WEEK 1 HOMEWORK – SAMPLE SOLUTIONS

IMPORTANT NOTE

These homework solutions show multiple approaches and some optional extensions for most of the questions in the assignment. You don't need to submit all this in your assignments; they're included here just to help you learn more – because remember, the main goal of the homework assignments, and of the entire course, is to help you learn as much as you can, and develop your analytics skills as much as possible!

Question 2.1

Describe a situation or problem from your job, everyday life, current events, etc., for which a classification model would be appropriate. List some (up to 5) predictors that you might use.

One possible answer:

Being students at Georgia Tech, the Teaching Assistants for the course suggested the following example. A college admissions officer has a large pool of applicants must decide who will make up the next incoming class. The applicants must be put into different categories – admit, waitlist, and deny – so a classification model is appropriate. Some common factors used in college admissions classification are high school GPA, rank in high school class, SAT and/or ACT score, number of advanced placement courses taken, quality of written essay(s), quality of letters of recommendation, and quantity and depth of extracurricular activities.

If the goal of the model was to automate a process to make decisions that are similar to those made in the past, then previous admit/waitlist/deny decisions could be used as the response. Alternatively, if the goal of the model was to make *better* admissions decisions, then a different measure could be used as the response – for example, if the goal is to maximize the academic success of students, then whether each admitted student's college GPA was above or below a certain threshold could be the response; if the goal is to maximize the post-graduation success of admitted students, then some measure of career success (e.g., whether each student got a good job after graduation) could be the response; etc.

Question 2.2

The files credit_card_data.txt (without headers) and credit_card_data-headers.txt (with headers) contain a dataset with 654 data points, 6 continuous and 4 binary predictor variables. It has anonymized credit card applications with a binary response variable (last column) indicating if the application was positive or negative. The dataset is the "Credit Approval Data Set" from the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/Credit+Approval) without the categorical variables and without data points that have missing values.

1. *Using the support vector machine function ksvm contained in the R package kernlab, find a good classifier for this data. Show the equation of your classifier, and how well it classifies the data points in the full data set. (Don't worry about test/validation data yet; we'll cover that topic soon.)*

   Notes on ksvm

   - *You can use scaled=TRUE to get ksvm to scale the data as part of calculating a classifier.*
   - *The term λ we used in the SVM lesson to trade off the two components of correctness and margin is called C in ksvm. One of the challenges of this homework is to find a value of C that works well; for many values of C, almost all predictions will be "yes" or almost all predictions will be "no".*
   - *ksvm does not directly return the coefficients a_0 and $a_1 \ldots a_m$. Instead, you need to do the last step of the calculation yourself. Here's an example of the steps to take (assuming your data is stored in a matrix called data):*[1]

     ```
     # call ksvm. Vanilladot is a simple linear kernel.
     model <- ksvm(data[,1:10],data[,11],type="C-svc",kernel="vanilladot",C=100,scaled=TRUE)
     # calculate a1…am
     a <- colSums(model@xmatrix[[1]] * model@coef[[1]])
     a
     # calculate a0
     a0 <- –model@b
     a0
     # see what the model predicts
     pred <- predict(model,data[,1:10])
     pred
     # see what fraction of the model's predictions match the actual classification
     sum(pred == data[,11]) / nrow(data)
     ```

   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.

[1] I know I said I wouldn't give you exact R code to copy, because I want you to learn for yourself. In general, that's definitely true – but in this case, because it's your first R assignment and because the ksvm function leaves you in the middle of a mathematical calculation that we haven't gotten into in this course, I'm giving you the code.

Note: If you get the error "Error in vanilladot(length = 4, lambda = 0.5) : unused arguments (length = 4, lambda = 0.5)", it means you need to convert data into matrix format:

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

SOLUTION:

There are multiple possible answers. See file solution 2.2-1.R for the R code for one answer. Please note that a good solution doesn't have to try both of the possibilities in the code; they're both shown to help you learn, but they're not necessary.

One possible linear classifier you can use, for scaled data z, is

$-0.0010065348z_1 - 0.0011729048z_2 - 0.0016261967z_3 + 0.0030064203z_4 + 1.0049405641z_5 - 0.0028259432z_6 + 0.0002600295z_7 - 0.0005349551z_8 - 0.0012283758z_9 + 0.1063633995z_{10} + 0.08158492 = 0.$

It predicts 565 points (about 86.4%) correctly. (Note that this is its performance on the training data; as you saw in Module 3, that's not a reliable estimate of its true predictive ability.) This quality of linear classifier can be found for a wide range of values of C (from 0.01 to 1000, and beyond). Using unscaled data, it's a lot harder to find a C that does this well.

2. *You are welcome, but not required, to try other (nonlinear) kernels as well; we're not covering them in this course, but they can sometimes be useful and might provide better predictions than vanilladot.*

It's also possible to find a better nonlinear classifier using a different kernel; kudos to those of you who went even deeper and tried this!

3. *Using the k-nearest-neighbors classification function kknn contained in the R kknn package, suggest a good value of k, and show how well it classifies that data points in the full data set. Don't forget to scale the data (scale=TRUE in kknn).*

   Notes on kknn

   - *You need to be a little careful. If you give it the whole data set to find the closest points to i, it'll use i itself (which is in the data set) as one of the nearest neighbors. A helpful feature of R is the index –i, which means "all indices except i". For example, data[-i,] is all the data except for the ith data point. For our data file where the first 10 columns are predictors and the 11th column is the response, data[-i,11] is the response for all but the ith data point, and data[-i,1:10] are the predictors for all but the ith data point.*
     (There are other, easier ways to get around this problem, but I want you to get practice doing some basic data manipulation and extraction, and maybe some looping too.)

- ***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).*

SOLUTION:
Here's one possible solution. See file solution 2.2-3.R for code. Please note that a good solution doesn't have to try all of the possibilities in the code.

As detailed in the code, we observe maximum accuracy for k=12 and k=15. (Again, as above, we're reporting performance on the training data, which is generally not good practice, as you saw in Module 3.) A summary of the number of correct predictions for different values of k is shown below (using scaled data).

Value of k (scaled data)	Correct predictions	Percent correct predictions
1-4	533	81.50%
6	553	84.56%
7,9	554	84.71%
8	555	84.86%
10,19-20	556	85.02%
5,11,13-14,16-18	557	85.17%
12,15	558	85.32%

As the table shows, the key (in the training data) is to use k ≥ 5; smaller values of k are significantly inferior. Although k=12 and k=15 look slightly better than the rest, it's not a statistically significant difference.

- Note that if we're just looking at fraction of correct predictions, it might be easy to get caught up in finding the very highest amount we can find. Don't lose sight of the fact that these differences might just be 1 data point out of 654 – which is not statistically significant.

We could do the same using unscaled data, by changing one word in the R code; replace

model=kknn(V11~V1+V2+V3+V4+V5+V6+V7+V8+V9+V10,data[-i,],data[i,],k=X, scale = TRUE)

with

model=kknn(V11~V1+V2+V3+V4+V5+V6+V7+V8+V9+V10,data[-i,],data[i,],k=X, scale = FALSE)

Using unscaled data, the results are significantly worse.

Value of k (unscaled data)	Correct predictions	Percent correct predictions
1-4	434	66.36%
10	443	67.74%

11	445	68.04%
12	447	68.35%
9,13	449	68.65%
14-15	450	68.81%
5,20	452	69.11%
7-8,16-19	453	69.27%
6	455	69.57%

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:

(a) using cross-validation (do this for the k-nearest-neighbors model; SVM is optional); and

(b) 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 (<u>you didn't need to</u>), 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.