McKenna Stout - Homework 1

Question 2.1 Answer

A classification model would currently best fit my life, rather my exercise habits, and more specifically my running schedule, essentially:
High Running Day vs Low Running Day based on work-related or my personal stress factors.

The reason this would be useful is that marathon training depends on consistently hitting weekly mileage goals. If I have many low running days in a row, that usually turns into a low running week.
Several low running weeks in a row would set me back from my planned training schedule, since I want to reach a certain total mileage by a specific date. By classifying days as high or low running days, I could better understand how work and life stress affect my ability to stay on track over time.

Observation: One Day
Predictors:
1. Number of Meetings that Day
2. Number of Active Projects
3. Number of Assigned Support Tickets
4. Number of Upcoming Bills
5. Day of Week (Workday vs Weekend)

Question 2.2

I am making my life more complicated by using R within VScode, which took the entire evening to setup correctly, but it seems to be working as expected now!
(I just like all the QOL that VScode gives, I use it for work and personal life projects, and I love that I can run it in dark-mode.)

Now that I have everything set up, I can run the recommended packages to use SVM and KNN:

```r
library(kernlab)
library(kknn)
```

But because this is the first time I am running these, I got the dreaded: `Error in library(kernlab) : there is no package called 'kernlab' Error in library(kknn) : there is no package called 'kknn'` So I ran:

```r
install.packages("kernlab")
install.packages("kknn")
```

Which then let me load the libraries (packages) as expected, so let me get started for real!

Using the `credit_card_data-headers.txt` file, I first ran:

```r
data <- read.table(
    "C:/Users/mstout/OneDrive - AANP/Documents/Workspace.MAIN/edX/GTx.ISYE6501/data/Homework1_ISYE6501/c
     header = TRUE
     )
```

Because I am re-learning R, I want to write out my reasoning for running each line (apologies in advance to my peers reviewing this).
`read.table()` reads a `.txt` file and turns it into data R can use, specifically a data frame R can use.
`header = TRUE` tells R that the first row of the file contains column names rather than data.
`data <-` assigns the resulting data frame to the variable named `data`.

I then ran:

```
dim(data)
```

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

Just to make sure it had the correct dimensions, as we know (`credit_card_data-headers.txt`) should be 654 rows, and 11 columns.
Now, I know I need to make sure R understand what the Predictors are, and what the Response is, and we know from the instructions that columns 1-10 are the Predictors, and column 11 is the Response so I ran:

```
x <- as.matrix(data[, 1:10])
y <- as.factor(data[, 11])
```

And verified it by running:

```
class(x)
```

```
## [1] "matrix" "array"
```

```
dim(x)
```

```
## [1] 654  10
```

```
class(y)
```

```
## [1] "factor"
```

```
levels(y)
```

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

```
table(y)
```

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

For x, it should respond with "matrix" as the class, and [654] [10] as the dimensions
For y, it should respond with "factor" as the class, and levels should show [0] and [1], because those are the two Response types found in column 11.
Something I did not know yet, but now have learned due to running `table(y)` is that there are [358] Responses [0], and [296] Responses [1]
Finally, with this setup, I will now run the SVM:

```r
model <- ksvm(
  x, y,
  type   = "C-svc",
  kernel = "vanilladot",
  C      = 10,
  scaled = TRUE
)
```

```
##  Setting default kernel parameters
```

I chose C = 10 first because I wanted to test out the hint from the Homework Instructions:
"Hint: You might want to view the predictions your model makes; if C is too large or too small, they'll almost all be the same (all zero or all one) and the predictive value of the model will be poor. Even finding the right order of magnitude for C might take a little trial-and-error."
I figured that instead of using the C = 100 from the example right away, I would try C = 10 first to see how "off" it might look. That way, I can start to build intuition for what it looks like when C is way off in the model.
(I also view homework as controlled experimentation, not obedience, so I plan to run both C = 10 and C = 100 and compare the results myself.)
Just to make sure everything loaded correctly, I also ran:

```r
class(model)
```

```
## [1] "ksvm"
## attr(,"package")
## [1] "kernlab"
```

And expect to see "ksvm" - which I did!
Now, I will run this:

```r
pred <- predict(model, x)
table(pred)
```

```
## pred
##   0   1
## 303 351
```

(I know they gave us `pred <- predict(model,data[,1:10])` to run, but earlier I just wanted to make it easier for myself by setting `x <- as.matrix(data[, 1:10])` so instead of having to write `data[, 1:10]` a bunch of times I can just write `x`)
Running it at `C = 10` gave me [303] at Response [0], and [351] at Response [1], which is actually not too bad?
But I know that isn't everything, so I ran:

```r
mean(pred == y)
```

```
## [1] 0.8639144
```

```
table(Predicted = pred, Actual = y)
```

```
##          Actual
## Predicted   0   1
##         0 286  17
##         1  72 279
```

Which told me the mean was: 0.8639144, meaning 86.4% of all 654 observations were classified correctly! (I know they gave us `sum(pred == data[,11]) / nrow(data)` to run, but I realized in this scenario, it's identical to calculating the mean, so when you run one or the other, it has the same result, I will always try to write the absolute shortest lines possible).
The small table of Predicted vs Actual told me that [286] true [0] were reported as [0], and [72] true [0] were reported as [1] (false positives), and [279] true [1] were reported as [1], and [17] true [1] were reported as [0] (false negatives).
Just based on real-world experience, I personally know that [72] false positives (approving someone risky) is unlikely to make the company requesting this report happy… However, the scope of this homework does not ask me to optimize for business risk, so, I will move on.

```
a <- colSums(model@xmatrix[[1]] * model@coef[[1]])
a0 <- -model@b
length(a)
```

```
## [1] 10
```

```
a0
```

```
## [1] 0.08157559
```

```
a
```

```
##            A1            A2            A3            A8            A9
## -0.0009033671 -0.0007891036 -0.0016972133  0.0026113629  1.0050221405
##           A10           A11           A12           A14           A15
## -0.0028363016 -0.0001569285 -0.0003925964 -0.0012784443  0.1064387167
```

Finally - I extracted the coefficients of the linear SVM to write out the classifiers.
Using the trained model, the intercept [a0] was 0.08157559, meaning that the model slightly favors class [1], which we did see with the [72] false positives
The remaining ten coefficients [an] were obtained directly from the support vectors. The one of note, [A9], was 1.0050221405, which is quite large! The next closest was [A15] which sits at 0.1064387167.
So, based on what we have learned so far, we can assume [A9] is large because changes in [A9] do far more than any other variable to separate approved from denied applications in this dataset!

Onto the KNN portion of the homework now, I ran:

```
y_num <- as.numeric(as.character(data[, 11]))
table(y_num)
```

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

I did this to make sure it comes back as numeric, because in the homework it says:
"Note that kknn will read the responses as continuous, and return the fraction of the k closest responses that are 1 (rather than the most common response, 1 or 0)."
Then I ran this:

```r
ks <- c(1, 3, 5, 7, 9, 11, 15, 21)
acc_knn <- numeric(length(ks))
```

I followed the advice to try many different k values, and stored them in the container `ks`, and created a place to store the accuracy for each k by creating `acc_knn` based on the length of `ks`.
With that set up, I now can attempt to find KNN using this:

```r
resp_name <- names(data)[11]

for (j in seq_along(ks)) {
  k <- ks[j]
  preds <- integer(nrow(data))

  for (i in 1:nrow(data)) {
    fit <- kknn(
      as.formula(paste(resp_name, "~ .")),
      train = data[-i, ],
      test  = data[i, , drop = FALSE],
      k     = k,
      scale = TRUE
    )

    p1 <- fitted(fit)
    preds[i] <- as.integer(p1 >= 0.5)
  }

  acc_knn[j] <- mean(preds == y_num)
}

data.frame(k = ks, accuracy = acc_knn)
```

```
##    k  accuracy
## 1  1 0.8149847
## 2  3 0.8149847
## 3  5 0.8516820
## 4  7 0.8470948
## 5  9 0.8470948
## 6 11 0.8516820
## 7 15 0.8532110
## 8 21 0.8486239
```

Essentially, I run `resp_name <- names(data)[11]` to get the name of the Response [11] column, then I enter a loop where I begin looking at the k values (`ks`), then use `preds <- integer(nrow(data))` to store the predictions.
Using the advice in the homework assignment, I wrote `for (i in 1:nrow(data)) {` to not let [i] be its own nearest neighbor, followed by the lines that actually fit / train a KNN model. I just used `example(kknn)` to help me best format the script, while making slight adjustments to best follow the guidance in the homework. But running that gave me what I wanted, a list of accuracy points for each k value, with `k= 15` being the highest accuracy, so I will select this as a reasonable choice for this assignment.

Just for fun

I wanted to test out multiple values of C at once, but could not for the life of me figure it out - So I begrudgingly used AI to help me fix and better formualte this code snippit.
I was unsure the correct way to setup a vector in this way, but once I saw how to set it up, I was able to plug it in to what we were already given in the homework.

```
Cs <- c(0.01, 0.1, 1, 10, 100, 1000)
acc <- numeric(length(Cs))
pred_counts <- vector("list", length(Cs))
models <- vector("list", length(Cs))

for (z in seq_along(Cs)) {
  models[[z]] <- ksvm(
    x, y,
    type   = "C-svc",
    kernel = "vanilladot",
    C      = Cs[z],
    scaled = TRUE
  )

  p <- predict(models[[z]], x)
  acc[z] <- mean(p == y)
  pred_counts[[z]] <- table(p)
}
```

```
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
```

```
data.frame(C = Cs, accuracy = acc)
```

```
##       C  accuracy
## 1 1e-02 0.8639144
## 2 1e-01 0.8639144
## 3 1e+00 0.8639144
## 4 1e+01 0.8639144
## 5 1e+02 0.8639144
## 6 1e+03 0.8623853
```

```
pred_counts
```

```
## [[1]]
## p
##   0   1
## 303 351
##
## [[2]]
## p
##   0   1
## 303 351
##
## [[3]]
## p
##   0   1
## 303 351
##
## [[4]]
## p
##   0   1
## 303 351
##
## [[5]]
## p
##   0   1
## 303 351
##
## [[6]]
## p
##   0   1
## 304 350
```

`Cs <- c(0.01, 0.1, 1, 10, 100, 1000)` Firstly, we need to create a vector containing all the values I want to test as the penalty marker, C. and I am holding these values in the `Cs` container.
Now running this, `acc <- numeric(length(Cs))`, we need a place to store the accuracy of each C, and this essentially creates: `c(0, 0, 0, 0, 0, 0)`.
`pred_counts <- vector("list", length(Cs))` this creates basically a list with 6 empty slots (aka `pred_counts <- vector("list", 6)`).
`models <- vector("list", length(Cs))` Very similar to line 3, but this one has to be a list beacause each call to ksvm() returns a complex object, and lists are required to store them as ar as I could tell.
Now the fun part - loops! `for (z in seq_along(Cs)) {...` I am telling it to loop over the C values we have set up in the Vector (Cs).
`models[[z]] <- ksvm(...` This should look familiar to the original code snippit we were given, the only difference is that it's now `models` which is a list, specifically referencing `z`.
The rest is essentially verbaitem the code snippit we were given up until `p <- predict(models[[z]], x)` which is generating predictions for every row of `x` using the models we have trained in `z`.
Now, to calculate the mean score for each of my C's… `acc[z] <- mean(p == y)`.
Next, I want to see the table that shows me how many predictions are 0 vs 1 for this model `pred_counts[[z]] <- table(p)`.
Out of the loop now, I needed to create a data frame that aligned my C number with its corresponding accuracy: `data.frame(C = Cs, accuracy = acc)`.
Last but not least, I run `pred_counts` which prints the list of prediction-count tables.
I will be the first to admit that this is most likely bloated with some unnecessary code, but as of right now it's the best I could come up with that seemingly is giving me what I want, so it will have to do!