y)
# print the summary of the response variable
summary(gene_df$y)
```

```
## -1  1
## 42 37
```

As we can see from the summary, the classes are quite balanced: we have 42 observation for the −1 class (53%) and 37 for the 1 class (47%). We then visualize this situation by means of a couple of plots:

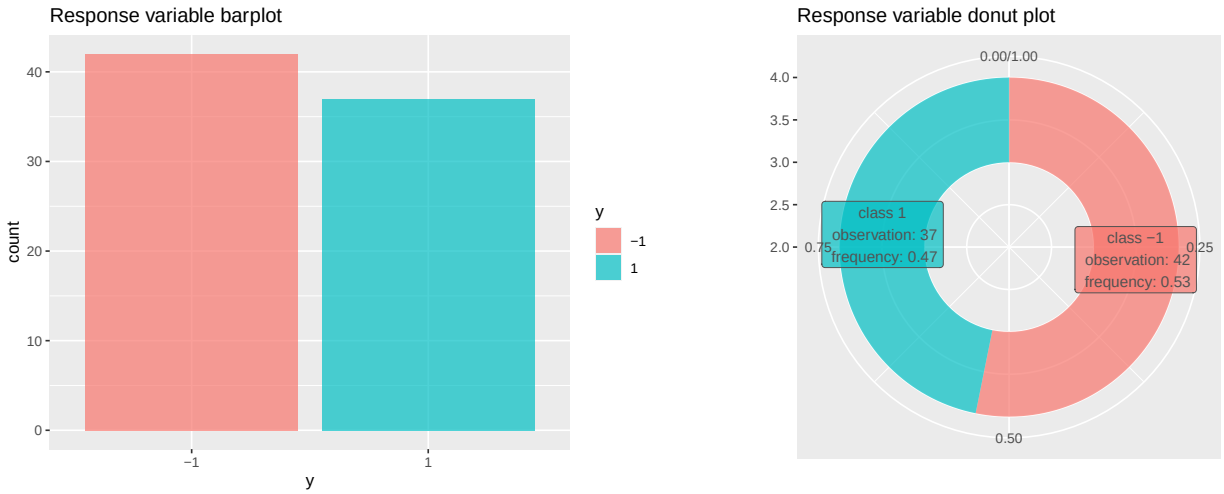
```
# response variable barplot
gene_bar = ggplot(gene_df, aes(x=y, fill=y)) +
  geom_bar(alpha=0.7) +
  ggtitle("Response variable barplot")

# response variable donut plot
# compute percentages
gene_df_plot = gene_df %>%
  group_by(y) %>%
  summarise(n = n()) %>%
  mutate(freq = n / sum(n))

# compute the cumulative percentages (top of each rectangle)
gene_df_plot$ymax = cumsum(gene_df_plot$freq)
# compute the bottom of each rectangle
gene_df_plot$ymin = c(0, head(gene_df_plot$ymax, n=-1))
# compute label position
gene_df_plot$labelPosition = (gene_df_plot$ymax + gene_df_plot$ymin) / 2
# compute a good label
gene_df_plot$label = paste0("class ", gene_df_plot$y, "\n observation: ",
                            gene_df_plot$n, "\n frequency: ", round(gene_df_plot$freq, 2))

# plot
gene_don = ggplot(gene_df_plot, aes(ymax=ymax, ymin=ymin, xmax=4, xmin=3, fill=y)) +
  geom_rect(alpha=0.7) +
  geom_label(x=3.5, aes(y=labelPosition, label=label), size=3.5, color="#525252", alpha=0.7) +
  coord_polar(theta="y") +
  xlim(c(2, 4)) +
  ylab("") +
  theme(legend.position = "none") +
  ggtitle("Response variable donut plot")

# show plots in a grid
ggarrange(gene_bar, gene_don, ncol = 2, nrow = 1)
```



Due to the huge number of the predictors we do not show summaries and plots concerning the predictors, their correlation and the outcome variable distribution within them. However, also given the nature of the data, we can imagine that some predictors could be correlated with each other and with the response variable.

Focusing now on column `sampleID`, we note that it contains a different numerical value for each observation:

```
# check if the unique values of the column are 79, as the number of observation
length(unique(gene_df$sampleID)) == dim(gene_df)[1]
```

```
## [1] TRUE
```

This is natural as this variable refers to the patients' ID. However, this data is not relevant for our analysis and its inclusion among the explanatory variables could compromise the fitting and testing of the models (however in a very lightweight manner, given the other numerous dimensions). For this reason we decide to remove this variable from the dataset:

```
gene_df_noID = gene_df %>%
  select(-sampleID)
```

1. Load the data and select a support vector machine for the task at hand. Evaluate different models and justify your final choice.

Support vector machines (SVM) are a powerful tool to analyze data with a number of predictors approximately equal or larger than the number of observations. In our case the number of predictors is in fact actually much larger than the number of observations.

SVC (linear kernel SVM) The first model we can test is the *support vector classifier*. This model uses a linear decision surface to classify non-separable classes, thus eventually allowing for misclassifications. In particular, it is the solution to the following optimization problem:

$$\begin{aligned}
&\text{maximize} \\
&\beta_0, \beta_1, \dots, \beta_p, \epsilon_1, \dots, \epsilon_n, M \\
&\text{subject to } \sum_{j=1}^{p} \beta_j^2 = 1 \\
&y_i \left(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip}\right) \geq M \left(1 - \epsilon_i\right) \\
&\epsilon_i \geq 0, \quad \sum_{i=1}^{n} \epsilon_i \leq C
\end{aligned}$$

Therefore, this type of models divide the p-dimensional space using a flat affine subspace of dimension $p-1$ (hyperplane), mapping observations to points in space so as to maximize the width of the gap between the two categories. New observations can then be mapped into that same space and predicted to belong to a category based on which side of the hyperplane they fall.

First, we can check if, considering all the 2000 predictors, the two response classes are linearly separable (whether it is possible to separate classes with a linear hyperplane) by fitting an SVC on the whole data. The hyperparameter

`cost` represents the cost of a violation to the margin (the smaller, the wider the margins) and it is inversely proportional to the slack budget parameter C in the formula below.

We will use a very high value for cost parameter to check if the data are linearly separable. In this way we impose an hard margin, not allowing for misclassifications.

```
# fit a SVC with an high cost (hard margin)
svc = svm(y ~ ., data=gene_df_noID, kernel="linear", cost=100)
summary(svc)
```

```
##
## Call:
## svm(formula = y ~ ., data = gene_df_noID, kernel = "linear", cost = 100)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  100
##
## Number of Support Vectors:  69
##
##  ( 33 36 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

First of all, the output shows us the call of the function and the parameters employed (either specified by the user or set by default). The number of support vectors is then reported, they represent those observations that characterise the hyperplane by being the closest to it. In particular, within brackets, we find how many support vectors belong to class -1 and to class 1. In our case, the number of support vectors is quite high (69, thus the 87% of the points), which is in general unusual for such a narrow margin. However, remember that our data has so many features, and therefore the hyperplane has many dimensions where points can touch the margin.

```
# confusion matrix
table(svc$fitted, gene_df_noID$y)
```

```
##
##      -1  1
##   -1 42  0
##    1   0 37
```

```
# (training) accuracy
mean(svc$fitted == gene_df_noID$y)
```

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

As we can notice from the confusion matrix and accuracy value, we have no missclassification, thus we can say that a linear hyperplane is able to separate our data. As we said, when we have a "$features \gg observations$" situation we are dealing with the curse of dimensionality: it is easier to find a separating hyperplane in such situation.

After this check, we can divide the data into train and test partitions and fit and test a SVC with the same previous parameter (high value for `cost`).

```
# set the seed for reproducible partitions
set.seed(22)

# split data into training and test sets
split = initial_split(gene_df_noID, prop=0.7)
x_train = training(split)
x_test = testing(split)
y_test = x_test$y
```

We fit the SVC using the training set and calculate the training accuracy:

```
# fit SVC using trainig set
svc_tr = svm(y ~ ., data=x_train, kernel="linear", cost=100)

# confusion matrix
table(svc_tr$fitted, x_train$y)
```

```
##
##      -1  1
##   -1 28  0
##   1   0 27
```

```
# training accuracy
mean(svc_tr$fitted == x_train$y)
```

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

Although using a high `cost`, we have no training error (remember that data are linearly separable).

However, let's see how the model performs on the test set:

```
# predict on the test set
pred_svc = predict(svc_tr, x_test)
# confusion matrix
table(pred_svc, y_test)
```

```
##         y_test
## pred_svc -1  1
##       -1  8  0
##       1   6 10
```

```
# test accuracy
mean(pred_svc == y_test)
```

```
## [1] 0.75
```

We have 6 misclassifications and the model reaches a test accuracy of 0.75 now. We can also calculate the precision for each class:

```
# precision for class -1
table(pred_svc, y_test)[2,2]/sum(table(pred_svc, y_test)[,2])
```

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

```
# precision for class 1
table(pred_svc, y_test)[1,1]/sum(table(pred_svc, y_test)[,1])
```

```
## [1] 0.5714286
```

As we can see we have a precision of 1 for class -1, while the class 1 achieves a precision of 0.57. We therefore understand that the model has some difficulty in classifying this latter class.

The high cost value (which build a hard margin) may not the optimal parameter for the model and tend to overfit the training data, thus it is important to tune this parameter using cross-validation. In particular we will use the `tune()` function, which performs a 10-fold cross-validation in order to choose the optimal parameter.

```
# set seed for reproducible results
set.seed(seed)
# tune cost parameter
tune.svc = tune(svm, y ~ ., data=x_train, kernel="linear",
                ranges=list(cost=c(0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1,
                                   1, 5, 10, 100)))
# get CV results
summary(tune.svc)
```

```
##
## Parameter tuning of 'svm':
##
```

```
## - sampling method: 10-fold cross validation
##
## - best parameters:
##   cost
##  0.005
##
## - best performance: 0.3366667
##
## - Detailed performance results:
##     cost     error dispersion
## 1  1e-04 0.7600000 0.09787874
## 2  5e-04 0.4700000 0.14006171
## 3  1e-03 0.3766667 0.19817065
## 4  5e-03 0.3366667 0.18820136
## 5  1e-02 0.3366667 0.18820136
## 6  1e-01 0.3366667 0.18820136
## 7  1e+00 0.3366667 0.18820136
## 8  5e+00 0.3366667 0.18820136
## 9  1e+01 0.3366667 0.18820136
## 10 1e+02 0.3366667 0.18820136
```

```
# extract the optimal model
bestmod_svc = tune.svc$best.model
```

The `tune()` function shows us the results of a 10-fold cross-validation on the training set using a ranges of `cost` values. The optimal cost parameter is 0.005, however, the function only returns the first cost value occurrence that achieves the best performance. In fact also our previous model cost (100) achieve the same best performance, thus the previous was already one of the optimal model for this partition.

```
# make prediction using the tune() optimal model (cost=0.005)
y_pred = predict(bestmod_svc, x_test)
# confusion matrix
table(y_pred, y_test)
```

```
##       y_test
## y_pred -1  1
##     -1  8  0
##     1   6 10
```

```
# accuracy
mean(y_pred == y_test)
```

```
## [1] 0.75
```

As we can see both misclassifications and accuracy are the same as the previous model.

Naturally, results may vary with other train-test partitions: the previous split is useful in order to show, interpret and have a first approach with the outputs of the model function.

However, to properly evaluate the performance of the models and to eventually select the optimal model, it is essential to use a nested cross-validation. In particular we will also re-optimize the model (tuning the `cost` parameter) at each fold iteration.

We will use a 10-fold cross-validation for both the parameter tuning and model assessment, for which we will use the accuracy measure. Each folds' accuracy value will then be averaged in order to obtain the cross-validation accuracy for the model.

Note that, due to the curse of dimensionality, computations are very time-consuming, thus, running the code may take a very long time.

```
# set seed for reproducible folds
set.seed(seed)
# get 10 set of index for training sets composition
gene_flds = createFolds(gene_df_noID$y, k = 10, list = T, returnTrain = T)

# initialize accuracy list
accuracy_CV_svc = c()
```

```r
#initialize precision lists
precision_svc_1 = c()
precision_svc_neg1 = c()
# initialize optimal costs list
optimal_cost = c()

# iterate over the folds
for (f in gene_flds) {
  # set seed for reproducible results
  set.seed(seed)

  # define index based on the i fold
  train_idx = f
  # split train and test sets according to the indexes
  train = gene_df_noID[train_idx,]
  test = gene_df_noID[-train_idx,]
  y_ts = test$y

  # tune cost parameter
  tune.out = tune(svm, y ~ ., data=train, kernel="linear",
                  ranges=list(cost=c(0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1,
                                     1, 5, 10, 100)))
  # summary(tune.out)
  # save the optimal cost parameter
  optimal_cost = append(optimal_cost, tune.out$best.parameters$cost)

  # extract optimal model
  bestmod = tune.out$best.model
  # make prediction using optimal model
  ypred = predict(bestmod, test)
  # calculate accuracy
  acc = mean(ypred == y_ts)
  # add accuracy to cv accuracy list
  accuracy_CV_svc = append(accuracy_CV_svc, acc)

  # precision for class 1
  prec_1 = table(ypred, y_ts)[2,2]/sum(table(ypred, y_ts)[,2])
  precision_svc_1 = append(precision_svc_1, prec_1)
  # precision for class -1
  prec_neg1 = table(ypred, y_ts)[1,1]/sum(table(ypred, y_ts)[,1])
  precision_svc_neg1 = append(precision_svc_neg1, prec_neg1)
}

# mean accuracy for SVC
mean(accuracy_CV_svc)
```

```
## [1] 0.7934524
```

As we can see the cross-validation accuracy for SVC is 0.79: some partition achieve in fact an higher accuracy with respect to ours. Note that, since we use a 10-fold cross-validation, the train and test proportion changes slightly from that previously used (0.7).

Out of curiosity we also stored the optimal values for cost parameter:

```r
# build a dataframe for optimal cost values
opt_cost_cv = data.frame(fold = seq(1,10), optimal_cost=optimal_cost)
knitr::kable(opt_cost_cv)
```

fold	optimal_cost
1	1e-03
2	1e-03
3	5e-04

fold	optimal_cost
4	5e-03
5	1e-03
6	1e-03
7	1e-03
8	1e-03
9	5e-03
10	1e-03

As we can see the most popular optimal cost value is 0.001, however, remember that the `tune()` function only returns the first occurrence as optimal value.

We can now look at the average precision value for each class:

```
# mean precision for class 1
mean(precision_svc_1)
```

```
## [1] 0.7833333
```

```
# mean precision for class -1
mean(precision_svc_neg1)
```

```
## [1] 0.805
```

As well as the accuracy, the precision values for the two classes are also quite good and similar. This suggests that the SVC model is an appropriate model for our data classification.

In general the linear kernel is in fact more suitable when the number of features is much larger than the number of observations; however, we can attempt to fit non-linear SVM models as well and see how they perform.

Polynomial kernel SVM The support vector machine (SVM) is an extension of the support vector classifier that results from enlarging the feature space in a specific way, using kernels. A kernel is a set of mathematical functions providing a window to manipulate the data, mapping them into higher dimension. In fact, the previous SVC can be also referred to as an SVM model with a linear kernel, which defines a linear boundary between the classes. However, sometimes we may want to enlarge our feature space in order to accommodate also non-linear boundary between the classes.

In particular, the SVM consists in the solution to the following optimization problem:

$$\underset{\beta_0,\beta_{11},\beta_{12},\ldots,\beta_{p1},\beta_{p2},\epsilon_1,\ldots,\epsilon_n,M}{\text{maximize}} M$$
$$\text{subject to } y_i\left(\beta_0+\sum_{j=1}^{p}\beta_{j1}x_{ij}+\sum_{j=1}^{p}\beta_{j2}x_{ij}^2\right)\geq M\left(1-\epsilon_i\right),$$
$$\sum_{i=1}^{n}\epsilon_i\leq C,\quad \epsilon_i\geq 0,\quad \sum_{j=1}^{p}\sum_{k=1}^{2}\beta_{jk}^2=1$$

We can now examine how a SVM model with a polynomial kernel performs. Using such a kernel with $d>1$, instead of the standard linear kernel in the support vector classifier algorithm, leads to a much more flexible decision boundary. It essentially amounts to fitting a support vector classifier in a higher-dimensional space involving polynomials of degree d, rather than in the original feature space. d is a positive integer and the polynomial kernel can be written as:

$$K\left(x_i,x_{i'}\right)=\left(1+\sum_{j=1}^{p}x_{ij}x_{i'j}\right)^d$$

However, given the linear separability of the data, we expect the model using such a kernel to perform worse than the previous one.

Since we have no idea about the optimal parameters (which in this case are the `degree` of the polynomial and the `cost`) we immediately use cross-validation to find the best combination for our train-test partitions.

```r
# set seed for reproducible results
set.seed(seed)
# tune cost and degree parameters
tune.poly = tune(svm, y ~ ., data=x_train, kernel="polynomial",
                 ranges=list(cost=c(0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1,
                                    1, 5, 10, 100),
                             degree=c(2, 3, 4, 5, 6)))
# get CV results
summary(tune.poly)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost degree
##   100      5
##
## - best performance: 0.5366667
##
## - Detailed performance results:
##     cost degree     error dispersion
## 1  1e-04      2 0.7266667 0.12352837
## 2  5e-04      2 0.7266667 0.12352837
## 3  1e-03      2 0.7266667 0.12352837
## 4  5e-03      2 0.7266667 0.12352837
## 5  1e-02      2 0.7266667 0.12352837
## 6  1e-01      2 0.7266667 0.12352837
## 7  1e+00      2 0.7266667 0.15539674
## 8  5e+00      2 0.5533333 0.21268420
## 9  1e+01      2 0.5733333 0.18311435
## 10 1e+02      2 0.5733333 0.18311435
## 11 1e-04      3 0.7266667 0.12352837
## 12 5e-04      3 0.7266667 0.12352837
## 13 1e-03      3 0.7266667 0.12352837
## 14 5e-03      3 0.7266667 0.12352837
## 15 1e-02      3 0.7266667 0.12352837
## 16 1e-01      3 0.7266667 0.12352837
## 17 1e+00      3 0.7266667 0.14639706
## 18 5e+00      3 0.5833333 0.18475877
## 19 1e+01      3 0.5633333 0.20027759
## 20 1e+02      3 0.5600000 0.19360135
## 21 1e-04      4 0.7266667 0.12352837
## 22 5e-04      4 0.7266667 0.12352837
## 23 1e-03      4 0.7266667 0.12352837
## 24 5e-03      4 0.7266667 0.12352837
## 25 1e-02      4 0.7266667 0.12352837
## 26 1e-01      4 0.7100000 0.14233502
## 27 1e+00      4 0.7633333 0.08527371
## 28 5e+00      4 0.6900000 0.15157608
## 29 1e+01      4 0.7100000 0.15157608
## 30 1e+02      4 0.6133333 0.17511901
## 31 1e-04      5 0.7266667 0.12352837
## 32 5e-04      5 0.7266667 0.12352837
## 33 1e-03      5 0.7266667 0.12352837
## 34 5e-03      5 0.7266667 0.12352837
## 35 1e-02      5 0.7266667 0.12352837
## 36 1e-01      5 0.6933333 0.17554149
## 37 1e+00      5 0.7300000 0.13917748
## 38 5e+00      5 0.6866667 0.20620019
```

```
## 39 1e+01      5 0.6166667 0.18872542
## 40 1e+02      5 0.5366667 0.25889127
## 41 1e-04      6 0.7266667 0.12352837
## 42 5e-04      6 0.7266667 0.12352837
## 43 1e-03      6 0.7266667 0.12352837
## 44 5e-03      6 0.7266667 0.12352837
## 45 1e-02      6 0.7266667 0.12352837
## 46 1e-01      6 0.6933333 0.17554149
## 47 1e+00      6 0.7466667 0.11674600
## 48 5e+00      6 0.7266667 0.15539674
## 49 1e+01      6 0.7266667 0.15539674
## 50 1e+02      6 0.7133333 0.11021864
```

As we can notice, in this case the function return us the model with the best combination of `cost` and `degree` parameters. The optimal model for this partition is the one with `degree` 5 and `cost` 100. We can then test the optimal parameter model:

```
# extract optimal model
bestmod_poly = tune.poly$best.model
# make prediction using optimal model
ypred = predict(bestmod_poly, x_test)
# confusion matrix
table(ypred, y_test)
```

```
##      y_test
## ypred -1  1
##    -1  1  0
##    1  13 10
```

```
# calculate accuracy
mean(ypred == y_test)
```

```
## [1] 0.4583333
```

As we imagined, the accuracy is much lower than in the previous model, reaching even less than 50% accuracy with this partition. In particular, the model fails to properly classify class -1 and tends to assign everything to class 1.

As usual, results may change according to different train-test partitions; hence we find the average accuracy by nested cross-validation (using the same folds and schema as the SVC):

```
# initialize accuracy list
accuracy_CV_poly = c()
# initialize precision lists
precision_poly_1 = c()
precision_poly_neg1 = c()
# initialize optimal costs list
optimal_cost_poly = c()
optimal_degree_poly = c()

# iterate over the folds
for (f in gene_flds) {
  # set seed for reproducible results
  set.seed(seed)
  # define index based on the i fold
  train_idx = f
  # split train and test sets according to the indexes
  train = gene_df_noID[train_idx,]
  test = gene_df_noID[-train_idx,]
  y_ts = test$y

  # tune cost parameter
  tune.out = tune(svm, y ~ ., data=train, kernel="polynomial",
                  ranges=list(cost=c(0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1,
                                     1, 5, 10, 100),
                              degree=c(2, 3, 4, 5, 6)))
```

```
  # summary(tune.out)
  # save the optimal cost parameter
  optimal_cost_poly = append(optimal_cost_poly, tune.out$best.parameters$cost)
  optimal_degree_poly = append(optimal_degree_poly, tune.out$best.parameters$degree)

  # extract optimal model
  bestmod = tune.out$best.model
  # make prediction using optimal model
  ypred = predict(bestmod, test)
  # calculate accuracy
  acc = mean(ypred == y_ts)
  # add accuracy to cv accuracy list
  accuracy_CV_poly = append(accuracy_CV_poly, acc)

  # precision for class 1
  prec_1 = table(ypred, y_ts)[2,2]/sum(table(ypred, y_ts)[,2])
  precision_poly_1 = append(precision_poly_1, prec_1)
  # precision for class -1
  prec_neg1 = table(ypred, y_ts)[1,1]/sum(table(ypred, y_ts)[,1])
  precision_poly_neg1 = append(precision_poly_neg1, prec_neg1)
}

# CV accuracy
mean(accuracy_CV_poly)
```

```
## [1] 0.5498016
```

```
# precision class -1
mean(precision_poly_neg1)
```

```
## [1] 0.465
```

```
# precision class 1
mean(precision_poly_1)
```

```
## [1] 0.65
```

The average CV accuracy is higher than the accuracy achieved on our partition, however it is just above 50%. Furthermore, neither class precision reaches high values: we record an average precision of 0.65 for class 1 and 0.47 for class -1.

The most popular `cost` values are in this case the highest (5, 10 and 100), while the most popular value for `degree` is 3.

In summary, performances using an SVM with polynomial kernel turn out to be quite lower than those of the SVC.

Radial kernel SVM For the sake of completeness, let us finally test an SVM model with a radial kernel. This model is a bit more local than the other two types of kernel, in the sense that only nearby training observations have an effect on the class label of a test observation. In fact, with this kernel the proximity between training and test observations in terms of Euclidean distance plays a key role in the prediction. In other words the closer training observations have a lot of influence on a test observation classification, while the training observations that are further away have relatively little influence on the prediction. Due to these characteristics the radial kernel SVM behaviour can be described as a weighted nearest neighbour model.

This type of kernel can be written as:

$$K\left(x_i, x_{i'}\right) = \exp\left(-\gamma \sum_{j=1}^{p} \left(x_{ij} - x_{i'j}\right)^2\right)$$

As the previous model we have the `cost` parameter. Additionally, with this kernel, another parameter to tune is the γ, which handles the amount influence that nearest observation have on each other, scaling their squared distance (formula below). If gamma is too large, the radius of the area of influence of the support vectors only includes the support vector itself and the model tends to overfit. When gamma is very small, the model is too constrained and

cannot capture the complexity or "shape" of the data. Moreover, the region of influence of any selected support vector would include the whole training set.

For these reason it is important to tune both `cost` and $_{\gamma}$ parameters using cross-validation (on the previous defined partition):

```
# set seed for reproducible results
set.seed(seed)
tune.rad = tune(svm, y ~ ., data=x_train, kernel="radial",
                ranges=list(cost=c(0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1,
                                   1, 5, 10, 100, 150),
                            gamma=c(0.1, 0.5, 1, 2, 3, 4, 6)))
summary(tune.rad)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##   cost gamma
##  1e-04   0.1
##
## - best performance: 0.7266667
##
## - Detailed performance results:
##       cost gamma     error dispersion
## 1  1.0e-04   0.1 0.7266667  0.1235284
## 2  5.0e-04   0.1 0.7266667  0.1235284
## 3  1.0e-03   0.1 0.7266667  0.1235284
## 4  5.0e-03   0.1 0.7266667  0.1235284
## 5  1.0e-02   0.1 0.7266667  0.1235284
## 6  1.0e-01   0.1 0.7266667  0.1235284
## 7  1.0e+00   0.1 0.7266667  0.1235284
## 8  5.0e+00   0.1 0.7266667  0.1235284
## 9  1.0e+01   0.1 0.7266667  0.1235284
## 10 1.0e+02   0.1 0.7266667  0.1235284
## 11 1.5e+02   0.1 0.7266667  0.1235284
## 12 1.0e-04   0.5 0.7266667  0.1235284
## 13 5.0e-04   0.5 0.7266667  0.1235284
## 14 1.0e-03   0.5 0.7266667  0.1235284
## 15 5.0e-03   0.5 0.7266667  0.1235284
## 16 1.0e-02   0.5 0.7266667  0.1235284
## 17 1.0e-01   0.5 0.7266667  0.1235284
## 18 1.0e+00   0.5 0.7266667  0.1235284
## 19 5.0e+00   0.5 0.7266667  0.1235284
## 20 1.0e+01   0.5 0.7266667  0.1235284
## 21 1.0e+02   0.5 0.7266667  0.1235284
## 22 1.5e+02   0.5 0.7266667  0.1235284
## 23 1.0e-04   1.0 0.7266667  0.1235284
## 24 5.0e-04   1.0 0.7266667  0.1235284
## 25 1.0e-03   1.0 0.7266667  0.1235284
## 26 5.0e-03   1.0 0.7266667  0.1235284
## 27 1.0e-02   1.0 0.7266667  0.1235284
## 28 1.0e-01   1.0 0.7266667  0.1235284
## 29 1.0e+00   1.0 0.7266667  0.1235284
## 30 5.0e+00   1.0 0.7266667  0.1235284
## 31 1.0e+01   1.0 0.7266667  0.1235284
## 32 1.0e+02   1.0 0.7266667  0.1235284
## 33 1.5e+02   1.0 0.7266667  0.1235284
## 34 1.0e-04   2.0 0.7266667  0.1235284
## 35 5.0e-04   2.0 0.7266667  0.1235284
```

```
## 36 1.0e-03   2.0 0.7266667  0.1235284
## 37 5.0e-03   2.0 0.7266667  0.1235284
## 38 1.0e-02   2.0 0.7266667  0.1235284
## 39 1.0e-01   2.0 0.7266667  0.1235284
## 40 1.0e+00   2.0 0.7266667  0.1235284
## 41 5.0e+00   2.0 0.7266667  0.1235284
## 42 1.0e+01   2.0 0.7266667  0.1235284
## 43 1.0e+02   2.0 0.7266667  0.1235284
## 44 1.5e+02   2.0 0.7266667  0.1235284
## 45 1.0e-04   3.0 0.7266667  0.1235284
## 46 5.0e-04   3.0 0.7266667  0.1235284
## 47 1.0e-03   3.0 0.7266667  0.1235284
## 48 5.0e-03   3.0 0.7266667  0.1235284
## 49 1.0e-02   3.0 0.7266667  0.1235284
## 50 1.0e-01   3.0 0.7266667  0.1235284
## 51 1.0e+00   3.0 0.7266667  0.1235284
## 52 5.0e+00   3.0 0.7266667  0.1235284
## 53 1.0e+01   3.0 0.7266667  0.1235284
## 54 1.0e+02   3.0 0.7266667  0.1235284
## 55 1.5e+02   3.0 0.7266667  0.1235284
## 56 1.0e-04   4.0 0.7266667  0.1235284
## 57 5.0e-04   4.0 0.7266667  0.1235284
## 58 1.0e-03   4.0 0.7266667  0.1235284
## 59 5.0e-03   4.0 0.7266667  0.1235284
## 60 1.0e-02   4.0 0.7266667  0.1235284
## 61 1.0e-01   4.0 0.7266667  0.1235284
## 62 1.0e+00   4.0 0.7266667  0.1235284
## 63 5.0e+00   4.0 0.7266667  0.1235284
## 64 1.0e+01   4.0 0.7266667  0.1235284
## 65 1.0e+02   4.0 0.7266667  0.1235284
## 66 1.5e+02   4.0 0.7266667  0.1235284
## 67 1.0e-04   6.0 0.7266667  0.1235284
## 68 5.0e-04   6.0 0.7266667  0.1235284
## 69 1.0e-03   6.0 0.7266667  0.1235284
## 70 5.0e-03   6.0 0.7266667  0.1235284
## 71 1.0e-02   6.0 0.7266667  0.1235284
## 72 1.0e-01   6.0 0.7266667  0.1235284
## 73 1.0e+00   6.0 0.7266667  0.1235284
## 74 5.0e+00   6.0 0.7266667  0.1235284
## 75 1.0e+01   6.0 0.7266667  0.1235284
## 76 1.0e+02   6.0 0.7266667  0.1235284
## 77 1.5e+02   6.0 0.7266667  0.1235284
```

Again, the function `tune()` returns the best combination of parameters, thus the best model in this case is the one with 0.0001 `cost` and 0.1 `gamma`.

```
# extract optimal model
bestmod_rad = tune.rad$best.model
# make prediction using optimal model
ypred = predict(bestmod_rad, x_test)
# confusion matrix
table(ypred, y_test)
```

```
##      y_test
## ypred -1  1
##    -1 14 10
##    1   0  0
```

```
# calculate accuracy
mean(ypred == y_test)
```

```
## [1] 0.5833333
```

In this case, the model with radial kernel achieves a higher accuracy than the model with polynomial kernel.

However, observing the confusion matrix we can note that this model classifies all observations in class -1, thus the accuracy value corresponds to the precision and the observed frequency of class -1 (while the precision for the other class is 0).

As with the previous models, we use nested 10-fold cross-validation to assess the performance of the optimal radial kernel model:

```
# initialize accuracy list
accuracy_CV_rad = c()
# initialize precision lists
precision_rad_1 = c()
precision_rad_neg1 = c()
# initialize optimal costs list
optimal_cost_rad = c()
# initialize optimal gamma list
optimal_gamma_rad = c()

# iterate over the folds
for (f in gene_flds) {
  # set seed for reproducible results
  set.seed(42)
  # define index based on the i fold
  train_idx = f
  # split train and test sets according to the indexes
  train = gene_df_noID[train_idx,]
  test = gene_df_noID[-train_idx,]
  y_ts = test$y

  # tune cost parameter
  tune.out = tune(svm, y ~ ., data=train, kernel="radial",
                  ranges=list(cost=c(0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1,
                                     1, 5, 10, 100),
                              gamma=c(0.1, 0.5, 1, 2, 3, 4, 6)))
  # summary(tune.out)
  # save the optimal cost parameter
  optimal_cost_rad = append(optimal_cost_rad, tune.out$best.parameters$cost)
  optimal_gamma_rad = append(optimal_gamma_rad, tune.out$best.parameters$gamma)

  # extract optimal model
  bestmod = tune.out$best.model
  # make prediction using optimal model
  ypred = predict(bestmod, test)
  # calculate accuracy
  acc = mean(ypred == y_ts)
  # add accuracy to cv accuracy list
  accuracy_CV_rad = append(accuracy_CV_rad, acc)

  # precision for class 1
  prec_1 = table(ypred, y_ts)[2,2]/sum(table(ypred, y_ts)[,2])
  precision_rad_1 = append(precision_rad_1, prec_1)
  # precision for class -1
  prec_neg1 = table(ypred, y_ts)[1,1]/sum(table(ypred, y_ts)[,1])
  precision_rad_neg1 = append(precision_rad_neg1, prec_neg1)
}
# CV accuracy
mean(accuracy_CV_rad)
```

```
## [1] 0.5325397
```

```
# precision class -1
mean(precision_rad_neg1)
```

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

```
# precision class 1
mean(precision_rad_1)
```

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

The cross-validation accuracy is slightly lower than that of the previous model (polynomial kernel) and reaches just over 50%. Beside that, we register an average precision of 1 for class -1 and 0 for class 1. Considering these precision values, the accuracy value tends to approach the observed frequency of class -1 (0.5316456). In fact, the radial kernel model classify each observation to class -1 in our case, that leads to the worst performance among the analysed models.

In conclusion, having tested SVM models using different kernels, the linear kernel model (SVC) seems to be the optimal one with these features. As we said before, this is due to the fact that, having so many features, it is easier to find a hyperplane able to separate the two classes. For this reason, in general, the SVC model is the most suitable when we have such a large number of features.

```
results_df = data.frame(model=c("SVC/SVM linear kernel", "SVM polynomial kernel",
                                "SVM radial kernel"),
                        CV_accuracy=c(mean(accuracy_CV_svc),
                                      mean(accuracy_CV_poly),
                                      mean(accuracy_CV_rad)),
                        precision_class_1=c(mean(precision_svc_1),
                                            mean(precision_poly_1),
                                            mean(precision_rad_1)),
                        precision_class_neg1=c(mean(precision_svc_neg1),
                                               mean(precision_poly_neg1),
                                               mean(precision_rad_neg1)))
knitr::kable(results_df)
```

model	CV_accuracy	precision_class_1	precision_class_neg1
SVC/SVM linear kernel	0.7934524	0.7833333	0.805
SVM polynomial kernel	0.5498016	0.6500000	0.465
SVM radial kernel	0.5325397	0.0000000	1.000

Unfortunately working with so many features is very computationally expensive, not to mention the fact that many of them generate noise. For the latter reason, the analysis and comparison of previous models may not be so reliable. In order to solve this problem (concerning the curse of dimensionality), variable selection or some dimensionality reduction technique would be useful.

2. A popular approach in gene expression analysis is to keep only the most variable genes for downstream analysis. Since most of the $2K$ genes have low expression or do not vary much across the experiments, this step usually minimizes the contribution of noise. Select then only genes whose standard deviation is among the top 5% and repeat the analyses performed in the previous task on the filtered data set.

As we have seen in the previous section, computations with a large number features are very expensive and time-consuming. Furthermore, so many features tend to create noise in the data and the optimal model selection becomes a quite challenging task.

We then proceed by selecting only those genes whose standard deviation is among the top 5%:

```
# Find the number of columns to keep
n_col_5 = 0.05*dim(dplyr:: select(gene_df_noID, -y))[2]

# Create a df and store gene and corresponding sd
genes_sd = gene_df_noID %>%
  select(-y) %>%
  summarise_if(is.numeric, sd)
# invert columns and rows
genes_sd = data.frame(gene = names(genes_sd), sd = as.numeric(genes_sd[1,]))
# order and keep the gene with the higher sd
gene_rank_best = genes_sd[order(-genes_sd$sd),][1:n_col_5,]
```

```
# show these gene
knitr::kable(head(gene_rank_best))
```

	gene	sd
1297	X38514_at	2.359599
653	X37280_at	1.633646
560	X41266_at	1.506192
1163	X1065_at	1.347584
918	X36575_at	1.346477
1248	X34362_at	1.332874

```
# get the column indexes and add response variable index
best_gene_idx = c(as.numeric(rownames(gene_rank_best)),2001)
# filter df
gene_df_5perc = gene_df_noID[,best_gene_idx]
knitr::kable(head(gene_df_5perc[,1:10]))
```

X38514_at	X37280_at	X41266_at	X1065_at	X36575_at	X34362_at	X40082_at	X33284_at	X37625_at	X40992_s_at
8.783013	6.793318	8.204418	8.265208	5.657863	8.716485	6.654734	8.770903	7.226242	6.840260
11.040864	5.035856	6.996015	6.693888	4.787349	6.260601	4.002936	5.850621	4.612264	4.309937
9.321789	7.515064	9.672395	8.625596	7.924031	7.892169	6.409294	7.620252	4.749547	8.443638
7.746815	8.138967	9.941716	9.721094	6.025907	10.209466	6.285863	6.184619	4.805881	7.868706
7.583775	7.561208	9.990832	8.970780	7.197957	9.188576	7.837010	6.116539	4.831491	5.766938
11.442418	8.227699	8.513482	6.277616	4.703743	6.612336	4.741875	6.082226	4.567798	5.454003

After this procedure, our dataset has 100 features (compared to 2000 in the full version).

We can finally repeat all the analysis training and testing SVM models on the filtered data.

SVC (linear kernel SVM) Starting from SVC, we can directly use nested cross-validation in order to get the average accuracy over multiple train-test partitions. As in the previous section, we will use 10-fold cross-validation for both parameter tuning and models evaluation.

```
# set seed for reproducible folds
set.seed(seed)
# get 10 set of index for training sets composition
gene_flds_2 = createFolds(gene_df_5perc$y, k = 10, list = T, returnTrain = T)

# initialize accuracy list
accuracy_CV_svc_sd = c()
#initialize precision lists
precision_svc_1_sd = c()
precision_svc_neg1_sd = c()
# initialize optimal costs list
optimal_cost_svc_sd = c()

# iterate over the folds
for (f in gene_flds_2) {
  # set seed for reproducible results
  set.seed(seed)

  # define index based on the i fold
  train_idx = f
  # split train and test sets according to the indexes
  train = gene_df_5perc[train_idx,]
  test = gene_df_5perc[-train_idx,]
  y_ts = test$y
```

```
  # tune cost parameter
  tune.out = tune(svm, y ~ ., data=train, kernel="linear",
                  ranges=list(cost=c(0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1,
                                     1, 5, 10, 100)))
  # summary(tune.out)
  # save the optimal cost parameter
  optimal_cost_svc_sd = append(optimal_cost_svc_sd, tune.out$best.parameters$cost)

  # extract optimal model
  bestmod = tune.out$best.model
  # make prediction using optimal model
  ypred = predict(bestmod, test)
  # calculate accuracy
  acc = mean(ypred == y_ts)
  # add accuracy to cv accuracy list
  accuracy_CV_svc_sd = append(accuracy_CV_svc_sd, acc)


  # precision for class 1
  prec_1 = table(ypred, y_ts)[2,2]/sum(table(ypred, y_ts)[,2])
  precision_svc_1_sd = append(precision_svc_1_sd, prec_1)
  # precision for class -1
  prec_neg1 = table(ypred, y_ts)[1,1]/sum(table(ypred, y_ts)[,1])
  precision_svc_neg1_sd = append(precision_svc_neg1_sd, prec_neg1)
}

# mean accuracy for SVC
mean(accuracy_CV_svc_sd)
```

```
## [1] 0.8501984
```

```
# precision class 1
mean(precision_svc_1_sd)
```

```
## [1] 0.8416667
```

```
# precision class -1
mean(precision_svc_neg1_sd)
```

```
## [1] 0.855
```

Note how, after reducing the number of features, computation is actually much faster.

The accuracy increased compared to the SVC trained on the complete dataset, reaching 85%. In addition, the precision of the classes has also improved and, besides that, their values are now slightly more similar than before.

```
# optimal models costs
optimal_cost_svc_sd
```

```
##  [1] 0.100 0.010 0.005 0.100 0.100 0.010 0.100 0.010 0.010 0.010
```

The most popular optimal value for `cost` parameter are now 0.01 and 0.1.

Polynomial kernel SVM Let us now use the same nested cross-validation schema to test the polynomial kernel SVM performance:

```
# initialize accuracy list
accuracy_CV_poly_sd = c()
# initialize precision lists
precision_poly_1_sd = c()
precision_poly_neg1_sd = c()
# initialize optimal costs list
optimal_cost_poly_sd = c()
optimal_degree_poly_sd = c()
```

```
# iterate over the folds
for (f in gene_flds_2) {
  # set seed for reproducible results
  set.seed(seed)
  # define index based on the i fold
  train_idx = f
  # split train and test sets according to the indexes
  train = gene_df_5perc[train_idx,]
  test = gene_df_5perc[-train_idx,]
  y_ts = test$y

  # tune cost parameter
  tune.out = tune(svm, y ~ ., data=train, kernel="polynomial",
                  ranges=list(cost=c(0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1,
                                     1, 5, 10, 100),
                              degree=c(2, 3, 4, 5, 6)))
  # summary(tune.out)
  # save the optimal cost parameter
  optimal_cost_poly_sd = append(optimal_cost_poly_sd, tune.out$best.parameters$cost)
  optimal_degree_poly_sd = append(optimal_degree_poly_sd, tune.out$best.parameters$degree)

  # extract optimal model
  bestmod = tune.out$best.model
  # make prediction using optimal model
  ypred = predict(bestmod, test)
  # calculate accuracy
  acc = mean(ypred == y_ts)
  # add accuracy to cv accuracy list
  accuracy_CV_poly_sd = append(accuracy_CV_poly_sd, acc)

  # precision for class 1
  prec_1 = table(ypred, y_ts)[2,2]/sum(table(ypred, y_ts)[,2])
  precision_poly_1_sd = append(precision_poly_1_sd, prec_1)
  # precision for class -1
  prec_neg1 = table(ypred, y_ts)[1,1]/sum(table(ypred, y_ts)[,1])
  precision_poly_neg1_sd = append(precision_poly_neg1_sd, prec_neg1)
}
# CV accuracy
mean(accuracy_CV_poly_sd)
```

```
## [1] 0.702381
```

```
# precision class 1
mean(precision_poly_1_sd)
```

```
## [1] 0.9166667
```

```
# precision class -1
mean(precision_poly_neg1_sd)
```

```
## [1] 0.515
```

Again, the model performs better than the one trained using all the features. However, the accuracy is still lower than both the SVC (the previous and the one trained on the full features dataset).

With regard to class precision, it rose for both classes, but especially for class 1, which reaches a value of 0.91. The model have in fact more trouble with class -1 classification.

```
# optimal cost for polynomial SVM
optimal_cost_poly_sd
```

```
##  [1] 5 5 5 5 5 5 5 5 5 5
```

```
# optimal degree for polynomial SVM
optimal_degree_poly_sd
```

```
##  [1] 3 3 3 3 3 3 3 3 3 3
```

The optimal model parameters for polynomial SVM are 5 for `cost` and 3 for `degree`. In the case of degree, the optimum value is the same as for the model trained on all features.

Radial kernel The last model we will examine is the SVM with radial kernel. Again, we directly use a nested 10-folds cross-validation in order to evaluate the performances.

```
# initialize accuracy list
accuracy_CV_rad_sd = c()
# initialize precision lists
precision_rad_1_sd = c()
precision_rad_neg1_sd = c()
# initialize optimal costs list
optimal_cost_rad_sd = c()
# initialize optimal gamma list
optimal_gamma_rad_sd = c()

# iterate over the folds
for (f in gene_flds_2) {
  # set seed for reproducible results
  set.seed(42)
  # define index based on the i fold
  train_idx = f
  # split train and test sets according to the indexes
  train = gene_df_5perc[train_idx,]
  test = gene_df_5perc[-train_idx,]
  y_ts = test$y

  # tune cost parameter
  tune.out = tune(svm, y ~ ., data=train, kernel="radial",
                  ranges=list(cost=c(0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1,
                                     1, 5, 10, 100),
                              gamma=c(0.1, 0.5, 1, 2, 3, 4, 6)))
  # summary(tune.out)
  # save the optimal cost parameter
  optimal_cost_rad_sd = append(optimal_cost_rad_sd, tune.out$best.parameters$cost)
  optimal_gamma_rad_sd = append(optimal_gamma_rad_sd, tune.out$best.parameters$gamma)

  # extract optimal model
  bestmod = tune.out$best.model
  # make prediction using optimal model
  ypred = predict(bestmod, test)
  # calculate accuracy
  acc = mean(ypred == y_ts)
  # add accuracy to cv accuracy list
  accuracy_CV_rad_sd = append(accuracy_CV_rad_sd, acc)

  # precision for class 1
  prec_1 = table(ypred, y_ts)[2,2]/sum(table(ypred, y_ts)[,2])
  precision_rad_1_sd = append(precision_rad_1_sd, prec_1)
  # precision for class -1
  prec_neg1 = table(ypred, y_ts)[1,1]/sum(table(ypred, y_ts)[,1])
  precision_rad_neg1_sd = append(precision_rad_neg1_sd, prec_neg1)
}
# CV accuracy
mean(accuracy_CV_rad_sd)
```

```
## [1] 0.5325397
```

```
# precision for class -1
mean(precision_rad_neg1_sd)
```

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

```
# precision for class 1
mean(precision_rad_1_sd)
```

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

In this case, the performance is exactly the same as the model trained on the full feature set, reaching an accuracy of 0.53 in both cases. The class precision values are the same too: 1 for class -1 and 0 for class 1. Hence, again, the model always predicts class -1. Furthermore, even the most popular values for optimal parameters (`cost` and `gamma`) are essentially the same.

Indeed, in this case, the model does not seem to be affected by the features reduction.

3. Draw some conclusions from the analyses that you have conducted.

As we have seen in the previous section, by decreasing the number of features we have not only decreased the computational effort, but in most cases also the accuracy.

```
results_df_2 = data.frame(model=c("SVC/SVM linear kernel (5% top sd fatures)",
                                  "SVM polynomial kernel (5% top sd fatures)",
                                  "SVM radial kernel (5% top sd fatures)"),
                          CV_accuracy=c(mean(accuracy_CV_svc_sd),
                                        mean(accuracy_CV_poly_sd),
                                        mean(accuracy_CV_rad_sd)),
                          precision_class_1=c(mean(precision_svc_1_sd),
                                              mean(precision_poly_1_sd),
                                              mean(precision_rad_1_sd)),
                          precision_class_neg1=c(mean(precision_svc_neg1_sd),
                                                 mean(precision_poly_neg1_sd),
                                                 mean(precision_rad_neg1_sd)))

results_df = rbind(results_df, results_df_2)
knitr::kable(results_df[order(-results_df$CV_accuracy),])
```

	model	CV_accuracy	precision_class_1	precision_class_neg1
4	SVC/SVM linear kernel (5% top sd fatures)	0.8501984	0.8416667	0.855
1	SVC/SVM linear kernel	0.7934524	0.7833333	0.805
5	SVM polynomial kernel (5% top sd fatures)	0.7023810	0.9166667	0.515
2	SVM polynomial kernel	0.5498016	0.6500000	0.465
3	SVM radial kernel	0.5325397	0.0000000	1.000
6	SVM radial kernel (5% top sd fatures)	0.5325397	0.0000000	1.000

As already mentioned, the SVC models achieve the best performance in both the full and filtered features cases, reaching a quite high accuracy and precision (for both classes).

This is followed by the polynomial SVM models that achieve a fairly good performance in the case of the filtered dataset, while the model trained on the complete dataset achieves just over 50% accuracy. Despite the sufficiently good accuracy in the first case, the precision values of the two classes are unbalanced, indicating that the model probably tends to always assign the new observation to class 1.

The weakest models are the radial SVMs that achieve an accuracy of just over 50%. In particular, this value does not change by training the model on filtered features. Besides that, the accuracy of this model is close to the observed frequency of class -1. In fact, it assigns every observation to that class, thus causing a total unbalance in classes precision. For this reason we can say that this model was not able to capture the complexity and any pattern in our data.

In conclusion, these results were somewhat expected, since, as we have proven, the data are linearly separable (even in the case of filtered features); therefore we had already guessed that the optimal SVM model would be the one with a linear kernel.

Lastly, in addition to the accuracy-based assessment of the models, it would be appropriate to display the ROC curves and thus provide the AUC measurement. They represents important diagnostic tools especially when dealing with unbalanced classes. However, as we noted in the data exploration, the classes of our response variable are rather

balanced. Still, this would be useful to better understand the behaviour of models that report quite unbalanced precision values and how their performances change with different probability cut-offs. For this reason let us build and display the models ROC curves using the filtered features dataset.

First we build a function to store false positive and true positive rates at each cut-offs threshold:

```
# define function that take as input models predicted value and groundtruth
roc_dfs = function(pred , truth , model) {
  predob = prediction(pred , truth)
  perf = performance(predob , "tpr", "fpr")
  # store fpr, tpr and cuoffs in a df
  roc = data.frame(cutoffs=perf@alpha.values[[1]], fpr=perf@x.values[[1]],
                   tpr=perf@y.values[[1]], model=model)
  return(roc)
}
```

Now we can calculate an optimal model sample for each SVM model using a train-test partition (splitting the filtered features dataset):

```
# set the seed for reproducible partitions
set.seed(seed)

# split data into training and test sets
split_2 = initial_split(gene_df_5perc, prop=0.7)
x_train_2 = training(split_2)
x_test_2 = testing(split_2)
y_test_2 = x_test_2$y

# set seed for reproducible results
set.seed(seed)

# tune SVC
tune.svc = tune(svm, y ~ ., data=x_train_2, kernel="linear",
                ranges=list(cost=c(0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1,
                                   1, 5, 10, 100)), decision.values = T)
# extract the optimal model
bestmod_svc = tune.svc$best.model
# extract predictions
fitted_svc = attributes(predict(bestmod_svc , x_test_2, decision.values = TRUE))$decision.values

# tune polynomial SVM
tune.poly = tune(svm, y ~ ., data=x_train_2, kernel="polynomial",
                ranges=list(cost=c(0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1,
                                   1, 5, 10, 100),
                            degree=c(2, 3, 4, 5, 6)), decision.values = T)
# extract optimal model
bestmod_poly = tune.poly$best.model
# extract predictions
fitted_poly = attributes(predict(bestmod_poly , x_test_2, decision.values = TRUE))$decision.values

# tune radial SVM
tune.rad = tune(svm, y ~ ., data=x_train_2, kernel="radial",
                ranges=list(cost=c(0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1, 1, 5, 10, 100, 150),
                            gamma=c(0.1, 0.5, 1, 2, 3, 4, 6)), decision.values = T)
# extract optimal model
bestmod_rad = tune.rad$best.model
# extract predictions
fitted_rad = attributes(predict(bestmod_rad , x_test_2, decision.values = TRUE))$decision.values
```

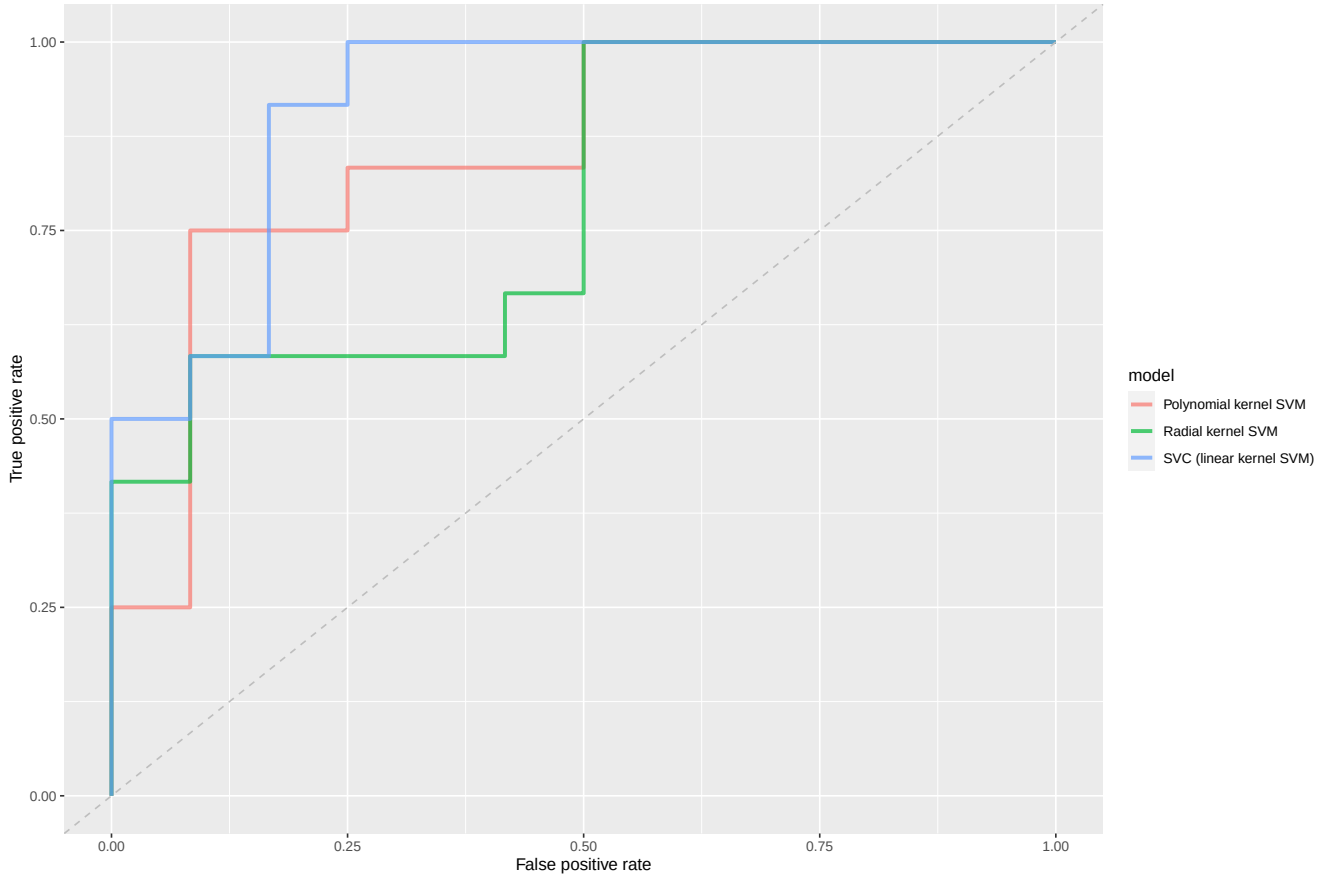
And we can finally apply the previous defined function and plot the SVMs ROC curves:

```
# models ROC data
roc_svc = roc_dfs(-fitted_svc, y_test_2, model="SVC (linear kernel SVM)")
roc_poly = roc_dfs(-fitted_poly, y_test_2, model="Polynomial kernel SVM")
```

```
roc_rad = roc_dfs(-fitted_rad, y_test_2, model="Radial kernel SVM")

# bind the results
roc_models = rbind(roc_svc, roc_poly, roc_rad)

# plot ROC curves
ggplot(roc_models, aes(x=fpr, y=tpr, group=model, color=model)) +
  geom_path(size=1.2, alpha=0.7) +
  xlab("False positive rate") +
  ylab("True positive rate") +
  # line corresponding to the "no information" classifier (e.g., random guess)
  geom_abline(linetype = "dashed", color="gray")
```



The ROC plot gives us a graphical idea of how the models perform with various probability cut-offs and make easy to identify the best decision threshold for each model (represented by the nearest point to the top left edges of the graph). In these terms a way to quantify the performance of each model is to compute the area under the ROC curve, named AUC:

```
# define a function to compute AUC
auc = function(pred , truth) {
  predob = prediction(pred , truth)
  auc_ROCR = performance(predob, measure = "auc")
  auc = auc_ROCR@y.values[[1]]
  return(auc)
}

# SVC AUC
auc(-fitted_svc, y_test_2)
```

```
## [1] 0.9166667
```

```
# Polynomial SVM AUC
auc(-fitted_poly, y_test_2)
```

```
## [1] 0.8541667
```

```
# Radial SVM AUC
auc(-fitted_rad, y_test_2)
```

```
## [1] 0.7847222
```

As we could already gather from the ROC curves plot, the model with the highest AUC value (0.91) is the SVC. This is followed by the Polynomial SVM, with a value of 0.85 and Radial SVM with 0.78.

We can observe that even in terms of AUC the models ranking based on CV accuracy does not change and the SVC is the best in this context as well.

However, the AUC value was calculated on a single train-test partition (using cross-validation was also too laborious and complex here), so we know that these values could vary, but probably without ever affecting the ranking.