Week 1 Assignment - SVM & KNN Classification

Omer Farooq (EDx ID: mfarooq4)

1/14/2020

Table of Contents

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.

At T-Mobile HQ in Seattle area where I am based, my team helps get analytics products built for our procurement and supply chain teams. One of the key insights we provide to our procurement teams is the visibility into company's spend data. We ingest transaction level invoices data from Accounts Payable and PO data from SAP. These datasets are stitched together and enriched with other datasets like HR org hierarchy data (to see which org or team spent what) and geographical data (to see which locations spend originated from).

One of the key enrichment of the spend data that is currently lacking is the spend categorization. This mean, putting spend transactions into right buckets based on what the spend type was. For example, a transaction to buy computer hardware gets categorized under "Hardware" category, an invoice to pay Accenture for their services gets categorized as "Consulting Services" and so on. This is a perfect problem to solve with a classification model like KNN. We even did a pilot with an Auto-ML software company and compared their model with the one from our internal data scientists. Further details are given below:

- **Data size:** ~ 1M rows of invoice transactions per month
- **Total features (columns) available:** 100+
- **Response:** a category (from a defined hierarchy of spend categories) of spend assigned to each invoice line (which is represented by a single row of the data)
- **Why is this important?** A key insight any procurement or finance team needs is details into where and how comapny's dollars are spend. If a procurement team can't even identify basic information like how much money the company spent on let's say 'software', they can't do further time series and trend analysis on that category to manage it better. A properly classified spend data enables procurement teams to

monitor the high spend categories and devise strategies to manage those categories better.

- **Predictors Useful for the Model:**
  - **Supplier Name** - an unstructure data type predictor with supplier names entered as free text. E.g. Accenture might indicate spend is for consulting services whereas Infosys or Tata Consulting might indicate it's for managed service.
  - **Business Unit**- a categorical data type with fixed names for different business units. Some BUs spend more of on one type of products or services. E.g. enterprise IT will almost always spend more on Hardware, Software, IT Services etc.
  - **Finance Coding** (General Ledger Account Code) - a categorical data type with code numbers indicating the type of spend as defined by the Finance team.
  - **Invoice Line Description** - an unstructured data type with free text explaining the what was purchased. There would most likely be hints in this text regarding what product or service was procured.
  - **Supplier Location** - a categorical data type, location of a given supplier would provide useful information regarding the type of product or service procured. E.g. an invoice from Google in the Bay Area could be for any of the services they offer whereas an invoice from Google in Seattle would most likely be for Cloud Services only.

QUESTION 2.2

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.)

Getting the needed libraries.

```
library(kernlab)
library(kknn)
library(caret)
library(data.table)
```

Let's take a glimpse of the data. The original **654 rows** of the data have been split into a trainig dataset (70% of the rows) and testing dataset (remaining 30%). 70/30 split is selected after research online and published work where this split has been recommneded the most. It ensures enough data (70%) for training and not 30% sample is also enough to check the validity of the model. 20% for testing is also mentioned but I went with 30% so that there is enough new data for the model to be properly vetted.

```
my_data <- read.delim("data_2.2/credit_card_data-headers.txt")
nrow(my_data)
```

```
## [1] 654

set.seed(101) #this ensures that same datasets are reproduced in the future.
sample <- sample.int(n = nrow(my_data), size = floor(.70*nrow(my_data)),
replace = F)
train_data <- my_data[sample,]
test_data  <- my_data[-sample,]

nrow(train_data)

## [1] 457

nrow(test_data)

## [1] 197

head(train_data)

##      A1    A2    A3    A8 A9 A10 A11 A12 A14 A15 R1
## 430   1 20.00 7.000 0.500  0   1   0   1   0   0  0
## 95    0 28.50 1.000 1.000  1   0   2   0 167 500  0
## 209   1 48.17 3.500 3.500  1   1   0   1 230   0  1
## 442   0 23.00 1.835 0.000  0   0   1   1 200  53  0
## 351   1 39.42 1.710 0.165  0   1   0   1 400   0  0
## 315   1 34.83 2.500 3.000  0   1   0   1 200   0  0
```

I decided to use the KSVM function from Kernlab package instead of SVM function from e1701 package. Main difference in KSVM are the different kernal method available to try out non-linear classifications. e1701 only offers linear, radial basis , polynomial and sigmoid fittings, whereas KSVM offers around 10 kernal functions including linear (vanilladot), hyperbolic (tanhdot), ANOVA (anovadot) etc. This is important b/c even though SVM theoretically finds the linear separation my maximizing the margin, in real life, most available datasets have non-linear decision boundaries. The kernal function of KSVM helps deals with that.

Let’s take a look at classifier from KSVM using linear kernel function (vanilladot) and a relatively high C value (=100). The model is trained and validated on the same training dataset as a starter.

```
#building model on training dataset
model_ksvm <- ksvm( x = as.matrix(train_data[,1:10]),
                    y = as.factor(train_data[,11]),
                    type='C-svc',
                    kernel= 'vanilladot',
                    C=100, scaled=TRUE
                    )

##  Setting default kernel parameters
```

```
#checking model on the training data itself
predictions <- predict(model_ksvm,newdata = train_data[,1:10])
confusionMatrix(factor(train_data[,11], levels = c(1,0)),predictions)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 192  55
##          1  13 197
##
##                Accuracy : 0.8512
##                  95% CI : (0.8152, 0.8826)
##     No Information Rate : 0.5514
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.7049
##
##  Mcnemar's Test P-Value : 6.627e-07
##
##             Sensitivity : 0.9366
##             Specificity : 0.7817
##          Pos Pred Value : 0.7773
##          Neg Pred Value : 0.9381
##              Prevalence : 0.4486
##          Detection Rate : 0.4201
##    Detection Prevalence : 0.5405
##       Balanced Accuracy : 0.8592
##
##        'Positive' Class : 0
##
```

The model shows an accuracy of **85.12%** on the training dataset. Let's check model's performance on testing dataset.

```
#checking predictions on testing dataset
predictions <- predict(model_ksvm,newdata = test_data[,1:10])
confusionMatrix(factor(test_data[,11], levels = c(1,0)),predictions)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 94 17
##          1  4 82
##
##                Accuracy : 0.8934
##                  95% CI : (0.8417, 0.9328)
##     No Information Rate : 0.5025
##     P-Value [Acc > NIR] : < 2.2e-16
##
```

```
##                   Kappa : 0.7869
##
##  Mcnemar's Test P-Value : 0.008829
##
##             Sensitivity : 0.9592
##             Specificity : 0.8283
##          Pos Pred Value : 0.8468
##          Neg Pred Value : 0.9535
##              Prevalence : 0.4975
##          Detection Rate : 0.4772
##    Detection Prevalence : 0.5635
##       Balanced Accuracy : 0.8937
##
##        'Positive' Class : 0
##
```

The model's accuracy goes up to **89.34%** on testing dataset. The accuracy scores are not expected to match in general across training & testing datasets. This is expected because because sometimes small overfitting of the training dataset is inevitable, making the model's performance on training data better, whereas other times the training dataset performs better indicating the model is truly working well.

Since I need to find the best C value for the mode, I will run a loop for different C values and document the accuracy to see which C value performs the best. 20 C values separated by 1e+/-01 are used in the loop.

```
TestCs <- c(10**(-10:10))

iter = length(TestCs)

results <- matrix(NA, nrow=iter, ncol=2)
colnames(results) <- c("Cvalue","Accuracy")

#looping Cs on the model and documenting accuracy percentage
for(i in 1:iter){
    model_ksvm <- ksvm( x = as.matrix(train_data[,1:10]),
                    y = as.factor(train_data[,11]),
                    type='C-svc',
                    kernel= 'vanilladot',
                    C=TestCs[[i]],
                    scaled=TRUE
                    )
    predictions <- predict(model_ksvm,newdata = test_data[,1:10])
    results[i,] <- c(TestCs[[i]], sum(predictions == test_data[,11]) /
nrow(test_data))
}

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

```
results
```

```
##       Cvalue  Accuracy
##  [1,]  1e-10 0.5634518
##  [2,]  1e-09 0.5634518
##  [3,]  1e-08 0.5634518
##  [4,]  1e-07 0.5634518
##  [5,]  1e-06 0.5634518
##  [6,]  1e-05 0.5634518
##  [7,]  1e-04 0.5634518
##  [8,]  1e-03 0.8324873
##  [9,]  1e-02 0.8934010
## [10,]  1e-01 0.8934010
## [11,]  1e+00 0.8934010
## [12,]  1e+01 0.8934010
## [13,]  1e+02 0.8934010
## [14,]  1e+03 0.8934010
## [15,]  1e+04 0.8934010
## [16,]  1e+05 0.8934010
## [17,]  1e+06 0.8781726
## [18,]  1e+07 0.7106599
## [19,]  1e+08 0.6903553
## [20,]  1e+09 0.6700508
## [21,]  1e+10 0.6700508
```

The results of the loop above show that for C values b/w 0.01 to 100,000 the accuracy of the model does not change i.e.89.34%. Accuracy drops below 0.01 and above 100,000.

I will use **C = 100** value to find the classifier from KSVM model. This C value is in the middle of the range for which accuracy is the same. Since C $\sim 1/\lambda$, this ensures that there is neither too much emphasis on the margin nor the total error. At the same time, it also indicates the right trade-off b/w margin and total error because for a higher than 100,000 C and lower than 0.01 C, accuracy is lower.

```
#final classifer using C=100

final_model_ksvm <- ksvm( x = as.matrix(train_data[,1:10]),
                    y = as.factor(train_data[,11]),
                    type='C-svc',
                    kernel= 'vanilladot',
                    C=100, scaled=TRUE
                    )

##  Setting default kernel parameters

final_predictions <- predict(final_model_ksvm,newdata = test_data[,1:10])
confusionMatrix(factor(test_data[,11], levels = c(1,0)),final_predictions)

## Confusion Matrix and Statistics
## 
##           Reference
## Prediction  0  1
##          0 94 17
##          1  4 82
##                                           
##                Accuracy : 0.8934          
##                  95% CI : (0.8417, 0.9328)
##     No Information Rate : 0.5025          
##     P-Value [Acc > NIR] : < 2.2e-16       
##                                           
##                   Kappa : 0.7869          
##                                           
##  Mcnemar's Test P-Value : 0.008829        
##                                           
##             Sensitivity : 0.9592          
##             Specificity : 0.8283          
##          Pos Pred Value : 0.8468          
##          Neg Pred Value : 0.9535          
##              Prevalence : 0.4975          
##          Detection Rate : 0.4772          
##    Detection Prevalence : 0.5635          
##       Balanced Accuracy : 0.8937          
##                                           
##        'Positive' Class : 0               
## 
```

Using this model, let’s find the coefficients for the equation.

```
#calculting a1...am
a <- colSums(final_model_ksvm@xmatrix[[1]] * final_model_ksvm@coef[[1]])
a
```

```
##            A1           A2           A3           A8           A9
## -0.0009206098 -0.0044367594 -0.0026209194  0.0047287318  1.0034635331
##           A10          A11          A12          A14          A15
## -0.0021271935 -0.0001701517 -0.0003920085 -0.0029128439  0.1051406257

#calculting a0
a0 <- -final_model_ksvm@b
a0

## [1] 0.1075138
```

Thus, the final equation for every point j would be:

$$-0.0009206098x_{A1j} - 0.0044367594x_{A2j} - 0.0026209194x_{A3j} + 0.0047287318x_{A8j} + 1.0034635331x_{A9j} - 0.0021271935x_{A10j} - 0.0001701517x_{A11j} - 0.0003920085x_{A12j} - 0.0029128439x_{A14j} + 0.1051406257x_{A15j} + 0.1075138 = 0$$

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.

In order to try the other kernels, let's loop over the model and use other available kernels. I will however skip the 'stringdot' kernel because it specializes operating on strings and our data does not have strings. Though I had landed on C=100 based on the analysis above, I will however try a few C values from the range for which accuracy of linear kernel was the same.

```
kernels <-
c('rbfdot','polydot','tanhdot','laplacedot','besseldot','anovadot','splinedot
')

iter = length(kernels)


kernel_results <- matrix(NA, nrow=iter, ncol=5)
colnames(kernel_results) <- c("Kernel","C=0.01","C=1.00","C=100","C=10000")

#looping different kernel on the model and documenting accuracy percentage
for 4 different C values
for(i in 1:iter){
    c0.01 <- ksvm( x = as.matrix(train_data[,1:10]),
                   y = as.factor(train_data[,11]),
                   type='C-svc',
                   kernel= kernels[[i]],
                   C=0.01,
                   scaled=TRUE
                   )
    c1.00 <- ksvm( x = as.matrix(train_data[,1:10]),
                   y = as.factor(train_data[,11]),
                   type='C-svc',
                   kernel= kernels[[i]],
                   C=1,
                   scaled=TRUE
                   )
    c100 <- ksvm( x = as.matrix(train_data[,1:10]),
                   y = as.factor(train_data[,11]),
                   type='C-svc',
                   kernel= kernels[[i]],
                   C=100,
                   scaled=TRUE
                   )
    c10000 <- ksvm( x = as.matrix(train_data[,1:10]),
                   y = as.factor(train_data[,11]),
                   type='C-svc',
                   kernel= kernels[[i]],
                   C=10000,
                   scaled=TRUE
                   )
    predc0.01 <- predict(c0.01,newdata = test_data[,1:10])
    predc1.00 <- predict(c1.00,newdata = test_data[,1:10])
    predc100 <- predict(c100,newdata = test_data[,1:10])
    predc10000 <- predict(c10000,newdata = test_data[,1:10])
    kernel_results[i,] <- c(kernels[[i]],
                            sum(predc0.01 == test_data[,11]) /
nrow(test_data),
                            sum(predc1.00 == test_data[,11]) /
nrow(test_data),
                            sum(predc100 == test_data[,11]) /
nrow(test_data),
                            sum(predc10000 == test_data[,11]) /
nrow(test_data)
                            )
}

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

```
kernel_results
```

```
##      Kernel       C=0.01              C=1.00              C=100
## [1,] "rbfdot"     "0.563451776649746" "0.883248730964467" "0.82741116751269"
## [2,] "polydot"    "0.893401015228426" "0.893401015228426" "0.893401015228426"
## [3,] "tanhdot"    "0.888324873096447" "0.705583756345178" "0.796954314720812"
## [4,] "laplacedot" "0.563451776649746" "0.893401015228426" "0.878172588832487"
## [5,] "besseldot"  "0.563451776649746" "0.873096446700508" "0.776649746192893"
## [6,] "anovadot"   "0.893401015228426" "0.893401015228426" "0.888324873096447"
## [7,] "splinedot"  "0.878172588832487" "0.812182741116751" "0.791878172588833"
##      C=10000
## [1,] "0.796954314720812"
## [2,] "0.893401015228426"
## [3,] "0.786802030456853"
## [4,] "0.878172588832487"
## [5,] "0.751269035532995"
## [6,] "0.873096446700508"
## [7,] "0.791878172588833"
```

The output of the analysis above shows that accuracy is varying quite a bit for some kernels over C values but for all kernels and C value iterations, I do not see any accuracy over 89.34% which was our final accuracy from C=100 linear model above. This means, even though there are non-linear models that are closer to the linear model's accuracy, none of them were better.

Among the seven kernel tried, polydot (Polynomial kernel-looks at not just input features but their combinations as well),laplacedot (which is similarto Gaussian & exponential kernels which are radial basis functions) and ANNOVAdot (also a radial basis function) came close to linear model's 89% accuracy mark.

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).

In this section, I am trying out the Weighted K-Nearest Neighbor model from KKNN package on the same training & test datasets I used for KSVN models.

Given my data is split into training & testing data, I do not need to exclude the ith point per the notes in the HW document. First attempt below is with k=10 neighbors and for the Optimal kernel. K=10 is picked randomly for now (we will run a loop of options for K to find best K below). Data is being scaled as well. I am using an *as.integer* function to turn probabilities to 0 and 1 (0.5 and above equals 1 and 0.5 andless equals 0)

```
model_kknn <- kknn( R1~.,
                    train_data, #no need to exclude ith row or run loop on
all rows b/c training and test data is separated already
                    test_data,
                    k=10,
                    distance = 2,
                    kernel = "optimal",
                    scale= TRUE
                    )

#using the predict function to create an array of probabilities for the test
data.
predictions <- predict(model_kknn,newdata = test_data[,1:10])

#converting probabilities to 0 and 1
predictions01 <- as.integer(predictions+0.5)

#Using confusion matrix function with predictions leveled out as well.
confusionMatrix(factor(test_data[,11], levels =
c(1,0)),factor(predictions01,levels=c(1,0)))

## Confusion Matrix and Statistics
##
##             Reference
## Prediction   1   0
##          1  76  10
##          0   9 102
##
##                Accuracy : 0.9036
##                  95% CI : (0.8535, 0.9409)
##     No Information Rate : 0.5685
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.8037
##
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.8941
##             Specificity : 0.9107
##          Pos Pred Value : 0.8837
##          Neg Pred Value : 0.9189
##              Prevalence : 0.4315
##          Detection Rate : 0.3858
##    Detection Prevalence : 0.4365
##       Balanced Accuracy : 0.9024
##
##        'Positive' Class : 1
##
```

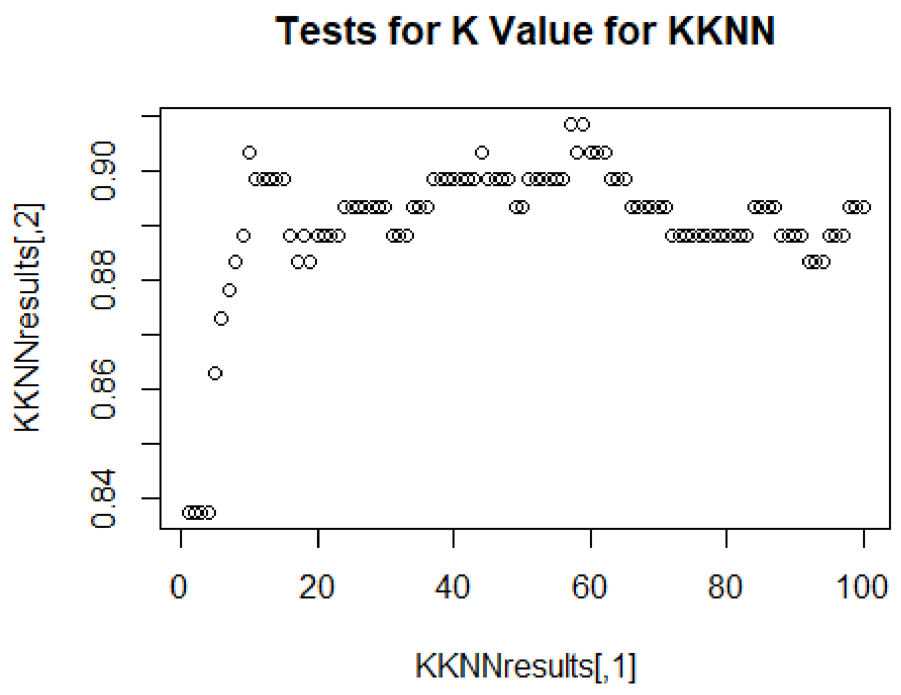
The first attempt shows good result with accuracy at **90.36%**.

Let's try 0 to 100 K values in a loop, log the accuracies and see if we can find a best K value for our model.

```r
iter = 100

KKNNresults <- matrix(NA, nrow=iter, ncol=2)
colnames(results) <- c("K-Value","Accuracy")

#looping Ks on the model and documenting accuracy percentage
for(i in 1:iter){
    model_kknn <- kknn( R1~.,
                    train_data,
                    test_data,
                    k=i,
                    distance = 2,
                    kernel = "optimal",
                    scale= TRUE
                    )
    predictions <- predict(model_kknn,newdata = test_data[,1:10])
    predictions01 <- as.integer(predictions+0.5)
    KKNNresults[i,] <- c(i, sum(predictions01 == test_data[,11]) /
nrow(test_data))
}

KKNNresults

##          [,1]      [,2]
##   [1,]      1 0.8375635
##   [2,]      2 0.8375635
##   [3,]      3 0.8375635
##   [4,]      4 0.8375635
##   [5,]      5 0.8629442
##   [6,]      6 0.8730964
##   [7,]      7 0.8781726
##   [8,]      8 0.8832487
##   [9,]      9 0.8883249
##  [10,]     10 0.9035533
##  [11,]     11 0.8984772
##  [12,]     12 0.8984772
##  [13,]     13 0.8984772
##  [14,]     14 0.8984772
##  [15,]     15 0.8984772
##  [16,]     16 0.8883249
##  [17,]     17 0.8832487
##  [18,]     18 0.8883249
##  [19,]     19 0.8832487
##  [20,]     20 0.8883249
##  [21,]     21 0.8883249
##  [22,]     22 0.8883249
##  [23,]     23 0.8883249
##  [24,]     24 0.8934010
##  [25,]     25 0.8934010
##  [26,]     26 0.8934010
##  [27,]   27 0.8934010
##  [28,]   28 0.8934010
##  [29,]   29 0.8934010
##  [30,]   30 0.8934010
##  [31,]   31 0.8883249
##  [32,]   32 0.8883249
##  [33,]   33 0.8883249
##  [34,]   34 0.8934010
##  [35,]   35 0.8934010
##  [36,]   36 0.8934010
##  [37,]   37 0.8984772
##  [38,]   38 0.8984772
##  [39,]   39 0.8984772
##  [40,]   40 0.8984772
##  [41,]   41 0.8984772
##  [42,]   42 0.8984772
##  [43,]   43 0.8984772
##  [44,]   44 0.9035533
##  [45,]   45 0.8984772
##  [46,]   46 0.8984772
##  [47,]   47 0.8984772
##  [48,]   48 0.8984772
##  [49,]   49 0.8934010
##  [50,]   50 0.8934010
##  [51,]   51 0.8984772
##  [52,]   52 0.8984772
##  [53,]   53 0.8984772
##  [54,]   54 0.8984772
##  [55,]   55 0.8984772
##  [56,]   56 0.8984772
##  [57,]   57 0.9086294
##  [58,]   58 0.9035533
##  [59,]   59 0.9086294
##  [60,]   60 0.9035533
##  [61,]   61 0.9035533
##  [62,]   62 0.9035533
##  [63,]   63 0.8984772
##  [64,]   64 0.8984772
##  [65,]   65 0.8984772
##  [66,]   66 0.8934010
##  [67,]   67 0.8934010
##  [68,]   68 0.8934010
##  [69,]   69 0.8934010
##  [70,]   70 0.8934010
##  [71,]   71 0.8934010
##  [72,]   72 0.8883249
##  [73,]   73 0.8883249
##  [74,]   74 0.8883249
##  [75,]   75 0.8883249
##  [76,]   76 0.8883249
##  [77,]   77 0.8883249
##  [78,]   78 0.8883249
##  [79,]   79 0.8883249
##  [80,]   80 0.8883249
##  [81,]   81 0.8883249
##  [82,]   82 0.8883249
##  [83,]   83 0.8883249
##  [84,]   84 0.8934010
##  [85,]   85 0.8934010
##  [86,]   86 0.8934010
##  [87,]   87 0.8934010
##  [88,]   88 0.8883249
##  [89,]   89 0.8883249
##  [90,]   90 0.8883249
##  [91,]   91 0.8883249
##  [92,]   92 0.8832487
##  [93,]   93 0.8832487
##  [94,]   94 0.8832487
##  [95,]   95 0.8883249
##  [96,]   96 0.8883249
##  [97,]   97 0.8883249
##  [98,]   98 0.8934010
##  [99,]   99 0.8934010
## [100,]  100 0.8934010
```

Plotting the matrix of results:

```
plot(KKNNresults)
title("Tests for K Value for KKNN")
```



```
max(KKNNresults[,2])

## [1] 0.9086294

which.max(KKNNresults[,2])

## [1] 57
```

After testing the model on 0 to 100 K values, it appears that maximum accuracy is **90.86%** at **K = 57**. Let's run the final model for K=57.

```
model_kknn <- kknn( R1~.,
                    train_data,
                    test_data,
                    k=57,
                    distance = 2,
                    kernel = "optimal",
                    scale= TRUE
                    )
predictions <- predict(model_kknn,newdata = test_data[,1:10])
predictions01 <- as.integer(predictions+0.5)
confusionMatrix(factor(test_data[,11], levels =
c(1,0)),factor(predictions01,levels=c(1,0)))

## Confusion Matrix and Statistics
##
##           Reference
## Prediction   1   0
##          1  78   8
##          0  10 101
##
##                Accuracy : 0.9086
##                  95% CI : (0.8594, 0.9449)
##     No Information Rate : 0.5533
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.8148
##
##  Mcnemar's Test P-Value : 0.8137
##
##             Sensitivity : 0.8864
##             Specificity : 0.9266
##          Pos Pred Value : 0.9070
##          Neg Pred Value : 0.9099
##              Prevalence : 0.4467
##          Detection Rate : 0.3959
##    Detection Prevalence : 0.4365
##       Balanced Accuracy : 0.9065
##
##        'Positive' Class : 1
##
```

KKNN model has only slightly improved the accuracy of prediction from 89.34% (from KSVM model) to 90.86%. Whether the small improvement is worth it or not depends on the business use of the model. If cost of misclassifying even 1 loan application is significant, even a small improvement could be useful. In some other use case like classifying spend data, as long as top 80% supplier's spend is properly classified, tail spend would not matter. In that scenario, small accuracy improvements would not be that exciting. It is also pertinent to see how much effort is spent trying to improve accuracy by a small percentage, at which point it becomes an ROI decision (i.e. is time and money spent improving the model worth it!)

Lastly, I will run a loop to test other kernal methods of KKNN just to see if they make any different (will keep K=57).

```
kernels <-
c("rectangular","triangular","epanechnikov","biweight","triweight","cos","inv
", "gaussian", "rank")

iter = length(kernels)


kernel_results <- matrix(NA, nrow=iter, ncol=2)
colnames(kernel_results) <- c("Kernel","Accuracy")

#looping on the model and documenting accuracy percentage
for(i in 1:iter){
    model_kknn <- kknn( R1~.,
                    train_data,
                    test_data,
                    k=57,
                    distance = 2,
                    kernel = kernels[[i]],
                    scale= TRUE
                    )
    predictions <- predict(model_kknn,newdata = test_data[,1:10])
    predictions01 <- as.integer(predictions+0.5)
    kernel_results[i,] <- c(kernels[[i]], sum(predictions01 ==
test_data[,11]) / nrow(test_data))
}

kernel_results

##       Kernel         Accuracy
##  [1,] "rectangular"  "0.888324873096447"
##  [2,] "triangular"   "0.913705583756345"
##  [3,] "epanechnikov" "0.898477157360406"
##  [4,] "biweight"     "0.893401015228426"
##  [5,] "triweight"    "0.888324873096447"
##  [6,] "cos"          "0.913705583756345"
##  [7,] "inv"          "0.893401015228426"
##  [8,] "gaussian"     "0.898477157360406"
##  [9,] "rank"         "0.903553299492386"
```

Trianuglar and cos kernels show improvement of accuracy over others. I could not find a good explanation of how these kernels diff, thus can't go into details of why the two kernels are showing better results. I have posted on Piazza to get more material on these kernels.