Homework #1: ISYE6501

Zach Olivier

5/15/2018

Question 2.1

Question:

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.

Answer:

One example I am excited about is an attempt to build a classification model to determine whether a person will purchase a certain product or not, say a new car.

Based on a list of attributes, a classification model could sort customers into groups of 'most likely will purchase' and 'most likely will not purchase' with high accuracy. This would help optimize our marketing budget by only targeting customers who are the most likely to purchase.

Predictors I would use for this model:

```
- Household Income
- Location
- Have they purchased a car before
- Time since last vehicle purchase
- Type of vehicle last purchased
```

Question 2.2.1

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

Question:

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.)

Answer:

First we read in the data and load the needed packages for this analysis. Then, we quickly look at the data and see if it matches the course description.

Our problem is classifying the data based on the 11th column - which I relabeled as response_y for clarity.

I also established a baseline accuracy that our model based classifiers should beat easily. My naive classifier will pick '1' as the answer for every value in the training set.

In this case my naive classifier will accurately classify ~45% of the observations in the training set.

Our proposed SVM and KNN models should aim to beat this baseline accuracy measure.

```
# set up directory and packages
setwd('~/Desktop/GTX/Homework 1/')

# load packages
pacman::p_load(tidyverse, kernlab, caret, kknn, modelr)

# get data and format
credit_df <- read.table('2.2credit_card_data-headersSummer2018.txt',header = T) %>%

        as_tibble() %>%
        dplyr::rename(., 'response_y' = R1)

# data summary
str(credit_df)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    654 obs. of  11 variables:
##  $ A1        : int  1 0 0 1 1 1 1 0 1 1 ...
##  $ 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        : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ A10       : int  0 0 1 0 1 1 1 1 1 1 ...
##  $ A11       : int  1 6 0 5 0 0 0 0 0 0 ...
##  $ A12       : int  1 1 1 0 1 0 0 1 1 0 ...
##  $ A14       : int  202 43 280 100 120 360 164 80 180 52 ...
##  $ A15       : int  0 560 824 3 0 0 31285 1349 314 1442 ...
##  $ response_y: int  1 1 1 1 1 1 1 1 1 1 ...
```

```
# investigate class imbalance - this is our 'baseline' accuracy
mean(credit_df$response_y == 1)
```

```
## [1] 0.4525994
```

```
# baseline error
(baseline <- 1 - mean(credit_df$response_y == 1))
```

```
## [1] 0.5474006
```

Answer:

The following code sets up the Support Vector Machine algorithm using the kernlab package in R.

The ksvm function is supplied with a formula to model the response_y binary variable based on all remaining predictors in the data set. The scaled argument is set to TRUE - the function will automatically scale the predictor data. Type is 'C-svc' to indicate we are using this function for classification. Finally, the kernal is set to 'vanilladot' which initializes a linear decision boundary.

I then set up a for loop to pass my model different regularization (C - Cost) parameters and then extract each model's accuracy. The model with the highest training set accuracy based on values of C will be selected as my 'best' model. **(Note: training set accuracy is not a valid measure of model performance!)**

Here are the results:

The best tuned model with C = 95.45 achieved 95% training set accuracy. This outperforms the naive classifer baseline accuracy greatly

However, this does not indicate we have a good model. We may have modeled our training set real and random effects accurately - but our model will likely not generalize well to new data. There is a high likelihood that we have over fit this model to the training set.

The formula for the best tuned linear decision boundary is:

y =

(-36.5 x A2) + (-8.36 x A3) + (54.2 x A8) + (48.6 x A9) + (-17.5 x A1) + (-22.6 x A10) + (14.5 x A11) + (-22.1 x A12) + (-56.4 x A14) + (50.2 x A15) + (-.778)

```
# set seed
set.seed(2)

# data frame of model parameters
model_param <- data.frame(
        type = 'C-svc',
        kernal = 'vanilladot',
        cross = 5,
        scaled = T,
        stringsAsFactors = F
)


# set up a list of possible C values: choose C from .5 to 100 by .05
# - will build a model for each C and determine the 'best' C based on accuracy
# originally tried values .05 to 200 - shortened to run for homework
cost_list <- as.list(seq(from = 50, to = 100, by = .05))

# set up a list to put the error metrics of each model into
error_list <- list()


# cost loop - for each value of cost - fit a model - extract accuracy - put into er
ror list
for (i in seq_along(cost_list)) {
```

```
        c <- cost_list[[i]]

        set.seed(2)

        # fit ksvm model based on model parameters
        ksvm_fit <- ksvm(
                response_y ~ .,
                data = credit_df,
                scaled = model_param$scaled ,
                type = model_param$type,
                kernal = model_param$kernal,
                C = c,
                cross = model_param$cross
        )

        error_list[[i]] = ksvm_fit@error
}


# reduce list of errors and cost parameters into a column in a data frame
error_df <- reduce(error_list, rbind.data.frame)
cost_df <- reduce(cost_list, rbind.data.frame)

# performance measures - put list of errors and costs next to each other to find hi
ghest accuracy
performance_train <- cbind(error_df, cost_df) %>%
        rename(
                'svm_error' = !!names(.[1]),
                'svm_cost' =  !!names(.[2])
        ) %>%
        mutate(svm_accuracy = 1-svm_error) %>%
        filter(svm_error == min(svm_error)) %>%
        arrange(., svm_cost) %>%
        .[1,]


# fit model with our best C determined by training error
svm_bestfit <- ksvm_fit <- ksvm(
                response_y ~ .,
                data = credit_df,
                scaled = model_param$scaled ,
                type = model_param$type,
                kernal = model_param$kernal,
                C = performance_train$svm_cost,
                cross = model_param$cross
        )

## extracting the model formula

# calculate a1...am the coefficients for each predictor!
svm_formula <- colSums(svm_bestfit@xmatrix[[1]] * svm_bestfit@coef[[1]]) %>%
                as.data.frame() %>%
                rownames_to_column('predictor') %>%
                rbind(., data.frame(predictor ='z.Intercept', . = svm_bestfit@b))
%>%
                spread(., key = predictor, value = .) %>%
```

```
                as_tibble()

# print results
performance_train
```

```
##    svm_error svm_cost svm_accuracy
## 1 0.04740061    90.35    0.9525994
```

```
svm_bestfit
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 90.35
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  0.0945349772139391
##
## Number of Support Vectors : 245
##
## Objective Function Value : -8315.259
## Training error : 0.047401
## Cross validation error : 0.198802
```

```
svm_formula
```

```
## # A tibble: 1 x 11
##      A1   A10   A11   A12   A14   A15    A2    A3    A8    A9 z.Intercept
## * <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>       <dbl>
## 1 -17.5 -22.6  14.5 -22.1 -56.4  50.2 -36.5 -8.36  54.2  48.6      -0.778
```

Question 2.2.2

Question:

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.

Answer:

Here is my experimentation with a non-linear kernal within a SVM model. I applied the same steps to hone in on a value of C that minimizes training set error.

The radial SVM model achieves a training set error of less than 4% with our best tuned Cost parameter.

As mentioned above - we are likely to have over fit to the training data. Training data should not be used to define the performance of our models.

Cross Validation is a better measure of real test set performance. Even though our flexible radial model

achieves less than 4% error on the training set, the cross validation error is almost 19%.

```
# set up list of possible cost values
# originally tried values .05 to 200 - shortened to run for homework
cost_list <- as.list(seq(from = 130, to = 175, by = .05))

# set up empty list to put error results for each cost into
error_list <- list()

# fit a radial svm for every value of cost...extract error to compare
for (i in seq_along(cost_list)) {

        c <- cost_list[[i]]

        set.seed(2)

        # fit ksvm model
        ksvm_fit <- ksvm(
                response_y ~ ., # formula
                data = credit_df, # training data
                scaled = T, # scale predictors
                type = 'C-svc', # svm for classification
                kernal = 'rbfdot', # radial decision boundary
                C = c, # set C to the input value from for loop
                cross = 10, # implement 10 fold cross validation
                kpar = 'automatic') # automatically detect best parameters for rad
ial kernal

        error_list[[i]] = ksvm_fit@error
}

# collapse lists to data frames
error_df <- reduce(error_list, rbind.data.frame)
cost_df <- reduce(cost_list, rbind.data.frame)

# find the optimal value of Cost - cost value with lowest error
performance_train <- cbind(error_df, cost_df) %>%
        rename(
                'svm_error' = !!names(.[1]),
                'svm_cost' =  !!names(.[2])
        ) %>%
        filter(svm_error == min(svm_error)) %>%
        arrange(., svm_cost) %>%
        .[1,]


# fit model with our best C determined by training error
svm_bestfit <- ksvm_fit <- ksvm(
                response_y ~ .,
                data = credit_df,
                scaled = T,
                type = 'C-svc',
                kernal = 'rbfdot',
                C = performance_train$svm_cost,
                cross = 10,
                kpar = 'automatic'
        )
```

```
)


# look at the results of our model
svm_bestfit@error
```

```
## [1] 0.03975535
```

```
svm_bestfit@cross
```

```
## [1] 0.1943823
```

Question 2.2.3

Question:

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).

Answer:

Here we apply Leave One Out Cross Validation (LOOCV) to the credit dataset in order to find the optimal K for our KNN model. To do this, I implement the train.kknn function to cycle through possible values of K and determine the best value based on LOOCV error. Variables are scaled within the algorithm by setting scale = True. The kmax arguement is upper bound of K values we would like to consider for the model.

In this implementation we find that the optimal K is 58.

Using this optimal K value, we fit a KNN model to the entire training set to determine accuracy.

The accuracy of the K = 58 KNN model is ~88%

Note: Using this model to predict back onto the training set will give misleading results. Smaller values for K will actually predict better on the training set. However, this will not generalize to an independent or new dataset. K = 1 may perform perfectly on the training set but K = 58 will likely generalize better to other datasets.

```
set.seed(2)

# set up train.kknn to find the optimal k using leave one out classification
(knn_fit_LOOCV <- train.kknn(
        response_y ~ .,
        data = credit_df,
        # test = credit_df,
        kmax = 100,
        # kcv = 10,
        scale = T)
)
```

```
##
## Call:
## train.kknn(formula = response_y ~ ., data = credit_df, kmax = 100,     scale =
T)
##
## Type of response variable: continuous
## minimal mean absolute error: 0.1850153
## Minimal mean squared error: 0.1073792
## Best kernel: optimal
## Best k: 58
```

```
# apply optimal K to the original training dataset
knn_fit <- kknn(
        response_y ~ .,
        train = credit_df,
        test = credit_df,
        k = 58,
        scale = T)

# accuracy measures
fitted <- fitted(knn_fit) %>%
     as_tibble() %>%
     mutate(value = ifelse(value > .5, 1, 0)) %>% # round to 1 if prob > .5, 0 oth
erwise
     cbind(., credit_df$response_y) %>%
     mutate(acc = value == credit_df$response_y) # if prediction == actual then Tru
e, else False

# percetage accuracy of the model
(test_accuracy <- mean(fitted$acc))
```

```
## [1] 0.8776758
```

Question 3.1

Question:

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)
b. splitting the data into training, validation, and test data sets (pick either KNN or SVM; the other is optional).

Answer: (a)

The kknn package provides a function ‘train.kknn’ which will search for the best value of K based on cross validation error. I specified kcv = 10 to run a 10-fold cross validation on the full credit dataset.

The train.kknn function will run the 10-fold cross validation and extract the K (nearest neighbor parameter K, not cross validation fold k) with the minimial error.

Based on the results the optimal **K is 58 and results in around 10% mean squared error**

I also provide the results for K = 30, 20, and 10 to illustrate that the accuracy improvement for using K = 58 is not that much smaller than using less neighbors.

Note: Using this model to predict back onto the training set will give misleading results. Smaller values for K will actually predict better on the training set. However, this will not generalize to an independent or new dataset. K = 1 may perform perfectly on the training set but K = 58 will likely generalize better to other datasets.

Cross validation is a better estimate of test set error and we should trust those results more than accuracy or error on the training set.

```
set.seed(4)

# from documentation - k fold cross validation needs kcv arguement
# kcv = Number of partitions for k-fold cross validation.


# find best value of K by cross validation - what is the estimated test accuracy?
knn_fit <- train.kknn(response_y ~ .,
                data = credit_df,
                kmax = 60, # search K values up to 600
                kcv = 10, # base accuarcy on 10-fold cross validation
                scale = T)

summary(knn_fit)
```

```
##
## Call:
## train.kknn(formula = response_y ~ ., data = credit_df, kmax = 60,     scale = T,
kcv = 10)
##
## Type of response variable: continuous
## minimal mean absolute error: 0.1850153
## Minimal mean squared error: 0.1073792
## Best kernel: optimal
## Best k: 58
```

```
# fit a model with less that the optimal K and compare accuracy
# K lower than our best fit should give inferior results
knn_fit2 <- train.kknn(response_y ~ .,
                data = credit_df,
                kmax = 30,
                kcv = 10,
                scale = T)

# mean squared error
summary(knn_fit2)
```

```
##
## Call:
## train.kknn(formula = response_y ~ ., data = credit_df, kmax = 30,     scale = T,
kcv = 10)
##
## Type of response variable: continuous
## minimal mean absolute error: 0.1850153
## Minimal mean squared error: 0.1081193
## Best kernel: optimal
## Best k: 30
```

```
# fit a model with less that the optimal K and compare accuracy
# K lower than our best fit should give inferior results
knn_fit3 <- train.kknn(response_y ~ .,
                data = credit_df,
                kmax = 20,
                kcv = 10,
                scale = T)

# mean squared error
summary(knn_fit3)
```

```
##
## Call:
## train.kknn(formula = response_y ~ ., data = credit_df, kmax = 20,     scale = T,
kcv = 10)
##
## Type of response variable: continuous
## minimal mean absolute error: 0.1850153
## Minimal mean squared error: 0.1099495
## Best kernel: optimal
## Best k: 20
```

```
# fit a model with less that the optimal K and compare accuracy
# K lower than our best fit should give inferior results
knn_fit4 <- train.kknn(response_y ~ .,
                data = credit_df,
                kmax = 10,
                kcv = 10,
                scale = T)

# mean squared error
summary(knn_fit4)
```

```
##
## Call:
## train.kknn(formula = response_y ~ ., data = credit_df, kmax = 10,     scale = T,
kcv = 10)
##
## Type of response variable: continuous
## minimal mean absolute error: 0.1850153
## Minimal mean squared error: 0.1148192
## Best kernel: optimal
## Best k: 10
```

Answer: (b)

I will use the caret package to split our data set into training, test, and validation data sets. **I will portion the available data 60% training, 20% test, and 20% validation.**

We train the model on the training data set, measure accuracy and tune parameters using the test set. Once we think we have a good model, we will re-train the model on the entire training and test sets, then calculate accuracy for our best model using the validation data set.

```
set.seed(2)

# develop the training and holdout partition
partition <- createDataPartition(credit_df$response_y, p = .6,
                                 list = F)
# develop the training set
train <- credit_df[partition,]

# develop the holdout set
holdout <- credit_df[-partition,]

# develop the split of the holdout data into test and validation
partition_valid <- createDataPartition(holdout$response_y, p = .5,
                                       list = F)
# split holdout into test
test <- holdout[partition_valid, ]

# split holdout in validation
validation <- holdout[-partition_valid, ]

# check splits
nrow(train)/nrow(credit_df); nrow(test)/nrow(credit_df); nrow(validation)/nrow(cred
it_df)
```

```
## [1] 0.6009174
```

```
## [1] 0.2003058
```

```
## [1] 0.1987768
```

Answer: (b)

Here we determine the best value of K by training our KNN model on the training data set, and measuring accuracy on the test data set. Based on this method the K value that produced the maximum accuracy on the test data set is **K = 40, with accuracy of ~88%**.

Now that we have our optimal value of K based on our test data set - we will apply this model to the final holdout data in the validation set. We do this to get an unbiased view of how our model will perform on new data.

The KNN model with K = 40 achieves ~85% accuracy on the validation dataset. This means we should expect our classifier to accurately predict 85% of new 'unseen' data into the correct category.

```
# set up list of possible K values from 1 to 100 by 3
possible_k <- as.list(seq(from = 1, to = 100, by = 3))

# set up a blank list to put accuracy values into
test_accuracy <- list()

# fit a model for each possible value of K and extract the accuracy from each model
for (i in seq_along(possible_k)) {

        k = possible_k[[i]]

        knn_fit <- kknn(
                response_y ~ .,
                train = train,
                test = test,
                # valid = validation$response_y,
                k = k,
                scale = T)

        fitted <- fitted(knn_fit) %>%
             as_tibble() %>%
             mutate(value = ifelse(value > .5, 1, 0)) %>%
             cbind(., test$response_y) %>%
             mutate(acc = value == test$response_y)

        test_accuracy[[i]] <- mean(fitted$acc)

}

# put the K and test accuracy lists into dataframes
k_df <- reduce(possible_k, rbind.data.frame)
test_acc_df <- reduce(test_accuracy, rbind.data.frame)

# find the best K associated with the highest accuracy
(performance_test <- cbind(k_df, test_acc_df) %>%
        rename(
                'knn_error' = !!names(.[2]),
                'knn_k' =  !!names(.[1])
        ) %>%
        filter(knn_error == max(knn_error)) %>%
        arrange(., knn_k) %>%
        .[1,])
```

```
##   knn_k knn_error
## 1    40 0.8778626
```

```
# fit the best model on the training + testing data - find accuracy on the validati
on set
knn_best_fit <-  kknn(
                response_y ~ .,
                train = rbind(train, test), # combine train to be train + test dat
asets
                test = validation,
                k = performance_test$knn_k, # best value of K found from test set
accuracy
                scale = T)

# view the validation set accuracy
best_fitted <- fitted(knn_best_fit) %>%
             as_tibble() %>%
             mutate(value = ifelse(value > .5, 1, 0)) %>% # kknn outputs a score f
or each value
             cbind(., validation$response_y) %>%
             mutate(acc = value == validation$response_y)

# accuracy of our best fit knn model on the validation dataset
(valid_accuracy <- mean(best_fitted$acc))
```

```
## [1] 0.8538462
```