Homework 1

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.

Situation:

A marathon organizer wants to select runners who are most likely to finish the race to minimize the number of participants who do not complete it. By building a classification model, it can predict whether a registered runner will finish the race or not based on historical data and characteristics of the runners.

Predictors I would use:

- Race Completion History: Previous races the runner has successfully completed
- Training Mileage: The average number of miles the runner trains per week
- Race Goal Time: The runner's target time, which might indicate their pacing strategy and realism in goal setting.
- Runner's Age: Age of the runner
- Health History: Information about prior injuries or medical conditions that could affect their performance

This approach could be very useful for classification because it assigns a binary outcome (finished or did not finish) to each potential runner, to ensure a higher completion rate.

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. Support Vector Machine Classification

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 lambda 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 a0 and a1…am. Instead, you need to do the last step of the calculation yourself.

Answer:

In this SVM classification task, I implemented a systematic approach to find a good classifier for the credit approval dataset. This application uses a linear kernel ('Vanilladot') as the base classifier, applied feature scaling (scaled = TRUE) to ensure all variables contribute appropriately, and I used a 5 fold cross-validation throughout the training process to ensure a robust model evaluation.

```
# Best practice
rm(list = ls())

# Set up directory and packages
setwd('~/Desktop/GTX/Homework 1/')
# Load packages
pacman::p_load(tidyverse, kernlab, caret, kknn, modelr, ggthemes, corrplot)

# Load the kernlab library which contains the ksvm function
library(kernlab)
library(tidyverse)
library("kknn")

# Data manipulation
credit_data <- read.table("/Users/cn/Desktop/GTX/Homework 1/credit_card_data-headers.txt
                          header = TRUE) %>%
  as_tibble() %>%
  dplyr::rename(.,target = R1) # Rename 11th column

# Set random seed for reproducible results
set.seed(222)

# Define model parameters
svm_params <- list(
  type = 'C-svc',
  kernel = 'vanilladot', # Linear kernel
  cross = 5,
  scaled = TRUE
```

```
)

# Define a smaller, more focused range of cost values
cost_values <- 10^seq(-2, 3, by = 0.5)#logarithmic sequence from 0.01 to 1000

# Create vectors to store results
error_results <- numeric(length(cost_values))
names(error_results) <- cost_values

# Train SVM for each cost value
for (i in seq_along(cost_values)) {
  current_cost <- cost_values[i]

  svm_model <- ksvm(
    as.factor(target) ~ .,
    data = credit_data,
    scaled = svm_params$scaled,
    type = svm_params$type,
    kernel = svm_params$kernel,
    C = current_cost,
    cross = svm_params$cross
  )

  error_results[i] <- cross(svm_model)
}

# Create performance summary
performance_summary <- data.frame(
  svm_cost = cost_values,
  svm_error = error_results,
  svm_accuracy = 1 - error_results
) %>%
  arrange(svm_error) %>%
  head(1)

# Fit final SVM model with the best cost
best_cost <- performance_summary$svm_cost
final_svm <- ksvm(
  as.factor(target) ~ .,
  data = credit_data,
  scaled = svm_params$scaled,
  type = svm_params$type,
  kernel = svm_params$kernel,
  C = best_cost,
```

```
  cross = svm_params$cross
)

# Get feature names
feature_names <- colnames(credit_data)[colnames(credit_data) != "target"]

# Extract coefficients
svm_coef <- final_svm@coef[[1]]
a <- colSums(final_svm@xmatrix[[1]] * svm_coef)

# Create the formula dataframe
svm_formula <- data.frame(
  predictor = c(feature_names, "Intercept"),
  value = c(a, final_svm@b)
) %>%
  pivot_wider(names_from = predictor, values_from = value) %>%
  as_tibble()
```

I focused on finding the best cost (C) value using a logarithmic sequence from 10^-2 to 10^3. This range was chosen deliberately to investigate both high-bias scenarios (low C values) and high-variance scenarios (high C values). Then I used a for loop to test each of these cost values to find which one gives the best performance, and C = 1000 performed the best among these options.

```
## [1] "SVM Model Formula Coefficients:"
```

```
## # A tibble: 1 x 11
##          A1       A2      A3       A8    A9       A10      A11       A12     A14
##       <dbl>    <dbl>   <dbl>    <dbl> <dbl>     <dbl>    <dbl>     <dbl>   <dbl>
## 1 -0.000215 0.000710 0.00116 0.000567 0.999 -0.000504 0.000716 -0.000913 7.97e-4
## # i 2 more variables: A15 <dbl>, Intercept <dbl>
```

The SVM formula is: f(x) = -0.000215(A1) + 0.000710(A2) + 0.00116(A3) + 0.000567(A8) + 0.999(A9) - 0.000504(A10) + 0.000716(A11) - 0.000913(A12) + 0.000797(A14) + A15(c0.000502) + Intercept

```
## [1] "Final SVM Model:"
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 1000
##
## Linear (vanilla) kernel function.
##
```

```
## Number of Support Vectors : 275
##
## Objective Function Value : -178024.8
## Training error : 0.137615
## Cross validation error : 0.137628
```

The training error is 0.137615, meaning about 13.76% of the training data was misclassified, and achieved an accuracy of approximately 86.25%. This data indicate that the model has been trained well and is likely to be as effective on new data.

2. Non-linear Kernels (Optional)

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:

In the next assignment, I implemented an SVM model using the Radial Basis Function Kernel, because this model can capture more complex decision boundaries than the linear kernel, which could be beneficial for this credit approval dataset where relationships between variables could be non-linear.

```r
# Flexible SVM

# Adjust cost range and kernel
cost_values_flex <- as.list(seq(from = 0.1, to = 100, by = 1))
error_results_flex <- list()

for (i in seq_along(cost_values_flex)) {
  current_cost <- cost_values_flex[[i]]

  svm_model_flex <- ksvm(
    target ~ .,
    data = credit_data,
    scaled = TRUE,
    type = 'C-svc',
    kernel = 'rbfdot',
    C = current_cost,
    cross = 10
  )
  error_results_flex[[i]] <- svm_model_flex@error
}

# Combine and find the best flexible SVM model
error_df_flex <- do.call(rbind, error_results_flex) %>% as.data.frame()
cost_df_flex <- do.call(rbind, cost_values_flex) %>% as.data.frame()
```

```r
# Ensure unique column names before combining
colnames(error_df_flex) <- "svm_error"
colnames(cost_df_flex) <- "svm_cost"

performance_summary_flex <- cbind(error_df_flex, cost_df_flex) %>%
  filter(svm_error == min(svm_error)) %>%
  arrange(svm_cost) %>%
  head(1)

# Final flexible SVM
best_cost_flex <- performance_summary_flex$svm_cost
final_svm_flex <- ksvm(
  target ~ .,
  data = credit_data,
  scaled = TRUE,
  type = 'C-svc',
  kernel = 'rbfdot',
  C = best_cost_flex,
  cross = 10
)
```

I explored cost values ranging from 0.1 to 100, using 10-fold cross validation for each cost value to ensure error estimation. In this model with the optimal model cost value of 97.1 the RBF kernel achieved an error rate of 0.0489 compared to 0.1376 to the error rate of the linear kernel.

```r
print(performance_summary_flex)
```

svm_error svm_cost 1 0.04281346 97.1

```r
print(final_svm_flex)
```

Support Vector Machine object of class “ksvm”

SV type: C-svc (classification) parameter : cost C = 97.1

Gaussian Radial Basis kernel function. Hyperparameter : sigma = 0.0897575422520958

Number of Support Vectors : 243

Objective Function Value : -9066.356 Training error : 0.04893 Cross validation error : 0.181865

3. k-Nearest Neighbors Classification

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. - 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).

Answer:

In this segment, I applied the k-NN classification, using the kknn function from the kknn package. I scaled the variables to ensure equal consideration of all variables in distance calculations.

```
# kNN Model

# Set seed and define k values
set.seed(333)
k_values <- seq(1,50,by = 2)

# Train kNN using cross-validation
knn_results <- train.kknn(
  target ~ .,
  data = credit_data,
  kmax = max(k_values),
  kcv = 10,
  scale = TRUE
)

# Extract optimal k value
summary(knn_results)
```

Call: train.kknn(formula = target ~ ., data = credit_data, kmax = max(k_values), scale = TRUE, kcv = 10)

Type of response variable: continuous minimal mean absolute error: 0.1850153 Minimal mean squared error: 0.107471 Best kernel: optimal Best k: 49

```
# Test final kNN model on entire dataset
final_knn <- kknn(
  target ~.,
  train=credit_data,
  test = credit_data,
  k = knn_results$best.parameters$k,
  scale = TRUE
)

#Calculate accuracy
predictions <- fitted(final_knn) %>%
  as_tibble() %>%
  mutate(predicted = ifelse(value > 0.5,1,0)) %>%
```

```
  cbind(actual = credit_data$target) %>%
  mutate(correct = predicted == actual)

final_accuracy <- mean(predictions$correct)
print(final_accuracy)
```

[1] 0.882263

```
# Knn Cross Validation
set.seed(111)
final_knn2 <- train.kknn(target ~.,
                         data = credit_data,
                         kmax = 60,
                         kcv= 10,
                         scale = TRUE)
summary(final_knn2)
```

Call: train.kknn(formula = target ~ ., data = credit_data, kmax = 60, scale = TRUE, 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

```
set.seed(111)
final_knn3 <- train.kknn(target ~.,
                         data = credit_data,
                         kmax = 70,
                         kcv= 10,
                         scale = TRUE)
summary(final_knn3)
```

Call: train.kknn(formula = target ~ ., data = credit_data, kmax = 70, scale = TRUE, 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

```
set.seed(111)
final_knn4 <- train.kknn(target ~.,
                         data = credit_data,
                         kmax = 10,
                         kcv= 10,
                         scale = TRUE)
summary(final_knn4)
```

Call: train.kknn(formula = target ~ ., data = credit_data, kmax = 10, scale = TRUE, 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

```
library(caret)
set.seed(555)

# Initial partition for training data (60%)
partition <-createDataPartition(credit_data$target, p=0.6,
                                list = F)
train <- credit_data[partition,]
holdout <- credit_data[-partition,]

# Split holdout into test and validation (50% each, so 20% of total each)
partition_valid <- createDataPartition(holdout$target, p=0.5,
                                       list = FALSE)
test <- holdout[partition_valid,]
validation <- holdout[-partition_valid,]

# Verify splits
print("Dimensions of splits:")
```

[1] “Dimensions of splits:”

```
print(dim(train))
```

[1] 393 11

```
print(dim(test))
```

[1] 131 11

```
print(dim(validation))
```

[1] 130 11

```
# Set up a 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 store accuracy values
test_accuracy <- list()

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

```
  k <- possible_k[[i]]

  final_knn <- kknn(
    target ~ .,
    train = train,
    test = test,
    k = k,
    scale = TRUE
  )

  predictions <- fitted(final_knn) %>%
    as_tibble() %>%
    mutate(value = ifelse(value > 0.5, 1, 0)) %>%
    cbind(., test$target) %>%
    mutate(acc = value == test$target)

  test_accuracy[[i]] <- mean(predictions$acc)
}

# Combine 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,]

# Fit the best model on the combined training + testing data
knn_best_fit <- kknn(
  target ~ .,
  train = rbind(train, test), # Combine train and test datasets
  test = validation,
  k = performance_test$knn_k, # Best K value
  scale = TRUE
)

# Calculate the validation set accuracy
best_fitted <- fitted(knn_best_fit) %>%
```

```
  as_tibble() %>%
  mutate(value = ifelse(value > 0.5, 1, 0)) %>%
  cbind(actual = validation$target) %>%  # Bind the actual target values
  mutate(acc = value == actual)

# Accuracy of the best-fit kNN model on the validation dataset
valid_accuracy <- mean(best_fitted$acc)
# Output results
print("Best K value from testing data:")
```

[1] “Best K value from testing data:”

```
print(performance_test$knn_k)
```

[1] 7

```
print("Validation accuracy of the best-fit kNN model:")
```

[1] “Validation accuracy of the best-fit kNN model:”

```
print(valid_accuracy)
```

[1] 0.8

Through iterative testing and validation, different neighborhood sizes were evaluated. While small k values showed signs of overfitting, and large values above 50 tended to smooth out important patterns, k=49 emerged as particularly effective, yielding 88.22% accuracy across the dataset.

The evaluation process involved splitting the data into three parts: 60% for initial training, with the remaining 40% divided equally between testing and validation sets. When applying the model to the validation data, it achieved 80% accuracy, suggesting robust real-world performance.