HW1

Question 2.1

I work for an engineer-to-order company which designs and manufactures packaging automation machinery. Each project is different based on customer product requirements, so it poses its own set of challenges. Sales and Engineering teams use past data and experience to identify what level of commitments can be made to a certain project, however it is not uncommon for projects to fail their expected delivery date due to challenging requirements, resulting in contractual penalties. A classification model can be made to predict if a project can be delivered in time.

What are we looking to answer? Will a card issuance packaging machine be delivered without penalty

No: Failure to deliver project or delivered late
Yes: Successfully delivered on time as per customer requirements

Predictors:

Required machine throughput (products per hour) – higher run speed is more challenging and requires more testing
Maximum allowable reject rate (%) – Achieving low reject rates typically requires additional testing time
Number of product variations machine is expected to run – Higher number of product variations requires more design and testing time
Avg Number of system processed required for each product variation – Higher number of system processes requires more modules resulting in more design, manufacturing, and testing time

Number of Field Engineers assigned to project: With the bottleneck typically being in the commissioning/testing stage, number of Field Engineers working on the project can be a predictor for a project's success

Number of weeks allocated to project**:** More time allocated to a project increases chances of successful on-time delivery

Question 2.2

Part 1

To find the most optimum value of C, we need to test our model with a wide range of C values and identify which value yields the highest prediction accuracy.

To start, I set C as a very small value, and scaled it by a magnitude of 10 while capturing the results, which are pasted in the table below.

C	Accuracy
1e-09	0.5474006
1e-08	0.5474006
1e-07	0.5474006
1e-06	0.5474006
1e-05	0.5474006
1e-04	0.5474006
1e-03	0.8379205
1e-02	0.8639144
1e-01	0.8639144
1e+00	0.8639144
1e+01	0.8639144
1e+02	0.8639144
1e+03	0.8623853
1e+04	0.8623853
1e+05	0.8639144
1e+06	0.6253823
1e+07	0.5458716
1e+08	0.6636086
1e+09	0.8027523
1e+10	0.4923547

As we can see, extreme small and large values of C both result in poor prediction accuracy. This may be because we are completely ignoring error as C gets large, or completely ignoring margin as C gets small which does not strike a good balance. Outside of this overall trend we also see accuracy jump up and down as C changes.

To zoom into the range that had the higher accuracy (1e-03 to 1e05), I re-ran the for loop starting at lambda=0.001 and tried smaller step sizes (ex: lambda=lambda*5, *2, *1.2 etc). This showed C value of 0.0014 to yield the highest accuracy of 86.69725%

Formula with a_m and a_0:

$0.0004345763\mathbf{x_{1j}} + 0.0178248684\mathbf{x_{2j}} + 0.0321827192\mathbf{x_{3j}} + 0.1129970897\mathbf{x_{4j}} + 0.4700066828\mathbf{x_{5j}} - 0.2271346763\mathbf{x_{6j}} + 0.1712967805\mathbf{x_{7j}} - 0.0058694133\mathbf{x_{8j}} - 0.0243132225\mathbf{x_{9j}} + 0.0929156233\mathbf{x_{10j}} - 0.1175208 = 0$

```r
library(readr)
library(kernlab)
library(ggplot2)
library(kknn)

data <- as.matrix(read.delim("credit_card_data-headers.txt"))

model_ksvm <- function(data,lambda){
  #x= training data (columns 1 to 10)
  #y= response vector (R1 column)
  model <- ksvm(data[,1:10],data[,11],type="C-svc",kernel="vanilladot",C=lamb
da,scaled=TRUE)

  #calculate a and a0
  a <- colSums(model@xmatrix[[1]] * model@coef[[1]])
  a0 <- -model@b

  #see what the model predicts
  pred <- predict(model,data[,1:10])

  #calculate percent of predictions that match the actual results
  accuracy <- (sum(pred == data[,11]) / nrow(data))*100
  return(accuracy)
}

#setup empty vectors to record results
lambdas = c()
accuracy = c()

#set C value start point
lambda = 0.001

#test incrementing c values and record results
for(i in 1:30){
  lambda = lambda*1.5
  lambdas = append(lambdas,lambda)
  accuracy = append(accuracy,model_ksvm(data,lambda))
}
```

```
lambda_results <- data.frame(c=lambdas, accuracy=accuracy)
print(lambda_results)
```

Part 2

rbfdot and laplacedot kernels surprisingly gave 100% prediction accuracy, with accuracy rising as C increases rbfdot -> C=2216837 100% laplacedot -> C=85 100%

Part 3

In order to train our model and test it for each data point without the test data point being considered in the nearest neighbors, I used a for loop which considered the following:

Train set: all data points except for ith point
Test set: ith point

I nested this loop in another for loop which tested various k values and recorded the prediction accuracies.

This test showed k-values of 12 and 15 both yield the highest prediction accuracy of 85.3211%.

We could also further narrow down a preferred k value by seeing which k value gives more conservative estimates in this case as we would prefer to not give a loan to someone who would likely repay it rather than give a loan to someone that wont repay it.

Extreme small and large k values result in decreasing accuracy as we consider too few points or paint too broad of a brush by considering a large area of data points.

```
library(readr)
library(kernlab)
library(ggplot2)
library(kknn)

data <- data.frame(read.delim("credit_card_data-headers.txt"))

#setup empty df to record accuracies
accuracies <- data.frame(k=c(),accuracy=c())

for(k_val in 1:30){
  #setup empty vector to record predictions
  results = c()

  for(i in 1:dim(data)[1]){
    model <- kknn(R1 ~ .,data[-i,],data[i,],k=k_val, scale=TRUE)
```

```r
    result = model$fitted.values

    #append rounded result to vector
    results <- c(results,round(result))
  }

  #append k-value and associated prediction accuracy to dataframe
  accuracy <- data.frame(k_val,sum(results == data[,11]) / nrow(data))

  accuracies <- rbind(accuracies,accuracy)
}

print(accuracies)
```

```
##    k_val sum.results....data...11...nrow.data.
## 1      1                             0.8149847
## 2      2                             0.8149847
## 3      3                             0.8149847
## 4      4                             0.8149847
## 5      5                             0.8516820
## 6      6                             0.8455657
## 7      7                             0.8470948
## 8      8                             0.8486239
## 9      9                             0.8470948
## 10    10                             0.8501529
## 11    11                             0.8516820
## 12    12                             0.8532110
## 13    13                             0.8516820
## 14    14                             0.8516820
## 15    15                             0.8532110
## 16    16                             0.8516820
## 17    17                             0.8516820
## 18    18                             0.8516820
## 19    19                             0.8501529
## 20    20                             0.8501529
## 21    21                             0.8486239
## 22    22                             0.8470948
## 23    23                             0.8440367
## 24    24                             0.8455657
## 25    25                             0.8455657
## 26    26                             0.8440367
## 27    27                             0.8409786
## 28    28                             0.8379205
## 29    29                             0.8394495
## 30    30                             0.8409786
```