MATH 3190 Final Project

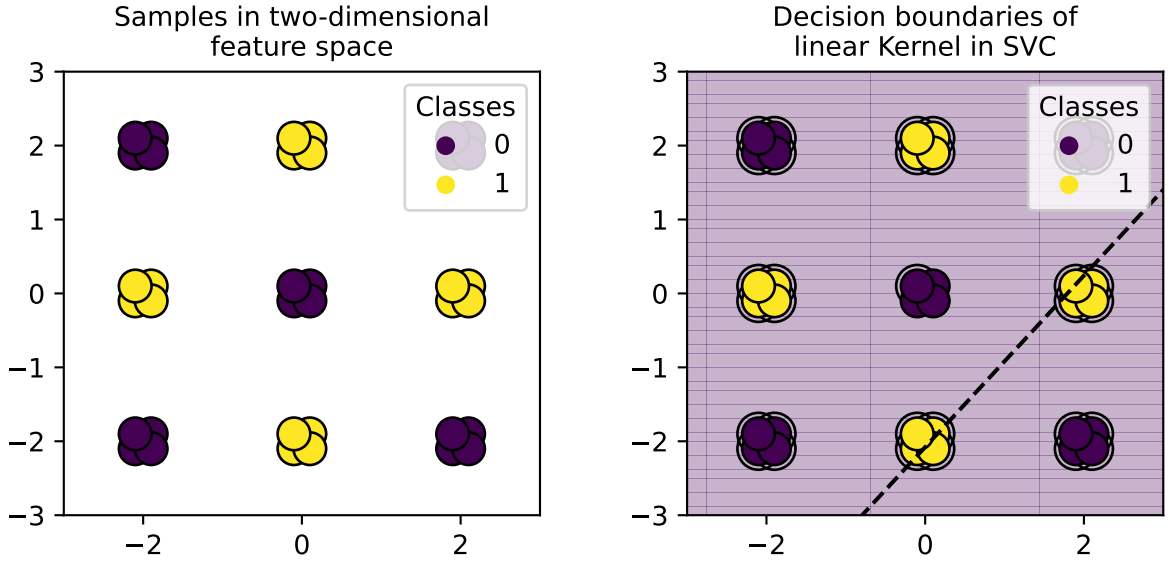
Jun Hanvey, Ian McFarlane, Kellen Nankervis

Due April 25, 2024

Introduction

In Notes 11, we explored Support Vector Machines, or SVMs, which are highly powerful classifiers. To put it briefly, they work by finding a linear hyperplane which maximizes the margin between two classes, called the decision boundary. But what happens if our data are not linearly separable and therefore a linear decision boundary cannot be formed? We can use a kernel trick, which allows us to transform the data into a higher dimension, becoming linearly separable, without actually transforming the data. The RBF Kernel in particular finds an infinite-dimensional projection of the data. In this project we will explore the Radial Basis Function Kernel and how it is an extremely powerful tool for classifying data that could not be classified with a linear kernel or other data with a extremely curved decision boundary.

Consider the following non-linearly separable data. Observe how the best SVM can do is classify everything as the majority class (purple).



There are some workarounds by using other **Kernels**. But, in order to understand what a Kernel is, we need to dive into the math for SVMs.

Disclaimer: For the math sections we'll be focusing on the hard margin case as it's easier to follow. Nonetheless, the insights we gain will also apply to the soft margin case.

Dual Problem

For the Support Vector Machine algorithm, our goal is to find a β 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

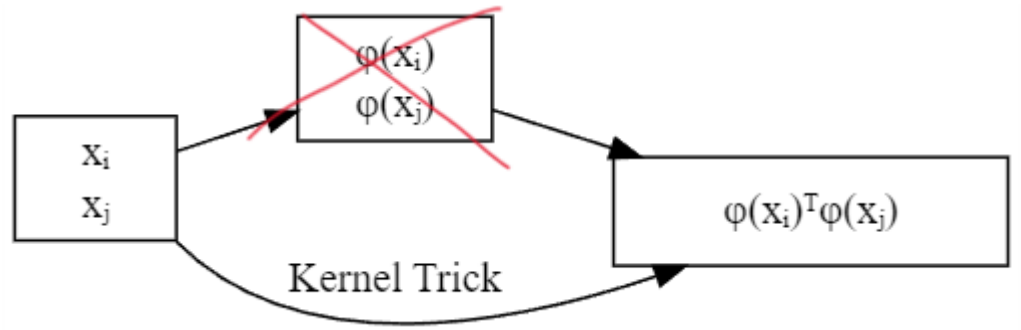
But, why did we go through all that trouble? In this form, we can observe that the Lagrangian depends on 3 components only:

- λ_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 ϕ is what we call the **Kernel Trick**.



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

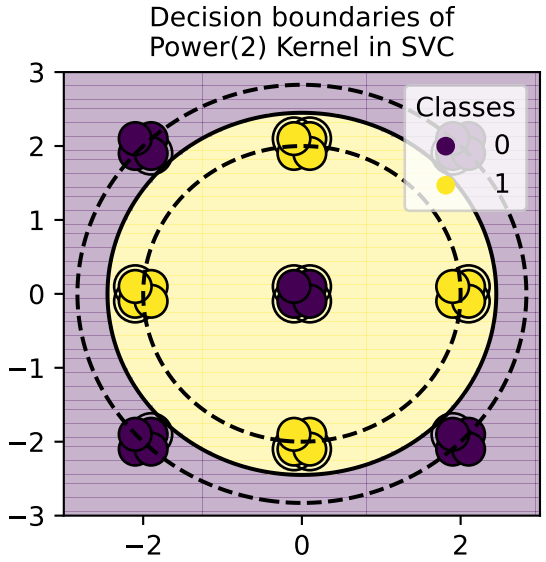
If we expand the dot product, we get:

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

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, $\phi_{Power(2)}(\boldsymbol{x}) = (x_1 x_1, x_1 x_2, \cdots, x_1 x_n, x_2 x_1, x_2 x_2, \cdots, x_2 x_n, \cdots, x_n x_1, x_n x_2, \cdots, x_n x_n)^T$ in this case 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. Applying this kernel on our data clearly results in a better classification, although not quite perfect:



Likewise, one can show that:

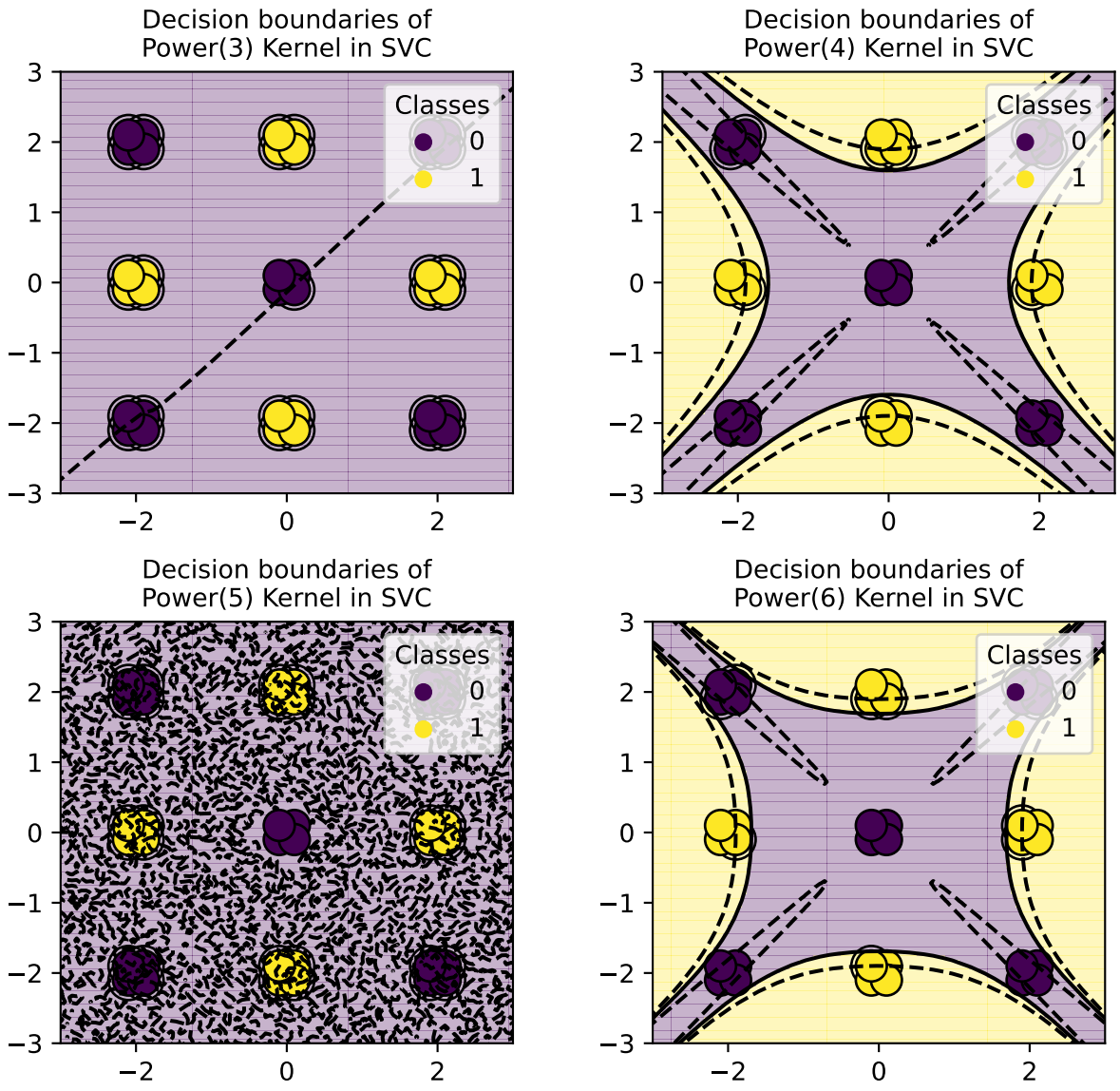
$K_{Power(3)}(\boldsymbol{a}, \boldsymbol{b}) = (\boldsymbol{a} \cdot \boldsymbol{b})^3$ corresponds to the transformation containing all 3-way interaction terms

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

$\vdots$

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

Similarly, we can try these Kernels on our data:

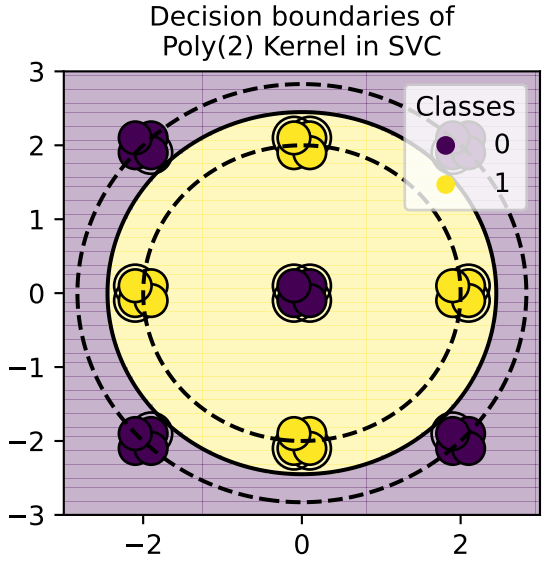


Like the base Kernel, the odd-power Kernels suck at classifying the data, while even power kernels do a fantastic job, specially after 4. It's also important to note that weird boundaries arise when our Kernel degree gets higher.

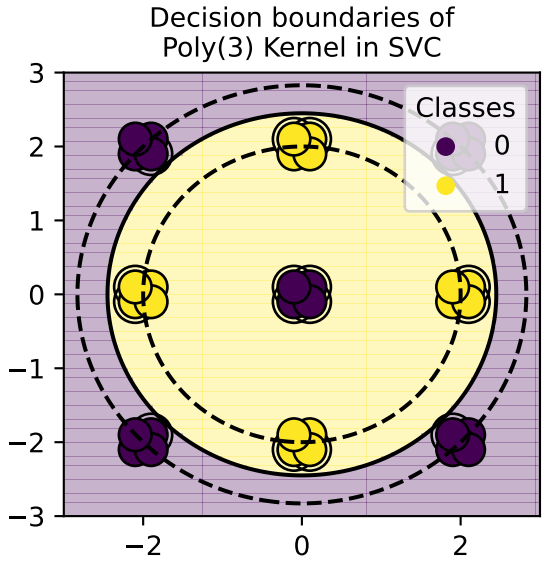
However, as we learned in Applied Statistics, whenever we include high order terms, we also want to include all lower level terms. So, let's inspect the following kernel:

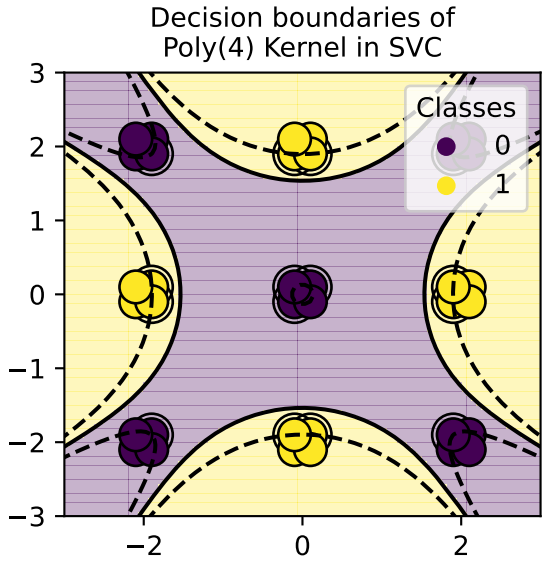
$$K_{Poly(2)}(\boldsymbol{a}, \boldsymbol{b}) = (\boldsymbol{a} \cdot \boldsymbol{b} + 1)^2 = \underbrace{(\boldsymbol{a} \cdot \boldsymbol{b})^2}_{\text{2-way}} + \underbrace{2(\boldsymbol{a} \cdot \boldsymbol{b})}_{\text{1-way}} + \underbrace{1}_{\text{0-way}}$$

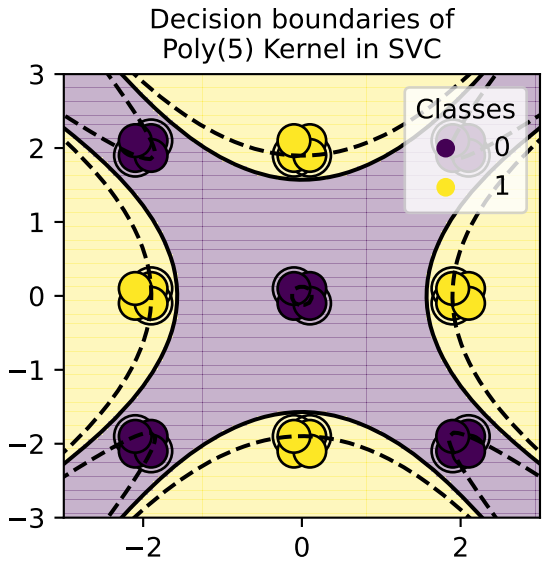
So, $K_{Poly(2)}$ gives us the 2-way interaction terms and all the lower order terms. Likewise, $K_{Poly(n)} = (\boldsymbol{a} \cdot \boldsymbol{b} + 1)^n$ gives us the n-way interaction terms and below, precisely what we wanted. This is also the polynomial kernel for SVM with $\gamma = 1$. After trying $K_{Poly(2)}$ on our data, we get a similar plot to the one produced by $K_{Power(2)}$:

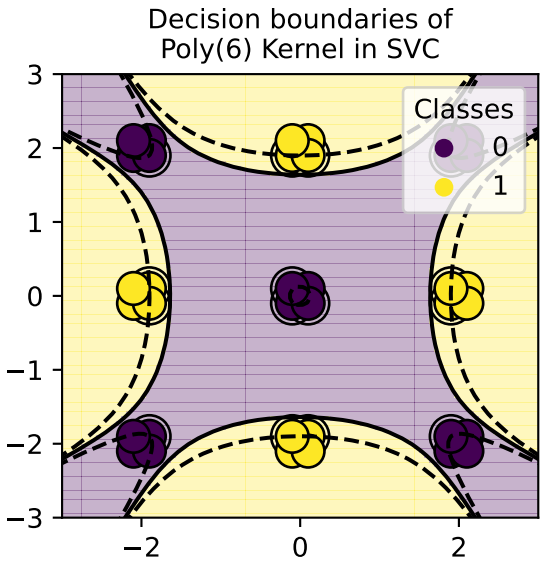


And when we run it with higher degree Kernels, we can observe that the even-degree Kernels look very similar. However, the odd degrees Kernels can in fact classify the data efficiently as they also "include" even-degree terms.

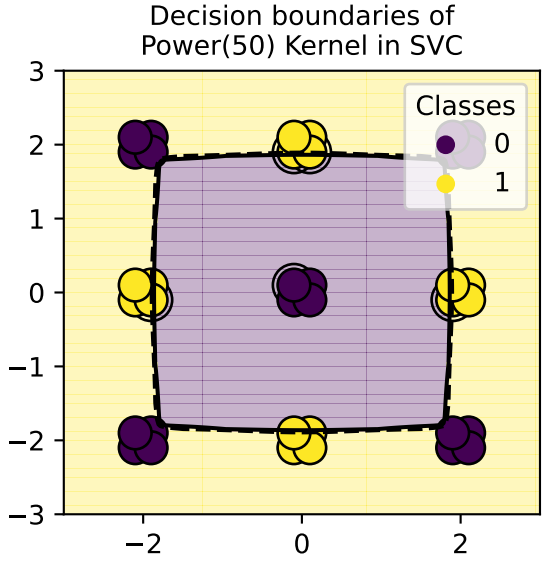


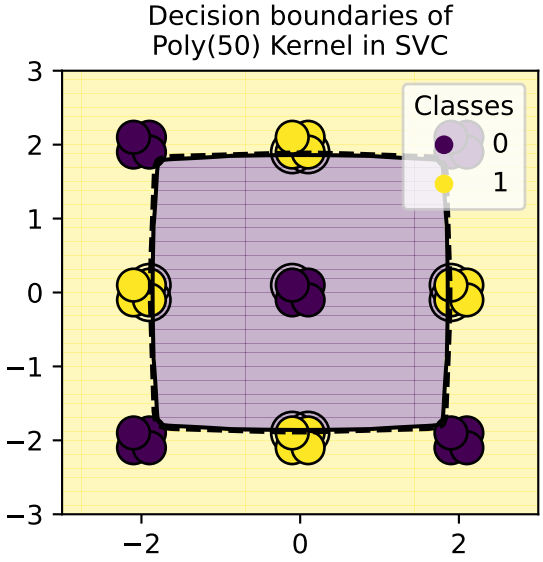






Once again, we get weirdness as our Kernel degree gets higher.





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:

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

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:

In practice, any SVM algorithm normalizes the data before performing the classification. So, let's multiply our kernel by $\exp\left(-\frac{\|\boldsymbol{a}\|_2^2 + \|\boldsymbol{b}\|_2^2}{2}\right)$. That way, outliers or observations that deviate a lot contribute less to our model.

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

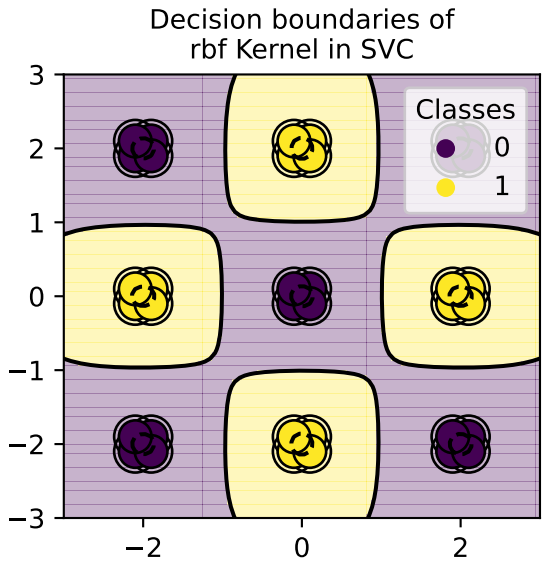
Now, let's raise it to the 2γ to add a control parameter. The control parameter adjusts for the importance of higher order terms. The higher it is, the more relevant they become.

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

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

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

After running the SVM with the RBF kernel, we get pretty nice non-linear decision boundaries.



Observe that the "majority class" becomes the "background", while the "minority class" creates some "splodges" across the plane.

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.

Strengths and Weaknesses

Strengths:

1. RBFs are able to perform optimization over an ∞-dimensional space using a relatively simple formula, making them computationally inexpensive while still having a lot of power.
2. There are no preassumptions that need to be made about the data, making it a great tool when the distribution of data is not known.
3. RBF decisions are local ones, making it a more robust algorithm that is not sensitive to outliers, in comparison to some other kernels.
4. The hyperparameter γ and the cost parameter (soft margin case) allow us to be more flexible about model specificity.

Weaknesses:

1. RBFs are non probablistic, meaning points are only classified as being in one class or the other. We cannot adjust the probability cutoff for classifying points like we might in logistic regression. However, We *can* FORCE a probabilistic model by fitting a logistic regression around the RBF model.
2. Like SVMs, we do not get easily interpretable coefficients. So, it becomes difficult to assess the influence of a specific variable on classification.
3. While this isn't necessarily a weakness per se, it's worth mentioning that the RBF kernel is non-parametric. This makes traditional types of inference significantly harder.
4. Because RBFs are more complex models, it is easier to overfit them.

Example Data

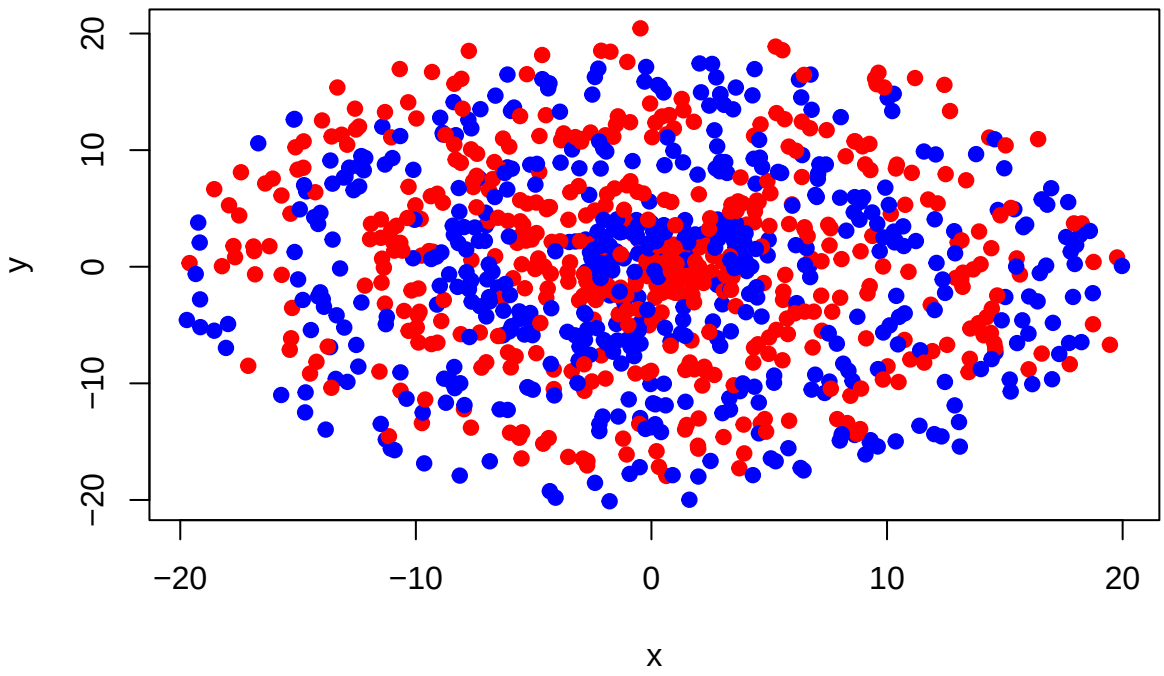
To show the power of the Radial Basis Function we will first use a generated data set.

Before we get started let's briefly discuss the cost and gamma parameters. The cost parameter is the cost of misclassifying a data point. A higher cost will lead to a more complex model that will try to classify all data points correctly. It trades off misclassification of training examples against increasing the margin. Larger values of cost will lead to a smaller margin. In this way cost behaves as a regularization parameter. The gamma parameter controls how far the influence of a single training example reaches. A low gamma will consider points far away from the decision boundary in calculations, while a high gamma will consider only points close to the decision boundary. The gamma parameters can be seen as the inverse of the radius of influence of samples selected by the model as support vectors.

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.

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")
```



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
```

```
# 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")
```

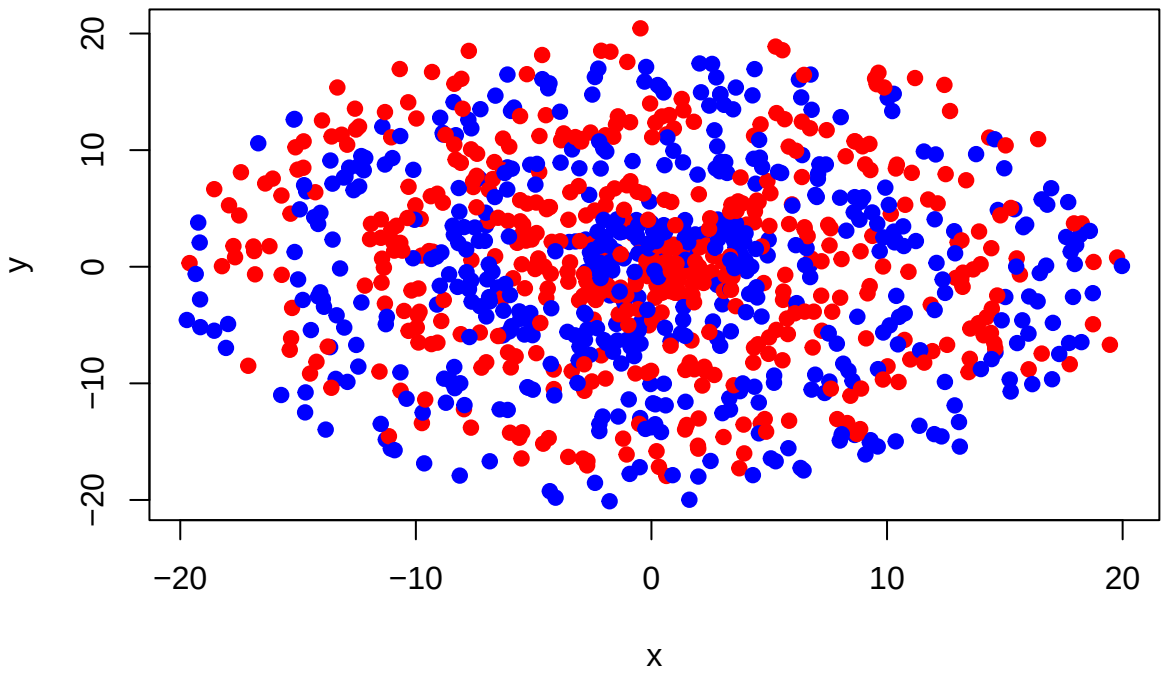
```
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##
## Number of Support Vectors:  945
```

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

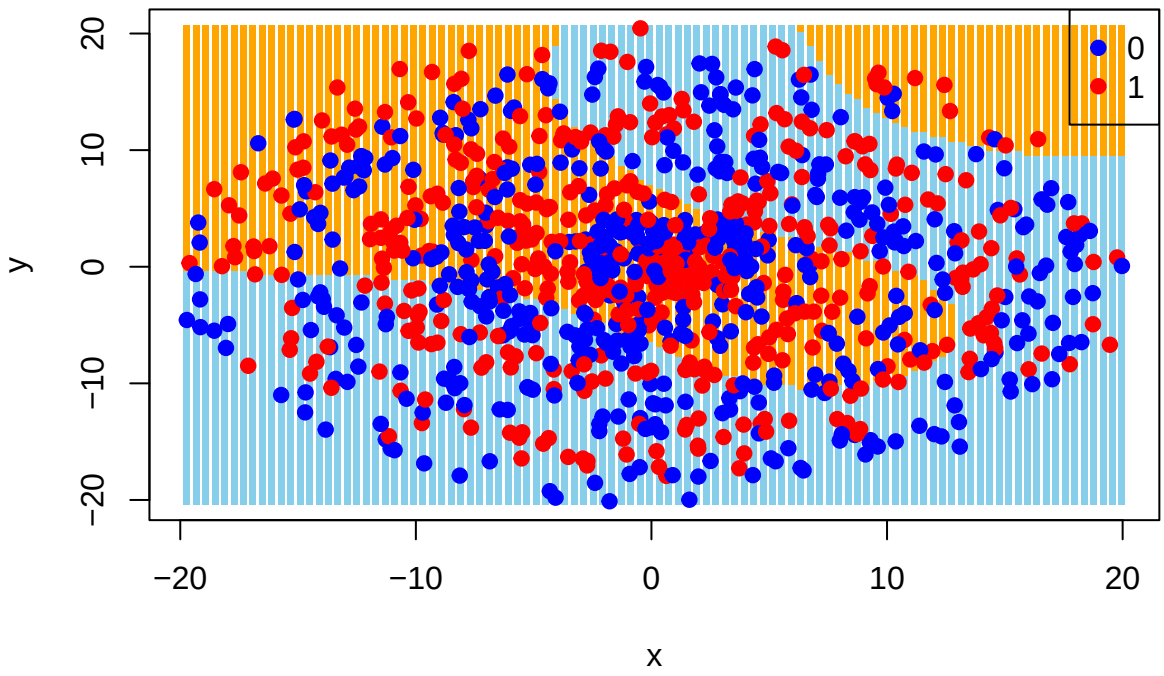
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 275 211
##          1 238 276
##
##                Accuracy : 0.551
##                  95% CI : (0.5196, 0.5821)
##     No Information Rate : 0.513
##     P-Value [Acc > NIR] : 0.008782
##
##                   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. For the data in this project we have a split near 50% for both classes and predicting one class incorrectly isn't worse than the other which means that accuracy works fine as an overall metric to gadge the models ability. For all future confusion matixes we will look at this and kappa only to save space. Let's look at the decision boundary to see what it thinks, here is the code to do that.

```
# 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")
```

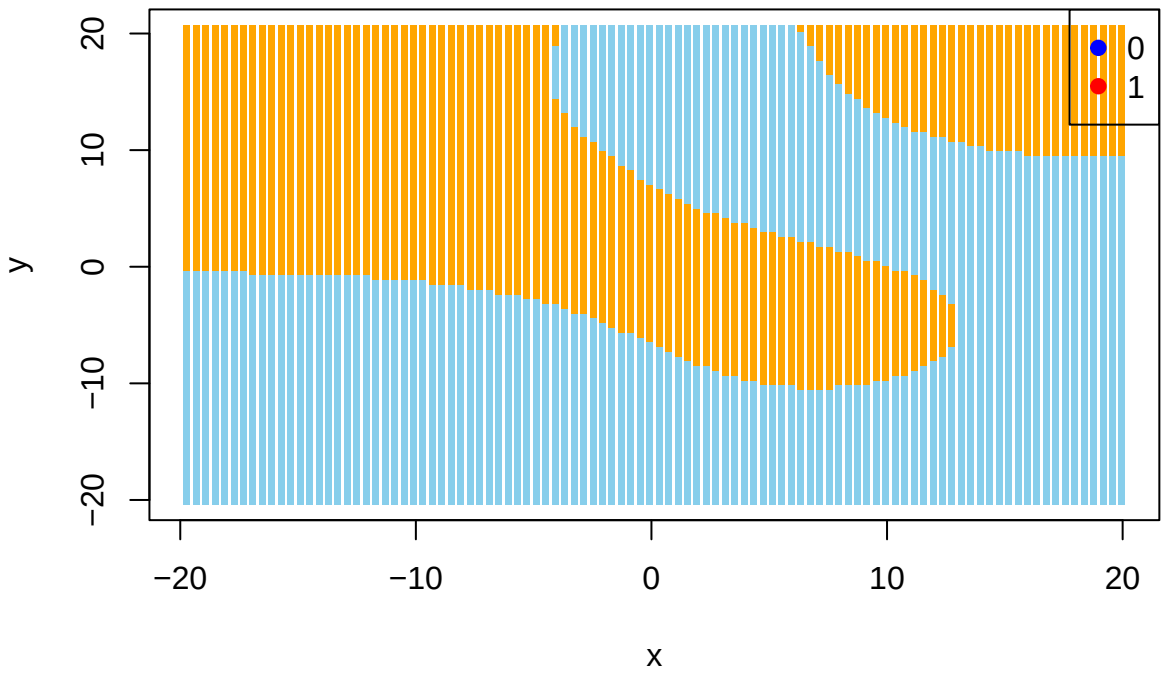



```r
# Plot the decision boundary
x1_grid <- seq(min(data2$x), max(data2$x), length.out = 100)
x2_grid <- seq(min(data2$y), max(data2$y), length.out = 100)
grid <- expand.grid(x = x1_grid, y = x2_grid)

predicted_labels <- predict(svmfit, newdata = grid)

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

# 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 just want to plot the decision boundary we can use the following code.

```
# Plot the decision boundary
plot(grid$x, grid$y, col = boundary_colors[predicted_labels], pch = ".",
     cex = 3.5, xlab = "x", ylab = "y")
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 = gamma)

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

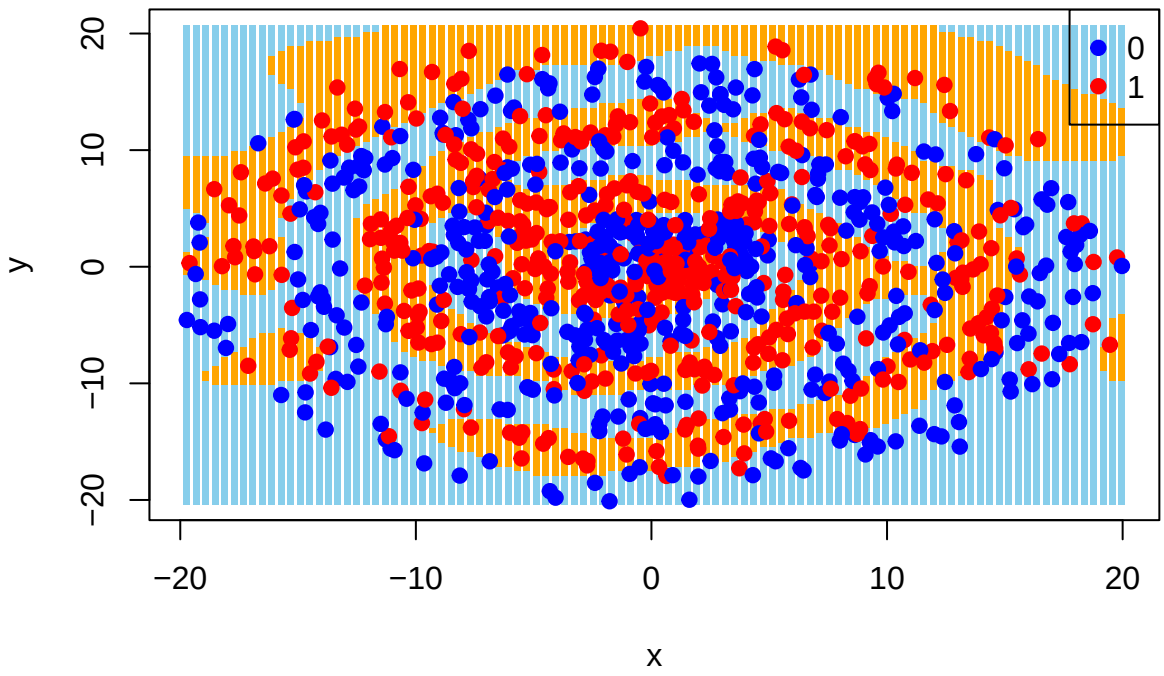
```
  return(results)
}

# Create a simpler confusion matrix function
simpler_cm <- function(cm.out) {
    print(cm.out[[2]])
    print(cm.out[[3]][1:2])
}
```

```
# 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, seed = 2024)

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

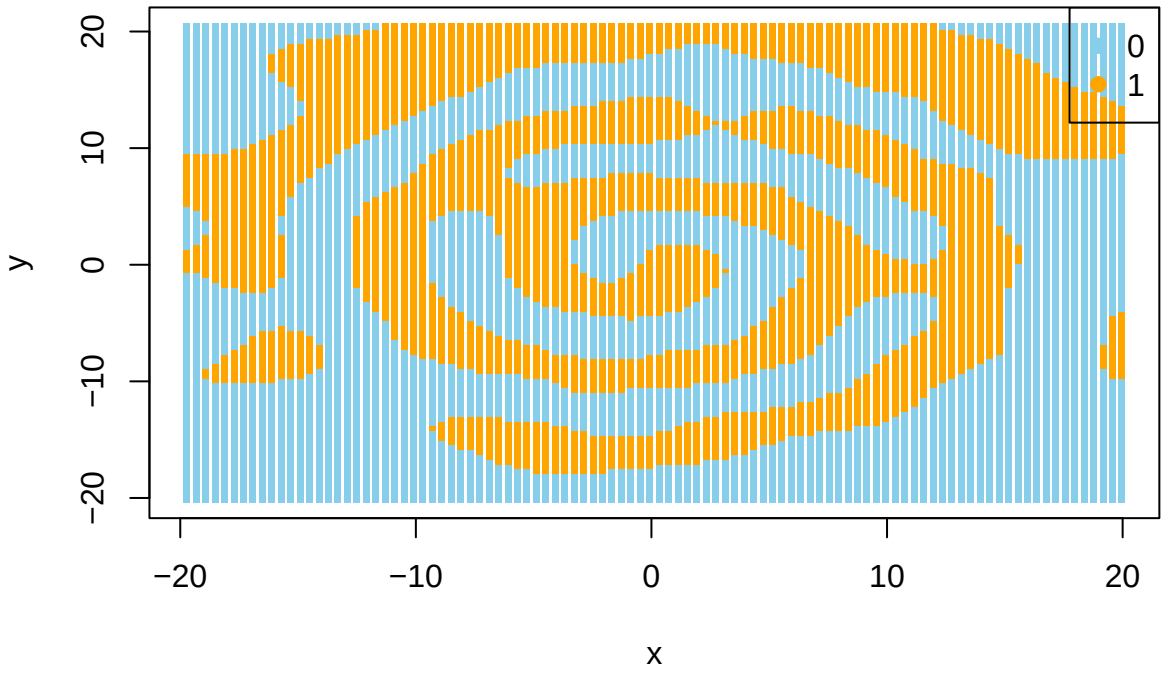
```
##    cost gamma accuracy
## 9     1    10     74.3
## 14   10    10     72.0
## 10    1   100     70.6
## 19  100    10     70.3
## 24 1000    10     67.6
```

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.

```
##           Reference
## Prediction   0   1
##          0 428  94
##          1  85 393
## Accuracy    Kappa
##  0.82100  0.64159
```



Let's look at just the decision boundary.



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.

```
##            Reference
## Prediction   0   1
##          0 375 128
##          1 116 381
##  Accuracy     Kappa
## 0.7560000 0.5120527
```

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.

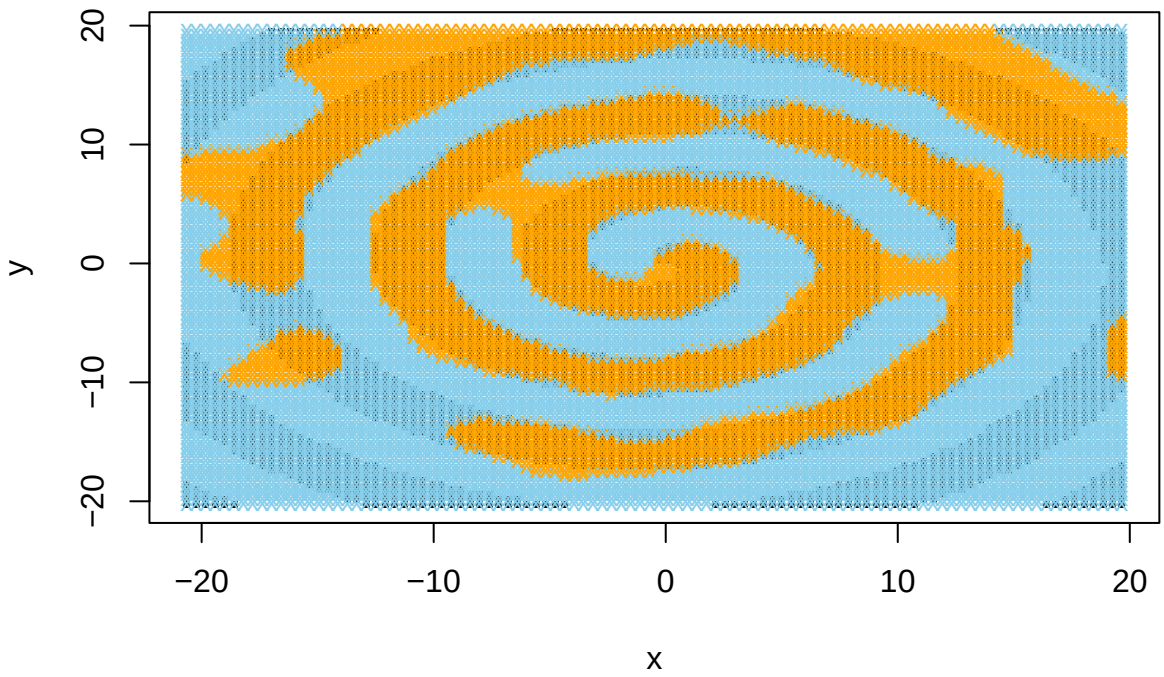
```
post_offset_class <- numeric(n)
new_r <- numeric(n)
new_theta <- numeric(n)

# This code is how I originally classified the data before offsetting it
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)
}
```

```
##            Reference
## Prediction   0   1
##          0 410 112
##          1  81 397
##  Accuracy     Kappa
## 0.8070000 0.6143055
```

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.



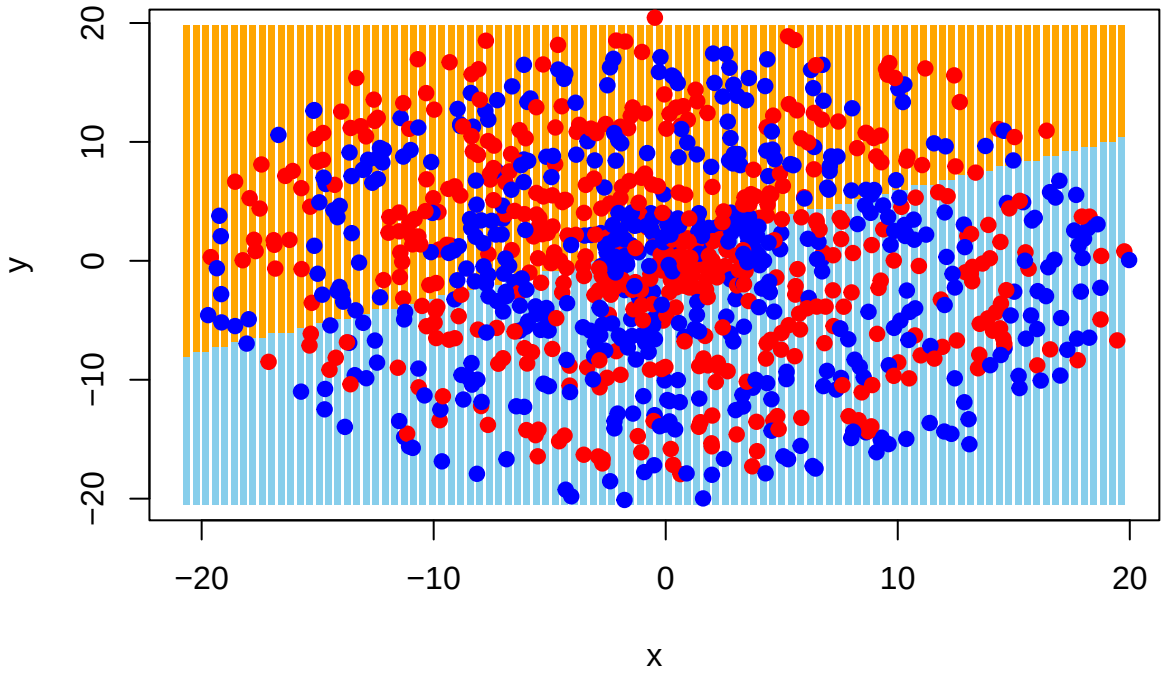
Now that we have seen the power of the RBF kernel we can compare it to other models. We will compare it to a linear kernel, a polynomial kernel, a logistic regression model without many interactions. We will also compare it to a KNN model.

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) |> simpler_cm()
```

```
##            Reference
## Prediction   0   1
##          0 296 268
##          1 217 219
##   Accuracy      Kappa
## 0.51500000 0.02676106
```

```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", ylab = "y")

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



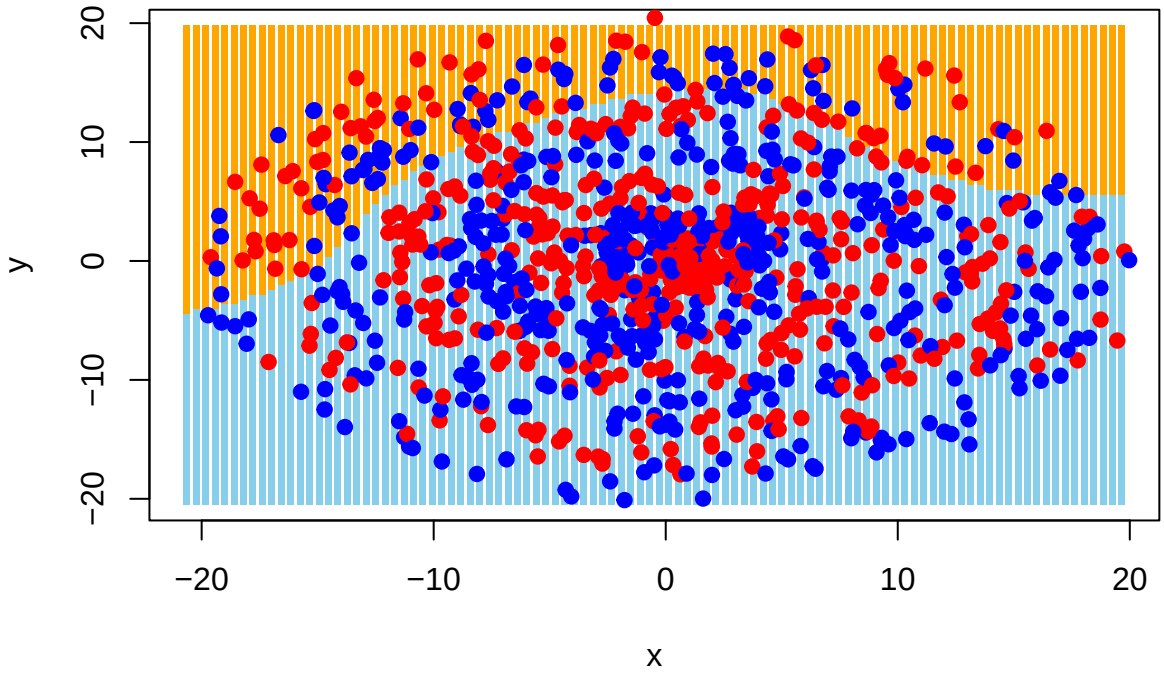
As we can see the linear kernel does not do a good job at classifying the data only getting a 51.5% accuracy (only slightly better than guessing) since no line cleanly fits the data. So lets try the polynomial kernel. I have used cross validation to find the best degree, cost, and gamma values for the polynomial kernel, but I will not run that code here since it takes a long time to run and the results are not good anyway.

From this we get that the best degree, cost, and gamma are 5, 10 and 0.1 respectively, although it does not seem like it will perform well. Let's check just to be sure.

```r
# 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) |> simpler_cm()
```

```
##            Reference
## Prediction   0   1
##          0 436 412
##          1  77  75
##    Accuracy       Kappa
## 0.511000000 0.003975949
```

```r
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", ylab = "y")

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



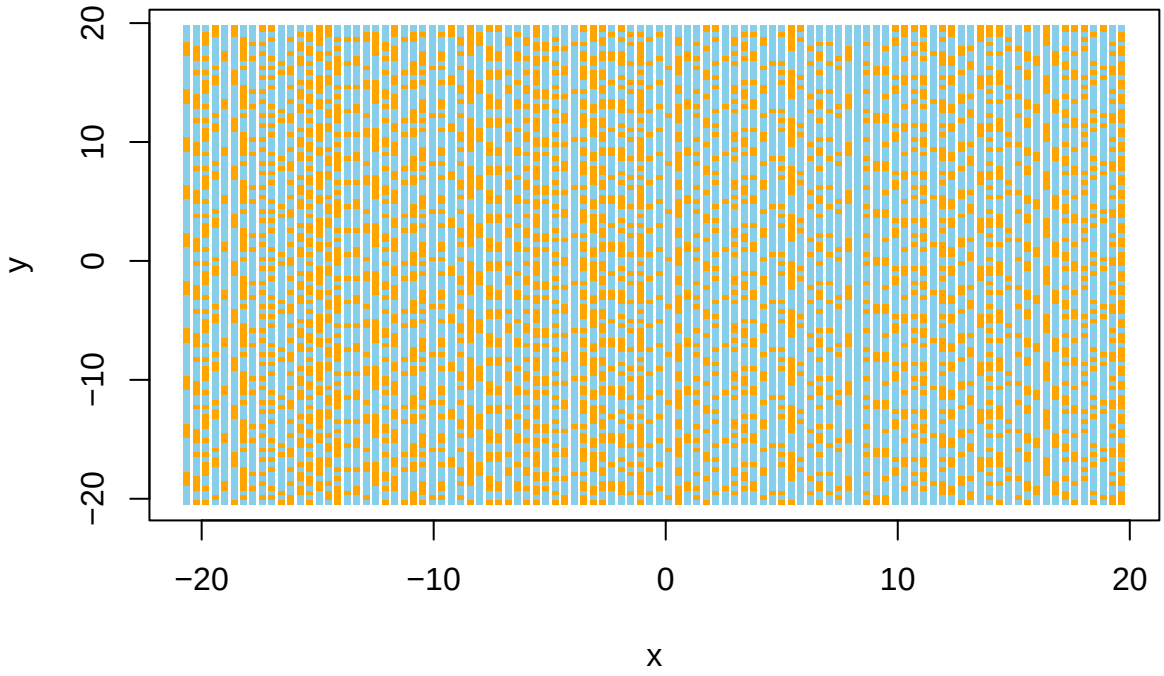
We can see that the very best polynomial kernel from our cross validation only gets an accuracy of 51.1% which is not very good, in fact it is worse than the linear kernel or just guessing the more likely (at least based on the test data) classification every time.

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

```r
# 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")

# 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
confusionMatrix(as.factor(predicted_probabilities), as.factor(data2$class)) |>
  simpler_cm()
```

```
##            Reference
## Prediction   0   1
##          0 352 295
##          1 161 192
##   Accuracy      Kappa
## 0.54400000 0.08097497
```

```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", ylab = "y")
```



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.

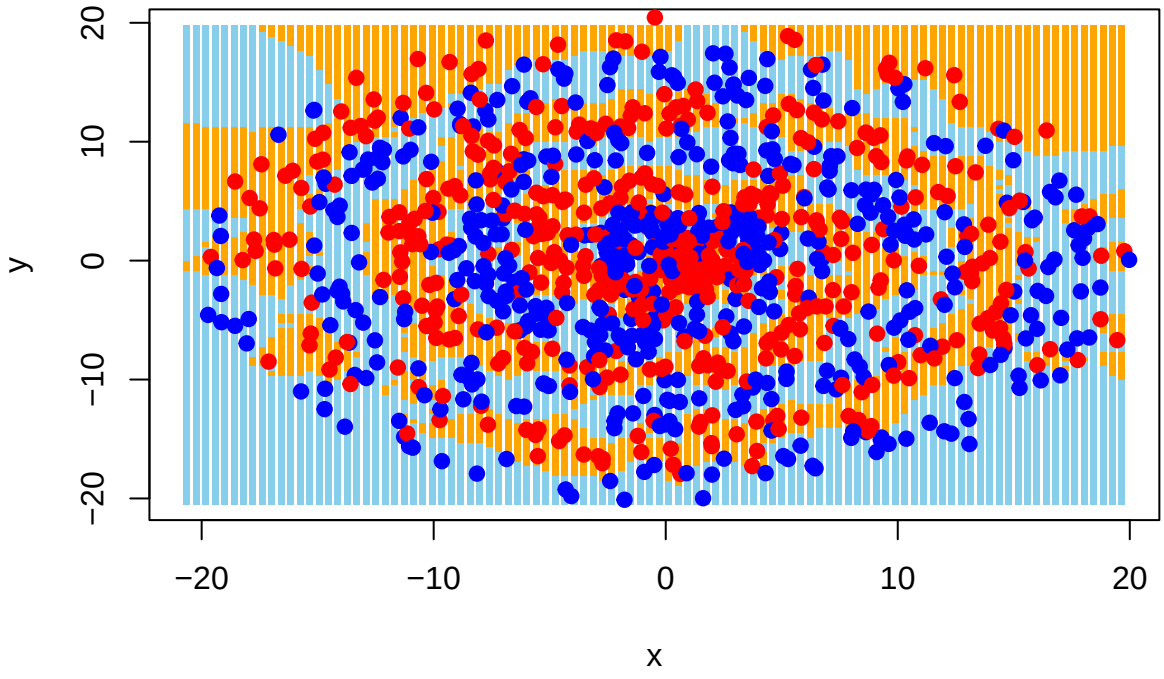
Finally lets check how a KNN model does on this data.

```
##    k  Accuracy     Kappa AccuracySD    KappaSD
## 2  3 0.7039671 0.4078728 0.01585862 0.03187415
## 3  5 0.6969572 0.3933439 0.02866241 0.05722388
## 5  9 0.6929621 0.3849269 0.03649458 0.07342050
## 4  7 0.6909369 0.3808089 0.03104536 0.06241822
## 6 11 0.6899670 0.3789681 0.04356385 0.08768379
```

We find that the best k value is 3 with an accuracy of 70.4%. I assume the k is so low because their is more correlation in the classification of the data points in certain directions at certain distances which makes the euclidean distance that KNN uses to classify (at least by default, although I won't be exploring other distance metrics in this example) not as useful, and therefore the model performs better looking at less neighbors. Now lets use this k value to classify the data and plot the decision boundary.

```
# Use the best k value found from cross validation
k <- 3

# Use a KNN model to classify the data
knn_model <- knn3(class ~ ., data = data2, k = k)
```

```
##           Reference
## Prediction   0   1
##          0 445  75
##          1  68 412
##  Accuracy     Kappa
## 0.8570000 0.7137023
```



This time the accuracy is quite good at 85.7% which is even better than the cross validated RBF kernel. I do think it has an unfair advantage in this category though since it can use the classification of itself as a neighbor to make a decision. This means if k = 1, the model would have 100% accuracy on the training set

as it would only look at it's own classification. The decision boundary is also looks quite good although it seems to have a harder time keeping a consistent width compared to the RBF SVM model. The results from the cross validation are also not as good as the RBF kernel, so lets see how it does on the test data to make a final conclusion.

```
##            Reference
## Prediction   0   1
##          0 359 142
##          1 132 367
##  Accuracy     Kappa
## 0.7260000 0.4520197
```

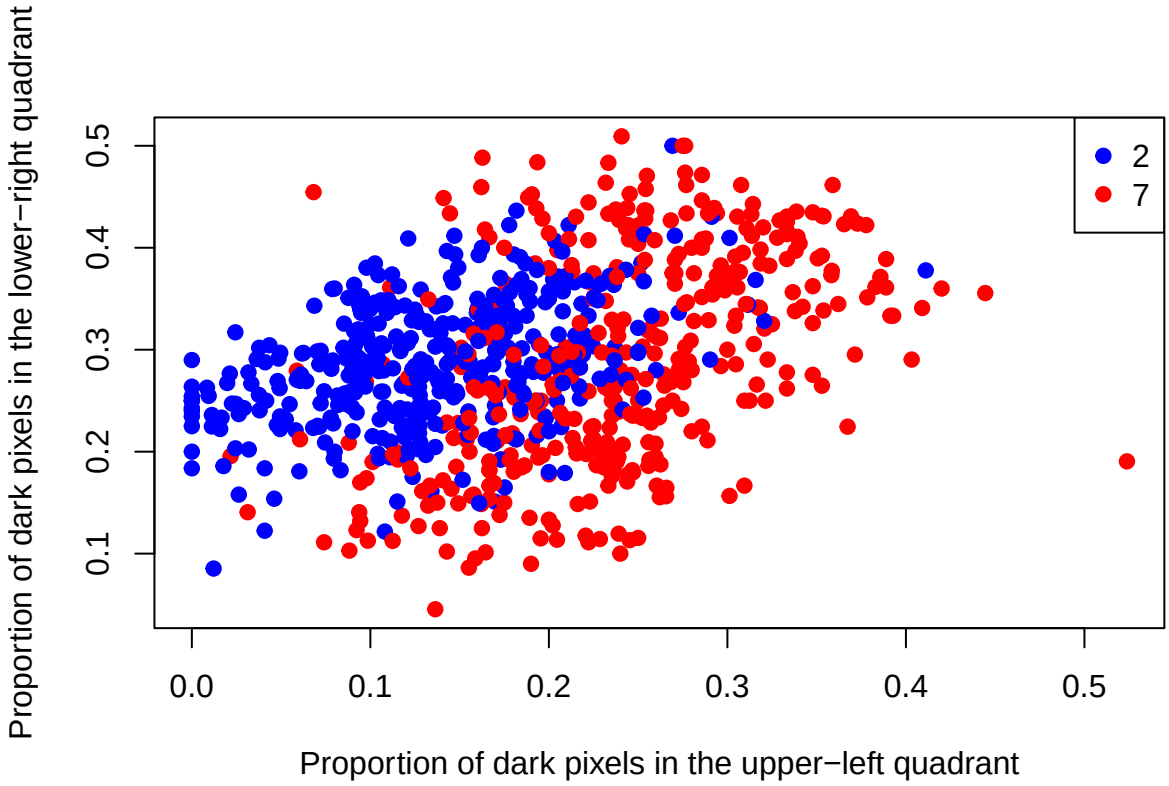
On the test data the KNN model gets an accuracy of 72.6% which is a bit lower than the 75.6% the RBF kernel got. This is likely due to the decision boundary not being as good as the RBF kernel. It also shows the the KNN model had more variance, and less bias then the RBF SVM, since it did better on classifying the training data, but worse on classifying the testing data. I do think this is somewhat of an unfair fight though, since on the training data the KNN model can use it's very own point as a neighbor, meaning it already knows the correct classification for at least that point. When there are only 3 neighbors this can have a big impact on the classification. Overall I would say the RBF kernel is the best model for this data since it did the best at classifying the test data and had the best decision boundary. The KNN model was a close second though.

Real Data Set

Now that we have demonstrated the power of the RBF kernel on a generated data set, we will now demonstrate the power of the RBF kernel on a real data set. We will use the famous MNIST data set which contains images of handwritten digits. We will only use the 2's and 7's from the data set since they are the most similar and thus the most difficult to classify. We will use the same process as before to classify the data and plot the decision boundary. This data set is also nice since it is a good example of where an RBF kernel would be useful since the decision boundary is not linear, and can also be graphed in 2 dimensions.

x_1 is the proportion of dark pixels in the upper-left quadrant, x_2 is the proportion of dark pixels in the lower-right quadrant, and y is the true classification for the digit (2 or 7).

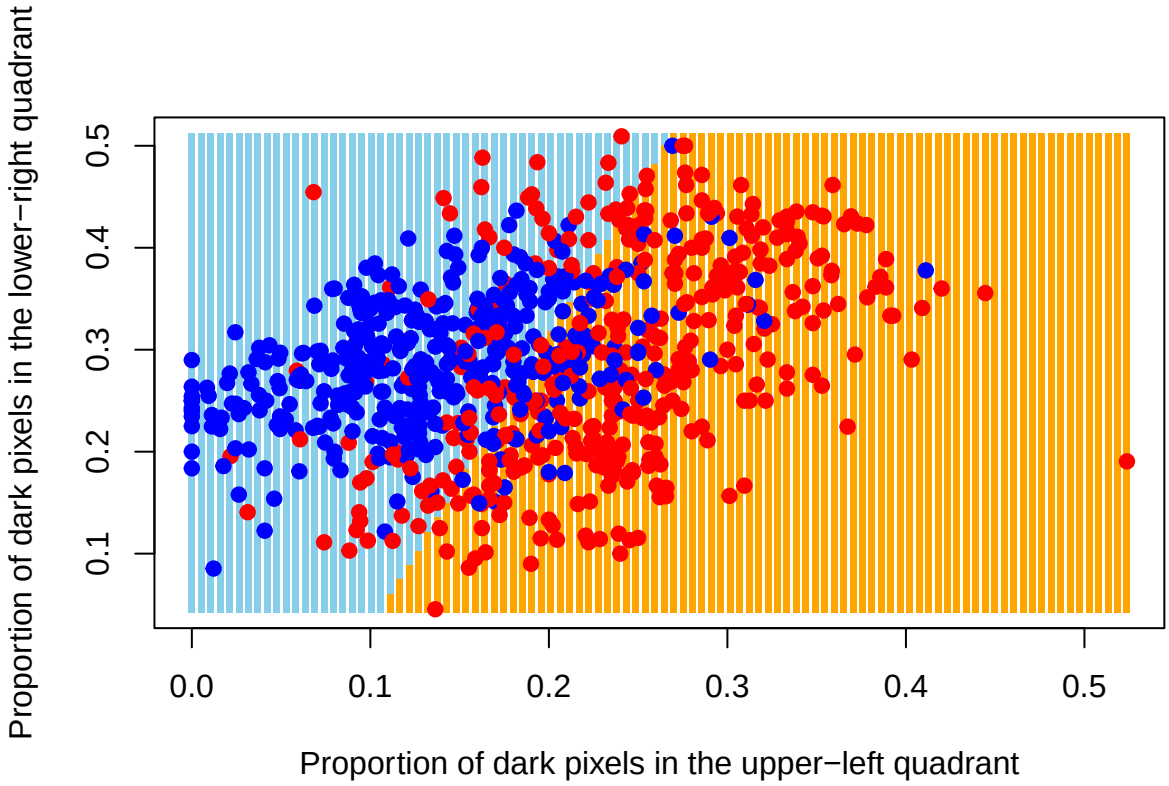
```
## Warning: package 'dslabs' was built under R version 4.3.3
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v lubridate 1.9.3     v tibble    3.2.1
## v purrr     1.0.2     v tidyr     1.3.1
## -- Conflicts ---------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## x purrr::lift()   masks caret::lift()
## x dplyr::select() masks MASS::select()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```



We can see from plotting the training data that the 2's and 7's are quite similar and thus will be difficult to classify. The sevens tend to be a bit more in the lower left corner meaning more of the pixels in the upper left are dark, and the twos tend to be a bit more in the upper right corner meaning slightly more of the pixels in the lower right are dark. This is a good example of where a linear kernel would not work well since the decision boundary is not linear.

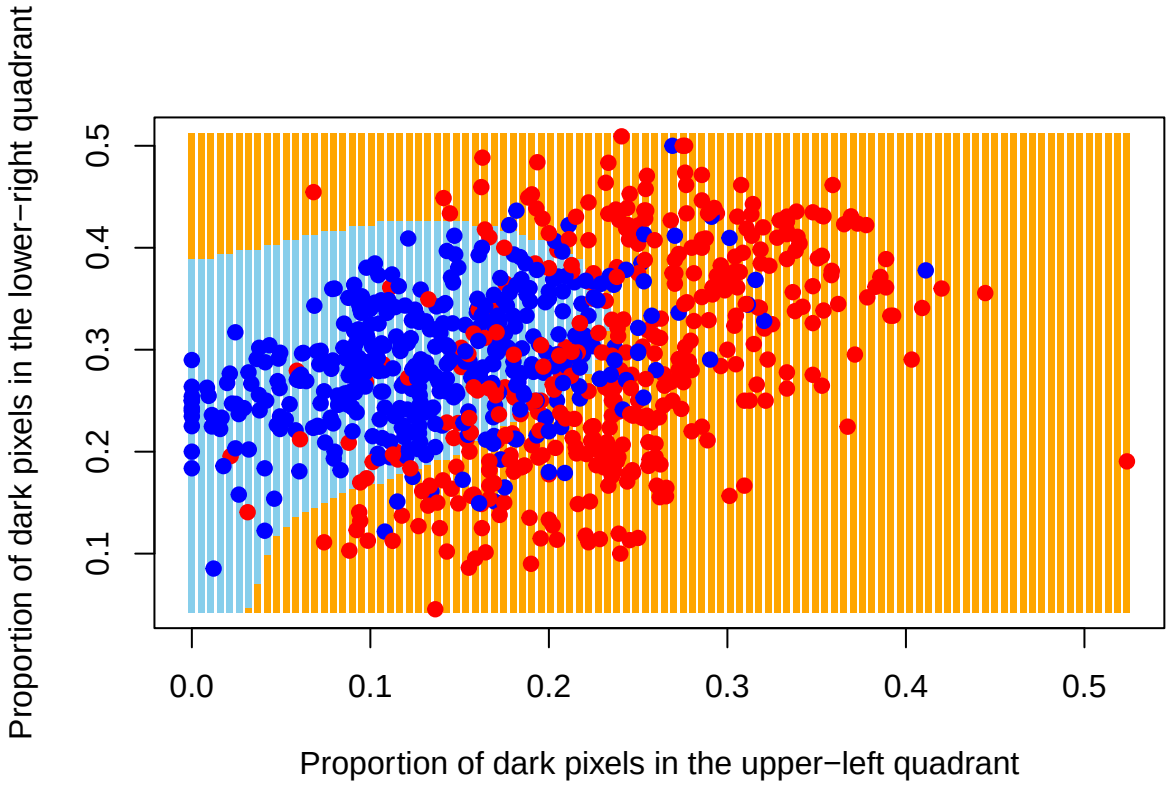
First let's try fitting a linear kernel to the data just to get a baseline, from here all the code will use the training data until we return to see how it does on the test data.

```
##            Reference
## Prediction   2   7
##          2 321  89
##          7  80 310
##  Accuracy     Kappa
## 0.7887500 0.5774736
```



We can see that our linear kernel does not quite do the job, it's accuracy of 78.88% is impressive, but not as good as we would like, but it gives us a good baseline to compare the other models to. Now let's try the RBF kernel.

```
##            Reference
## Prediction   2   7
##          2 342  74
##          7  59 325
##  Accuracy     Kappa
## 0.8337500 0.6674667
```

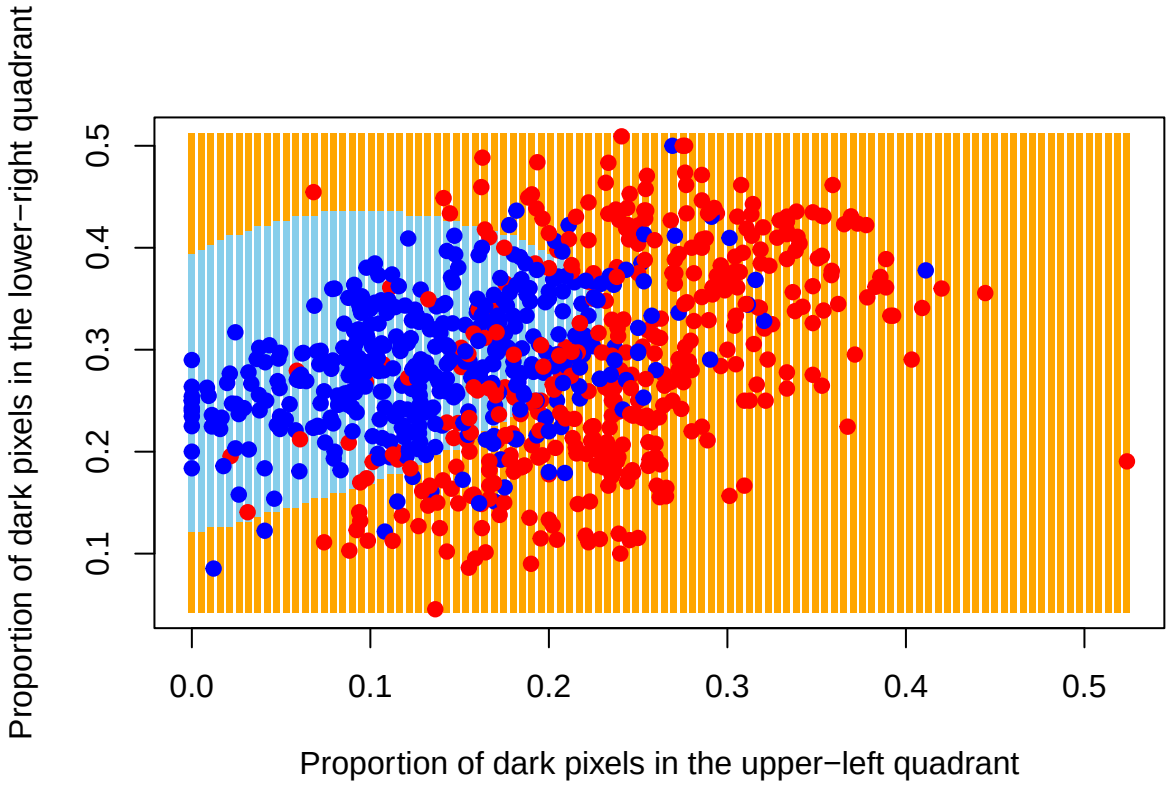


Even with the default values of cost and gamma the RBF kernel seems to do a better job of classifying the data than the linear kernel. The accuracy of 83.38% is better than the linear kernel, but still not as good as we would like. The decision boundary also looks better than the linear kernel, seeming to follow the data better without overfitting. Now let's try to find the best values for cost and gamma using cross validation.

```
##     cost gamma accuracy
## 3  1e-01  1.00   83.375
## 12 1e+01  0.10   83.250
## 21 1e+03  0.01   83.250
## 13 1e+01  1.00   83.125
## 17 1e+02  0.10   83.125
```

From this we see that lowering the cost down to 0.1 and keeping ht gamma at 1 seems to do the best at predicting new unseen data getting an accuracy of 83.375% on the test folds. Let's use this to train a new model and see how the decision boundary and confusion matrix change as a result.

```
##            Reference
## Prediction   2   7
##          2 335  63
##          7  66 336
## Accuracy    Kappa
## 0.838750 0.677504
```



I don't see much of a difference between the two decision boundaries and the accuracy has only increased to 83.88%, an improvement by 0.5%, it seems to only reclassify the bottom left corner of the data from 2's to 7's and slightly pull in the amount of area classified as twos along the edge causing more points to be classified as 7's overall. I do trust in the power of cross validation though so I will trust that this model is better than the default model. Now let's see how it does on the test data.

```
##           Reference
## Prediction  2  7
##          2 82 19
##          7 14 85
##  Accuracy     Kappa
## 0.8350000 0.6701319
```

We can see that on the test data which was 20% of the whole data set the model got an accuracy of 83.5% which is basically the same as the training. This means we have found a good model that generalizes well to new data and also did well on it's own training data. This means we have found a good mix of bias and variance in our model.

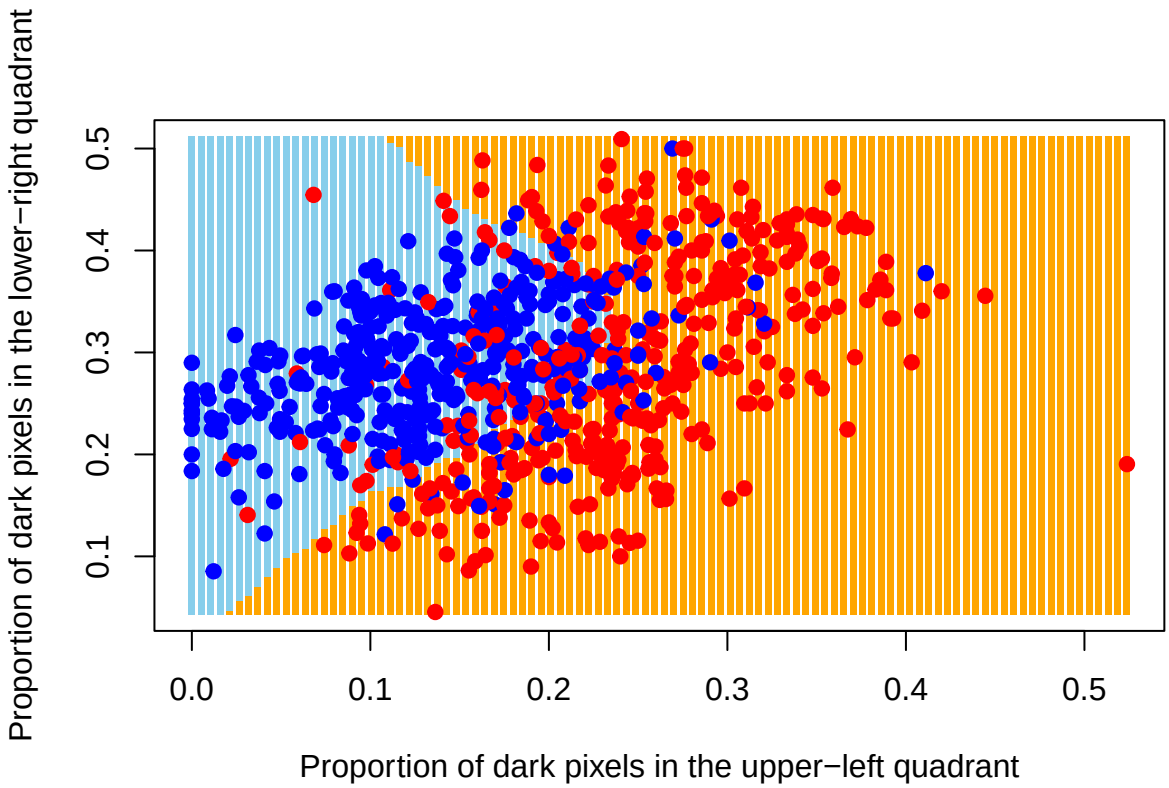
From the example data set we know that k-Nearest Neighbors also does well on these types of data sets so let's see how it does on this data set to make comparisons and see if we have found the best model.

```
##     k  Accuracy     Kappa AccuracySD    KappaSD
## 42 83 0.8338862 0.6677464 0.03817962 0.07631988
## 43 85 0.8338862 0.6677464 0.03817962 0.07631988
## 44 87 0.8338703 0.6677054 0.03819001 0.07633036
## 34 67 0.8326203 0.6652054 0.03640234 0.07275124
## 24 47 0.8301670 0.6603074 0.03938112 0.07872667
```

From the cross validation we see that a k value of 83 is the best value for this data set with an accuracy of 83.38% on the test folds. Let's use this k value to classify the data and plot the decision boundary.

```
##           Reference
```

```
## Prediction   2   7
##          2 343  75
##          7  58 324
##  Accuracy     Kappa
## 0.8337500 0.6674626
```



This decision boundary looks similar to the one the RBF kernel produced with the default values of cost and gamma, and the accuracy is the exact same at 83.38%. It seems that this decision boundary just slightly predicts more 2's than the RBF kernel, but overall they are very similar. Now let's see how it does on the test data so we can make a final conclusion.

```
##           Reference
## Prediction  2  7
##          2 83 23
##          7 13 81
##  Accuracy     Kappa
## 0.8200000 0.6408619
```

On the training data the KNN model got an accuracy of 82% which means it predicts worse by 1.5% than the RBF kernel which is equivalent to 3 more misclassifications. This means that the RBF kernel has again done the best job at classifying the data, this time both on the training and test data even if only by a small margin. This again shows the power of the RBF kernel on non-linear data sets for classification.

Conclusion

From the examples of the generated data set and the MNIST data set we can see that the RBF kernel is a powerful tool for classifying non-linear data sets. It was able to classify the generated data of the spiral with a decision boundary very close to the true decision boundary and an accuracy of 75.6% on the test data. This was better than the linear kernel, polynomial kernel, logistic regression model, and KNN model. It was

also able to classify the MNIST data set of 2's and 7's with an accuracy of 83.5% on the test data. The KNN model was a close second with an accuracy of 82% on the test data. One downside of the RBF kernel is that it is computationally expensive and can take a long time to train on large data sets. It is also crucial to perform cross validation to find the best values for cost and gamma since the default values are not always the best which further increases the computational cost. However once the model is trained it is very fast at classifying new data points and does a great job at generalizing to new data assuming the chosen parameters are good. Overall the RBF kernel in support vector machines is a great tool for classifying non-linear data sets and a tool every data scientist should have in their toolbox.