# MATH 3190 Final Project

Jun Hanvey, Ian McFarlane, Kellen Nankervis

Due April 25, 2024

Hello World!

```r
print("Hello world!")
```

```
## [1] "Hello world!"
```

## Introduction

- TODO

## Dual Problem

For the Support Vector Machine algorithm, our goal is to find a $\beta$ and $c$ under the following optimization objective:

$$\begin{aligned}&\underset{\boldsymbol{\beta},c}{\text{minimize}}\ \|\boldsymbol{\beta}\|_2^2\\&\text{subject to } y_i(\boldsymbol{\beta}\cdot\boldsymbol{x}_i - c) \geq 1 \text{ for all } i\end{aligned}$$

For convenience, let's divide our objective function by 2, which doesn't affect the results:

$$\begin{aligned}&\underset{\boldsymbol{\beta},c}{\text{minimize}}\ \frac{1}{2}\|\boldsymbol{\beta}\|_2^2\\&\text{subject to } y_i(\boldsymbol{\beta}\cdot\boldsymbol{x}_i - c) \geq 1 \text{ for all } i\end{aligned}$$

Then we can find the dual optimization problem, using Lagrange Multipliers:

$$\underset{\boldsymbol{\lambda}}{\text{maximize}}\ \underset{\boldsymbol{\beta},c}{\text{minimize}}\ L(\boldsymbol{\beta},c,\boldsymbol{\lambda}) = \frac{1}{2}\|\boldsymbol{\beta}\|_2^2 - \sum_{i=1}^{n}\lambda_i\Big(y_i(\boldsymbol{\beta}\cdot\boldsymbol{x}_i - c) - 1\Big)$$

The dual problem will be satisfied when all partial derivatives are zero, Which leads us to the following results:

$$\begin{aligned}\frac{\partial L}{\partial\boldsymbol{\beta}} &= \boldsymbol{\beta} - \sum_{i=1}^{n}\lambda_i y_i\boldsymbol{x}_i \overset{\text{set}}{=} 0 \iff \boldsymbol{\beta} = \sum_{i=1}^{n}\lambda_i y_i\boldsymbol{x}_i\\ \frac{\partial L}{\partial\boldsymbol{\lambda}} &= \sum_{i=1}^{n} y_i(\boldsymbol{\beta}\cdot\boldsymbol{x}_i - c) - 1 \overset{\text{set}}{=} 0 \iff c = \frac{\sum_{i=1}^{n} y_i\boldsymbol{\beta}\cdot\boldsymbol{x}_i - n}{\sum_{i=1}^{n} y_i}\\ \frac{\partial L}{\partial c} &= \sum_{i=1}^{n}\lambda_i y_i \overset{\text{set}}{=} 0 \iff \boldsymbol{\lambda}\cdot\boldsymbol{y} = 0\end{aligned}$$

Observe how $\boldsymbol{\beta}$ and $c$ can be derived from $\boldsymbol{\lambda}$, $\boldsymbol{y}$ and $X$. Substituting these results into the Lagrangian should lead to some nice results. Although, substituting for $c$ won't be necessary (it gets multiplied by zero). Then, replacing $\boldsymbol{\beta}$ in our Lagrangian yields:

$$\begin{aligned}
L(\boldsymbol{\beta}, c, \boldsymbol{\lambda}) &= \frac{1}{2}\|\boldsymbol{\beta}\|_2^2 - \sum_{i=1}^{n} \lambda_i \Big( y_i(\boldsymbol{\beta} \cdot \boldsymbol{x}_i - c) - 1 \Big) \\
&= \frac{1}{2}\|\boldsymbol{\beta}\|_2^2 - \sum_{i=1}^{n} \lambda_i y_i \boldsymbol{\beta} \cdot \boldsymbol{x}_i + c \sum_{i=1}^{n} \lambda_i y_i + \sum_{i=1}^{n} \lambda_i \\
&= \frac{1}{2}\| \sum_{j=1}^{n} \lambda_j y_j \boldsymbol{x}_j \|_2^2 - \sum_{i=1}^{n} \left( \sum_{j=1}^{n} \lambda_j y_j \boldsymbol{x}_j \right) \cdot \Big( \lambda_i y_i \boldsymbol{x}_i \Big) + c \cdot 0 + \sum_{i=1}^{n} \lambda_i \\
&= \frac{1}{2} \left( \sum_{i=1}^{n} \lambda_i y_i \boldsymbol{x}_i \right) \cdot \left( \sum_{j=1}^{n} \lambda_j y_j \boldsymbol{x}_j \right) - \sum_{i=1}^{n} \sum_{j=1}^{n} (\lambda_j y_j \boldsymbol{x}_j) \cdot (\lambda_i y_i \boldsymbol{x}_i) + \sum_{i=1}^{n} \lambda_i \\
&= \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} (\lambda_i y_i \boldsymbol{x}_i) \cdot (\lambda_j y_j \boldsymbol{x}_j) - \sum_{i=1}^{n} \sum_{j=1}^{n} (\lambda_i y_i \boldsymbol{x}_i) \cdot (\lambda_j y_j \boldsymbol{x}_j) + \sum_{i=1}^{n} \lambda_i \\
&= -\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i \cdot \boldsymbol{x}_j + \sum_{i=1}^{n} \lambda_i \\
L(\boldsymbol{\beta}, c, \boldsymbol{\lambda}) &= \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i \cdot \boldsymbol{x}_j
\end{aligned}$$

This shows a version of the Lagrangian that doesn't depend on $\boldsymbol{\beta}$ nor $c$, which means the optimization problem reduces to:

$$\underset{\boldsymbol{\lambda}}{\text{maximize}} \; L(\boldsymbol{\beta}, c, \boldsymbol{\lambda}) = \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i \cdot \boldsymbol{x}_j$$

## Kernel Trick

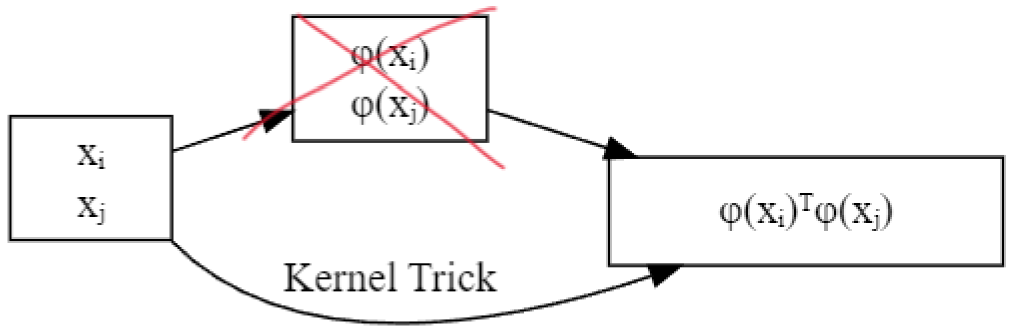
But, why did we go through all that trouble? In this form, we can observe that the Lagrangian depends on 3 components only:

- $\lambda_i$, an optimization artifact
- $y_i$, a fixed "binary" variable
- $\boldsymbol{x}_i$, the features we're using to predict the class

Notice, $\boldsymbol{x}_i$ is the only aspect of our model we can modify (via feature engineering). So, let $\phi(\boldsymbol{x})$ be our feature engineer transformation, then our Lagrangian becomes:

$$L(\boldsymbol{\beta}, c, \boldsymbol{\lambda}) = \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j \phi(\boldsymbol{x}_i) \cdot \phi(\boldsymbol{x}_j)$$

From this we can observe that we don't really need to calculate $\phi(\boldsymbol{x})$, a function that finds $\phi(\boldsymbol{x}_i) \cdot \phi(\boldsymbol{x}_j)$ from $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ is good enough. We call that function a Kernel and denote it by $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$. To avoid having complicated sub-indices, we'll relabel the arguments of $K$ to $\boldsymbol{a}$ and $\boldsymbol{b}$, so we'll have $K(\boldsymbol{a}, \boldsymbol{b})$. Finding the $\phi(\boldsymbol{a}) \cdot \phi(\boldsymbol{b})$, without having to calculate $\phi$ is what we call the **Kernel Trick**.



One of the simplest transformations we can investigate is $K_{Power(2)}(\boldsymbol{a}, \boldsymbol{b}) = (\boldsymbol{a} \cdot \boldsymbol{b})^2$

If we expand the dot product, we get:

$K_{Power(2)}(\boldsymbol{a}, \boldsymbol{b}) = (a_1b_1 + a_2b_2 + \cdots + a_nb_n)^2$

Expanding the square, gives:

$$\begin{aligned}
K_{Power(2)}(\boldsymbol{a}, \boldsymbol{b}) =& (a_1b_1)(a_1b_1) + (a_1b_1)(a_2b_2) + \cdots + (a_1b_1)(a_nb_n) + \\
& (a_2b_2)(a_1b_1) + (a_2b_2)(a_2b_2) + \cdots + (a_2b_2)(a_nb_n) + \\
& \vdots \\
& (a_nb_n)(a_1b_1) + (a_nb_n)(a_2b_2) + \cdots + (a_nb_n)(a_nb_n) \\
\\
=& (a_1a_1)(b_1b_1) + (a_1a_2)(b_1b_2) + \cdots + (a_1a_n)(b_1b_n) + \\
& (a_2a_1)(b_2b_1) + (a_2a_2)(b_2b_2) + \cdots + (a_2a_n)(b_2b_n) + \\
& \vdots \\
& (a_na_1)(b_nb_1) + (a_na_2)(b_nb_2) + \cdots + (a_na_n)(b_nb_n)
\end{aligned}$$

Observe that the expanded sum is equivalent to the dot product of the vectors containing all pair-wise interaction terms. So, in this case

$$\phi_{Power(2)}(\boldsymbol{x}) = \begin{bmatrix} x_1x_1 \\ x_1x_2 \\ \vdots \\ x_1x_n \\ x_2x_1 \\ x_2x_2 \\ \vdots \\ x_2x_n \\ \vdots \\ x_nx_1 \\ x_nx_2 \\ \vdots \\ x_nx_n \end{bmatrix}$$

and $K_{Power(2)}(\boldsymbol{a}, \boldsymbol{b}) = \phi_{Power(2)}(\boldsymbol{a}) \cdot \phi_{Power(2)}(\boldsymbol{b}) = (\boldsymbol{a} \cdot \boldsymbol{b})^2$. However, notice we don't have to compute $\phi_{Power(2)}$ if we take $K_{Power(2)}(\boldsymbol{a}, \boldsymbol{b}) = (\boldsymbol{a} \cdot \boldsymbol{b})^2$ instead.

Likewise, one can show that:

$$\begin{aligned}
K_{Power(3)}(\boldsymbol{a},\boldsymbol{b}) =& (\boldsymbol{a}\cdot\boldsymbol{b})^3 \text{ corresponds to the transformation containing all 3-way interaction terms}\\
K_{Power(4)}(\boldsymbol{a},\boldsymbol{b}) =& (\boldsymbol{a}\cdot\boldsymbol{b})^4 \text{ corresponds to the transformation containing all 4-way interaction terms}\\
&\vdots\\
K_{Power(n)}(\boldsymbol{a},\boldsymbol{b}) =& (\boldsymbol{a}\cdot\boldsymbol{b})^n \text{ corresponds to the transformation containing all n-way interaction terms}
\end{aligned}$$

However, in Applied Statistics we learned that whenever we include high order terms, we also want to include all lower level terms. So, let's inspect the following kernel:

$K_{Poly(2)}(\boldsymbol{a},\boldsymbol{b}) = (\boldsymbol{a}\cdot\boldsymbol{b}+1)^2 = \underbrace{(\boldsymbol{a}\cdot\boldsymbol{b})^2}_{\text{2-way}} + 2\underbrace{(\boldsymbol{a}\cdot\boldsymbol{b})}_{\text{1-way}} + \underbrace{1}_{\text{0-way}}$

So, $K_{Poly(2)}$ gives us the 2-way interaction terms and all the lower order terms. Likewise, $K_{Poly(n)} = (\boldsymbol{a}\cdot\boldsymbol{b}+1)^n$ gives us the n-way interaction terms and below, precisely what we wanted. This is also the polynomial kernel for SVM with $\gamma = 1$.

This proposes an interesting conundrum: which n should we pick? We could try using using cross-validation. However, our friend ***Taylor*** might have a way of trying out all n values at the same time, while giving more weight to lower order terms than the higher order terms (following the principle of parsimony).

Recall:

$\exp(x) = 1 + \frac{1}{1!}x^1 + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \frac{1}{4!}x^4 + \cdots$

Then by plugging $\boldsymbol{a}\cdot\boldsymbol{b}$ for $x$, we get:

$\exp(\boldsymbol{a}\cdot\boldsymbol{b}) = \underbrace{1}_{\text{0-way}} + \frac{1}{1!}\underbrace{(\boldsymbol{a}\cdot\boldsymbol{b})^1}_{\text{1-way}} + \frac{1}{2!}\underbrace{(\boldsymbol{a}\cdot\boldsymbol{b})^2}_{\text{2-way}} + \frac{1}{3!}\underbrace{(\boldsymbol{a}\cdot\boldsymbol{b})^3}_{\text{3-way}} + \frac{1}{4!}\underbrace{(\boldsymbol{a}\cdot\boldsymbol{b})^4}_{\text{4-way}} + \cdots$

Which gives us ALL n-way interaction terms, weighing the lower terms more and the higher terms less. This also means we're essentially computing the dot product of an infinite-dimensional transformation, without having to compute infinite transformations. However, to get to the Radial Basis Function, we need a couple transformations:

First, let's multiply it by $\exp\left(-\frac{\|\boldsymbol{a}\|_2^2 + \|\boldsymbol{b}\|_2^2}{2}\right)$:

$$\begin{aligned}
\exp\Big(\boldsymbol{a}\cdot\boldsymbol{b}\Big)\exp\left(-\frac{\|\boldsymbol{a}\|_2^2 + \|\boldsymbol{b}\|_2^2}{2}\right) &= \exp\left(\boldsymbol{a}\cdot\boldsymbol{b} - \frac{\|\boldsymbol{a}\|_2^2 + \|\boldsymbol{b}\|_2^2}{2}\right)\\
&= \exp\left(-\frac{\|\boldsymbol{a}\|_2^2 + \|\boldsymbol{b}\|_2^2 - 2\boldsymbol{a}\cdot\boldsymbol{b}}{2}\right)\\
&= \exp\left(-\frac{1}{2}\|\boldsymbol{a}-\boldsymbol{b}\|_2^2\right)
\end{aligned}$$

Now, let's raise it to the $2\gamma$ to add a control parameter.

$\left(\exp\left(-\frac{1}{2}\|\boldsymbol{a}-\boldsymbol{b}\|_2^2\right)\right)^{2\gamma} = \exp\Big(-\gamma\|\boldsymbol{a}-\boldsymbol{b}\|_2^2\Big)$

Finally, after 4 pages, we have arrived to the Radial Basis Function Kernel:

$K_{RBF}(\boldsymbol{a},\boldsymbol{b}) = \exp\Big(-\gamma\|\boldsymbol{a}-\boldsymbol{b}\|_2^2\Big)$

In essence RBF is so special because it performs the optimization over an infinite-dimensional feature space. All that with a really simple formula, which allows for very intricate decision boundaries with minimal computational power.

#TODO: Find good value for n and fix colors so they are the same as the later plot. - Kellen Nankervis
Create example data to test RBF kernel function

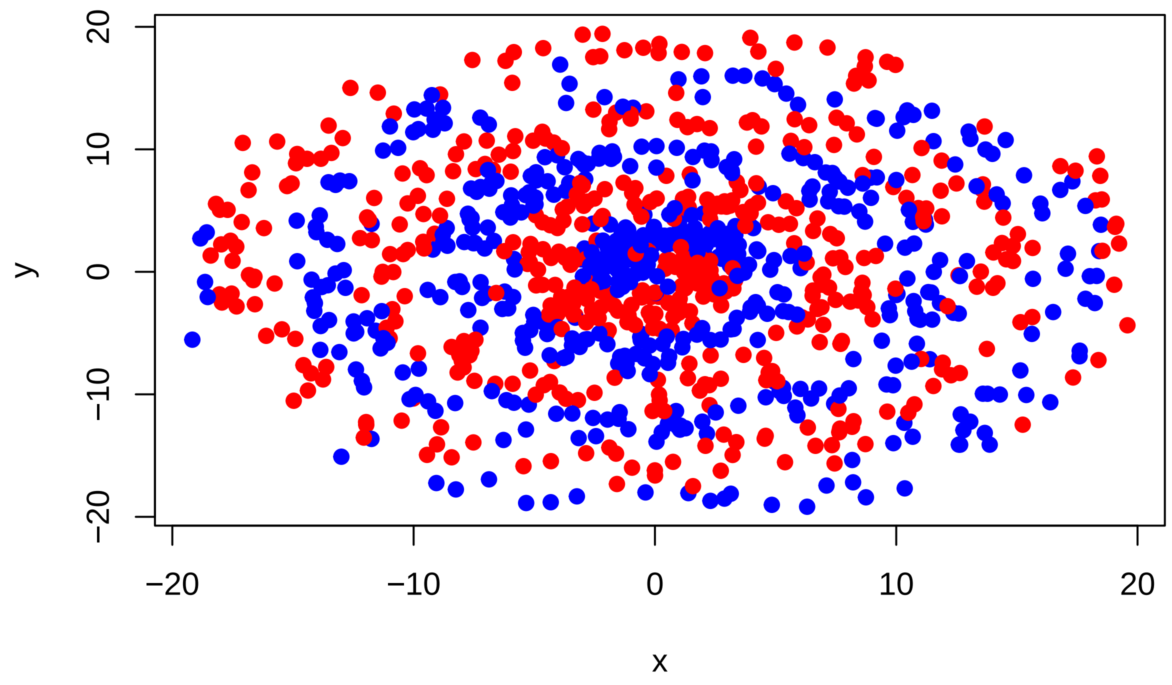
```r
set.seed(123)
n <- 1000
for (i in 1:10) {
  r <- runif(n, 1, 6 * pi + 1)
  theta <- runif(n, 0, 2 * pi)
}
# Make a sample classification n observations long
class <- sample(c(0, 1), n, replace = TRUE)
# Now fill the classification vector with the correct values
for (j in 1:n) {
  if ((r[j] + theta[j]) %% (2 * pi) < pi) {
    class[j] <- 1
  } else {
    class[j] <- 0
  }
}

# Create a data frame with the data
data <- data.frame(r, theta, class)
# Create a scatter plot of the data in x and y coordinates
data$color <- ifelse(data$class == 1, "red", "blue")
data$x <- data$r * cos(data$theta)
data$y <- data$r * sin(data$theta)

# Now offset the data a bit so their is some overlap
for (j in 1:n) {
  r <- runif(1, 0, 1)
  theta <- runif(1, 0, 2 * pi)
  # Convert to x and y coordinates
  data$x[j] <- data$x[j] + r * cos(theta)
  data$y[j] <- data$y[j] + r * sin(theta)
  # Convert the new x and y coordinates back to r and theta
  data$r[j] <- sqrt(data$x[j]^2 + data$y[j]^2)
  data$theta[j] <- atan2(data$y[j], data$x[j])
}
```

#TODO: Decide which plots to show and which to delete. Commented out ones would be my current suggestions to delete or at least move to later in the document. Make plots look good when knitted to pdf and/or slides. - Kellen Nankervis

```r
# Plot the data in the x and y coordinates
plot(data$x, data$y, col = data$color, pch = 19, xlab = "x", ylab = "y")
```



```
# Plot the data in polar coordinates
# plot(data$r, data$theta, col = data$color, pch = 19, xlab = "r", ylab = "theta")

# Plot r*theta vs. r^2 * theta^2
# plot(data$r + data$theta, data$r * data$theta, col = data$color, pch = 19, xlab = "r*theta", ylab = "
```

#TODO: Decide on good cost to not overfit. Could be a good spot to also show how we can use cross-validation to find the best cost. Make plots look good when knitted to pdf and/or slides. - Kellen Nankervis

```
# Load the required svm library
library(e1071)
library(caret)
```

```
## Loading required package: ggplot2
## Warning: package 'ggplot2' was built under R version 4.3.2
## Loading required package: lattice
```

```
# Convert class to a factor
data$class <- as.factor(data$class)

# Create a data frame with only the class and the x and y coordinates
data2 <- data.frame(class = data$class, x = data$x, y = data$y)
data2$class <- as.factor(data2$class)

# Use a radial basis function kernel to classify the data with the SVM function
svmfit <- svm(class ~ ., data = data2, kernel = "radial", cost = 1000, gamma = 1)

print(svmfit)
```

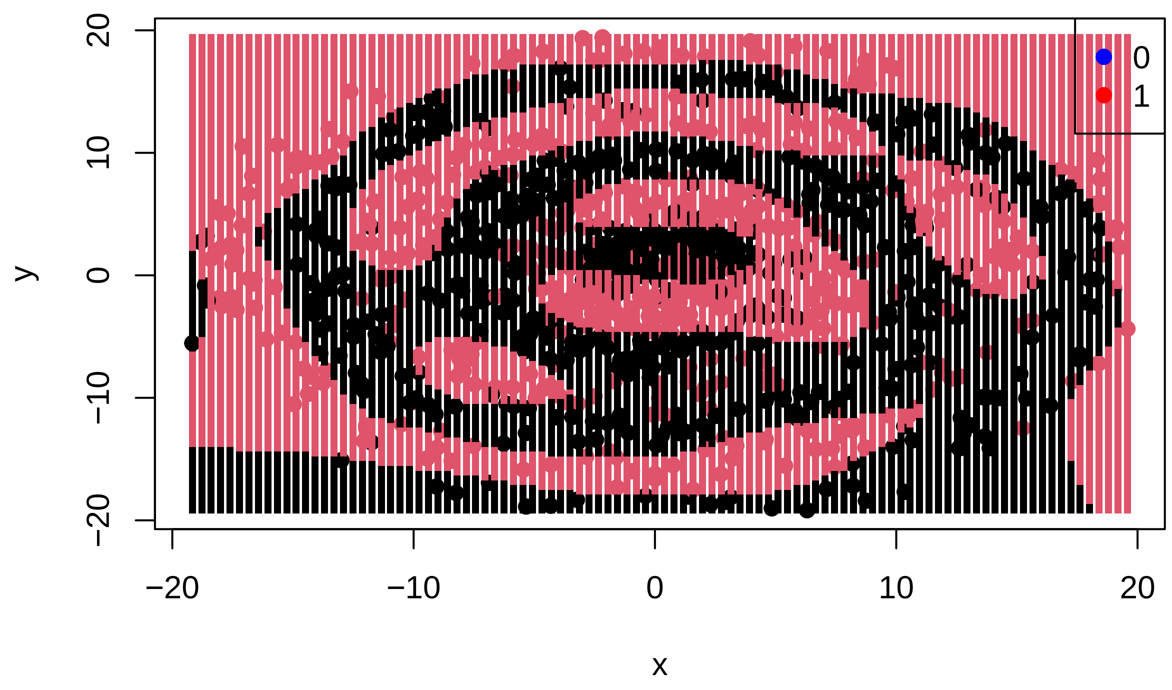
```
##
## Call:
## svm(formula = class ~ ., data = data2, kernel = "radial", cost = 1000,
##     gamma = 1)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1000
##
## Number of Support Vectors:  739
```

```
# Plot the data points
plot(data2$x, data2$y, col = data2$class, pch = 19, xlab = "x", ylab = "y")

# Plot the decision boundary
# plot(svmfit, data2$class, grid = 100, dataSymbol = 16)
x1_grid <- seq(min(data2$x), max(data2$x), length.out = 100)
x2_grid <- seq(min(data2$y), max(data2$y), length.out = 100)
grid <- expand.grid(x = x1_grid, y = x2_grid)

predicted_labels <- predict(svmfit, newdata = grid)

plot(data2$x, data2$y, col = data2$class, pch = 19, xlab = "x", ylab = "y")
points(grid$x, grid$y, col = factor(predicted_labels), pch = ".", cex = 3.5)
legend("topright", legend = levels(data2$class), col = c("blue", "red"), pch = 19)
```



```r
# Create a confusion matrix to evaluate the SVM model
confusionMatrix(predict(svmfit, data2), data2$class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 427 145
##          1  80 348
##
##                Accuracy : 0.775
##                  95% CI : (0.7478, 0.8005)
##     No Information Rate : 0.507
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.5491
##
##  Mcnemar's Test P-Value : 1.984e-05
##
##             Sensitivity : 0.8422
##             Specificity : 0.7059
##          Pos Pred Value : 0.7465
##          Neg Pred Value : 0.8131
##              Prevalence : 0.5070
##          Detection Rate : 0.4270
##    Detection Prevalence : 0.5720
##       Balanced Accuracy : 0.7740
##
##        'Positive' Class : 0
##
```

#This was some example code I found to make the decision boundary plot. I'm leaving it for others to see but it will be removed in the final version obviously. - Kellen Nankervis

```r
print("")
```

```
## [1] ""
```

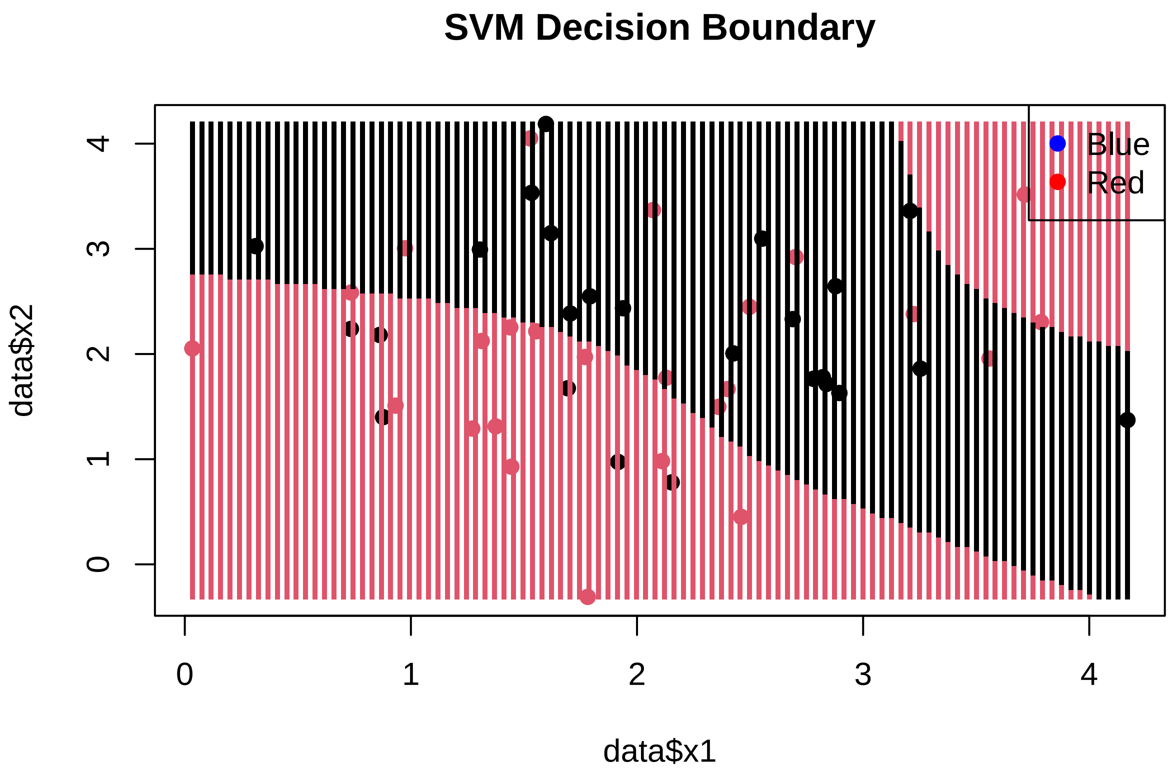
```r
# Create a toy dataset
set.seed(123)
data <- data.frame(
  x1 = rnorm(50, mean = 2),
  x2 = rnorm(50, mean = 2),
  label = c(rep("Red", 25), rep("Blue", 25)) |> as.factor()
)

# Train an SVM
svm_model <- svm(label ~ ., data = data, kernel = "radial")

# Create a grid of points for prediction
x1_grid <- seq(min(data$x1), max(data$x1), length.out = 100)
x2_grid <- seq(min(data$x2), max(data$x2), length.out = 100)
grid <- expand.grid(x1 = x1_grid, x2 = x2_grid)

# Predict class labels for the grid
predicted_labels <- predict(svm_model, newdata = grid)
```

```r
# Plot the decision boundary
plot(data$x1, data$x2, col = factor(data$label), pch = 19, main = "SVM Decision Boundary")
points(grid$x1, grid$x2, col = factor(predicted_labels), pch = ".", cex = 2.5)
legend("topright", legend = levels(data$label), col = c("blue", "red"), pch = 19)
```


SVM Decision Boundary
data$x2
data$x1
Blue
Red