Task 1: Lab 1: Block 1

```
library(kknn)
library(tidyverse)
setwd("C:/Users/Sofia Danielson/OneDrive - Linköpings universitet/R programmering/maskininlärning/ML_lab
#read in data
data <- read_csv("Task_1/optdigits.csv")

colnames(data)[ncol(data)] <- "digit"
data$digit <- factor(data$digit)
#partition data
n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.5))
train=data[id,]
id1=setdiff(1:n, id)
set.seed(12345)
id2=sample(id1, floor(n*0.25))
valid=data[id2,]
id3=setdiff(id1,id2)
test=data[id3,]
```

2. The misclassification level for both the train and test data seem to be at a fairly low level, at 0.0424 and 0.0492 respectively

```
# Predict on training data
knn_model <- kknn(digit ~ ., train = train, test = train, k = 30, kernel = "rectangular")
train_pred <- fitted(knn_model)

# Predict on test data
test_knn_model <- kknn(digit ~ ., train = train, test = test, k = 30, kernel = "rectangular")
test_pred <- fitted(test_knn_model)

# Misclassification error
train_misclassification <- sum(train$digit != train_pred) / nrow(train)
print(paste("Training Misclassification Error:", round(train_misclassification, 4)))
```

```
## [1] "Training Misclassification Error: 0.0424"
```

```
test_misclassification <- sum(test$digit != test_pred) / nrow(test)
print(paste("Test Misclassification Error:", round(test_misclassification, 4)))
```

```
## [1] "Test Misclassification Error: 0.0492"
```

Based from the calculations of the error rate for each digit on the training and testing data, the digits that appears the easiest to classify correctly for the knn model are 0, 2, 3, 6 and 7

The digits that seem to be the harder to predict are 1, 4, 5, 8, 9

```r
#confusion matrices
train_conf_matrix <- table(train$digit, train_pred)
print("Training Confusion Matrix:")
```

```
## [1] "Training Confusion Matrix:"
```

```r
print(train_conf_matrix)
```

```
##    train_pred
##       0   1   2   3   4   5   6   7   8   9
##   0 177   0   0   0   1   0   0   0   0   0
##   1   0 174   9   0   0   0   1   0   1   3
##   2   0   0 170   0   0   0   0   1   2   0
##   3   0   0   0 197   0   2   0   1   0   0
##   4   0   1   0   0 166   0   2   6   2   2
##   5   0   0   0   0   0 183   1   2   0  11
##   6   0   0   0   0   0   0 200   0   0   0
##   7   0   1   0   1   0   1   0 192   0   0
##   8   0  10   0   1   0   0   2   0 190   2
##   9   0   3   0   4   2   0   0   2   4 181
```

```r
test_conf_matrix <- table(test$digit, test_pred)
print("Test Confusion Matrix:")
```

```
## [1] "Test Confusion Matrix:"
```

```r
print(test_conf_matrix)
```

```
##    test_pred
##      0  1  2  3  4  5  6  7  8  9
##   0 97  0  0  0  0  0  1  0  0  0
##   1  0 91  3  0  0  0  0  0  0  3
##   2  0  0 93  1  0  0  0  0  1  0
##   3  0  0  0 95  0  0  0  2  1  0
##   4  1  0  0  0 89  0  1  5  1  3
##   5  0  1  0  1  0 79  1  0  0  5
##   6  0  0  0  0  0  0 94  0  0  0
##   7  0  2  0  0  0  1  0 91  1  0
##   8  0  3  0  1  0  0  1  0 86  0
##   9  0  0  0  4  0  0  0  2  1 94
```

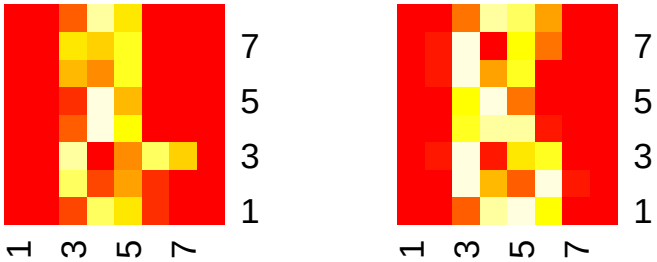
3. The first two digits seemed pretty definitely easier to recognize as an eight (although the first one of them was maybe a little difficult to recognize as well). The other did appear much less recognizable as symbolizing the digit eight. Looking at the results for these cases, the easiest to predict where indeed correct predictions, while the hardest ones had larger probabilities on 6, 1 and 1 respectively.

```r
# # Extract predicted probabilities and identify cases for digit "8"
train_probs <- knn_model$prob

# Filter for only rows where the true digit is "8"
```

```r
eight_indices <- which(train$digit == 8)

# Get predicted probabilities of class 8
eight_probs <- vapply(eight_indices, function(i) as.numeric(train_probs[i, "8"]), numeric(1))

# Find indices of the 2 highest and 3 lowest probabilities
easiest_cases <- order(eight_probs, decreasing = TRUE)[1:2]
hardest_cases <- order(eight_probs)[1:3]

easiest_indexes <- eight_indices[easiest_cases]
hardest_indexes <- eight_indices[hardest_cases]

#Function to visualize a digit as an 8x8 heatmap
visualize_digit <- function(data_row) {
  matrix_data <- matrix(as.numeric(data_row[1:64]), nrow = 8, ncol = 8, byrow = TRUE)
  heatmap(matrix_data, Colv = NA, Rowv = NA, scale = "none",
          col = heat.colors(16))
}

# Visualize easiest cases
cat("Easiest cases of digit '8':\n")
```

```
## Easiest cases of digit '8':
```

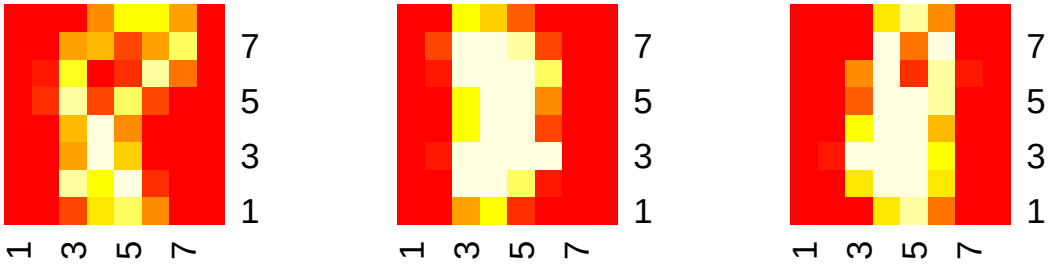
```r
for (i in easiest_indexes) {
  visualize_digit(train[i,])
}
```




```r
# Visualize hardest cases
cat("Hardest cases of digit '8':\n")
```

```
## Hardest cases of digit '8':
```

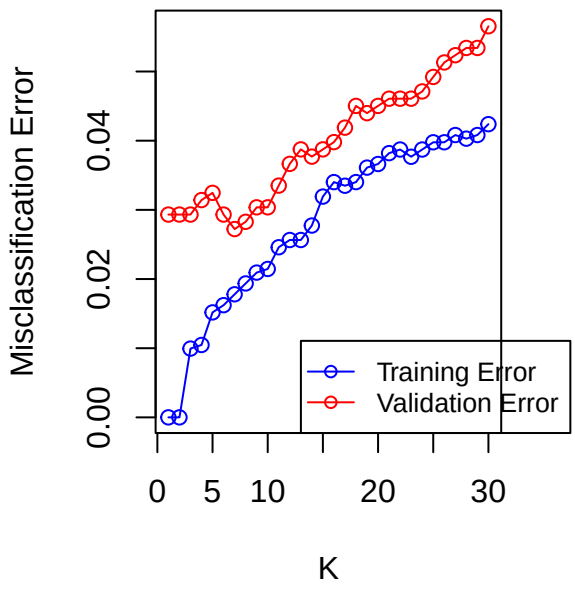
```r
for (i in hardest_indexes) {
  visualize_digit(train[i,])
}
```




```
# cat("probabilities for hardest cases of digit '8':\n")
# print(knn_model$prob[hardest_indexes,])
#
# cat("probabilities for easiest cases of digit '8':\n")
# print(knn_model$prob[easiest_indexes,])
```

4. As observed from the graph below, the misclassification seems to be generally lower for predictions on the training data than the validation data, which is to be expected since the model is trained to minimize the error on the training data. The error appears to only increase with K for the training data. This is simply because the model becomes more sensitive to local anomalies and is more likely to become over fitted to the training data for few K's. The validation error decreases a little bit from 1 to 7 K, but later grows as K also grows. The model was probably slightly over fit for the lower values of K, but reached a local optimum at around K = 7. As K grew, it probably began to become under fitted as it averages to many neighbors, including those far away from the test point, which explains why the validation error began to grow after K > 7.

```
train_errors <- numeric(30)
valid_errors <- numeric(30)

for (k in 1:30) {

  knn_model <- kknn(digit ~ ., train = train, test = valid, k = k, kernel = "rectangular")

  # Training predictions
  train_pred <- fitted(kknn(digit ~ ., train = train, test = train, k = k, kernel = "rectangular"))
  train_errors[k] <- mean(train_pred != train$digit)

  # Validation predictions
  valid_pred <- fitted(knn_model)
  valid_errors[k] <- mean(valid_pred != valid$digit)
}

par(xpd = TRUE, mar = c(5, 4, 4, 7))  # Increase the right margin
plot(1:30, train_errors, type = "o", col = "blue", ylim = c(0, max(train_errors, valid_errors)), xlab =
lines(1:30, valid_errors, type = "o", col = "red")
legend("bottomright", inset = c(-0.2, 0), legend = c("Training Error", "Validation Error"), col = c("blu
```




```r
# optimal K
optimal_k <- which.min(valid_errors)
cat("Optimal K:", optimal_k, "\n")
```

```
## Optimal K: 7
```

```r
# Estimate the test error for optimal K
knn_test <- kknn(digit ~ ., train = train, test = test, k = optimal_k, kernel = "rectangular")
test_pred <- fitted(knn_test)
test_error <- mean(test_pred != test$digit)
cat("Optimal K:", optimal_k, "\n")
```

```
## Optimal K: 7
```

```r
cat("Test Error for optimal K:", test_error, "\n")
```

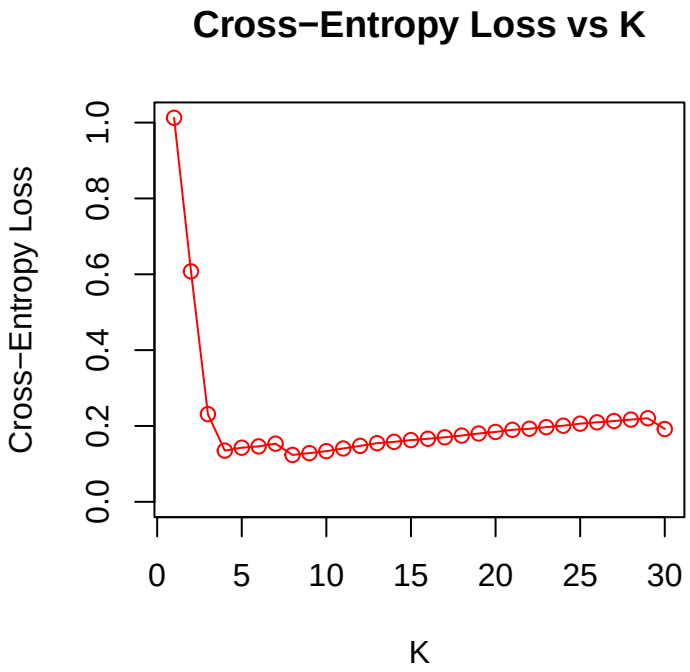
```
## Test Error for optimal K: 0.03870293
```

5.

For the cross entropy results, observations of the graph shows how the loss is the highest for low values of K (specifically $K > 4$), reaches a local minimum at $K = 8$, and starts to slowly increase again as K increases. The high loss for the lower values of K most can probably be explained by the models tendency to become over fitted with low number of neighbors, resulting in incorrect or uncertain predictions. After K grows larger than 8, we can see how the cross entropy loss also increases, which yet again likely due to becoming under fitted.

for a classification task that has a multinomial distribution, using cross entropy will most likely be favorable. Cross entropy loss will take prediction certainty into consideration and will simply not just treat all faulty and correct classifications the same way. For example, when using regular error rate as the error function, the model will be "penalized" the same for making an incorrect classification with 99% certainty as it would have been if it made an incorrect classification with 42% certainty. A similar problem arises for correct corrections, as the regular error rate function would not recognice the difference between a correct classification with high confidence against a correct classification with low confidence. Cross entropy would penalize these predictions more accordingly to the degree of confidence, which will allow for better calibrations and adjustments during the training process, allowing for training models with higher accuracy and confidence.

```
max_k <- 30
cross_entropy_losses <- numeric(max_k)
epsilon <- 1e-15

for (k in 1:max_k) {
  knn_model <- kknn(digit ~ ., train = train, test = valid, k = k, kernel = "rectangular")
  probs <- knn_model$prob
  #Compute cross-entropy loss
  cross_entropy_loss <- 0
  for (i in 1:nrow(valid)) {
    true_class <- as.numeric(valid$digit[i])
    predicted_prob <- probs[i, true_class]  # Get the probability for the true class
    cross_entropy_loss <- cross_entropy_loss - log(predicted_prob + epsilon)
  }
  cross_entropy_losses[k] <- cross_entropy_loss / nrow(valid)  # Normalize by number of
}

# Plot cross-entropy losses
plot(1:max_k, cross_entropy_losses, type = "o", col = "red",
     xlab = "K", ylab = "Cross-Entropy Loss", ylim = c(0, max(cross_entropy_losses)),
     main = "Cross-Entropy Loss vs K")
```




```
# Optimal K
optimal_k <- which.min(cross_entropy_losses)
cat("Optimal K (based on cross-entropy loss):", optimal_k, "\n")
```

```
## Optimal K (based on cross-entropy loss): 8
```