# Question 3.1

# *Using the same data set (credit\_card\_data.txt or credit\_card\_data-headers.txt) as in Question 2.2, use the ksvm or kknn function to find a good classifier:*

1. *using cross-validation (do this for the k-nearest-neighbors model; SVM is optional); and*
2. *splitting the data into training, validation, and test data sets (pick either KNN or SVM; the other is optional).*

**SOLUTIONS:**

**(a)**

There are different ways to do this. Three different methods are shown in solution 3.1-a.R. Just having one method is fine for your homework solutions. All three are shown below, for learning purposes. Another optional component shown below is using cross-validation for ksvm; this too did not need to be included in your solutions.

*METHOD 1*

The simplest approach, using kknn’s built-in cross-validation, is fine as a solution. train.kknn uses leave-one-out cross-validation, which sounds like a different type of cross-validation that I didn’t mention in the videos – but if you watched the videos, you know it implicitly already! For each data point, it fits a model to *all* the other data points, and uses the remaining data point as a test – in other words, if n is the number of data points, then leave-one-out cross-validation is the same as n-fold cross-validation.

Using this approach here are the results (using scaled data):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| k | Correct | Percent correct | k | Correct | Percent correct |
| 1,2,3,4 | 533 | 81.50% | 18 | 557 | 85.17% |
| 5 | 557 | 85.17% | 19-20 | 556 | 85.02% |
| 6 | 553 | 84.56% | 21 | 555 | 84.86% |
| 7 | 554 | 84.71% | 22 | 554 | 84.71% |
| 8 | 555 | 84.86% | 23 | 552 | 84.40% |
| 9 | 554 | 84.71% | 24-25 | 553 | 84.56% |
| 10-11 | 557 | 85.17% | 26 | 552 | 84.40% |
| 12 | 558 | 85.32% | 27 | 550 | 84.10% |
| 13-14 | 557 | 85.17% | 28 | 548 | 83.79% |
| 15-17 | 558 | 85.32% | 29 | 549 | 83.94% |
|  |  |  | 30 | 550 | 84.10% |

As before k < 5 is clearly worse than the rest, and value of k between 10 and 18 seem to do best. For unscaled data, the results are significantly worse (not shown here, but generally between 66% and 71%).

Note that technically, these runs just let us choose a model from among k=1 through k=30, but because there might be random effects in validation, to find an estimate of the model quality we’d have to run it on some test data that we didn’t use for training/cross-validation.

*METHOD 2*

Some of you used the cv.kknn function in the kknn library. This approach is also shown in solution 3.1-a.R.

*METHOD 3*

And others of you found the caret package in R that has the capability to run k-fold cross-validation (among other things). The built in functionality of the caret package gives ease of use but also the flexibility to tune different parameters and run different models. It’s worth trying. This approach is also shown in solution 3.1-a.R.

The main line of code is:

knn\_fit <- train(as.factor(V11)~V1+V2+V3+V4+V5+V6+V7+V8+V9+V10,

data,

method = "knn", # choose knn model

trControl=trainControl(

method="repeatedcv", # k-fold cross validation

number=10, # number of folds (k in cross validation)

repeats=5), # number of times to repeat k-fold cross

validation

preProcess = c("center", "scale"), # standardize the data

tuneLength = kmax) # max number of neighbors (k in nearest

neighbor)

The trainControl method allows us to determine the number of resampling iterations (“number”) and the number of folds to perform ("repeats"). The train function finally trains the model while allowing us to preprocess the data (scale and center) as well as select the number of k values to choose from.

*ksvm Cross Validation*

If you also tried cross-validation with ksvm (you didn’t need to), you could do that by including “cross=k” for k-fold cross-validation – for example, “cross=10” gives 10-fold cross-validation.

In the R code for Question 2.2 Part 1, you would replace the line

model <- ksvm(as.matrix(data[,1:10]),as.factor(data[,11]),type = "C-svc",kernel = "vanilladot",C = 100,scaled = TRUE)

with

model <- ksvm(as.matrix(data[,1:10]),as.factor(data[,11]),type = "C-svc",kernel = "vanilladot",C = 100,scaled = TRUE,cross=10)

model@cross shows the error measured by cross-validation, so 1–model@cross is the estimate of the model’s fraction of points correctly classified; instead of the 86.4% correct classifications found in Question 2.2 Part 1 using scaled data, the cross-validated estimate is a little bit lower: 86.2%. That’s a difference of only about 1 correct prediction out of 654, so it’s not a big difference – meaning our initial model is a good one, and doesn’t seem to have been over-fit.

To compare models with different values of C, we can use that modification in the code in solution 2.2-1.R.

The results with scaled data to show that for C=0.00001 or C=0.00001, only about 55% of points are classified correctly. At C=0.001, about 83% are classified correctly. At 0.01 and higher, the model achieves the 86.2% classification correctness we got above – a wide range of values of C gives a good model. With unscaled data, just as before, finding a value of C to give a good model is harder.

|  |  |  |
| --- | --- | --- |
| *C* | *Percent correct (scaled data)* | *Percent correct (unscaled data)* |
| 0.00001 | 54.76% | 66.19% |
| 0.0001 | 54.74% | 68.50% |
| 0.001 | 82.86% | 75.38% |
| 0.01 | 86.23% | 81.50% |
| 0.1 | 86.24% | 85.48% |
| 1 | 86.24% | 78.89% |
| 10 | 86.26% | 58.81% |
| 100 | 86.25% | 70.04% |
| 1000 | 86.25% | 65.36% |

**(b)**

As usual, there are lots of possible answers. File solution 3.1-b.R contains one approach.

In this approach, we first split the data into training, validation, and test sets. We used 60%, 20%, and 20%, but other splits are fine too as long as training has at least half.

Then, we fit 9 SVM models and 20 k-nearest-neighbor models to the training data, and evaluated them on the validation data.

We report the SVM model that does best in validation, and the KNN model that does best in validation. The code chose C=0.01 as the best SVM model, though it was equal with C=0.1, 1, 10, 100, and 1000, so any of them could’ve been chosen. The best KNN model was with k=16.

Then, we have an if statement that checks to see which model – the best SVM model or the best KNN model – performed best on the validation data. Whichever one it is, is the model we suggest using (and we report its performance on the test set).

**Important note**: In our code, the best SVM model (C=0.01) performs best on the validation data… but the best KNN model performs best on the test data. It might be tempting to therefore say, “Oh, let’s use the best KNN model.” Don’t give in to this temptation! If you do, you’re losing the value of separating validation and test sets. You’d essentially be using the test set to pick the best model, and then you’d (incorrectly) be using that same test set to estimate its quality – the selection bias from the validation step will be incorrectly included in the quality estimate.

You could’ve used a different approach – for example, only testing SVM models or only testing KNN models.

Some people also went beyond what we’ve covered, and tested models with different kernels – that’s also a good idea, and it’s possible to get better models that way.