# MATH 3190 Final Project

Jun Hanvey, Ian McFarlane, Kellen Nankervis

Due April 25, 2024

Hello World!

```
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

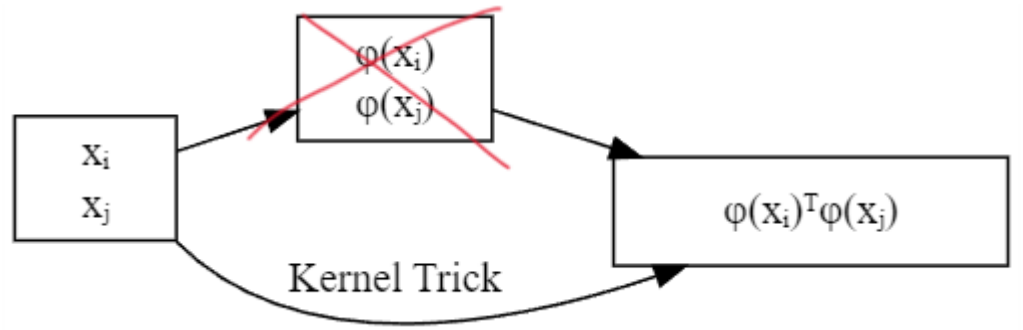
But, why did we go through all that trouble? In this form, we can observe that the Lagrangian depends on 3 components only:

- $\lambda_i$, 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:

$K_{Power(2)}(\boldsymbol{a}, \boldsymbol{b}) = (a_1 b_1 + a_2 b_2 + \cdots + a_n b_n)^2$

Expanding the square, gives:

$$\begin{aligned} K_{Power(2)}(\boldsymbol{a}, \boldsymbol{b}) =& (a_1 b_1)(a_1 b_1) + (a_1 b_1)(a_2 b_2) + \cdots + (a_1 b_1)(a_n b_n) + \\ & (a_2 b_2)(a_1 b_1) + (a_2 b_2)(a_2 b_2) + \cdots + (a_2 b_2)(a_n b_n) + \\ & \vdots \\ & (a_n b_n)(a_1 b_1) + (a_n b_n)(a_2 b_2) + \cdots + (a_n b_n)(a_n b_n) \\ \\ =& (a_1 a_1)(b_1 b_1) + (a_1 a_2)(b_1 b_2) + \cdots + (a_1 a_n)(b_1 b_n) + \\ & (a_2 a_1)(b_2 b_1) + (a_2 a_2)(b_2 b_2) + \cdots + (a_2 a_n)(b_2 b_n) + \\ & \vdots \\ & (a_n a_1)(b_n b_1) + (a_n a_2)(b_n b_2) + \cdots + (a_n a_n)(b_n b_n) \end{aligned}$$

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

$$\phi_{Power(2)}(\boldsymbol{x}) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ \vdots \\ x_1 x_n \\ x_2 x_1 \\ x_2 x_2 \\ \vdots \\ x_2 x_n \\ \vdots \\ x_n x_1 \\ x_n x_2 \\ \vdots \\ x_n x_n \end{bmatrix}$$

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:

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

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.

## Example Data

- TODO: Fix colors so they are the same as the later plot. Add some more surrounding text to the whole example portion. - Kellen Nankervis To show the power of the Radial Basis Function we will first use a generated data set.

First we will generate the data. For this example I am creating a sort of spiral pattern using 1000 data points. I'm using this pattern because it is a case where a linear kernel would not work well. The data is generated in polar coordinates and then converted to x and y coordinates. The data is then offset a bit so there is some overlap between the two classes.

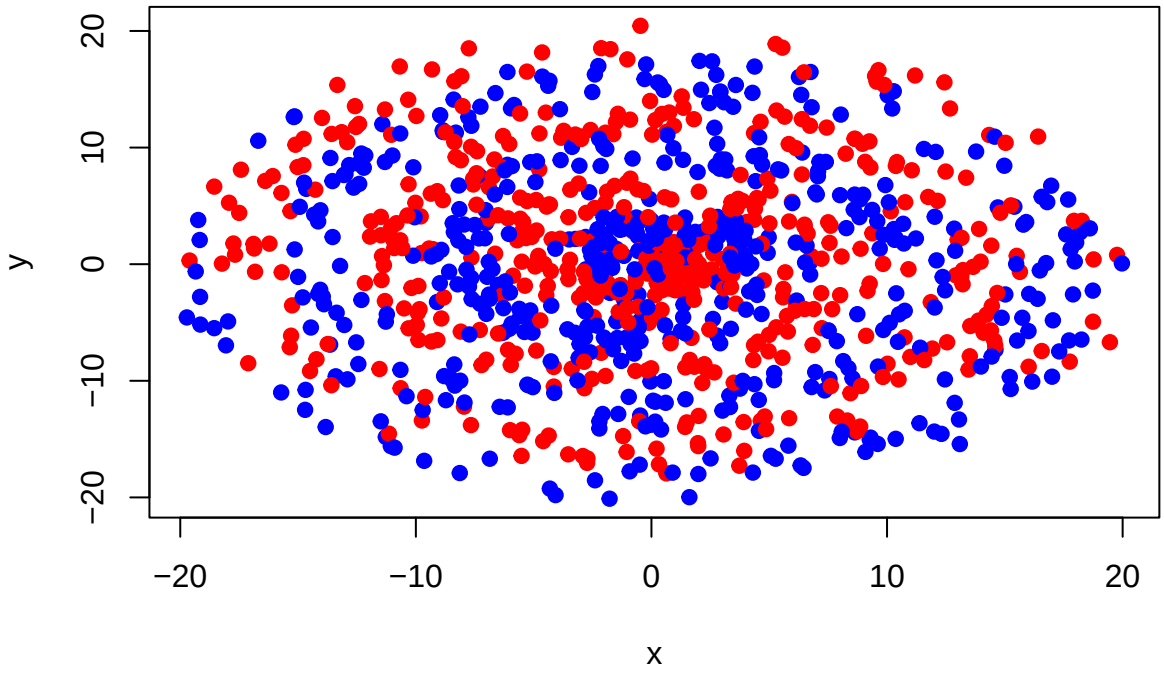
```
n <- 1000
r <- 6 * pi + 1
r_offset <- 2
colors <- c("blue", "red")

generate_spiral <- function(n, r, r_offset, colors, seed) {
  set.seed(seed)

  # Generate random values for r and theta
  r <- runif(n, 1, 6 * pi + 1)
  theta <- runif(n, 0, 2 * pi)

  # Classify observations based on r and theta
  class <- ifelse((r + theta) %% (2 * pi) < pi, 1, 0)

  # Create a data frame with the data
  data <- data.frame(r, theta, class)

  # Create colors based on class
  data$color <- ifelse(data$class == 1, colors[2], colors[1])

  # Convert polar coordinates to Cartesian coordinates
  data$x <- data$r * cos(data$theta)
  data$y <- data$r * sin(data$theta)

  # Offset the data
  for (j in 1:n) {
    r_offset_val <- runif(1, 0, r_offset)
    theta_offset_val <- runif(1, 0, 2 * pi)

    # Convert to Cartesian coordinates
    data$x[j] <- data$x[j] + r_offset_val * cos(theta_offset_val)
    data$y[j] <- data$y[j] + r_offset_val * sin(theta_offset_val)

    # Convert back to polar coordinates
    data$r[j] <- sqrt(data$x[j]^2 + data$y[j]^2)
    data$theta[j] <- atan2(data$y[j], data$x[j])
  }

  return(data)
}

data <- generate_spiral(n, r, r_offset, colors, seed = 2024)
```

- 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

Now lets take a look at this generated data.

```
# 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 = "
```

As we can see it matches a spiral pattern with a bit of overlap between the two classes. Since we know how the data was genreated it might be smart to convert to polar coordinates, but if this data weren't generated we might not make that connection. This is where the RBF kernel can be very useful.

Now lets use the RBF kernel to classify this data. Right now we will use a cost of 1 and a gamma of 1, the default values of the function, but later we can use cross-validation to find the best values for these parameters.

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

```
## Loading required package: ggplot2
## Loading required package: lattice
```

```r
# 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")
print(svmfit)
```

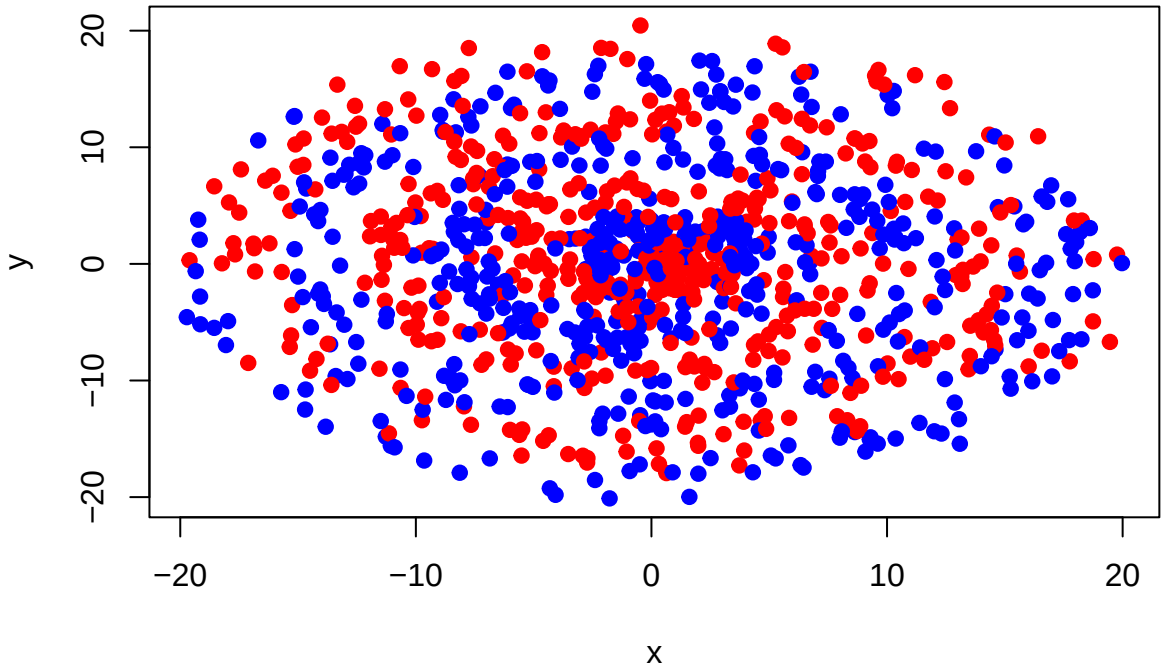
```
##
## Call:
## svm(formula = class ~ ., data = data2, kernel = "radial")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##
## Number of Support Vectors:  945
```

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

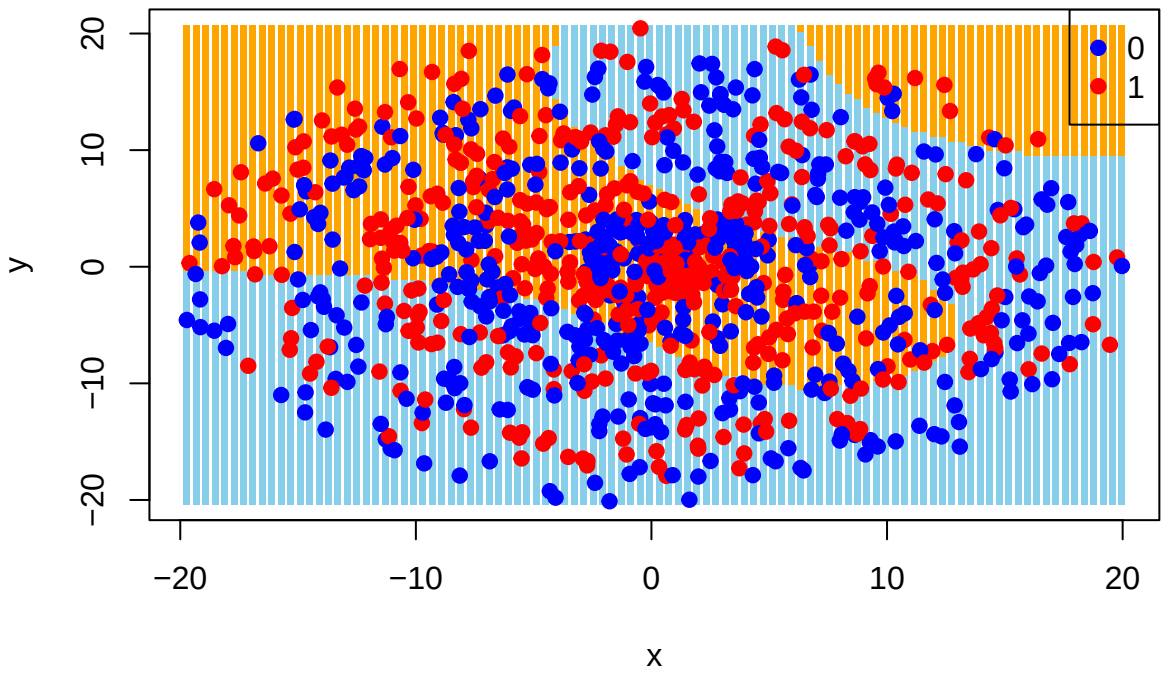
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 275 211
##          1 238 276
##
##                Accuracy : 0.551
##                  95% CI : (0.5196, 0.5821)
##     No Information Rate : 0.513
##     P-Value [Acc > NIR] : 0.008782
##
##                   Kappa : 0.1027
##
##  Mcnemar's Test P-Value : 0.219817
##
##             Sensitivity : 0.5361
##             Specificity : 0.5667
##          Pos Pred Value : 0.5658
##          Neg Pred Value : 0.5370
##              Prevalence : 0.5130
##          Detection Rate : 0.2750
##    Detection Prevalence : 0.4860
##       Balanced Accuracy : 0.5514
##
##        'Positive' Class : 0
##
```

This doesn't look that good as it is hardly better than the trivial approach which would get 51.3% correct compared to the 55.1% of our model. Let's look at the decision boundary to see what it thinks.

```r
# Define colors for data points
point_colors <- c("blue", "red")

# Define colors for decision boundary
boundary_colors <- c("skyblue", "orange")

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

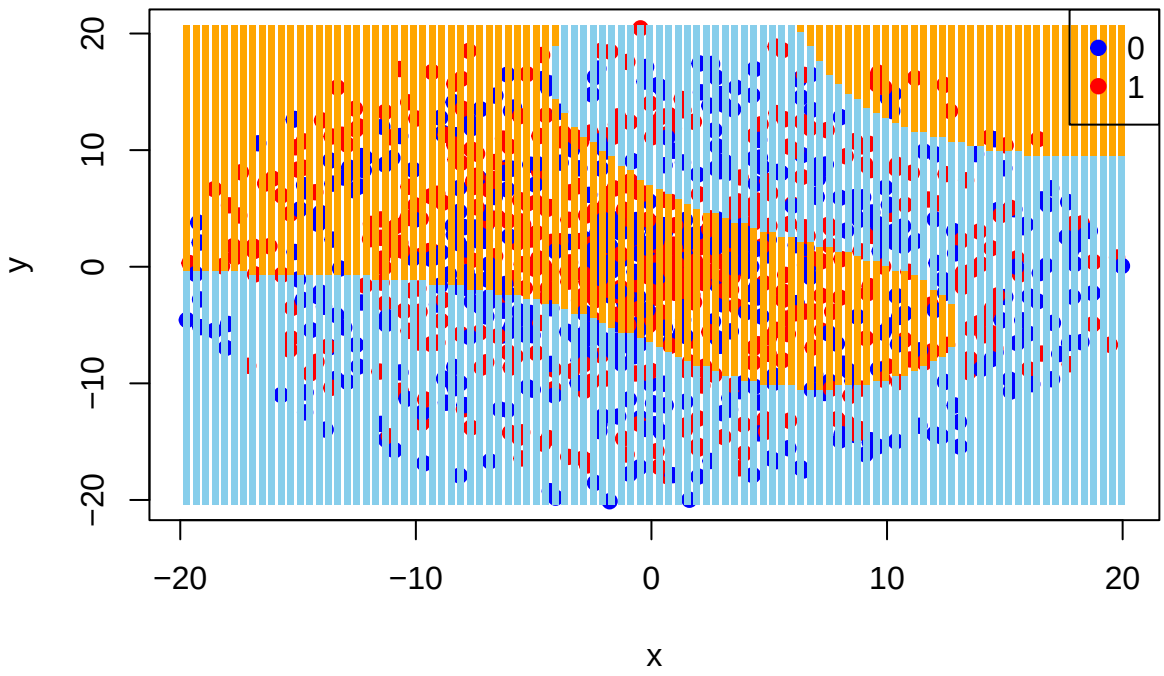



```r
# Plot the decision boundary
x1_grid <- seq(min(data2$x), max(data2$x), length.out = 100)
x2_grid <- seq(min(data2$y), max(data2$y), length.out = 100)
grid <- expand.grid(x = x1_grid, y = x2_grid)

predicted_labels <- predict(svmfit, newdata = grid)

plot(grid$x, grid$y, col = boundary_colors[predicted_labels], pch = ".", cex = 3.5, xlab = "x", ylab =

# Plot the data points
points(data2$x, data2$y, col = point_colors[data2$class], pch = 19)
legend("topright", legend = levels(data2$class), col = point_colors, pch = 19)
```



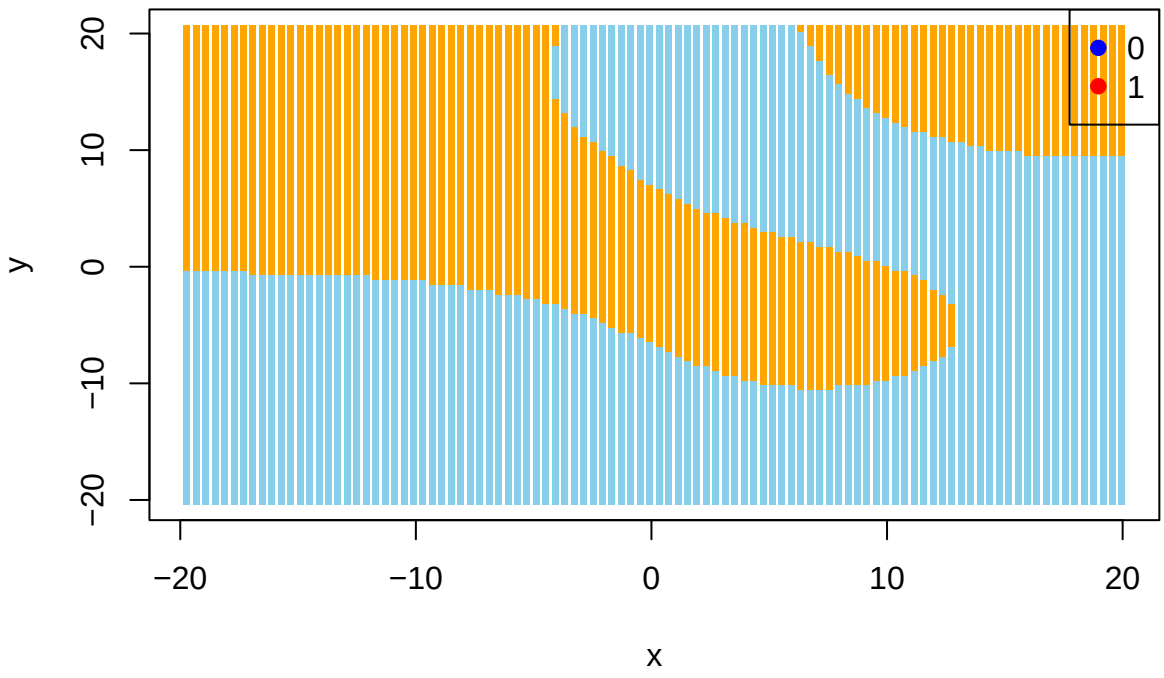

- If we want to plot the decision boundary above the points we can use the following code. - Kellen Nankervis

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

# Plot the decision boundary
points(grid$x, grid$y, col = boundary_colors[predicted_labels], pch = ".", cex = 3.5)
legend("topright", legend = levels(data2$class), col = point_colors, pch = 19)
```




- Or to just plot the decision boundary we can use the following code. - Kellen Nankervis

```
# Plot the decision boundary
plot(grid$x, grid$y, col = boundary_colors[predicted_labels], pch = ".", cex = 3.5, xlab = "x", ylab =
legend("topright", legend = levels(data2$class), col = point_colors, pch = 19)
```




We can see that the decision boundary is not very good. This is because the default values for cost and gamma are not good for this data. However hope is not lost since we can use cross-validation to find the best values for these parameters. Luckily we can do this with the svm function by setting 'cross' in the svm function to 5. This will use 5-fold cross-validation to find the best values for cost and gamma.

```
# First write a simple cross validation function
cross_validate <- function(folds, costs, gammas, data, seed) {
  # Create a data frame to store the results
  results <- data.frame(cost = numeric(0), gamma = numeric(0), accuracy = numeric(0))

  # Loop through each cost and gamma value
  for (cost in costs) {
    for (gamma in gammas) {
      # Set seed so the folds should be the same each time
      set.seed(seed)

      # Use cross-validation to find the best cost and gamma values
      svm_cross <- svm(class ~ ., data = data, kernel = "radial", cross = folds, cost = cost, gamma = ga

      # Store the results
      results <- rbind(results, data.frame(cost = cost, gamma = gamma, accuracy = svm_cross$tot.accuracy
    }
  }

  return(results)
}
```

```r
# Run our function
validation_data_frame <- cross_validate(5, c(0.1, 1, 10, 100, 1000), c(0.01, 0.1, 1, 10, 100), data2, s

# Print the top 5 best cost and gamma values
print(validation_data_frame[order(-validation_data_frame$accuracy), ][1:5, ])
```

```
##    cost gamma accuracy
## 9     1    10     74.3
## 14   10    10     72.0
## 10    1   100     70.6
## 19  100    10     70.3
## 24 1000    10     67.6
```

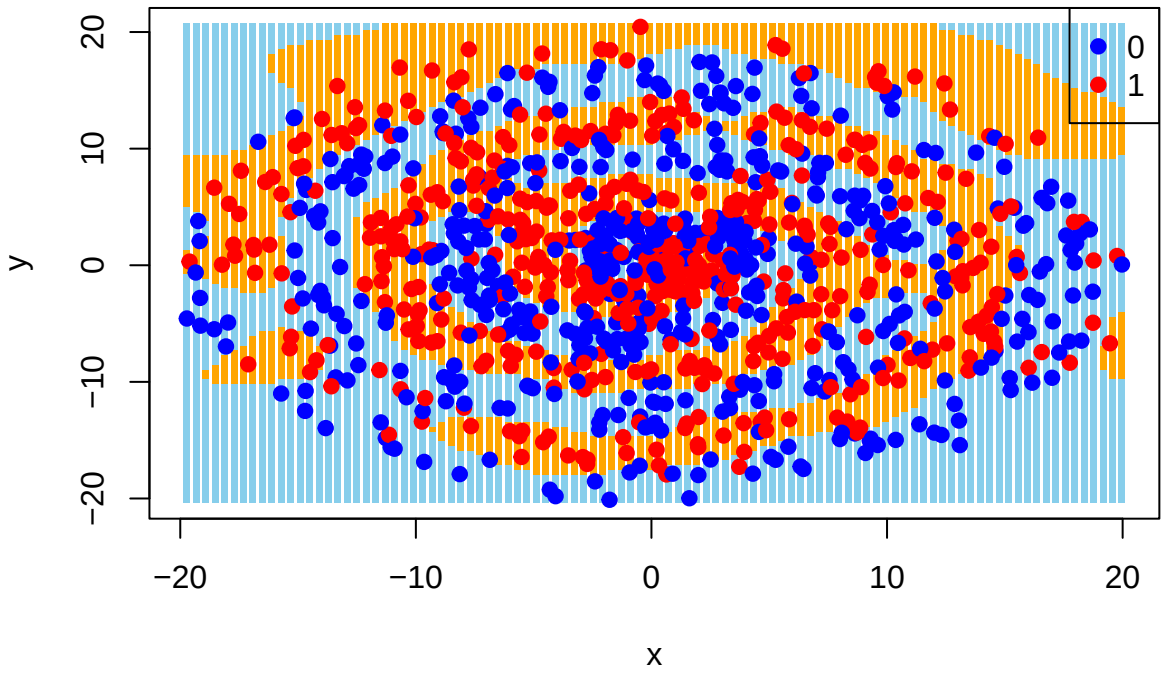
From these results we see that a cost of 1 and a gamma of 10 are the best values for this data with an total accuracy of 74.3% on the test folds. Now lets use these values to classify the data and plot the decision boundary.

```r
# Use the best cost and gamma values to classify the data
svmfit_best <- svm(class ~ ., data = data2, kernel = "radial", cost = 1, gamma = 10)

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

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 428  94
##          1  85 393
##
##                Accuracy : 0.821
##                  95% CI : (0.7958, 0.8443)
##     No Information Rate : 0.513
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.6416
##
##  Mcnemar's Test P-Value : 0.5499
##
##             Sensitivity : 0.8343
##             Specificity : 0.8070
##          Pos Pred Value : 0.8199
##          Neg Pred Value : 0.8222
##              Prevalence : 0.5130
##          Detection Rate : 0.4280
##    Detection Prevalence : 0.5220
##       Balanced Accuracy : 0.8206
##
##        'Positive' Class : 0
##
```

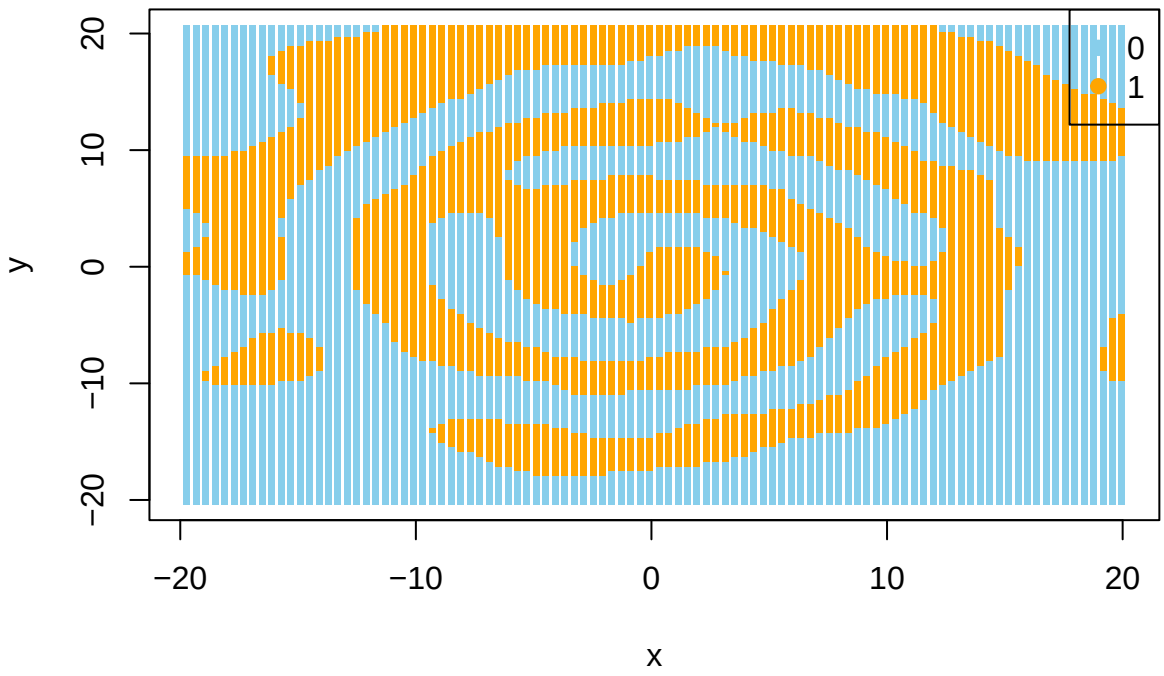
```r
predicted_labels_best <- predict(svmfit_best, newdata = grid)

plot(grid$x, grid$y, col = boundary_colors[predicted_labels_best], pch = ".", cex = 3.5, xlab = "x", yl

# Plot the data points
points(data2$x, data2$y, col = point_colors[data2$class], pch = 19)
```

```
legend("topright", legend = levels(data2$class), col = point_colors, pch = 19)
```




- Or to just plot the decision boundary which I think is my preferred method right now. - Kellen Nankervis

```
# Plot the decision boundary
plot(grid$x, grid$y, col = boundary_colors[predicted_labels_best], pch = ".", cex = 3.5, xlab = "x", yla
legend("topright", legend = levels(data2$class), col = boundary_colors, pch = 19)
```




Now this looks pretty good with a total accuracy of 82.1% on the entire training set. This is a huge improvement over the 55.1% we got with the default values. The plot also shows that the decision boundary is much better than before. It isn't quite perfect, with some gaps and strange connections occasionally, but it is much closer to the true decision boundary of how the data was classified than before.

The final step is to see how this model performs on some test data generated the same way. We will use the same confusion matrix function as before to evaluate the model.

```
# Create new test data
n <- 1000
r <- 6 * pi + 1
r_offset <- 2
colors <- c("blue", "red")
test_data <- generate_spiral(n, r, r_offset, colors, seed = 2025)

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

# Create a data frame with only the class and the x and y coordinates
test_data2 <- data.frame(class = test_data$class, x = test_data$x, y = test_data$y)
test_data2$class <- as.factor(test_data2$class)

# Create a confusion matrix to evaluate the SVM model
confusionMatrix(predict(svmfit_best, test_data2), test_data2$class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
```

```
## Prediction   0   1
##          0 375 128
##          1 116 381
##
##                Accuracy : 0.756
##                  95% CI : (0.7281, 0.7823)
##     No Information Rate : 0.509
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.5121
##
##  Mcnemar's Test P-Value : 0.4813
##
##             Sensitivity : 0.7637
##             Specificity : 0.7485
##          Pos Pred Value : 0.7455
##          Neg Pred Value : 0.7666
##              Prevalence : 0.4910
##          Detection Rate : 0.3750
##    Detection Prevalence : 0.5030
##       Balanced Accuracy : 0.7561
##
##        'Positive' Class : 0
##
```

We see we get an accuracy of 75.6% on the test data. This is a bit lower than the 82.1% we got on the training data, but it is actually better than what we got when doing cross validation on the training data likely since we trained on the full data set where in cross validation we only trained on 80% of the data. This shows that the model is generalizing well to new data.

To see what the model accuracy would have been had the model found the true decision boundary before applying the offset we can use the following code.

```
post_offset_class <- numeric(n)
new_r <- numeric(n)
new_theta <- numeric(n)

for (j in 1:n) {
  new_r[j] <- sqrt(test_data2$x[j]^2 + test_data2$y[j]^2)
  new_theta[j] <- atan2(test_data2$y[j], test_data2$x[j])

  ifelse(new_theta[j] < 0, new_theta[j] <- new_theta[j] + 2 * pi, new_theta[j] <- new_theta[j])

  # Classify observations based on r and theta
  post_offset_class[j] <- ifelse((new_r[j] + new_theta[j]) %% (2 * pi) < pi, 1, 0)
}

# Add the results to the data frame
test_data2$post_offset_class <- post_offset_class
test_data2$new_r <- new_r
test_data2$new_theta <- new_theta

# Make the post offset class a factor
test_data2$post_offset_class <- as.factor(test_data2$post_offset_class)

# Now compare the true class to the predicted class given by the true decision boundary
```

```r
confusionMatrix(test_data2$post_offset_class, test_data2$class)
```

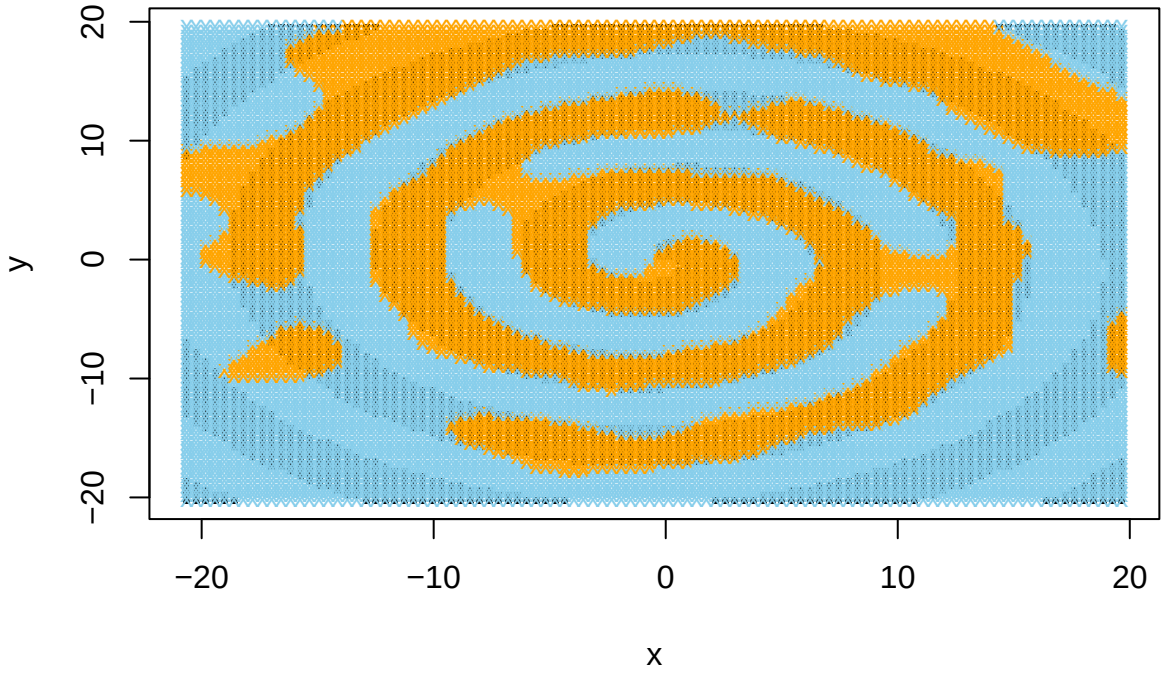
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 410 112
##          1  81 397
##
##                Accuracy : 0.807
##                  95% CI : (0.7811, 0.831)
##     No Information Rate : 0.509
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.6143
##
##  Mcnemar's Test P-Value : 0.03082
##
##             Sensitivity : 0.8350
##             Specificity : 0.7800
##          Pos Pred Value : 0.7854
##          Neg Pred Value : 0.8305
##              Prevalence : 0.4910
##          Detection Rate : 0.4100
##    Detection Prevalence : 0.5220
##       Balanced Accuracy : 0.8075
##
##        'Positive' Class : 0
##
```

We see that due to the offset being applied to the data, even if we re applied the true decision boundary before offsetting the data we would only get an accuracy of 80.7%. This is good since that means our model is only 5.2% off a model that found the true decision boundary before the offset was applied.

Finally lets plot the true decision boundary and the decision boundary found by the model for comparison.

```r
# Create a grid of points for prediction
x1_grid <- seq(min(test_data2$x), max(test_data2$x), length.out = 100)
x2_grid <- seq(min(test_data2$y), max(test_data2$y), length.out = 100)

grid <- expand.grid(x = x1_grid, y = x2_grid)

# Make the grid a data frame of x and y coordinates
grid_data <- data.frame(x = grid$x, y = grid$y)

for (j in 1:nrow(grid_data)) {
  new_r <- sqrt(grid_data$x[j]^2 + grid_data$y[j]^2)
  new_theta <- atan2(grid_data$y[j], grid_data$x[j])

  ifelse(new_theta < 0, new_theta <- new_theta + 2 * pi, new_theta <- new_theta)

  # Classify observations based on r and theta
  grid_data$class[j] <- ifelse((new_r + new_theta) %% (2 * pi) < pi, 1, 0)
}

# Plot the true decision boundary
```

```r
plot(grid_data$x, grid_data$y, col = grid_data$class, pch = ".", cex = 3.5, xlab = "x", ylab = "y")

# Plot the decision boundary found by the model
predicted_labels_best <- predict(svmfit_best, newdata = grid)

points(grid$x, grid$y, col = boundary_colors[predicted_labels_best], pch = "x", cex = .85)
```




Now that we have seen the power of the RBF kernel we can compare it to other models. We will compare it to a linear kernel, a polynomial kernel, a logistic regression model without many interactions. We will also compare it to a KNN model.

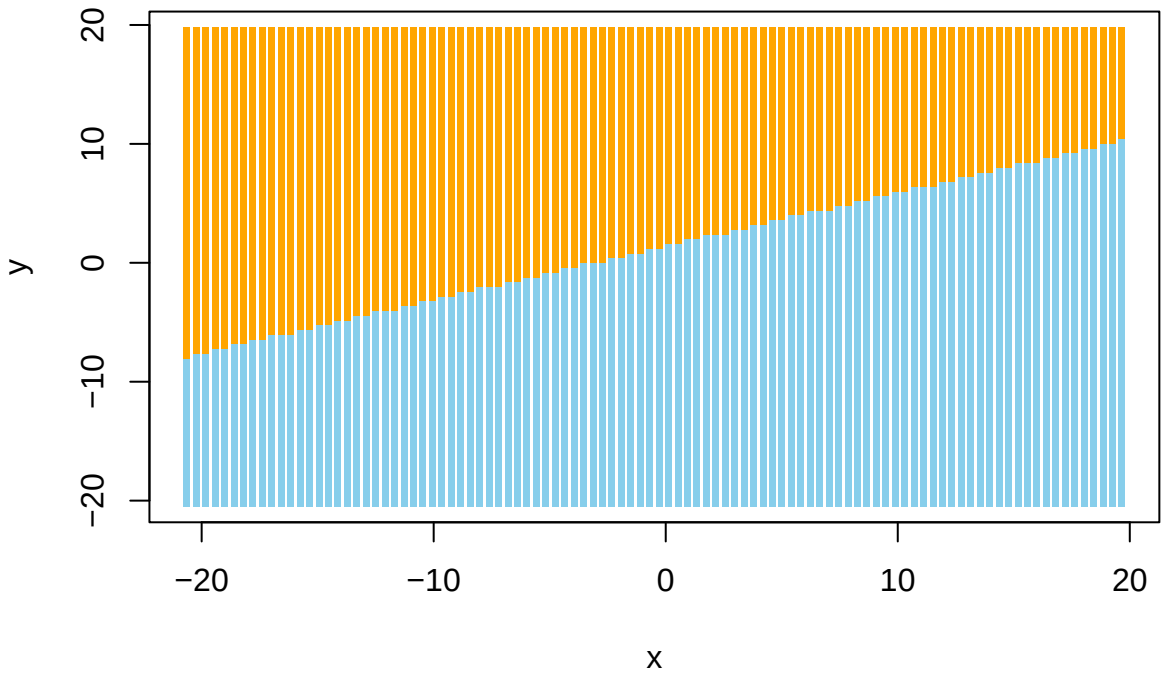
Lets start with a linear kernel and a polynomial kernel since those are fairly easy.

```r
# Use a linear kernel to classify the data with the SVM function
svmfit_linear <- svm(class ~ ., data = data2, kernel = "linear", cost = 1000)

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

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 296 268
##          1 217 219
##
##                Accuracy : 0.515
##                  95% CI : (0.4835, 0.5464)
```

```
##     No Information Rate : 0.513
##     P-Value [Acc > NIR] : 0.46231
##
##                   Kappa : 0.0268
##
##  Mcnemar's Test P-Value : 0.02318
##
##             Sensitivity : 0.5770
##             Specificity : 0.4497
##          Pos Pred Value : 0.5248
##          Neg Pred Value : 0.5023
##              Prevalence : 0.5130
##          Detection Rate : 0.2960
##    Detection Prevalence : 0.5640
##       Balanced Accuracy : 0.5133
##
##        'Positive' Class : 0
##
```

```r
predicted_labels_linear <- predict(svmfit_linear, newdata = grid)

plot(grid$x, grid$y, col = boundary_colors[predicted_labels_linear], pch = ".", cex = 3.5, xlab = "x", y
```



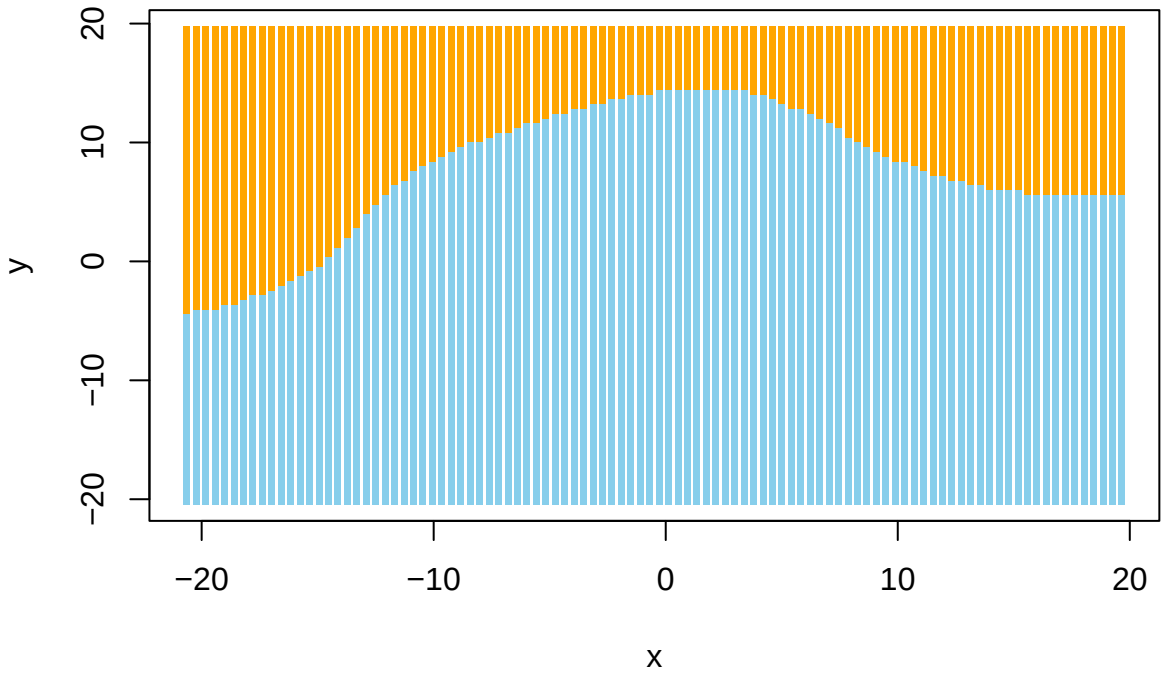

As we can see the linear kernel does not do a good job at classifying the data only getting a 51.5% accuracy (only slightly better than guessing) since no line cleanly fits the data. So lets try the polynomial kernel.

```r
# Use a polynomial kernel to classify the data with the SVM function

# Write a cross validation function for polynomial kernels that takes degrees, cost, and gamma into acc
cross_poly <- function(folds, degrees, costs, gammas, dataset, seed) {
  results <- data.frame(degree = numeric(0), cost = numeric(0), gamma = numeric(0), accuracy = numeric(
  iteration <- 0

  for (degree in degrees) {
    for (cost in costs) {
      for (gamma in gammas) {
        set.seed(seed)
        print(iteration)
        iteration <- iteration + 1
        svm_cross <- svm(class ~ ., data = dataset, kernel = "polynomial", degree = degree, cross = fol

        results <- rbind(results, data.frame(degree = degree, cost = cost, gamma = gamma, accuracy = svm
      }
    }
  }

  return(results)
}

# Run our function
degrees <- c(4, 5, 6)
costs <- c(1, 10, 100)
gammas <- c(0.1, 1, 10)

# Running this code will take awhile, maybe uncomment it when knitting the pdf for the final time.
# validation_data_frame_poly <- cross_poly(2, degrees, costs, gammas, data2, seed = 2024)

# Print the top 5 best degree, cost, and gamma values
# print(validation_data_frame_poly[order(-validation_data_frame_poly$accuracy), ][1:5, ])

# Make the best poly model based on the cross validation
svm_poly <- svm(class ~ ., data = data2, kernel = "polynomial", degree = 5, cost = 10, gamma = 0.1)

# Check the svm_poly model on a confusion matrix
confusionMatrix(predict(svm_poly, data2), data2$class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 436 412
##          1  77  75
##
##                Accuracy : 0.511
##                  95% CI : (0.4795, 0.5424)
##     No Information Rate : 0.513
##     P-Value [Acc > NIR] : 0.5629
##
##                   Kappa : 0.004
##
```

```
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.8499
##             Specificity : 0.1540
##          Pos Pred Value : 0.5142
##          Neg Pred Value : 0.4934
##              Prevalence : 0.5130
##          Detection Rate : 0.4360
##    Detection Prevalence : 0.8480
##       Balanced Accuracy : 0.5020
##
##        'Positive' Class : 0
##
```

```
predicted_labels_poly <- predict(svm_poly, newdata = grid)

plot(grid$x, grid$y, col = boundary_colors[predicted_labels_poly], pch = ".", cex = 3.5, xlab = "x", yla
```




We can see that the very best polynomial kernal from our cross validation only gets an accuracy of 51.1% which is not very good, in fact it is worse than the linear kernel or just guessing the more likely (at least based on the test data) classification every time.

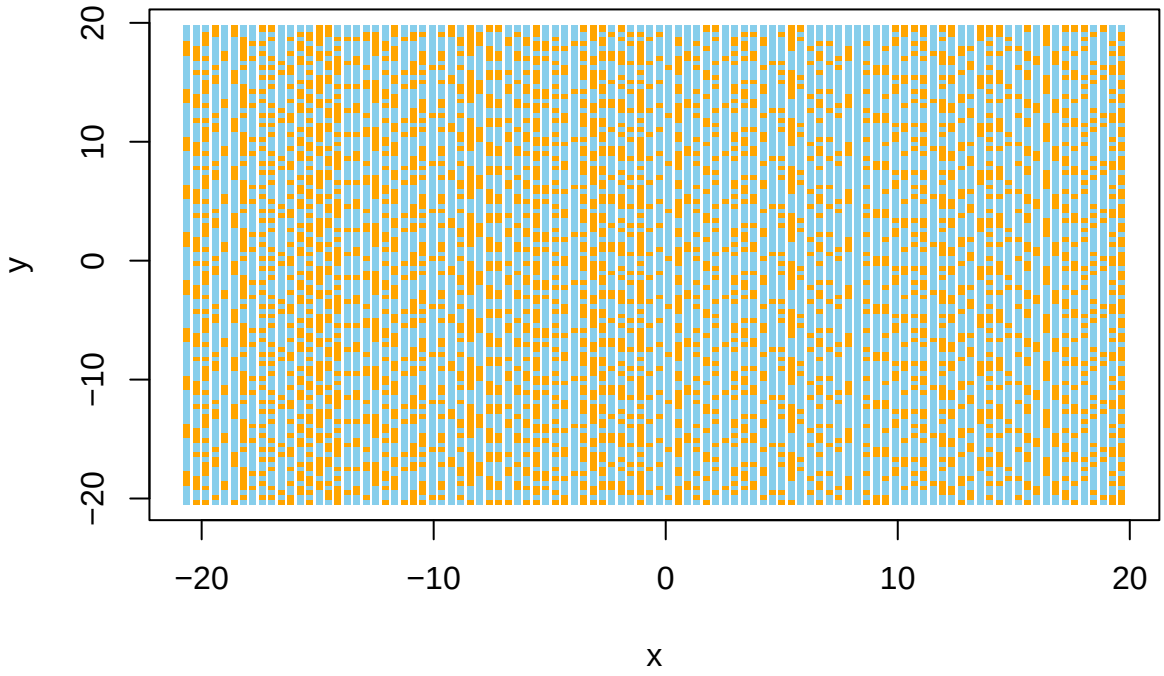
Now let's check how a logistic regression model does on this data.

```
# Use a logistic regression model to classify the data
logit_model <- glm(class ~ x + y, data = data2, family = "binomial")
# Predict using the model
predicted_probabilities <- predict(logit_model, type = "response")
```

```r
# Change all the Falses to 0 and Trues to 1
predicted_probabilities <- ifelse(predicted_probabilities > 0.5, 1, 0)

# Create a confusion matrix to evaluate the model
library(caret)
confusionMatrix(as.factor(predicted_probabilities), as.factor(data2$class))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 352 295
##          1 161 192
##
##                Accuracy : 0.544
##                  95% CI : (0.5125, 0.5752)
##     No Information Rate : 0.513
##     P-Value [Acc > NIR] : 0.02675
##
##                   Kappa : 0.081
##
##  Mcnemar's Test P-Value : 4.715e-10
##
##             Sensitivity : 0.6862
##             Specificity : 0.3943
##          Pos Pred Value : 0.5440
##          Neg Pred Value : 0.5439
##              Prevalence : 0.5130
##          Detection Rate : 0.3520
##    Detection Prevalence : 0.6470
##       Balanced Accuracy : 0.5402
##
##        'Positive' Class : 0
##
```

```r
# Plot the decision boundary
predicted_labels_logit <- predicted_probabilities > 0.5
plot(grid$x, grid$y, col = boundary_colors[predicted_labels_logit + 1], pch = ".", cex = 3.5, xlab = "x
```




As we can see from the decision boundary this model isn't very good at classifying the data, only getting an accuracy of 54.4%. This is better than the linear kernel, but still not very good, and looking at the decision boundary we can clearly see that it is no where close to the true decision boundary.

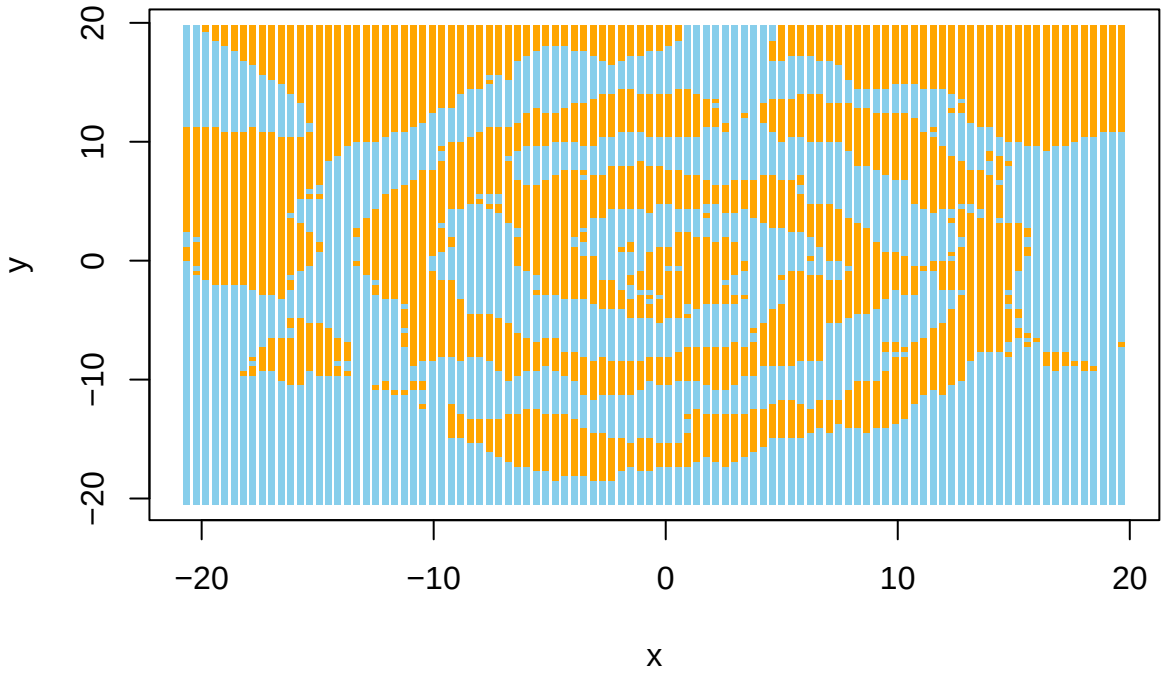
Finally lets check how a KNN model does on this data.

```
# Load the required class library
library(class)

# Use a KNN model to classify the data
knn_model <- knn(train = data2[, c("x", "y")], test = data2[, c("x", "y")], cl = data2$class, k = 5)

# Convert knn_model to numeric
knn_model <- as.numeric(knn_model)

# Convert 1s to 0s and 2s to 1s
knn_model <- knn_model - 1

# Create a confusion matrix to evaluate the model
library(caret)
confusionMatrix(as.factor(knn_model), data2$class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 430  99
##          1  83 388
```

```
##
##                Accuracy : 0.818
##                  95% CI : (0.7927, 0.8415)
##     No Information Rate : 0.513
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.6355
##
##  Mcnemar's Test P-Value : 0.2662
##
##             Sensitivity : 0.8382
##             Specificity : 0.7967
##          Pos Pred Value : 0.8129
##          Neg Pred Value : 0.8238
##              Prevalence : 0.5130
##          Detection Rate : 0.4300
##    Detection Prevalence : 0.5290
##       Balanced Accuracy : 0.8175
##
##        'Positive' Class : 0
##
```

```r
# Create the grid for the decision boundary
predicted_labels_knn <- knn(train = data2[, c("x", "y")], test = grid, cl = data2$class, k = 5)

# Convert knn_model to numeric
predicted_labels_knn <- as.numeric(predicted_labels_knn)

# Convert 1s to 0s and 2s to 1s
predicted_labels_knn <- predicted_labels_knn - 1

# Plot the decision boundary
plot(grid$x, grid$y, col = boundary_colors[predicted_labels_knn + 1], pch = ".", cex = 3.5, xlab = "x",
```




This time the accuracy is quite good at 81.8% which is almost as good as the RBF kernel. However, the decision boundary doesn't look quite as good as the RBF kernel and struggles to keep a consistent width throughout the decision boundary. It also has some strange gaps similar to the RBF kernel. Let's see how it does on the test data just to be sure.

```
# Create a confusion matrix to evaluate the KNN model
knn_model_test <- knn(train = data2[, c("x", "y")], test = test_data2[, c("x", "y")], cl = data2$class,

# Convert knn_model to numeric
knn_model_test <- as.numeric(knn_model_test)

# Convert 1s to 0s and 2s to 1s
knn_model_test <- knn_model_test - 1

# Create a confusion matrix to evaluate the model
library(caret)
confusionMatrix(as.factor(knn_model_test), test_data2$class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 358 145
##          1 133 364
##
##                Accuracy : 0.722
##                  95% CI : (0.6931, 0.7496)
```

```
##     No Information Rate : 0.509
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.4441
##
##  Mcnemar's Test P-Value : 0.5094
##
##             Sensitivity : 0.7291
##             Specificity : 0.7151
##          Pos Pred Value : 0.7117
##          Neg Pred Value : 0.7324
##              Prevalence : 0.4910
##          Detection Rate : 0.3580
##    Detection Prevalence : 0.5030
##       Balanced Accuracy : 0.7221
##
##        'Positive' Class : 0
##
```

- Potential TODO: Try a naive bayes model on this data. - Kellen Nankervis

On the test data the KNN model gets an accuracy of 72.2% which is a bit lower than the 75.6% the RBF kernel got. This is likely due to the decision boundary not being as good as the RBF kernel. Still this would be a reasonable candidate for a model to use on this data, especially since we do not need to run cross-validation to find the best parameters for the model. If we want the very best fit however, the RBF kernel with parameters picked through cross-validation would be the best choice.

- TODO: Work on 2's and 7's example. - Kellen Nankervis
- TODO: Write final conclusion. - Kellen Nankervis
- Example of how to plot I found online. Should be removed for final paper but I thought it might be helpful to look at. - 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)

# Plot the decision boundary
```

```r
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)
```