American University - DATA 312

Instructions for Homework 4-B

Unit 4 – Machine Learning
Part B - Supervised learning using Support Vector Machines

Pre-requisite work:
Please finish Homework 4-A before starting this assignment. You will need the following libraries installed:

- tidyverse
- **(NEW!)** e1071
- **(NEW!)** kernlab

Caution: the e1071 library and the GENEAread library both provide short time Fourier transforms (stft()), but with differently structured inputs and outputs. Although we aren't using the stft() function in this worksheet, you might in the future! If you happen to need both libraries, you can get the stft() we used previously via GENEAread::stft().

Objectives:

Consider a dataset containing several variables (columns). Some of the variables are designated as the "response variables" (or "outputs"). You'd like to predict the response variables from the "explanatory variables" (or "inputs"). There are also variables that you didn't (or couldn't) measure, which are the "latent variables". These distinctions between variable roles are a bit arbitrary, and it can be a bit of a problem to determine which variables should play which role, especially because the same variable can play different roles depending on your research question. Exploring your data (on the training set, like in HW4A) is an important part of determining possible variable roles.

"Machine learning" is a term that you've probably heard, perhaps surrounded by a bit of mystery. While there are several different kinds of algorithms that are usually considered "learning", they are essentially just fancier versions of statistical regression. Machine learning comes in two varieties: "supervised" and "unsupervised". In supervised learning, the goal is to predict the response variables from the explanatory ones, given that the training data contains the correct answers, just like traditional regression. Sometimes you discover new latent variables in the process. In unsupervised learning, the goal is a little different; you want to determine which variables should play which roles.

As in HW4A, careful attention to the train-query-test process with the data is critical, because modern supervised learning algorithms are extremely flexible. Moreover, since they are quite a bit more sensitive to the data than (say) linear regression, it is very easy to get a misleading performance measurement if you don't follow the correct process.

In this worksheet, we'll focus our attention on one supervised learning technique, called support vector machines (SVMs). The goal of an SVM is simple: given a table of numerical variables, assign a boolean (true/false) classification to each observation. You use the training sample of your data to help the SVM give the correct classification to a set of observations with known classifications. Once "trained"you can use the SVM to classify other observations that don't have known classifications. For instance, a common task for an SVM might be to determine if a pedestrian is in view of a car's collision avoidance camera. Given a digital image -- a big collection of numerical variables -- the output of such an SVM would simply be "pedestrian" (true) or "not pedestrian" (false).

A given SVM is a rather elaborate algorithm, whose behavior is determined by quite a few numerical "parameters". The training data are used internally to determine these parameters. Thankfully, that process is handled internally by the e1071 library, so you can just tell an SVM to "train thyself" given some data and it will!

Actually, there are many kinds of SVMs, though they ultimately look about the same to the data scientist from the standpoint of inputs and outputs. The e1071 library provides four different types of SVM, though there are many others that are in wide usage. In the machine learning jargon, this means that in addition to the parameters of each SVM, there are "hyperparameters" that select the particular SVM algorithm you want to use. In our train-query-test process, the query stage, where we select one algorithm without changing it, can be thought of as the training stage for the hyperparameters. It sounds fancy, but it's really pretty easy: try a handful of versions of your algorithm, and then pick the best.

<u>Work Process:</u>

1. As usual, clear the workspace

```
rm(list=ls())
gc()
```

2. Load the libraries:

```
library(tidyverse)
library(e1071)
```

3. The data we'll be using for this worksheet is kind of amusing

```
data("spam",package='kernlab')
```

It is a single table, called `spam`, in which each row corresponds to an email message. The columns are normalized word frequencies (recall our Text Mining module? Yes, that's what made the values!) for various words present in the textual content of the messages. Finally, there is a `type` column that either contains the string "spam" or "nonspam". You guessed it! We are going to make an email spam filter!

4. Now it happens that this particular dataset is rather easy to "cheat" because it includes a few oddities in how it was collected. Specifically, this dataset was collected before "spear phishing" was common (where a spammer impersonates a trusted sender whose account has been compromised). As a result, words that are characteristic of the data collector's organization are a dead giveaway that a message is "nonspam". These correspond to two columns: `george` and `num650`, so we'll deselect these.

Just as in HW4A, we need to create a sampling frame

```
raw_data_samplingframe <- spam %>%
  select(-george, -num650 ) %>%  # Remove the "cheating" columns...
  mutate(snum=sample.int(n(),n())/n())
```

5. And, just as in HW4A, we split our data into training, query, and test sets.

```
training <- raw_data_samplingframe %>%
  filter(snum<0.6) %>%
  select(-snum)

query<- raw_data_samplingframe %>%
  filter(snum>=0.6,snum<0.8) %>%
  select(-snum)

test<- raw_data_samplingframe %>%
  filter(snum>=0.8) %>%
  select(-snum)
```

6. Save these off in CSV files. For your assignment, make sure to upload these all to Canvas!

```
training %>% write_csv('spam_training.csv')
query %>% write_csv('spam_query.csv')
test %>% write_csv('spam_test.csv')
```

7. In a few places we will want just the input variables. It will be frequently useful to have the correct answers in a separate table as well. Let's split these off now:

```
training_input <- training %>% select(-type)
training_truth <- training$type

query_input <- query %>% select(-type)
query_truth <- query$type

test_input <- test %>% select(-type)
test_truth <- test$type
```

TRAINING STAGE

8. Let's start with a quick visualization! Since all the explanatory variables are numerical (they're normalized word frequencies) and there are quite few of them, principal components analysis (PCA) is a good choice.

```
spam_pca <- training_input %>% prcomp()
```

Now we can plot the messages as a scatterplot, and color by `type`:

```
spam_pca$x %>%
  as_tibble() %>%
  mutate(type=training_truth) %>%
  ggplot(aes(PC1,PC2,color=type)) +
  geom_point()
```

Each message corresponds to a point in this plot. Messages with similar word usage end up near each other, while messages using different words tend to be further away from each other.

Notice how the spam messages spread out a bit further from the nonspam messages. These outliers are easier to detect than the spam messages that do a better job of looking like a nonspam message.

9. If you remember HW3D, you may recall using *k*-means to identify notes within a PCA scatterplot. We needed to know the number of notes to make it work. Clustering with *k*-means is a kind of "semi"-supervised learning, in that we tell it the number of classes we want. In the case of spam detection, there are two classes: spam and nonspam. So, here we go!

```
spam_kmeans <- training_input %>% kmeans(2)
```

10. Did *k*-means correctly identify the spam? We can tell if we make a table comparing the k-means cluster ID (an integer) with the true classes:

```
kmeans_results <- tibble(training_truth,km=spam_kmeans$cluster)
```

Then we can count the number of times we get a match between the two. I think this looks nicest as a contingency table, so we'll need to pivot_wider():

```
kmeans_results %>%
  count(training_truth,km) %>%
  pivot_wider(names_from=km,values_from=n)
```

Have a look at the results closely. The best possible performance consists of having each row and each column having exactly one zero entry. Because *k*-means has an element of randomness to it, I can't tell you specifically which class ID number (the `km` column) means spam or not, but it's not very effective.

11. Just as a sanity check, though, you can ask whether *k*-means is detecting "something" about the spamminess of a message. The way to do this is with chi-squared. We've done this many times before, so the following block of code should make perfect sense to set up the contingency table:

```
ct <- kmeans_results %>%
  count(training_truth,km) %>%
  pivot_wider(names_from=km,values_from=n) %>%
  column_to_rownames('training_truth') # Look out! column named the same as a variable!
```

And then let's just run the test:

```
chisq.test(ct)
```

Well, I got a small *p*-value. So *k*-means detects a statistically significant feature of the spam, but it's not really conclusive either.

12. Why did *k*-means do poorly? Let's label the PCA plot we made in Step 8 with the *k*-means clusters instead of the true values:

```
spam_pca$x %>%
  as_tibble() %>%
  ggplot(aes(PC1,PC2,color=spam_kmeans$cluster)) +
  geom_point()
```

What you should see is that *k*-means splits the messages into an "inner core" and "outliers". It labels all of the inner core as nonspam, and the outliers as spam. While this works for the really far outliers (which are obviously spam messages!), it doesn't do a good job closer to the core.

13. Let's give up on *k*-means, and switch over to SVM training. You train an SVM using the svm() function. The svm() function can take two styles of input, though they both do the same thing internally. If you're interested, you can read the documentation:

```
?svm
```

Here's the format that the e1071 authors use more frequently in the documentation:

```
svm_linear <- svm(type~.,   # Watch the syntax.  It's a tilde, a period, and then comma
  data=training,     # Unlike tidyverse, the data input is not the first argument.  We can't pipe...
  kernel='linear')   # "linear" is the hyperparameter (see Step 14)
```

The other option is tidyverse compatible, but you have to split the data into input and output first:

```
svm_linear <- training_input %>%
 svm(y=training_truth,
 kernel='linear')
```

These both do the same thing, so pick whichever you like. I'm going to use the first version in what follows.

14. The way SVMs work is that they split the space of observations (imagine the PCA plot) into two halves by way of a "separating surface". That surface is high dimensional and takes quite a few parameters to define, but you need not worry too much about them. Additionally, there are many kinds of surfaces you might try to use, each defined by a set of equations. Choose a different set of equations, and your surfaces will look different! The choice of a set of equations is called a "hyperparameter", and in the case of svm(), the hyperparameter is called `kernel`. There are four kernels supported by e1071:

- 'linear' : The separating surface is a flat plane
- 'polynomial' : The separating surface is a given by a polynomial... like z=ax^2+by^2+cxy+ ... (lots more)
- 'radial' : The separating surface is an ellipsoidal blob
- 'sigmoid' : The separating surface is kind of "S"-shaped

Since our dataset isn't too big, it doesn't hurt to try all of these! We can use the query dataset to select the one that does the best job at correctly classifying the email messages as spam or nonspam.

How do we do this? Well, we just repeat Step 13 with a different `kernel`:

```
svm_poly <- svm(type~.,
                data=training,
                kernel='polynomial')
svm_radial <- svm(type~.,
                data=training,
                kernel='radial')
svm_sigmoid <-svm(type~.,
                  data=training,
                  kernel='sigmoid')
```

Note: each of the different kernels have additional hyperparameters in them (like `degree`, `gamma`, and the like), that you might need to change in some circumstances. We don't need to do that here, but you may have to change them if you don't get good results on other data...

15. Now that we've trained the SVMs for our data, it's a good idea to anticipate their performance. Plotting is perhaps the best way to do this, but plotting in the original data variables is not reasonable. That's unfortunate because that's how our SVMs in Step 14 were trained. But, since we are using the training data, there is nothing wrong with training an additional set of SVMs to work on the PCA data since these plot better. The hope (which will be validated against the query sample) is that these plots will be representative.

To that end, we need to transform the training data using PCA, and add back the true classifications ('spam' / 'nonspam').

```
training_pca <- spam_pca$x %>%
  as_tibble() %>%
  mutate(type=training_truth)
```

16. OK, let's plot each SVM. It happens that the trained SVMs made by the svm() function already know how to plot themselves, though this doesn't use ggplot(). Instead, it uses the base R plot() function. Here's how to do it:

```
sl<-svm(type~.,data=training_pca,kernel='linear')  # Watch the punctuation! (see Step 13)

plot(sl,training_pca,PC1~PC2)  # This plots PC1 versus PC2
```

If you look closely at this, you can see a line slicing through the plot, separating the spam from the nonspam. It does a pretty good job!

Try this with the other three kernels, changing what evidently needs to be changed.

Notice the different shapes of the different spam/nonspam regions! They all look a little different. Which do you think best matches the shape of the true classes? (See the plots from Step 8.)

QUERY STAGE

17. Now we're ready to choose the SVM kernel that will serve as our final version! Since our SVMs are already trained (Step 14), we can just apply them to the new input data. The way to do this is via the predict() function. It works like this:

```
predict(svm_linear, query_input)
```

You should get spammed with some "spam" and "nonspam"! Not very helpful, but it lets us see that the SVM is working. Let's bundle it all into one big table:

```
query_results <- tibble(query_truth,
        linear=predict(svm_linear, query_input),
        poly=predict(svm_poly, query_input),
        radial=predict(svm_radial, query_input),
        sigmoid=predict(svm_sigmoid, query_input))
```

18. We want to check how many instances we have of where the `query_truth` matches with each of the other four columns: each one is where we correctly identified a message. The way that we built the query_results table is a little annoying, because we really want to do these comparisons systematically. So, let's pivot the data longer by reworking the columns:

```
query_results1 <- query_results %>%  pivot_longer(cols=!query_truth)
```

19. After this, we check for matches. There are four kinds of these!

```
query_results2 <- query_results1 %>%
  mutate(tp=(query_truth=='spam' & value=='spam'),  # True positives (SVM got it right!)
            tn=(query_truth=='nonspam' & value=='nonspam'),  # True negatives (SVM got it right!)
            fp=(query_truth=='nonspam' & value=='spam'),  # False positive (SVM made a mistake)
            fn=(query_truth=='spam' & value=='nonspam'))  # False negative (SVM made a mistake)
```

20. Count up the occurrences of each of these

```
query_results3 <- query_results2 %>% group_by(name) %>% # Group by the kernel names
  summarize(tp=sum(tp),
                 tn=sum(tn),
                 fp=sum(fp),
                 fn=sum(fn))
```

21. With the counts of true/false positives/negatives, there are many possible kinds of scores you can make to determine which SVM kernel works best. Different disciplines tend to prefer some of these scores over others: medical tests often report *sensitivity* and *specificity*, while most deep learning researchers prefer *F1 scores*. If you're curious, have a look at

https://en.wikipedia.org/wiki/Sensitivity_and_specificity

Just using the formulas on that page, we can compute a few of these:

```
query_results3 %>%
  mutate(accuracy=(tp+tn)/(tp+tn+fp+fn),
        sensitivity=tp/(tp+fn),
        specificity=tn/(tn+fp),
        ppv=tp/(tp+fp),
        npv=fn/(tn+fn),
        f1=2*tp/(2*tp+fp+fn))
```

I noticed that 'linear' was the best in my case at least in terms of the *accuracy* and the *F1 score*, but it's awfully close to the 'radial' score. This is now a situation where plotting can help, if you like.

22. Finally, you can repeat Step 17-21 with just the svm_linear on the test data!

<u>Assignment</u>:

For this assignment, please submit your R script file and data sets (one CSV or R package and three CSV files for samples).

1. Locate a dataset (you can consult the HW1B-Dataset assignment on Canvas, which has many suggestions) that has a number of numerical variables that you can use as explanatory variables and a binary (boolean) variable that you can use as a response variable. **Every student must have a different dataset for this assignment**, which is easy because there are hundreds of freely available datasets!

Caution: some of these datasets do not separate well according to the response variables. It's better if the data separate, but don't worry if you can't find a good dataset. The process will "work", but it might not work well. Be sure to explain your findings; that's more important than getting good results!

2. Split your chosen dataset into the three training/query/test samples. Save each off each as a CSV file and upload these to Canvas.

3. Train at least three SVMs on your training data, being careful to explain which variables are the response and the explanatory variables. Feel free to experiment with different hyperparameters beyond what was done in the worksheet above. The hyperparameters you need might be quite different!

4. Using your query set, select which SVM performs the best... and then...

5. ... determine its performance using the testing set.

6. Make sure to explain Steps 4 and 5 in comments in your R script file!