Kernels and Discriminant Analysis

Exercise 01

In this exercise, we will use randomly generated data. Solve the following questions:

(a) Implement a function that returns the kernel matrix K using a polynomial kernel of degree 3 with a constant c = 1. In other words, calculate the $K \in \mathbb{R}^{m \times m}$ matrix such that

$$K_{ij} = (x_{i1} \cdot x_{j1} + 1)^3$$

(b) Calculate the α coefficients using the Ridge Regression method with Kernel and a penalty parameter of $\lambda = 0.001$. Remember that these coefficients are given by

$$\alpha = (K + \lambda I)^{-1} y.$$

For simplification purposes, let the y-intercept b = 0.

(c) Create a two-dimensional graph of the resulting function using $x = x1$ and $y =$ "predicted value". Remember that Kernel Ridge regression makes predictions according to the following expression:

$$f(x) = \sum_{i=1}^{m} \alpha_i k(x_i, x) + b,$$

This means that, for training data, the expression for f is given by $f = \alpha K + b$, where K is the kernel matrix calculated in (a), and f is the vector containing predicted values for the training set.

```r
library(tidyverse)
library(knitr)

# Load the dataset
df <- tibble( x1 = runif(100,-4,4),
y = x1**3 - 2*x1**2 + 5*x1 + 13 + rnorm(100,0,5.0))

# Function to generate the Kernel matrix
kernel <- function(df) {
  m <- nrow(df)
  K <- matrix(0, m, m)

  for (i in 1:m) {
    for (j in 1:m) {
      x1_i <- df$x1[i]
      x1_j <- df$x1[j]
      K[i, j] <- (x1_i * x1_j + 1)**3
    }
  }

  return(K)
}

# a) Calculate the kernel matrix for the given dataset
K <- kernel(df)
print(kable(head(K)))
```

```
##
##
## |           |           |             |           |          |             |             |
## |-----------:|----------:|------------:|----------:|---------:|------------:|------------:|---------
## |  2381.76999| 2111.78018| -1106.701525|  769.45520| 34.212881| -1144.188387| -117.6568053|  2912.065
## |  2111.78018| 1873.15694|  -959.080734|  685.33737| 31.282839|  -991.708838| -100.5055709|  2580.30
## | -1106.70153| -959.08073|  1487.593004| -274.05455| -1.199616|  1529.436415|  264.2777357| -1402.48
## |   769.45520|  685.33737|  -274.054548|  261.42260| 15.332616|  -283.827983|  -24.3528196|   934.01
## |    34.21288|   31.28284|    -1.199616|   15.33262|  2.793920|    -1.272157|   -0.0003854|    39.81
## | -1144.18839| -991.70884|  1529.436415| -283.82798| -1.272157|  1572.495177|  271.1507673| -1449.66
```

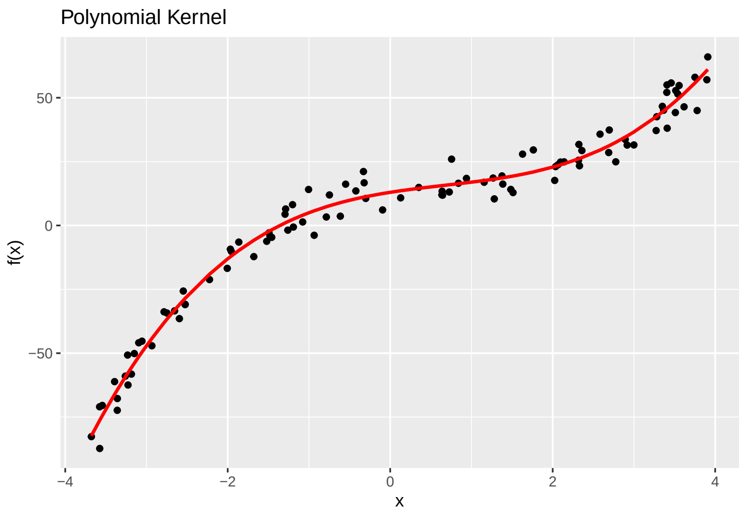
```r
# Calculate the identity matrix multiplied by the penalty
n <- nrow(df)
I <- diag(n) * 0.001

# b) Calculate the alpha coefficients
alpha <- solve(K + I) %*% as.matrix(df$y, ncol = 1)
print(kable(head(alpha)))
```

```
##
##
## |           |
## |----------:|
## |  4057.2066|
## |   309.4900|
## | -4764.8330|
## |  5598.7423|
## | -2209.1733|
## |   329.5568|
```

```r
# Multiply the kernel matrix by the alpha coefficients,
# creating a nx1 with predicted values for Y.
# We store these values in a column named y_k:
Y <- K %*% alpha
Y_K <- as.data.frame(Y)
df$y_k <- Y_K$V1
print(kable(head(df)))
```

```
##
##
## |         x1|         y|       y_k|
## |----------:|---------:|---------:|
## |  3.5149215|  52.81356|  48.75635|
## |  3.3655637|  45.11861|  44.80913|
## | -3.2273044| -62.45169| -57.68686|
## |  2.3225253|  31.72575|  26.12701|
## |  0.6390922|  13.39936|  15.60854|
## | -3.2601630| -58.98614| -59.31569|
```

```
# c) Create a 2D plot for the output function
plot <- ggplot(df, aes(x = x1, y = y_k)) +
  geom_point(aes(x = x1, y = y)) +
  geom_line(color = "red", size = 1) +
  xlab("x") +
  ylab("f(x)") +
  ggtitle("Polynomial Kernel")
print(plot)
```



Exercise 02

Repeat the previous exercise for the $sinc(x)$ function, which is defined as follows:

$$sinc(x) = \begin{cases} sin(x)/x & \text{if } x \neq 0, \\ 1 & \text{otherwise.} \end{cases}$$

The $sinc(x)$ function is often used to test regression algorithms. Assume that the ϵ error follows a normal distribution with mean $= 0$ and variance $= 0.05$. For this exercise, use the Gaussian kernel with $\sigma = 1$. The kernel is defined as follows:

$$K_{ij} = e^{-(x_{i1} - x_{j1})^2}.$$

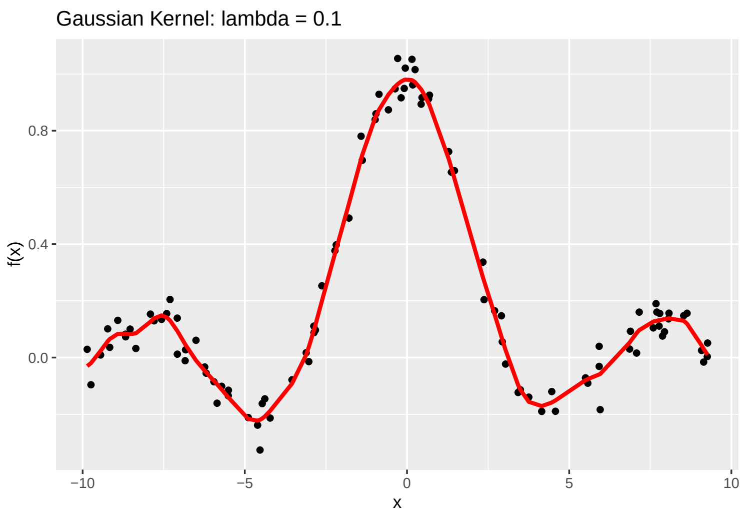
Test different values for the λ penalty.

```
# Load the data
df <- tibble( x1 = runif(100,-10,10),
y = if_else(x1==0,1,sin(x1)/x1) + rnorm(100,0,0.05))

# Function to generate the Kernel matrix
```

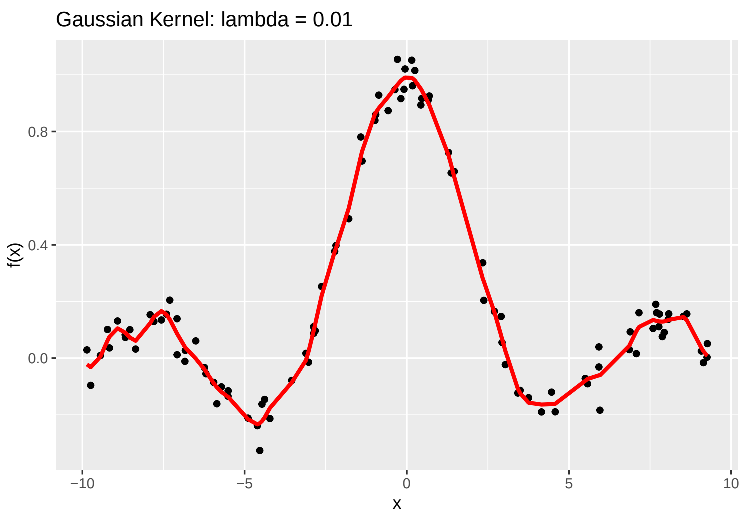
```r
kernel <- function(df) {
  m <- nrow(df)
  K <- matrix(0, m, m)

  for (i in 1:m) {
    for (j in 1:m) {
      x1_i <- df$x1[i]
      x1_j <- df$x1[j]
      K[i, j] <- exp(-(x1_i - x1_j)**2)
    }
  }

  return(K)
}

# a) Calculate the kernel matrix for the given dataset
K <- kernel(df)

# Calculate the identity matrix multiplied by the penalty
# In this case, we compare 3 different values for lambda
n <- nrow(df)
I1 <- diag(n) * 0.1
I2 <- diag(n) * 0.01
I3 <- diag(n) * 0.001

# b) Calculate the alpha coefficients
alpha1 <- solve(K + I1) %*% as.matrix(df$y, ncol = 1)
alpha2 <- solve(K + I2) %*% as.matrix(df$y, ncol = 1)
alpha3 <- solve(K + I3) %*% as.matrix(df$y, ncol = 1)

# Multiply the kernel matrix by the alpha coefficients,
# creating a nx1 with predicted values for Y.
# We store these values in new columns named y_k_01,
# y_k_001 e y_k_0001, for each value of lambda:
Y1 <- K %*% alpha1
Y2 <- K %*% alpha2
Y3 <- K %*% alpha3
Y_K_01 <- as.data.frame(Y1)
Y_K_001 <- as.data.frame(Y2)
Y_K_0001 <- as.data.frame(Y3)
df$y_k_01 <- Y_K_01$V1
df$y_k_001 <- Y_K_001$V1
df$y_k_0001 <- Y_K_0001$V1
print(kable(head(df)))
```

```
##
##
## |         x1|          y|     y_k_01|    y_k_001|   y_k_0001|
## |----------:|----------:|----------:|----------:|----------:|
## | -4.5315428| -0.3260372| -0.2194787| -0.2295085| -0.2344624|
## | -0.1810688|  0.9159216|  0.9735013|  0.9792279|  0.9841436|
## |  0.6933227|  0.9252952|  0.8903347|  0.8940234|  0.8969534|
## |  7.7721881|  0.1110125|  0.1315647|  0.1298474|  0.1293347|
## | -1.3779221|  0.6958073|  0.7149812|  0.7298504|  0.7394302|
## |  2.3753893|  0.2040828|  0.2700559|  0.2711231|  0.2705296|
```

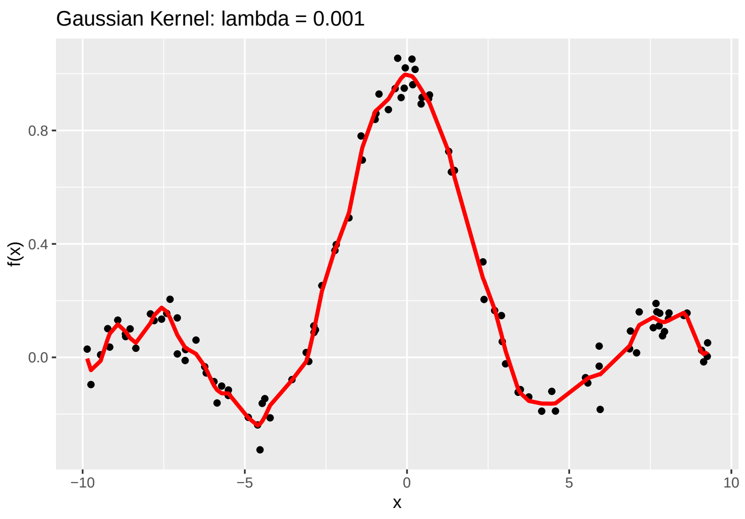
```
# c) Create a 2D plot for the output function

# Plot 1:  lambda = 0.1
plot <- ggplot(df, aes(x = x1, y = y_k_01)) +
  geom_point(aes(x = x1, y = y)) +
  geom_line(color = "red", size = 1.2) +
  xlab("x") +
  ylab("f(x)") +
  ggtitle("Gaussian Kernel: lambda = 0.1")
print(plot)
```



```
# Plot 2: lambda = 0.01
plot <- ggplot(df, aes(x = x1, y = y_k_001)) +
  geom_point(aes(x = x1, y = y)) +
  geom_line(color = "red", size = 1.2) +
  xlab("x") +
  ylab("f(x)") +
  ggtitle("Gaussian Kernel: lambda = 0.01")
print(plot)
```



```r
# Plot 3: lambda = 0.001
plot <- ggplot(df, aes(x = x1, y = y_k_0001)) +
  geom_point(aes(x = x1, y = y)) +
  geom_line(color = "red", size = 1.2) +
  xlab("x") +
  ylab("f(x)") +
  ggtitle("Gaussian Kernel: lambda = 0.001")
print(plot)
```



The solution appears to become more flexible as the value for lambda decreases. With lambda = 0.001, the graph displays more cusps and corners compared that of the function with lambda = 0.1.

Exercise 03

Answer the following questions:

(a) If the Bayes decision boundary is linear, do we expect LDA or QDA to perform better on the training set? On the test set? Explain.

(b) If the Bayes decision boundary is non-linear, do we expect LDA or QDA to perform better on the training set? On the test set? Explain.

a) If the decision boundary is linear, QDA is expected to perform better on the training set, while LDA will perform better on the test set. This is because QDA would fit the data better than LDA because of its increased flexibility. However, for the test set, LDA is expected to outperform QDA as the flexibility of QDA can lead to an overfit model.

b) If the decision boundary is nonlinear, QDA is expected to perform better on both the test and training sets. As mentioned earlier, QDA results in more flexible models that can better capture nonlinear relationships in both datasets.

Exercise 04

In this exercise, we will use the "Ionosphere" data set on radar data. This data is often used to test machine learning algorithms and consists of a binary classification problem. See ?IonoSphere for more details. This dataset is available in the mlbench library. We will not use the columns V1 and V2, so remove these columns using select$(-V1, -V2)$. The goal of this exercise is to compare the LDA, QDA, and Naive Bayes methods. All of these methods can be found in the discrim library of tidymodels. You will also need to install the klaR library, which defines the engine for these methods. For all methods, generate a k-fold cross-validation set with k = 10 or k = 5 (depending on the time required to run on your machine).

Which method yields the best results?

```r
library(mlbench)
library(tidyverse)
library(discrim)
library(klaR)
library(dplyr)
library(tidymodels)
library(recipes)

# Prepare the data
data(Ionosphere)
df <- as.data.frame(Ionosphere)
df <- subset(df, select = -c(V1, V2))

rs <- vfold_cv(df, v = 10)
rcp <- recipe(Class ~ ., data = df)

# LDA
lda <- discrim_regularized(frac_common_cov = 1) %>%
  set_engine("klaR")

fit_lda <-
  workflow() %>%
  add_model(lda) %>%
  add_recipe(rcp) %>%
  fit_resamples(
    resamples = rs,
    control = control_resamples(save_pred = TRUE)
  )

collect_metrics(fit_lda)
```

```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.855    10  0.0137 Preprocessor1_Model1
## 2 roc_auc  binary     0.871    10  0.0183 Preprocessor1_Model1
```

```
# QDA
qda <- discrim_regularized(frac_common_cov = 0) %>%
  set_engine("klaR")

fit_qda <-
  workflow() %>%
  add_model(qda) %>%
  add_recipe(rcp) %>%
  fit_resamples(
    resamples = rs,
    control = control_resamples(save_pred = TRUE)
  )

collect_metrics(fit_qda)
```

```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.935    10 0.0141  Preprocessor1_Model1
## 2 roc_auc  binary     0.982    10 0.00549 Preprocessor1_Model1
```

```
# Naive Bayes
naive <- naive_Bayes() %>% set_engine("klaR")

fit_naive <-
  workflow() %>%
  add_model(naive) %>%
  add_recipe(rcp) %>%
  fit_resamples(
    resamples = rs,
    control = control_resamples(save_pred = TRUE)
  )

collect_metrics(fit_naive)
```

```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.917    10  0.0162 Preprocessor1_Model1
## 2 roc_auc  binary     0.955    10  0.0162 Preprocessor1_Model1
```

Based on these results, QDA exhibits higher accuracy and the highest ROC AUC value, indicating that it is better at distinguishing between the classes. In contrast, LDA yielded the worst results in terms of the same metrics, whereas Naive Bayes demonstrated intermediate performance.