Homework 2 (ISYE 6501) — Question 3.1

Question 3.1 Answer

Sorry in advance to my peer reviewers! (— —) I like to write out my reasoning and can be… wordy… But I will try to be more brief this time!

First, I need to set up my environment!
`rm(list=ls())` is something I usually run at the very beginning of my time within R to ensure I am starting fresh and not somehow meshing variables I've used before on accident, possibly.
(To be honest though, I don't actually know if that is true. Does running that really clear my memory? If anyone reviewing my code knows, please let me know.)

```
 rm(list=ls())  # Uncomment only if starting fresh

library(kknn)
library(caret)
library(kernlab)

set.seed(16)  # Keep fixed for reproducibility, I chose 16 because it's my favorite number
```

`set.seed(16)` is just "locking in" my random number generator so that you (peers) can see the same results as me, since this makes it so that it produces the same sequence of 'random' numbers every time I (or a peer) knit this file.
The rest should look familiar, just loading in `kknn`, `kernlab`, and although the homework does not specify a package for splitting or scaling the data, I use `caret` to create better balanced data splits than I could create on my own, and also to scale predictors using only the training set, avoiding data leakage.

```
## [1] 654  11
```

```
##    A1    A2     A3    A8 A9 A10 A11 A12 A14   A15 R1
## 1   1 30.83  0.000 1.250  1   0   1   1 202     0  1
## 2   0 58.67  4.460 3.040  1   0   6   1  43   560  1
## 3   0 24.50  0.500 1.500  1   1   0   1 280   824  1
## 4   1 27.83  1.540 3.750  1   0   5   0 100     3  1
## 5   1 20.17  5.625 1.710  1   1   0   1 120     0  1
## 6   1 32.08  4.000 2.500  1   1   0   0 360     0  1
## 7   1 33.17  1.040 6.500  1   1   0   0 164 31285  1
## 8   0 22.92 11.585 0.040  1   1   0   1  80  1349  1
## 9   1 54.42  0.500 3.960  1   1   0   1 180   314  1
## 10  1 42.50  4.915 3.165  1   1   0   0  52  1442  1
## 11  1 22.08  0.830 2.165  0   1   0   0 128     0  1
## 12  1 29.92  1.835 4.335  1   1   0   1 260   200  1
## 13  0 38.25  6.000 1.000  1   1   0   0   0     0  1
```

```
## 14  1 48.08  6.040 0.040  0   1   0   1   0  2690  1
## 15  0 45.83 10.500 5.000  1   0   7   0   0     0  1
## 16  1 36.67  4.415 0.250  1   0  10   0 320     0  1
```

Second, I load the credit card dataset and confirm the dimensions (654 rows, 11 columns), and a few rows of the data. This is just a quick check I always like to do to make sure the file loaded correctly.

Now that the data is loaded in, and the libraries are set correctly, we can start for real!
I wanted to try a different way of making `knn` this time, as I saw a lot of variations during office hours, and reading different ways to go about scripting it online.
Although I did not directly take any one specific line of code, I did use the code shown during office hours, and from https://www.datacamp.com/tutorial/k-nearest-neighbors-knn-classification-with-r-tutorial as reference material.

```
df$R1 <- factor(df$R1, levels = c(0, 1))
table(df$R1)
```

```
##
##   0   1
## 358 296
```

`R1` is the response variable, that is just the title of the column given in the dataset, so I will use that as my variable name as well.
But, even though it is coded as 0/1, this is a classification problem, so I explicitly have to convert it to a factor.

```
bin_cols <- c("A1","A9","A10","A11","A12")
bin_cols <- intersect(bin_cols, names(df))

for (cc in bin_cols) {
  df[[cc]] <- factor(df[[cc]])
}

str(df)
```

```
## 'data.frame':    654 obs. of  11 variables:
##  $ A1 : Factor w/ 2 levels "0","1": 2 1 1 2 2 2 2 1 2 2 ...
##  $ A2 : num  30.8 58.7 24.5 27.8 20.2 ...
##  $ A3 : num  0 4.46 0.5 1.54 5.62 ...
##  $ A8 : num  1.25 3.04 1.5 3.75 1.71 ...
##  $ A9 : Factor w/ 2 levels "0","1": 2 2 2 2 2 2 2 2 2 2 ...
##  $ A10: Factor w/ 2 levels "0","1": 1 1 2 1 2 2 2 2 2 2 ...
##  $ A11: Factor w/ 23 levels "0","1","2","3",..: 2 7 1 6 1 1 1 1 1 1 ...
##  $ A12: Factor w/ 2 levels "0","1": 2 2 2 1 2 1 1 2 2 1 ...
##  $ A14: int  202 43 280 100 120 360 164 80 180 52 ...
##  $ A15: int  0 560 824 3 0 0 31285 1349 314 1442 ...
##  $ R1 : Factor w/ 2 levels "0","1": 2 2 2 2 2 2 2 2 2 2 ...
```

Here I am just creating a vector of column names, and ensuring `bin_cols` is only going to use actual, real column names (I am OCD).
Then I run a loop to get the column whose name is stored in the variable `cc` to convert it to a factor. I do this so it knows it is not a continuous numerical value.

```r
num_cols <- names(df)[sapply(df, is.numeric)]
num_cols <- setdiff(num_cols, "R1")

df[num_cols] <- scale(df[num_cols])
```

Now I run this, and set `num_cols` as a variable that includes all numeric columns, except for `R1` which is the response column, so that I can apply `scale()`.

```r
form <- as.formula(
  paste("R1 ~", paste(setdiff(names(df), "R1"), collapse = " + "))
)
```

Running this just constructs the formula `R1 ~ A1 + A2 + ... + A10`.

This block of code is doing `10-fold` cross-validation to decide the reasonable value of `k` for the `knn` classifier. The data is first split into 10 folds, and then for each value of `k` from [1:21], the model is trained on 9 folds and evaluated on the remaining fold (90%/10%).
I chose [1:21] because in the last homework, with this same data, we discovered that `k = (12, 15)`, and although it might be wrong to make assumptions since we are using a different method here, I cannot imagine that `k` will change more than 5 digits.
(Is it cheating to use knowledge we already know from Homework 1 to make assumptions on Homework 2? I hope not!)
This process is repeated so that each fold is used as the test set once, and the accuracies are averaged.
The value of `k` that gives the highest average accuracy is selected as the best `k` (`best_k_cv`), and both the selected `k` and its cross-validated accuracy are reported.

```r
K <- 10
fold_id <- sample(rep(1:K, length.out = nrow(df)))
k_grid <- 1:21

cv_acc <- sapply(k_grid, function(k) {

  fold_acc <- sapply(1:K, function(f) {
    tr <- df[fold_id != f, , drop = FALSE]
    te <- df[fold_id == f, , drop = FALSE]

    fit <- kknn(
      form,
      train = tr, # training
      test  = te, # testing
      k = k,
      distance = 2,
      scale = FALSE
    )

    pred <- fitted(fit)
    mean(pred == te$R1)
  })

  mean(fold_acc)
})

best_k_cv <- k_grid[which.max(cv_acc)]
best_k_cv
```

```
## [1] 18
```

```
max(cv_acc)
```

```
## [1] 0.8500466
```

Based on `10-fold` cross-validation, a `knn` classifier with `k = 18` does well, achieving [0.8500466] (85%) classification accuracy!
I would consider this a good classifier for Part 1!
Though this surprised me a little bit, I thought it would actually be a lot closer to Homework 1 `k = (12, 15)`, but it just goes to show how different methods of validation can change results ever so slightly.
(I also know that small <4 digit changes between k values as small as these really don't have that much of an overall or substantial effect at least on this dataset, I'm sure any k value between [15:20] would do alright!)

Now, onto splitting the data into the recommended (or just most common practice): [60%] training, [20%] validation, and [20%] test.
I once again use `set.seed(16)` to fix the randomness so the same rows are split each time (I am unsure if I really need to run this again here but just to be safe!)
'createDataPartition()' splits the data while preserving class balance in 'R1', and the first split was hard-coded to create the 60% training set, with the remaining 40% split evenly into the validation and test sets (20% each)
and as always, my favorite simple sanity check, I used dim() to check to confirm the split sizes were what I expected… and they were!

```
set.seed(16) # Keep fixed for reproducibility, I chose 16 because it's my favorite number

idx_train <- createDataPartition(df$R1, p = 0.60, list = FALSE)
train_df <- df[idx_train, , drop = FALSE]
temp_df  <- df[-idx_train, , drop = FALSE]

idx_valid <- createDataPartition(temp_df$R1, p = 0.50, list = FALSE)
valid_df <- temp_df[idx_valid, , drop = FALSE]
test_df  <- temp_df[-idx_valid, , drop = FALSE]

dim(train_df); dim(valid_df); dim(test_df)
```

```
## [1] 393  11
```

```
## [1] 131  11
```

```
## [1] 130  11
```

Because the total number of rows split into 60% (392.4 rows rounded up to 393 rows) left 261 rows to be split evenly, we ended up with 393 training, 131 validation, and 130 test.
As we learned in the lecture, the training set is used to fit the model, the validation set is used to tune `k`, and the test set is held out entirely until the very end.
The test set is only used once, so it gives a clean estimate of final model performance.

```
num_cols2 <- names(train_df)[sapply(train_df, is.numeric)]
num_cols2 <- setdiff(num_cols2, "R1")

sc <- preProcess(train_df[, num_cols2, drop = FALSE],
```

```
                  method = c("center", "scale"))

train_df[, num_cols2] <- predict(sc, train_df[, num_cols2])
valid_df[, num_cols2] <- predict(sc, valid_df[, num_cols2])
test_df[,  num_cols2] <- predict(sc, test_df[,  num_cols2])
```

Here, `num_cols2` is created to store the names of all numeric predictor columns in the training data. `setdiff(num_cols2, "R1")` should look familiar by now, I use this to remove the response variable so that only predictors are scaled as responses should never be scaled. `preProcess()` is then used to compute the mean and standard deviation, making sure to only use the training and validation sets.
The `predict()` portion applies this same transformation to the training, validation, and test sets.

```
k_grid2 <- 1:21

val_acc <- sapply(k_grid2, function(k) {

  fit <- kknn(
    form,
    train = train_df,
    test  = valid_df,
    k = k,
    distance = 2,
    scale = FALSE
  )

  pred <- fitted(fit)
  mean(pred == valid_df$R1)
})

best_k_val <- k_grid2[which.max(val_acc)]
best_k_val
```

```
## [1] 20
```

```
max(val_acc)
```

```
## [1] 0.8854962
```

`k_grid2` defines the range of values of `k` to evaluate, and so for each value of `k`, a knn model is trained on the training set and then evaluated on the validation set.
`mean(pred == valid_df$R1)` computes the validation accuracy for that specific value of `k`.
The value of `k` that maximizes validation accuracy is selected as `best_k_val`, which turned out to be `k = 20`, achieving [0.8854962] (88.5%) classification accuracy!
But, it's important to remember this is just the validation set, so we can only really use this to select the correct model. Finally, I run:

After the final tuning is complete, the training and validation sets are combined using `rbind()`, and then the final knn model is then fit using the selected value of `k = 20`.
`mean(pred_test == test_df$R1)` computes the final test accuracy, and `confusionMatrix()` provides a more detailed breakdown of classification performance (much more detailed than needed, very wordy, like me).
I found that `k = 20` achieved a [0.8153846] (81.5%) classification accuracy! Honestly, I'm not sure that this

is a good accuracy point, I feel like with the specific data we are working with.. It's not the best, and should be slightly more accurate.
According to the small table provided by `confusionMatrix()`, Predicted vs Reference told me that [60] true [0] were reported as [0], and [11] true [0] were reported as [1] (false positives), and [46] true [1] were reported as [1], and [13] true [1] were reported as [0] (false negatives).

```
fp_test <- cm$table["1", "0"]
n_test  <- sum(cm$table)
n_total <- nrow(df)

fp_test / n_test * n_total
```

```
## [1] 55.33846
```

In a perfect world, that would mean there would be roughly [55] false positives (approving someone risky) in this model, which is actually ever so slightly better than what we saw on HW1… But still not something I would want to show off, if that makes sense?
However, once again similar to HW1, the scope of this homework does not ask me to optimize for business risk, so, I will move on.

Just For Fun

In this section I wanted to do the optional portions of question 3.1!

```
X <- model.matrix(R1 ~ . - 1, data = df)
y <- df$R1

dim(X)
```

```
## [1] 654  32
```

The `ksvm()` function requires the predictors to be provided as a numeric matrix
In case you see that the dimenstions are no longer [654] and [11], don't be alarmed.
The increase in the number of columns occurs because model.matrix() expands factor predictors into multiple dummy variables, producing a fully numeric design matrix required by the SVM.

```
best_C_linear
```

```
## [1] 0.01
```

```
best_acc_linear
```

```
## [1] 0.8606527
```

Here I use the same 10-fold cross-validation structure as before, but now to tune the SVM cost parameter C.
For each value of C, the model is trained on nine folds and evaluated on the remaining fold, and the average accuracy across folds is computed.
Overall, I found that the linear SVM performs comparably to knn on this dataset, and in this case achieves slightly higher cross-validated accuracy. Among all values of C I tested, [C = 0.01] produced the highest average cross-validated accuracy, that accuracy being [0.8606527] (86%). This section is mainly exploratory, but it was helpful for seeing how different classifiers behave on the same problem!