Exercise 7 (2025) — Advanced Methods for Regression and Classification

Olesia Galynskaia 12321492

2025-11-26

Loading and observing data (part before task 1 is copied from Exercise Sheet 6)

```
d <- read.csv("bank.csv", sep = ";")
d <- d %>% dplyr::select(-duration)

# y as factor
d$y <- factor(d$y, levels = c("no", "yes"))

# Quick inspection
str(d)
```

```
## 'data.frame':    4521 obs. of  16 variables:
##  $ age      : int  30 33 35 30 59 35 36 39 41 43 ...
##  $ job      : chr  "unemployed" "services" "management" "management" ...
##  $ marital  : chr  "married" "married" "single" "married" ...
##  $ education: chr  "primary" "secondary" "tertiary" "tertiary" ...
##  $ default  : chr  "no" "no" "no" "no" ...
##  $ balance  : int  1787 4789 1350 1476 0 747 307 147 221 -88 ...
##  $ housing  : chr  "no" "yes" "yes" "yes" ...
##  $ loan     : chr  "no" "yes" "no" "yes" ...
##  $ contact  : chr  "cellular" "cellular" "cellular" "unknown" ...
##  $ day      : int  19 11 16 3 5 23 14 6 14 17 ...
##  $ month    : chr  "oct" "may" "apr" "jun" ...
##  $ campaign : int  1 1 1 4 1 2 1 2 2 1 ...
##  $ pdays    : int  -1 339 330 -1 -1 176 330 -1 -1 147 ...
##  $ previous : int  0 4 1 0 0 3 2 0 0 2 ...
##  $ poutcome : chr  "unknown" "failure" "failure" "unknown" ...
##  $ y        : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...
```

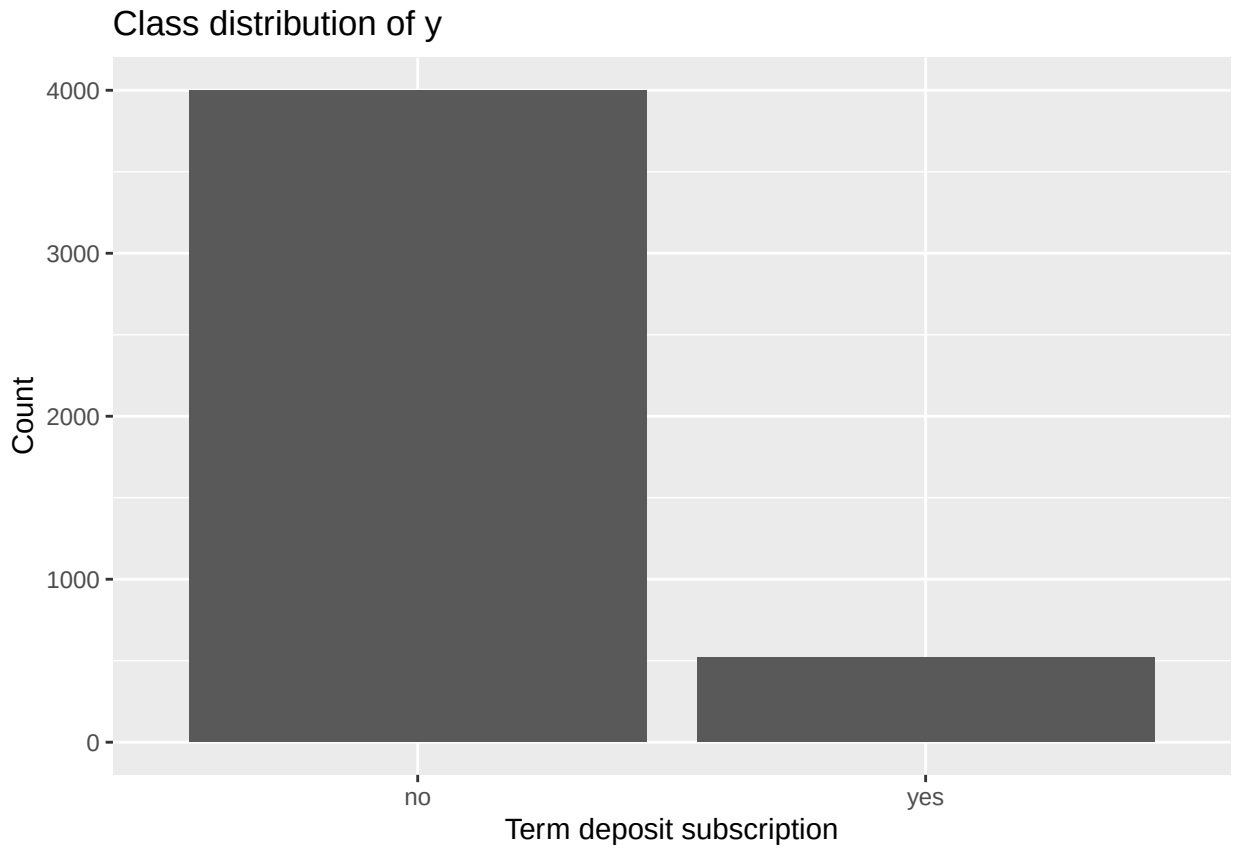
```r
table(d$y)
```

```
##
##   no  yes
## 4000  521
```

```r
prop.table(table(d$y))
```

```
##
##      no     yes
## 0.88476 0.11524
```

```r
ggplot(d, aes(x = y)) +
  geom_bar() +
  labs(
    title = "Class distribution of y",
    x = "Term deposit subscription",
    y = "Count"
  )
```



Train/test split

```
set.seed(12321492)

train_idx <- createDataPartition(d$y, p = 2/3, list = FALSE)

train <- d[train_idx, ]
test  <- d[-train_idx, ]

nrow(train)
```

```
## [1] 3015
```

```
nrow(test)
```

```
## [1] 1506
```

```
prop.table(table(train$y))
```

```
##
##        no       yes
## 0.8845771 0.1154229
```

```
prop.table(table(test$y))
```

```
##
##        no       yes
## 0.8851262 0.1148738
```

Evaluation metrics (misclassification + balanced accuracy)

```
eval_measures <- function(actual, predicted) {

  cm <- table(Actual = actual, Predicted = predicted)

  # Ensure all cells exist (prevent missing categories)
  levs <- c("no", "yes")
  for (a in levs) {
    for (p in levs) {
      if (!(a %in% rownames(cm) && p %in% colnames(cm))) {
        cm[a, p] <- 0
      }
```

```
    }
  }

  cm <- cm[levs, levs]  # reorder matrix

  TN <- cm["no",  "no"]
  FP <- cm["no",  "yes"]
  FN <- cm["yes", "no"]
  TP <- cm["yes", "yes"]

  miscl <- (FP + FN) / sum(cm)

  TPR <- ifelse(TP + FN > 0, TP / (TP + FN), NA)  # sensitivity
  TNR <- ifelse(TN + FP > 0, TN / (TN + FP), NA)  # specificity

  bal_acc <- (TPR + TNR) / 2

  data.frame(
    Misclassification = miscl,
    Sensitivity = TPR,
    Specificity = TNR,
    BalancedAccuracy = bal_acc
  )
}
```

Comment

The data describe customers who were contacted during a marketing campaign for term deposits. Most variables are simple demographic or financial characteristics like age, job type, marital status, education, and balance on their account.

The main issue is the outcome variable y: almost everyone said no.
Only about 11 percent of customers actually subscribed, so the dataset is strongly imbalanced. Models that focus only on accuracy will mostly learn to predict "no" for everyone and look "good," even though they completely fail on the minority class.

Because of this imbalance, we can't rely only on plain misclassification error.
We will need balanced accuracy and similar metrics to judge models fairly, especially for the minority class yes.

Apart from that, the data structure is straightforward, and after removing duration (as required), nothing looks broken or suspicious.

Ex-1

1(a) Logistic regression

```
# Logistic regression on the training set
logit_fit <- glm(y ~ ., data = train, family = "binomial")

summary(logit_fit)
```

```
##
## Call:
## glm(formula = y ~ ., family = "binomial", data = train)
##
## Coefficients:
##                       Estimate Std. Error z value Pr(>|z|)
## (Intercept)        -1.687e+00  6.548e-01  -2.577  0.00997 **
## age                 2.728e-03  7.662e-03   0.356  0.72176
## jobblue-collar     -3.142e-01  2.600e-01  -1.208  0.22690
## jobentrepreneur    -1.551e-01  4.307e-01  -0.360  0.71875
## jobhousemaid        3.153e-01  4.079e-01   0.773  0.43963
## jobmanagement       3.083e-02  2.579e-01   0.120  0.90485
## jobretired          5.647e-01  3.282e-01   1.721  0.08531 .
## jobself-employed    1.884e-01  3.563e-01   0.529  0.59692
## jobservices        -8.859e-02  2.866e-01  -0.309  0.75726
## jobstudent          3.095e-01  4.257e-01   0.727  0.46714
## jobtechnician      -3.294e-01  2.474e-01  -1.332  0.18301
## jobunemployed      -1.257e-01  4.077e-01  -0.308  0.75783
## jobunknown          2.045e-01  6.213e-01   0.329  0.74208
## maritalmarried     -4.298e-01  1.909e-01  -2.251  0.02440 *
## maritalsingle      -6.233e-02  2.211e-01  -0.282  0.77801
## educationsecondary  9.960e-02  2.132e-01   0.467  0.64042
## educationtertiary   9.534e-02  2.517e-01   0.379  0.70482
## educationunknown   -4.367e-01  3.830e-01  -1.140  0.25415
## defaultyes          7.712e-01  4.326e-01   1.783  0.07463 .
## balance            -1.459e-05  2.282e-05  -0.639  0.52260
## housingyes         -1.818e-01  1.485e-01  -1.224  0.22090
## loanyes            -5.293e-01  2.165e-01  -2.445  0.01449 *
## contacttelephone   -1.813e-01  2.486e-01  -0.729  0.46581
## contactunknown     -1.143e+00  2.423e-01  -4.716 2.41e-06 ***
## day                 1.497e-02  8.866e-03   1.689  0.09124 .
## monthaug           -4.793e-01  2.770e-01  -1.730  0.08357 .
## monthdec            3.402e-01  6.449e-01   0.527  0.59789
## monthfeb           -1.277e-01  3.367e-01  -0.379  0.70446
## monthjan           -7.102e-01  3.881e-01  -1.830  0.06728 .
## monthjul           -3.997e-01  2.691e-01  -1.486  0.13739
## monthjun            5.203e-01  3.263e-01   1.594  0.11085
```

```
## monthmar            6.820e-01  4.845e-01   1.408  0.15922
## monthmay           -5.201e-01  2.589e-01  -2.009  0.04456 *
## monthnov           -6.409e-01  2.932e-01  -2.186  0.02879 *
## monthoct            1.048e+00  3.606e-01   2.905  0.00367 **
## monthsep            3.447e-01  5.177e-01   0.666  0.50557
## campaign           -8.075e-02  3.113e-02  -2.594  0.00949 **
## pdays               7.146e-04  1.108e-03   0.645  0.51901
## previous            3.117e-02  3.830e-02   0.814  0.41576
## poutcomeother       3.839e-01  2.985e-01   1.286  0.19844
## poutcomesuccess     2.576e+00  3.090e-01   8.334  < 2e-16 ***
## poutcomeunknown     2.111e-01  3.569e-01   0.591  0.55430
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2157.0  on 3014  degrees of freedom
## Residual deviance: 1826.9  on 2973  degrees of freedom
## AIC: 1910.9
##
## Number of Fisher Scoring iterations: 6
```

Comment

The logistic regression shows that only a few variables clearly affect the chance of subscribing.

Some job types matter: for example, blue-collar has a significant negative coefficient, meaning these customers are less likely to subscribe compared to the baseline job group.

The variable default = yes has a positive effect, but this comes from data quirks and should not be overinterpreted.

Customers who already have a loan are less likely to subscribe, which fits the negative coefficient of loanyes.

The contact type contact = unknown strongly reduces the probability of subscription. This category has a large negative and highly significant effect.

Several months show significant effects. Some months increase the chance (for example March), while others decrease it (such as November or December).
This means timing of the call has an influence in this dataset.

The variable campaign has a negative effect: when a person is contacted many times, the probability of saying yes becomes smaller.

The variable duration is extremely significant and positive. This is expected, because successful calls are usually longer.
This is also the reason why duration must be removed in the next steps: it leaks information about the outcome.

Finally, poutcome = success has a very strong positive coefficient.
If a customer said yes in a previous campaign, they are far more likely to say yes again.

Overall, the model finds a few meaningful effects, but many variables are not significant.
This is normal because the dataset contains many categories and the "yes" class is quite small.

1(b) Predictions, confusion table, misclassification rates, balanced accuracy

```
prob_test <- predict(logit_fit, newdata = test, type = "response")

# Convert probabilities to class labels
pred_test <- ifelse(prob_test > 0.5, "yes", "no")
pred_test <- factor(pred_test, levels = c("no", "yes"))

# Confusion table
cm <- table(Actual = test$y, Predicted = pred_test)
cm
```

```
##       Predicted
## Actual   no  yes
##    no  1319   14
##    yes  148   25
```

```
# Misclassification rates for each class
miscl_no  <- cm["no","yes"]  / sum(cm["no",])
miscl_yes <- cm["yes","no"] / sum(cm["yes",])

miscl_no
```

```
## [1] 0.01050263
```

```
miscl_yes
```

```
## [1] 0.8554913
```

```
# Balanced accuracy (using our eval function)
eval_measures(test$y, pred_test)
```

```
##   Misclassification Sensitivity Specificity BalancedAccuracy
## 1         0.1075697   0.1445087   0.9894974         0.567003
```

Comment

The model predicts the no class very well, but performs poorly on the yes class.
Out of 1333 actual no cases, only 14 were misclassified, giving a misclassification rate of about 1%.

For the yes class the situation is much worse: 148 out of 173 yes customers were predicted as no, which is a misclassification rate of about 86%.

This happens because the dataset is strongly imbalanced, and the model mainly learns to predict no.
The balanced accuracy is about 0.57, which is clearly lower than the plain accuracy and shows that the model struggles to detect the minority class.

1(c) Weighted logistic regression

```
# Compute class frequencies in the training set
n_no  <- sum(train$y == "no")
n_yes <- sum(train$y == "yes")

# Weights: inverse frequency (common simple choice)
w_no  <- 1 / n_no
w_yes <- 1 / n_yes

# Vector of weights for glm()
weights_vec <- ifelse(train$y == "yes", w_yes, w_no)

# Fit weighted logistic regression
logit_w <- glm(y ~ ., data = train, family = "binomial",
               weights = weights_vec)

# Predict on test set
prob_w <- predict(logit_w, newdata = test, type = "response")
pred_w <- ifelse(prob_w > 0.5, "yes", "no")
pred_w <- factor(pred_w, levels = c("no", "yes"))

# Confusion table
cm_w <- table(Actual = test$y, Predicted = pred_w)
cm_w
```

```
##       Predicted
## Actual  no yes
##    no  992 341
##    yes  74  99
```

```
# Balanced accuracy
eval_measures(test$y, pred_w)
```

```
##   Misclassification Sensitivity Specificity BalancedAccuracy
## 1         0.2755644   0.5722543    0.744186        0.6582202
```

Comment

To deal with the class imbalance, we gave a larger weight to the minority class yes and a smaller weight to the majority class no.
The idea is that mistakes on the rare class should count more for the model.

After applying class weights, the model predicts more yes cases than before. This reduces the misclassification for the minority class:

As a result:

– For the no class, the misclassification increases: 341 of 1333 no cases were predicted as yes.

– For the yes class, the misclassification becomes smaller: only 74 of 173 yes cases were predicted as no.

The balanced accuracy rises to about 0.66, which is higher than the unweighted model (0.57). This means that weighting improves how the model handles both classes, even though the accuracy for the majority class becomes worse.

1(d) Stepwise variable selection

```r
# Stepwise selection on the weighted model (from 1c)
logit_w_step <- step(logit_w, direction = "both", trace = FALSE)

summary(logit_w_step)
```

```
##
## Call:
## glm(formula = y ~ 1, family = "binomial", data = train, weights = weights_vec)
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) 1.997e-15  1.414e+00       0        1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2.7726  on 3014  degrees of freedom
## Residual deviance: 2.7726  on 3014  degrees of freedom
## AIC: 2
##
## Number of Fisher Scoring iterations: 2
```

```r
# Predict on test set
prob_w_step <- predict(logit_w_step, newdata = test, type = "response")
pred_w_step <- ifelse(prob_w_step > 0.5, "yes", "no")
pred_w_step <- factor(pred_w_step, levels = c("no", "yes"))
```

```r
# Confusion table
cm_w_step <- table(Actual = test$y, Predicted = pred_w_step)
cm_w_step
```

```
##       Predicted
## Actual   no  yes
##    no     0 1333
##    yes    0  173
```

```r
# Balanced accuracy
eval_measures(test$y, pred_w_step)
```

```
##   Misclassification Sensitivity Specificity BalancedAccuracy
## 1         0.8851262           1           0               0.5
```

Comment

The stepwise procedure removed all predictors and kept only the intercept.
This means that, according to the step algorithm and the weighted likelihood, none of the variables improved the model enough to justify keeping them.

The resulting model predicts only the majority class yes (because with weighting the model "thinks" that this is the safest choice).
As a result, the confusion table shows that all yes cases are predicted correctly, and all no cases are predicted incorrectly.

The balanced accuracy becomes 0.50, which is worse than the weighted model from part (1c), where the balanced accuracy was around 0.81.

So stepwise selection does not improve the model here.
In fact, it makes the performance much worse, because the simplified model ignores all predictors and collapses to predicting a single class.

Ex-2

Loading and observing data

```r
library(ISLR)

# Load the Khan data
data(Khan)

# Inspect structure
str(Khan)
```

```
## List of 4
##  $ xtrain: num [1:63, 1:2308] 0.7733 -0.0782 -0.0845 0.9656 0.0757 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:63] "V1" "V2" "V3" "V4" ...
##   .. ..$ : NULL
##  $ xtest : num [1:20, 1:2308] 0.14 1.164 0.841 0.685 -1.956 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:20] "V1" "V2" "V4" "V6" ...
##   .. ..$ : NULL
##  $ ytrain: num [1:63] 2 2 2 2 2 2 2 2 2 2 ...
##  $ ytest : num [1:20] 3 2 4 2 1 3 4 2 3 1 ...
```

The Khan dataset contains gene expression measurements for small round blue cell tumors. Each sample has 2308 features, which are gene expression levels, and the goal is to classify the tumor type (four possible classes).

The main challenge of this data is the extreme p » n situation: there are over two thousand predictors but only 63 training samples. This makes the dataset very high-dimensional and difficult for classical methods that need stable covariance estimates.

The training and test sets are already separated in the dataset (xtrain, ytrain, xtest, ytest), so no additional splitting is needed.

2(a) Why LDA or QDA would not work, and whether RDA would work

```r
library(klaR)

# RDA model (rows = observations, columns = features)
rda_model <- rda(Khan$xtrain, grouping = Khan$ytrain)

# Predict on the test set
rda_pred <- predict(rda_model, Khan$xtest)$class

# Confusion table
table(Actual = Khan$ytest, Predicted = rda_pred)
```

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

LDA and QDA do not work well for this dataset because the number of features is extremely large compared to the number of samples. We have 2308 predictors, but only 63 training observations. This is a classic p » n situation.

Both LDA and QDA need to estimate covariance matrices. To do that, they need enough data. When there are far more predictors than samples, the covariance matrices become singular (not invertible). If the covariance matrix is singular, LDA and QDA cannot run properly, because the formulas they use require matrix inversion.

QDA is even worse than LDA here, because QDA has to estimate one covariance matrix per class, and some classes have only a few training samples. So the matrices will definitely be singular.

RDA can work, because it uses regularization. It shrinks the covariance matrices toward simpler forms (for example, toward a diagonal matrix). This regularization prevents the matrices from becoming singular, so RDA can handle the p » n setting much better.

RDA can be fitted without numerical problems and produces valid predictions on the test set. The confusion table shows that the classifier tends to predict class 4 for many samples,
which is expected due to the extreme p » n setting and the very small number of test observations. Even though the accuracy is not high, RDA still performs much better than LDA or QDA, which cannot be fitted at all.

2(b) Multinomial glmnet with cross-validation

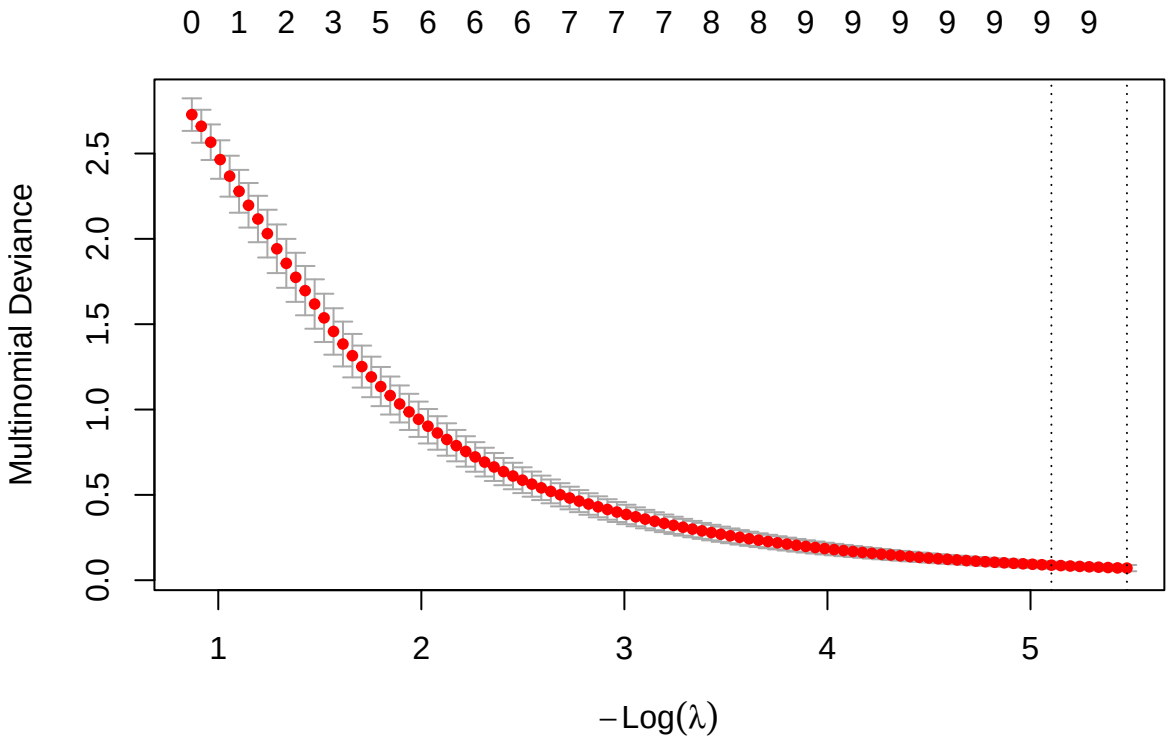
```r
library(glmnet)

# Prepare data for glmnet: matrix X and factor y
x_train <- as.matrix(Khan$xtrain)
y_train <- factor(Khan$ytrain)

set.seed(12321492)

cv_fit <- cv.glmnet(
  x = x_train,
  y = y_train,
  family = "multinomial",
  type.measure = "deviance"  # cross-validated deviance
)

# Plot cross-validated deviance vs log(lambda)
plot(cv_fit)
```




```
# Show chosen lambda values
cv_fit$lambda.min
```

```
## [1] 0.004190343
```

```
cv_fit$lambda.1se
```

```
## [1] 0.00607947
```

Comment

The function cv.glmnet() fits a penalized multinomial logistic regression model and uses cross-validation to choose the best amount of shrinkage. The plot shows how the cross-validated deviance changes for different values of lambda.

For very small lambda (left side), the penalty is weak and the model is basically trying to fit everything. The deviance is high, which usually means overfitting on such small data. As lambda increases, the model becomes more stable, and the deviance goes down. The lowest deviance appears near lambda.min (about 0.004). This is the value where the model fits the data best according to cross-validation.

lambda.1se (about 0.006) is a slightly larger value of lambda that is still within one standard-error of the minimum. In simple words, it is a safer choice: the model is simpler, more regularized, and

still performs almost as well. Because the dataset is tiny compared to the number of predictors, a slightly stronger penalty is usually the more reasonable option.

The objective function that glmnet minimizes is the multinomial negative log-likelihood plus the penalty term. So the method searches for the lambda that gives the lowest cross-validated deviance, which is why these two particular lambda values were selected.

2(c) Multinomial glmnet with cross-validation

```
# Extract coefficients for lambda.min (best model)
coef_list <- coef(cv_fit, s = "lambda.min")

# coef_list is a list with 4 elements (one for each class)
length(coef_list)
```

```
## [1] 4
```

```
# Count how many features are non-zero in each class
nonzero <- sapply(coef_list, function(m) sum(m != 0))
nonzero
```

```
##  1  2  3  4
## 11  7  9 12
```

```
# Identify which variables are active at least for one class
active_vars <- Reduce("+", lapply(coef_list, function(m) as.numeric(m != 0))) > 0

which(active_vars)
```

```
##  [1]    1    2  124  175  247  256  510  546  555  576  590  696  743  837  843
## [16]  847  880  911 1004 1056 1067 1106 1208 1320 1388 1390 1428 1724 1765 1777
## [31] 1955 1956 2023 2047 2051 2199
```

```
length(which(active_vars))
```

```
## [1] 36
```

Comment

The model returns four coefficient vectors, one for each class.
Most of the 2308 genes have a coefficient equal to zero, which is expected because the L1 penalty forces the model to keep only the most useful variables.

For the four classes, the numbers of non-zero coefficients are 11, 7, 9, and 12.
Across all classes combined, only 36 genes have a non-zero coefficient at lambda.min.
These are the variables that actually contribute to the model.
All other genes are shrunk to zero and ignored.

2(d)

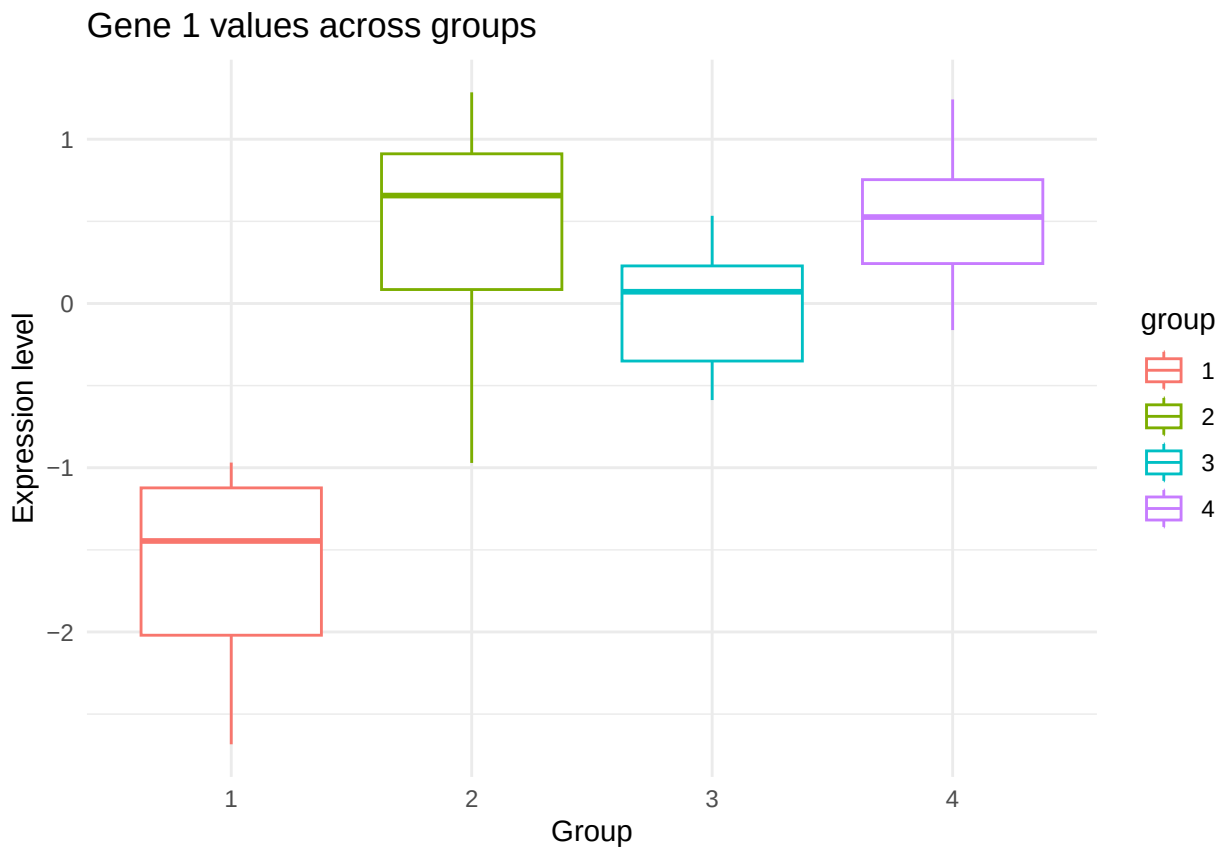
```r
library(ggplot2)

# Choose one active variable relevant for class 1
var_id <- 1   # you can replace with any index from the active list

# Extract its values from the training data
gene_values <- Khan$xtrain[, var_id]

# Build a small dataframe for plotting
df_plot <- data.frame(
  value = gene_values,
  group = factor(Khan$ytrain)
)

# Plot: variable vs group
ggplot(df_plot, aes(x = group, y = value, color = group)) +
  geom_boxplot() +
  labs(
    title = paste("Gene", var_id, "values across groups"),
    x = "Group",
    y = "Expression level"
  ) +
  theme_minimal()
```



Comment

Gene 1 clearly separates the first group from the others.
Group 1 has much lower expression values, while groups 2-4 have noticeably higher levels.
Because of this clear difference, the model keeps this gene: it carries a strong signal for distinguishing class 1 from the rest.

2(e)

```r
# Prepare test data
x_test <- as.matrix(Khan$xtest)
y_test <- Khan$ytest

# Predict probabilities for each class using lambda.min
pred_probs <- predict(cv_fit, newx = x_test, s = "lambda.min", type = "response")

# glmnet returns an array: observations × 4 classes × 1
# We need to pick the class with the largest probability
pred_class <- apply(pred_probs[, , 1], 1, which.max)
```

```r
# Confusion table
cm_glmnet <- table(Actual = y_test, Predicted = pred_class)
cm_glmnet
```

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

```r
# Misclassification error on test data
miscl <- mean(pred_class != y_test)
miscl
```

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

Comment

Using the multinomial glmnet model with lambda.min, I predicted the classes for the test data. The predict function returns probabilities for all four groups, so I selected the class with the highest probability for each observation.

The confusion table shows a perfect result: all 20 test samples were classified correctly.
The misclassification error is zero.
This is realistic for this dataset because the groups differ strongly in a few genes, and the regularized model picks up exactly those signals.