MATH 3190 Final Project

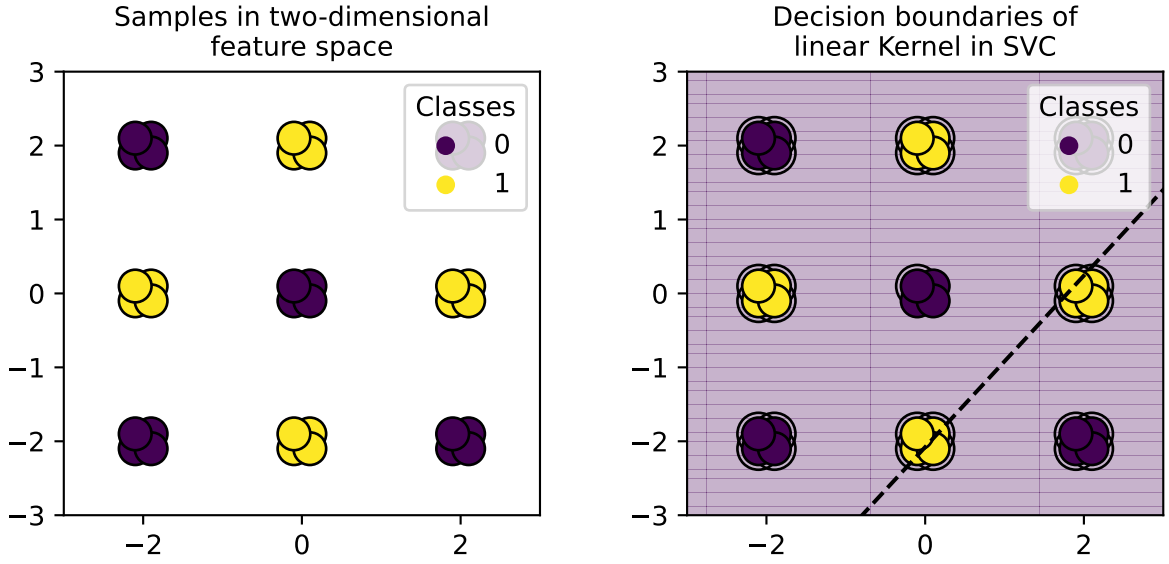
Jun Hanvey, Ian McFarlane, Kellen Nankervis

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.

Mathematics behind SVM

We'll now dive into the mathematics behind SVM's. The mathematics can be a little bit advanced, but understanding the results is sufficient. It's important to note that 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:

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

Then, we can find the dual optimization problem by the use 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 the following results:

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

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

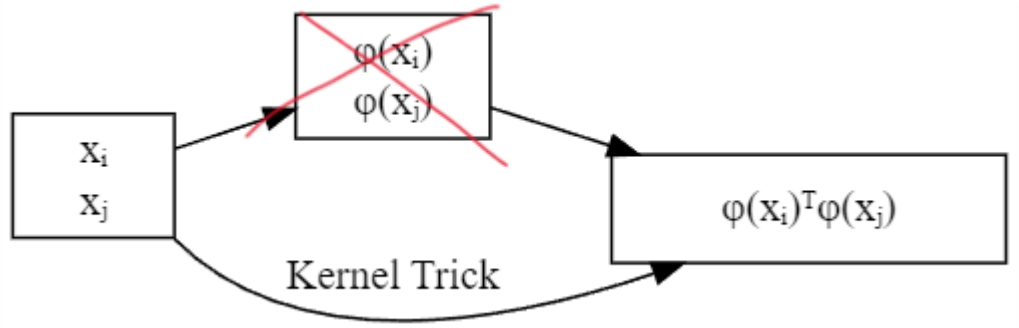
But, why did we go through all that trouble? In this form, we can observe that the Lagrangian depends on 3 components only:

- 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 $K_{Power(2)}(\boldsymbol{a}, \boldsymbol{b}) = (\boldsymbol{a} \cdot \boldsymbol{b})^2$

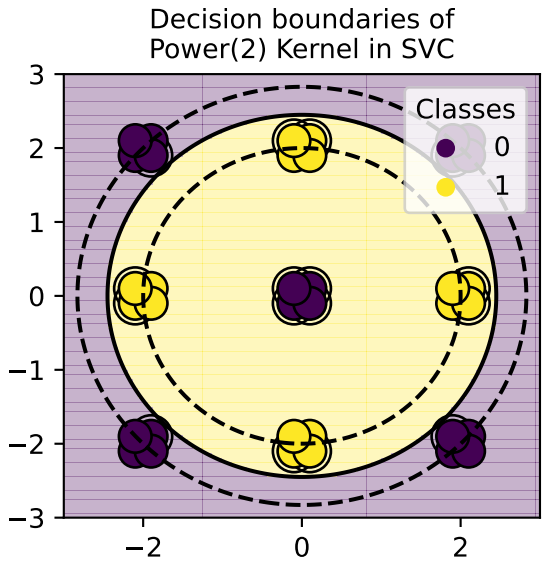
If we expand the dot product, we get:

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

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:

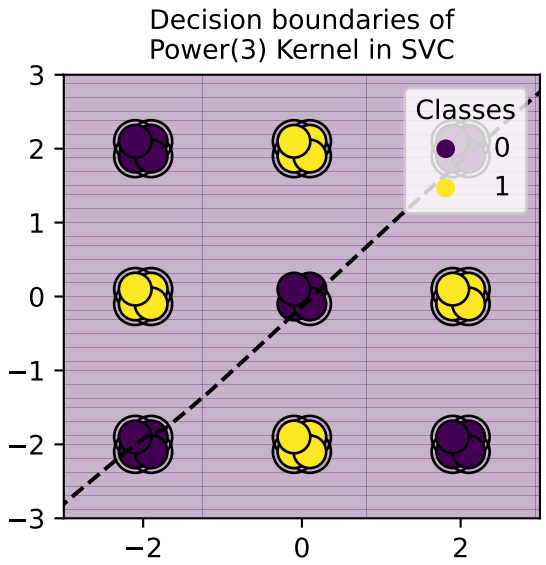
$K_{Power(3)}(\boldsymbol{a}, \boldsymbol{b}) = (\boldsymbol{a} \cdot \boldsymbol{b})^3$ corresponds to the transformation containing all 3-way interaction terms

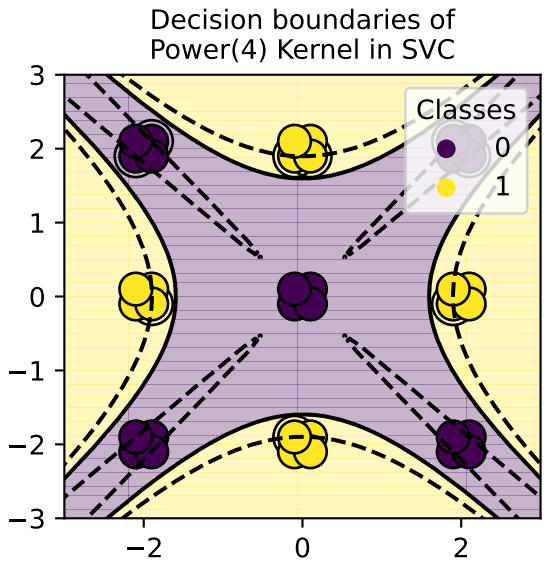
$K_{Power(4)}(\boldsymbol{a}, \boldsymbol{b}) = (\boldsymbol{a} \cdot \boldsymbol{b})^4$ corresponds to the transformation containing all 4-way interaction terms

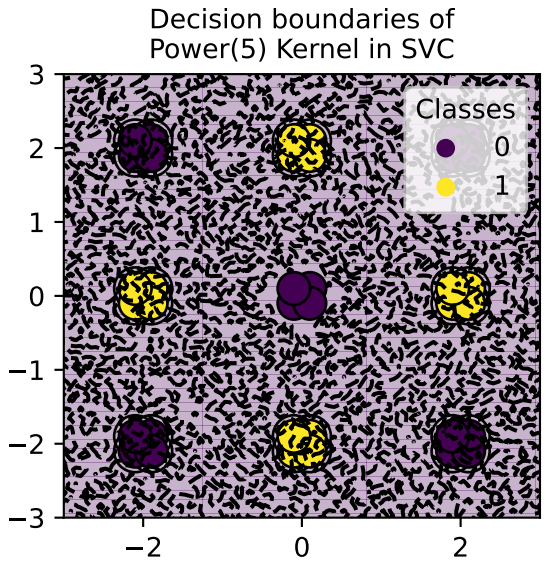
$\vdots$

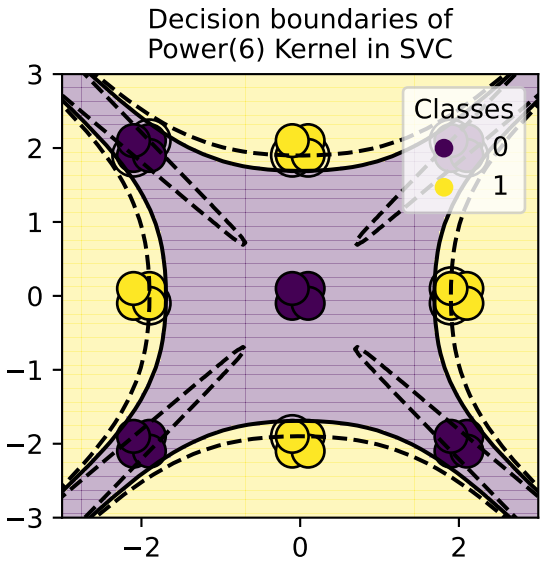
$K_{Power(n)}(\boldsymbol{a}, \boldsymbol{b}) = (\boldsymbol{a} \cdot \boldsymbol{b})^n$ corresponds to the transformation containing all n-way interaction terms

Let's try out several Power Kernels:





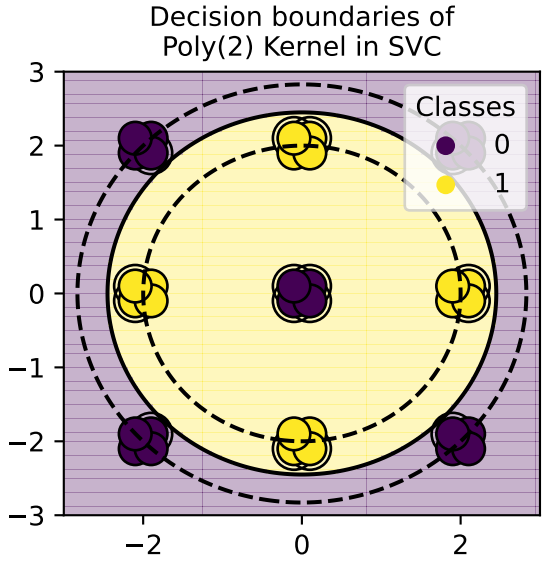




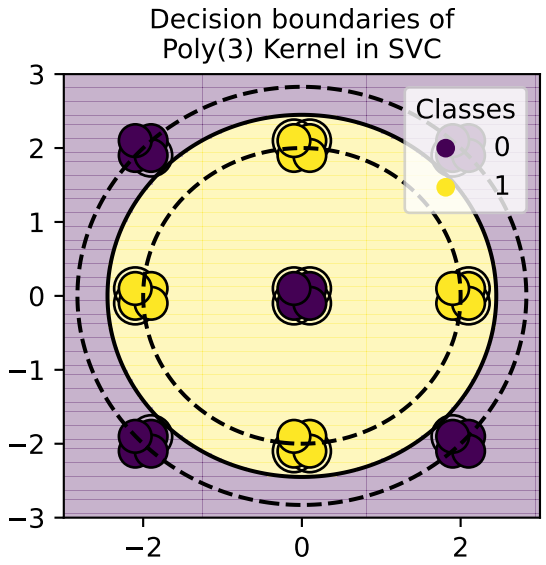
Like the base Kernel, the odd-power Kernels suck at classifying the data, while even power kernels do a fantastic job, specially after 4. This oddity can be mitigated by a principle learned in Applied Statistics - whenever we include high order terms, we also want to include all lower level terms. So, let's inspect the following kernel:

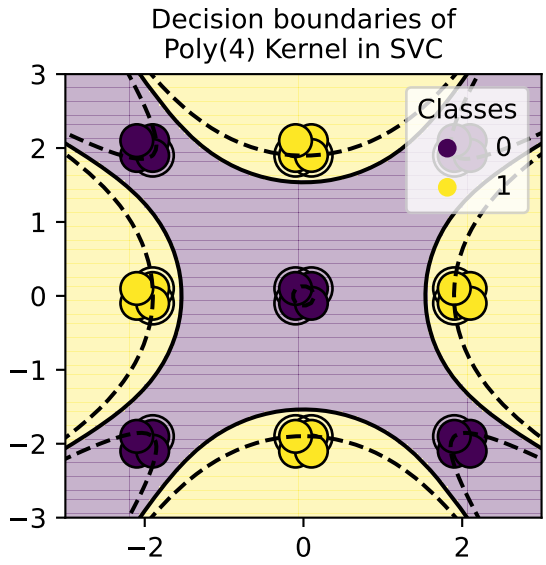
$$K_{Poly(2)}(\boldsymbol{a}, \boldsymbol{b}) = (\boldsymbol{a} \cdot \boldsymbol{b} + 1)^2 = \underset{\text{2-way}}{(\boldsymbol{a} \cdot \boldsymbol{b})^2} + \underset{\text{1-way}}{2(\boldsymbol{a} \cdot \boldsymbol{b})} + \underset{\text{0-way}}{1}$$

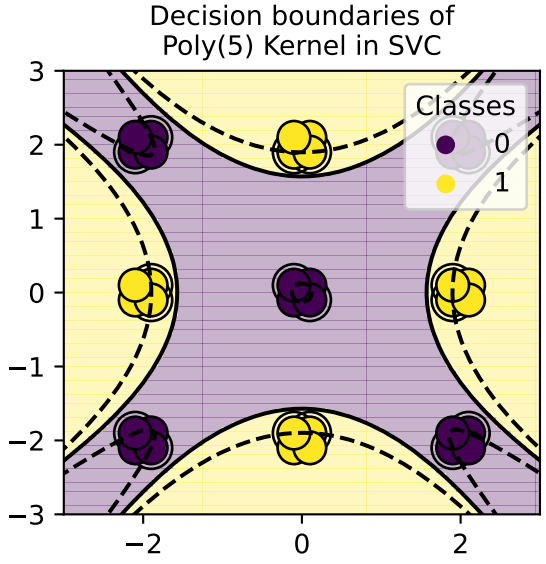
So, $K_{Poly(2)}$ gives us the 2-way interaction terms and all the lower order terms. Likewise, $K_{Poly(n)} = (\boldsymbol{a} \cdot \boldsymbol{b} + 1)^n$ gives us the n-way interaction terms and below, precisely what we wanted. This is also the polynomial kernel for SVM with $\gamma = 1$. After trying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

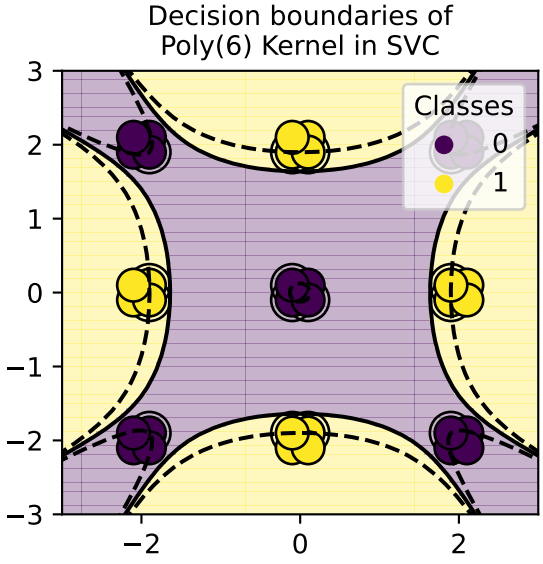


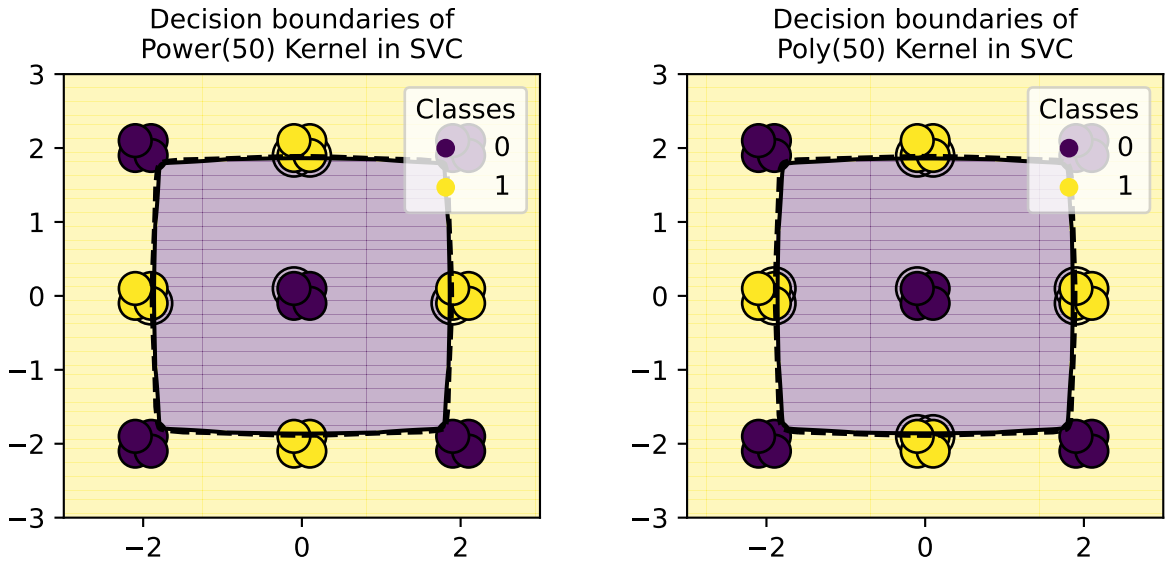
And when we run it with higher degree Kernels, we can observe that the even-degree Kernels look very similar. However, the odd degrees Kernels can in fact classify the data efficiently as they also "include" even-degree terms.


Decision boundaries of
Poly(3) Kernel in SVC
Classes
0
1
3
2
1
0
−1
−2
−3
−2
0
2


Decision boundaries of
Poly(4) Kernel in SVC
Classes
0
1
3
2
1
0
−1
−2
−3
−2
0
2


Decision boundaries of
Poly(5) Kernel in SVC
Classes
0
1
3
2
1
0
−1
−2
−3
−2
0
2


Decision boundaries of
Poly(6) Kernel in SVC
Classes
0
1
3
2
1
0
−1
−2
−3
−2
0
2



It's also important to note that weird boundaries arise when our Kernel degree gets higher. $K_{Power(50)}$ is similar to $K_{Power(2)}$, but inside-out. 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.

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

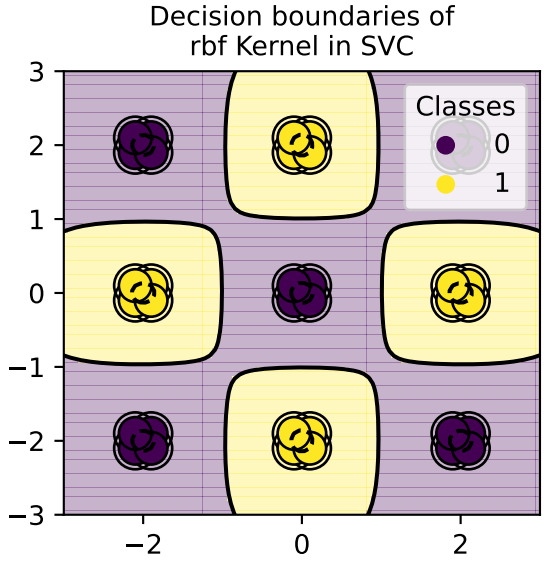
Now, let's raise it to the 2γ to add a control parameter. The control parameter adjusts for the importance of higher order terms. The higher it is, the more relevant they become.

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

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.

Classifying the data

Now we know all about fitting an SVM model with a custom Kernel. However, once the model has been fit, how do we classify our data? We compute the kernel between $\boldsymbol{\beta}$ and an observation, subtract c, and take the sign:

$$\text{if } K(\boldsymbol{\beta}, \boldsymbol{x}_i) - c > 0 \rightarrow y_i = 1$$
$$\text{if } K(\boldsymbol{\beta}, \boldsymbol{x}_i) - c < 0 \rightarrow y_i = -1$$

Radial Basis Function Considerations

Assumptions for Custom Kernel SVM

1.- There are two classes to classify

2.- K, the matrix composed of all pairwise kernels ($k_ij = K(\boldsymbol{x}_i, \boldsymbol{x}_j)$), is positive semi-definite. This is satisfied by the Radial Basis Kernel.

Model Parameters

The RBF model has two tunne-able parameters: cost and gamma.

1.- The cost parameter (ξ) is the cost of missclassifying a data point (Soft Margin case only). A higher ξ will lead to a more complex model that will try to classify all data points correctly. It trades off missclassification of training examples against increasing the margin. Larger values of cost will lead to a smaller margin. In this way ξ behaves as a regularization parameter.

2.- The gamma (γ) parameter controls how far the influence of a single training example reaches. A low γ will consider points far away from the decision boundary in calculations, while a high γ will consider only points close to the decision boundary. The γ parameters can be seen as the inverse of the radius of influence of samples selected by the model as support vectors.

Strengths and Weaknesses

Strengths:

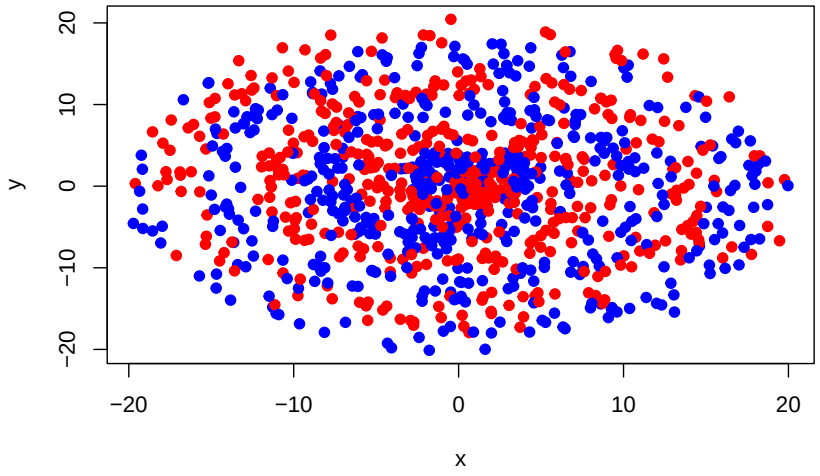
1. RBFs are able to perform optimization over an ∞-dimensional space using a relatively simple formula, making them computationally inexpensive while still having a lot of power.
2. There are no preassumptions that need to be made about the data, making it a great tool when the distribution of data is 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 Spiral Data

To show the power of the Radial Basis Function we will first use a generated data set: it matches a spiral pattern with a bit of overlap (noise) between the two classes. The data was generated with polar coordinates, so we might think about converting to polar coordinates first. However, if this data weren't generated we might not make that connection. This is where the RBF kernel can be very useful.



Fitting the Model

Now lets use the RBF kernel to classify this data. First, let's use a cost of 1 and a gamma of 1 (the default values).

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

# 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"))
```

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

Diagnostics

Confusion Matrix As RBF (and SVM) are classification algorithms, the confusion matrix diagnostics serve as a well tested diagnostic toolkit. Below we have the confusion matrix for our RBF model. The accuracy is 55.1%, which is not much better than random guessing. Note: The data in this project has a split close enough to 50-50. This means that accuracy works fine as an overall metric to gauge the models ability. For all future confusion matrices we will look at just accuracy and kappa.

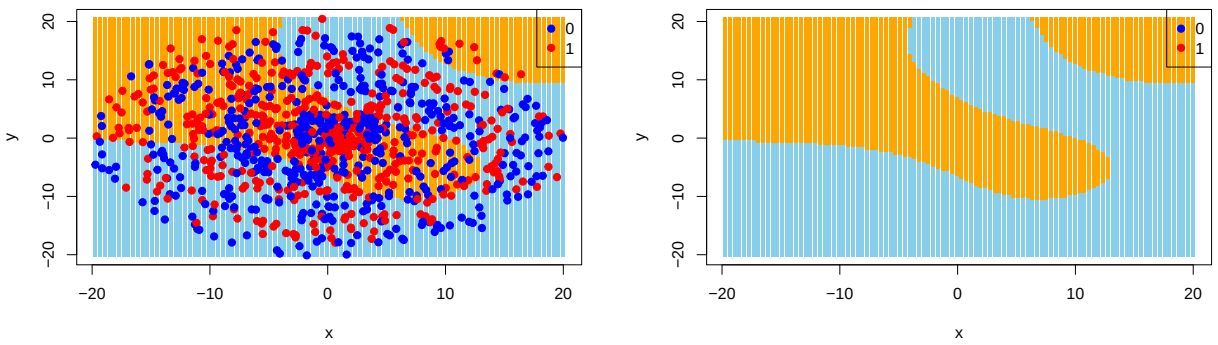
```
# Create a confusion matrix to evaluate the SVM model
confusionMatrix(predict(svmfit, data2), data2$class)
```

```
## 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
##
```

Decision Boundary Another way of assessing the model is to look at the decision boundary. The following code plots the decision boundary for our model (both with and without the points):

```r
# Define colors for data points and decision boundary
point_colors <- c("blue", "red")
boundary_colors <- c("skyblue", "orange")

# 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 the decision boundary with the data points
# Plot the decision boundary
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)

# Plot the decision boundary with the data points
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 observe that the decision boundary doesn't capture the spiraly-ness of the data very well. This is in part since we used the default ξ and γ.

Cross Validating RBF

We can use cross-validation to find the best values for ξ and γ. Fortunately, the `svm` function has built in cross-validation. Setting the `cross` parameter to the number of folds, in this case 5. However, the `svm` function does not *find* the best set of parameters, it only calculates statistics for a given set of parameters. The following function performs the full cross-validation for a RBF model:

```
# 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)
}
```

After running our function and filtering for the best 5 results, we get the following table. We can observe that a cost of 1 and a gamma of 10 are the best values for this data with an average accuracy of 74.3% on the test folds.

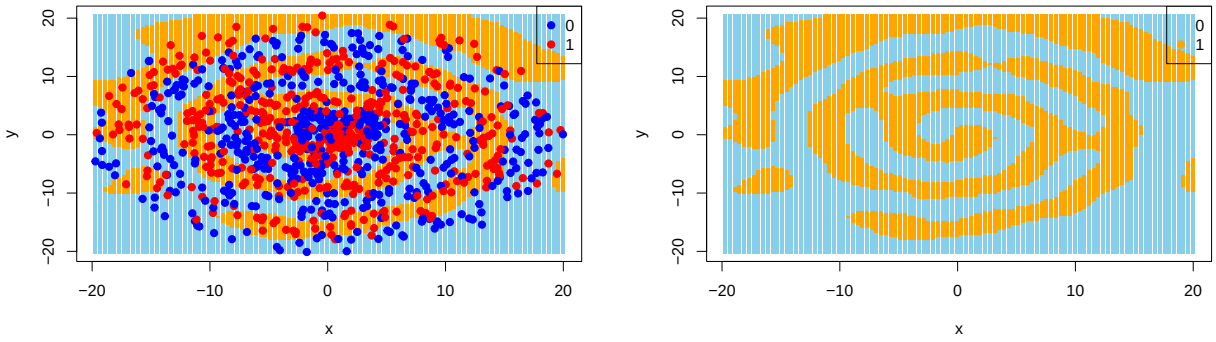
```
# Run our function
validation_data_frame <- cross_validate(5, c(0.1, 1, 10, 100, 1000),
                                        c(0.01, 0.1, 1, 10, 100),
                                        data2, seed = 2024)

# 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
```

Now let's use these values to fit a new model. By analyzing the Confusion Matrix, we can observe that the accuracy rose all the way to 82.1% on the training set. This is a huge improvement!!!

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

The decision boundary also looks way better. It isn't quite perfect, Once again the majority class becomes the background and the minority class generates weird patches. Nonetheless, the fitted decision boundary is much closer to the true decision boundary of the data than before.



Testing Dataset

The final step is to see how this model performs on some test data generated the same way. By observing the Test Confusion Matrix, we get a test accuracy of 75.6% on the test data. This is a lower than the 82.1% training accuracy. However, the test accuracy is higher than the best average cross-validated accuracy (74.3%). This shows that the model is generalizing well to new data.

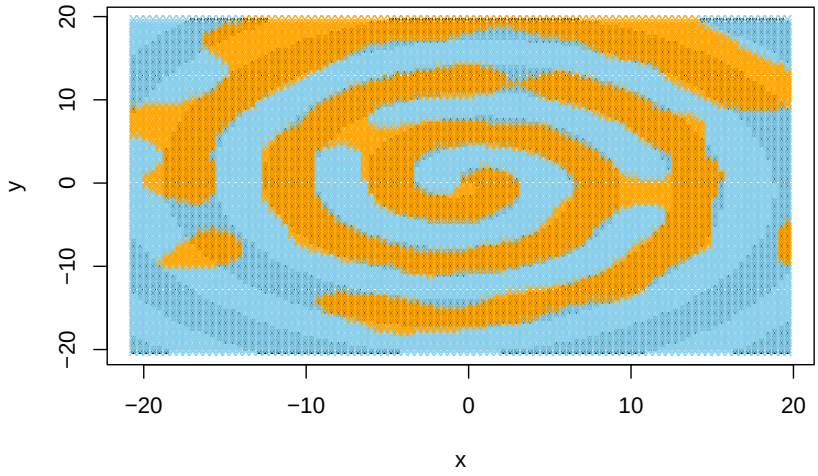
```
##           Reference
## Prediction   0   1
##          0 375 128
##          1 116 381
##  Accuracy     Kappa
## 0.7560000 0.5120527
```

Best Case Scenario

Since this is generated data, we can also classify the data using the true decision boundary (spiral - noise) and compare the results to the ones obtained by our model. From the Confusion Matrix, we can observe that the added noise causes the best possible accuracy to be 80.7%. This implies our model is only 5.2% below the accuracy of the best model we *could* get.

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

We can also overlay the plot of the fit decision boundary (lightblue-orange) and a plot for the true decision boundary (light-dark). By doing so, we can observe that the fit decision boundary estimates the true decision boundary surprisingly well.



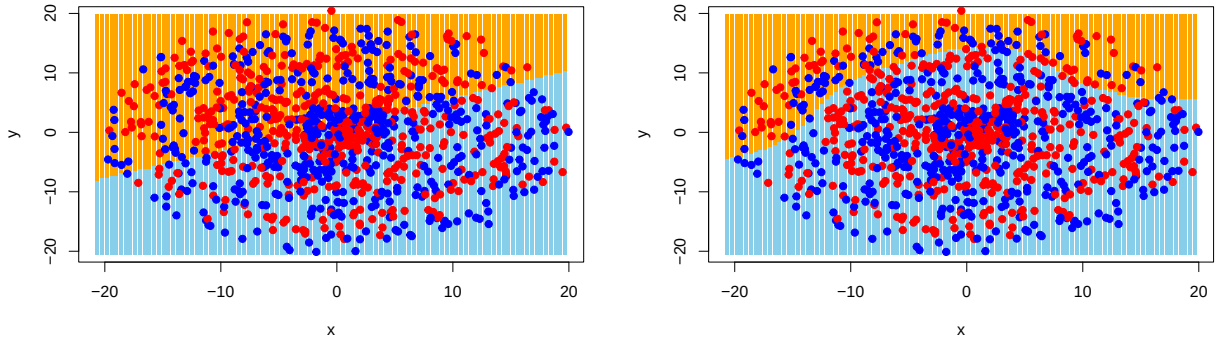
Facing Other Models

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 SVM, a polynomial kernel SVM, a logistic regression model without many interactions. We will also compare it to a KNN model.

Lets start with a linear kernel SVM and a polynomial kernel SVM since those are part of the same family. After cross-validation (for the polynomial kernel the best degree, cost, and gamma are 5, 10 and 0.1) and testing, we obtained each models' confusion Matrix to realize they're marginally better than just randomly guessing. From the confusion matrix, we can observe that the linear kernel model is basically guessing. This is in part since no clear line or simple curve splits the data.

```
## [1] Linear SVM:
##           Reference
## Prediction   0   1
##          0 296 268
##          1 217 219
##   Accuracy      Kappa
## 0.51500000 0.02676106
## [1] Polynomial SVM:
##           Reference
## Prediction   0   1
##          0 436 412
##          1  77  75
##    Accuracy       Kappa
## 0.511000000 0.003975949
```

Once again, the plots show that no single line or simple curve efficiently separates the data.



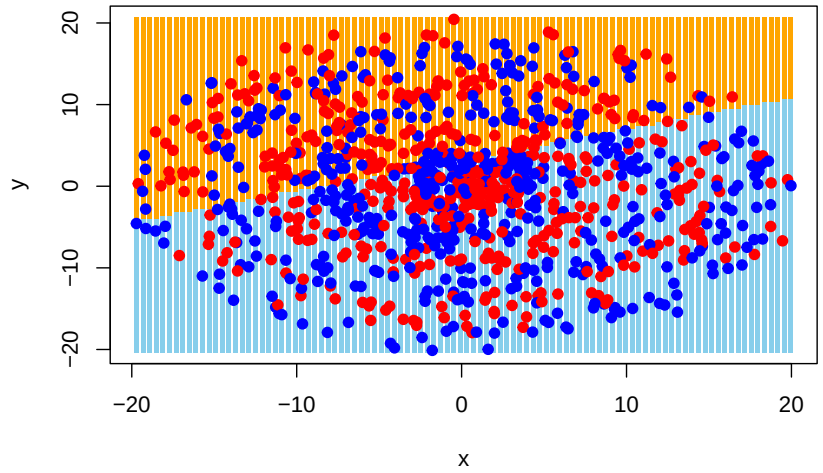
Now let's check how a Logistic Regression model does on this data. After fitting the model, we get the following confusion Matrix. The Logistic Regression has an accuracy of 54.4%, which is better than the previous 2 SVM models, however, it is still not much better than random guessing.

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

If we peek at the predicted probabilities, we can observe the model is not very confident about any decisions.

```
##         1         2         3         4         5         6
## 0.4095280 0.5171398 0.5398165 0.4295237 0.4407871 0.5510954
```

Finally, the plot looks like the linear kernel since we do not have any interaction terms. The decision boundary is a straight line that doesn't capture the spiral shape of the data. Adding interaction terms actually made model accuracy worse.

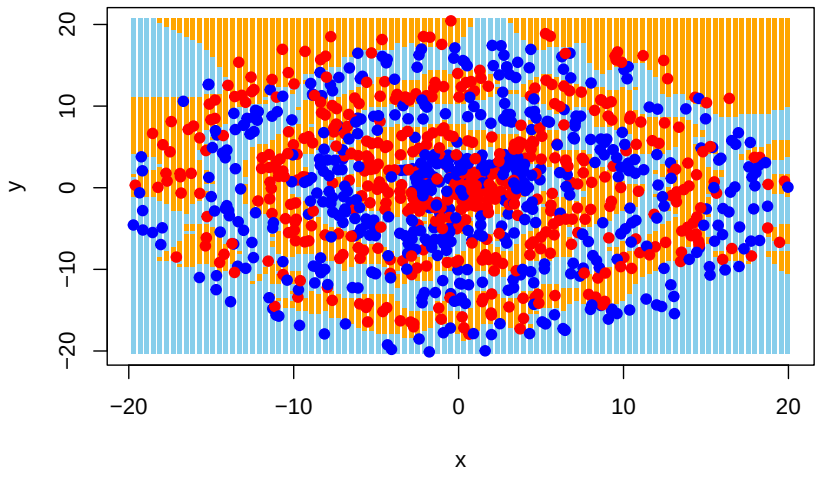


Finally let's check how a KNN model does on this data. We find that the best k value is 3. We can expect the KNN model to perform much better for weird decision boundaries, so let's calculate both confusion matrices: training and testing. We can observe that the training accuracy of 85.7% and a testing accuracy of 70.4%. While the training accuracy is greater when compared to RBF (82.1%), the training accuracy is in fact lower

greater when compared to RBF (75.6%). This is an indicator that the KNN model is significantly overfit and performs worse when predicting new data than RBF in this scenario.

```
## [1] Training KNN Confusion Matrix:
##           Reference
## Prediction   0   1
##          0 445  75
##          1  68 412
##  Accuracy     Kappa
## 0.8570000 0.7137023
## [1] Testing KNN Confusion Matrix:
##           Reference
## Prediction   0   1
##          0 359 142
##          1 132 367
##  Accuracy     Kappa
## 0.7260000 0.4520197
```

While the KNN model does capture the overall spiral shape, it's not as consistant as the RBF model.

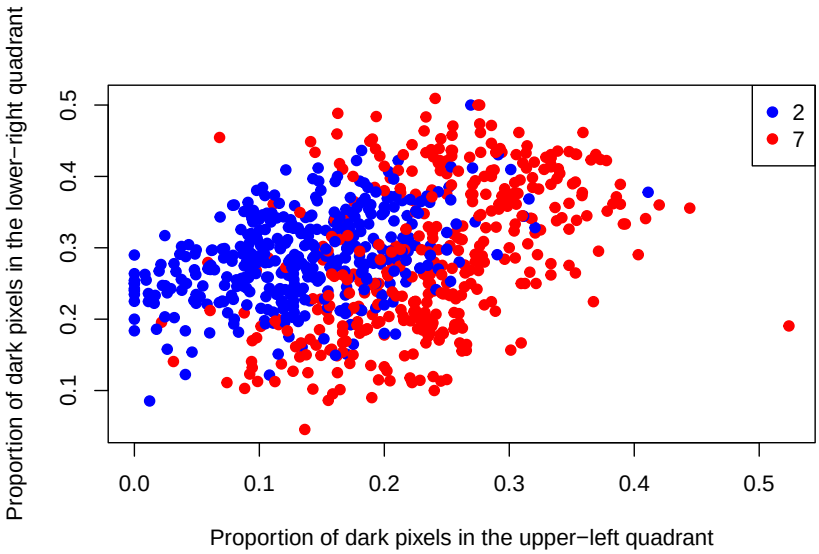


In conclusion, the RBF model was one of the only two models capable of capturing such intricate decision boundaries. And when compared to the other alternative (KNN), RBF was far superior in this scenario.

Real Data Set

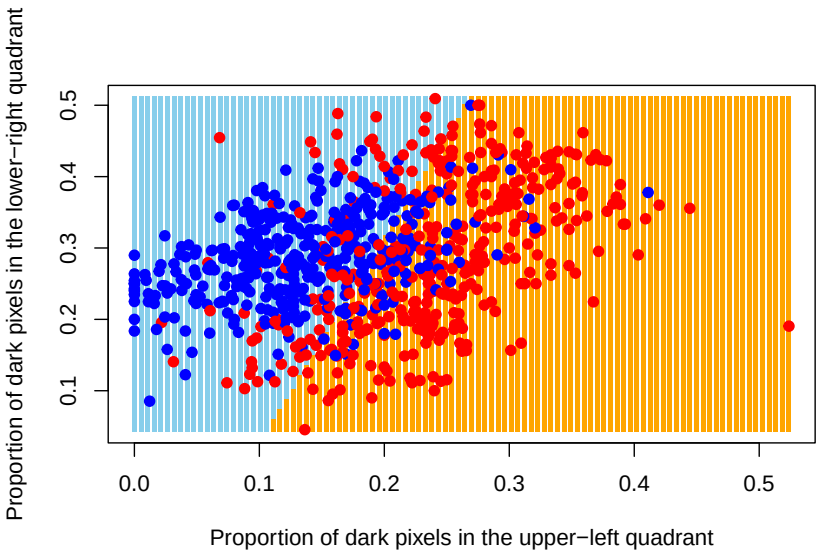
Now that we have demonstrated the power of the RBF kernel on a generated data set, we will now demonstrate the power of the RBF kernel on a real data set. We will use the famous MNIST data set which contains images of handwritten digits. We will only use the 2's and 7's from the data set since they are 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.

For this dataset `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). By plotting the data, we can observe that a single curve should be able to separate the data. More specifically, a parabola facing to the left should enclose the 2's good enough.



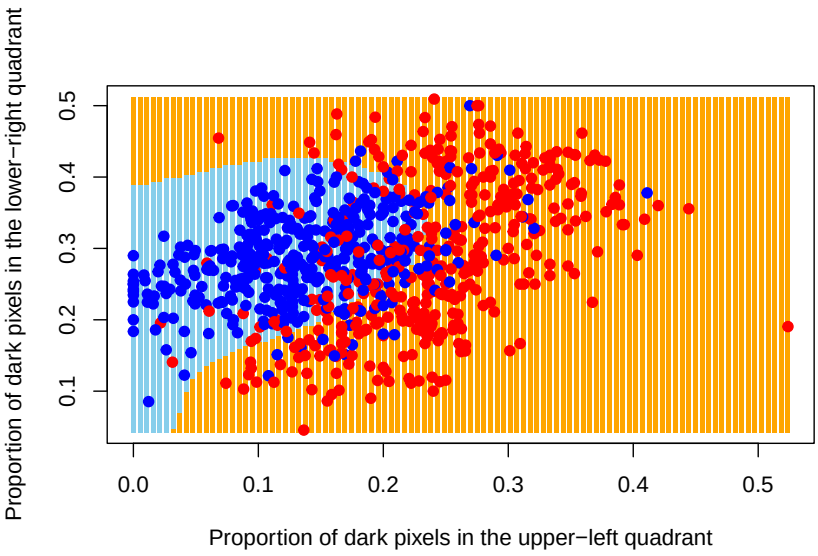
First let's try fitting a linear kernel to the data to get a baseline. From here, all the code will use the training data until we return to see how it does on the test data. From the Confusion Matrix we can observe that our linear kernel does not quite do the job. It's accuracy of 78.88% is not as good as we would like, but it gives us a good baseline to compare the other models to. Now let's try an untunned RBF kernel.

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



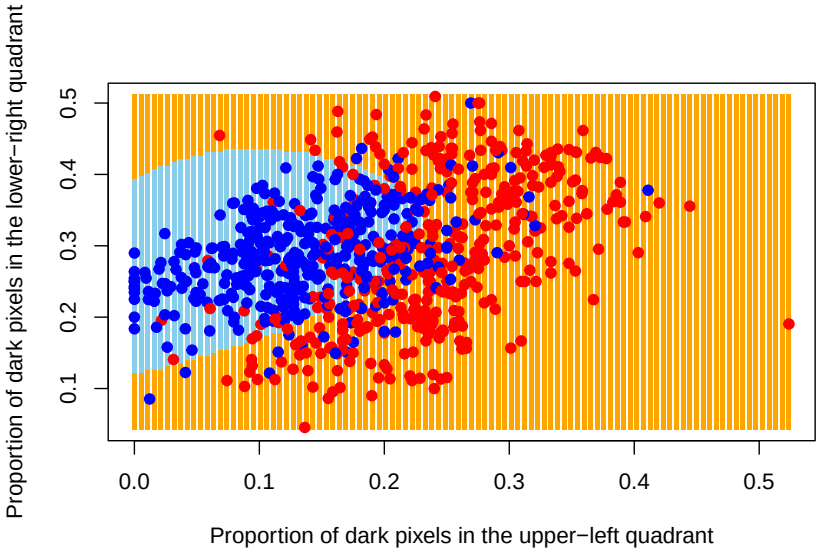
Even with the default values of cost and gamma the RBF kernel seems to do a better job of classifying the data than the linear kernel. The accuracy of 83.38% is better than the linear kernel. The decision boundary also looks better than the linear kernel, seeming to follow the data better. Now let's try to find the best values for cost and gamma using cross-validation.

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



After 10-fold cross-validation, we get that a cost of 0.1 and a gamma of 1 seems to do the best at predicting new unseen data getting an accuracy of 83.375% on the training dataset. From the confusion matrix we observe an accuracy of 83.88%, a marginal 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 2's along the edge causing more points to be classified as 7's overall.

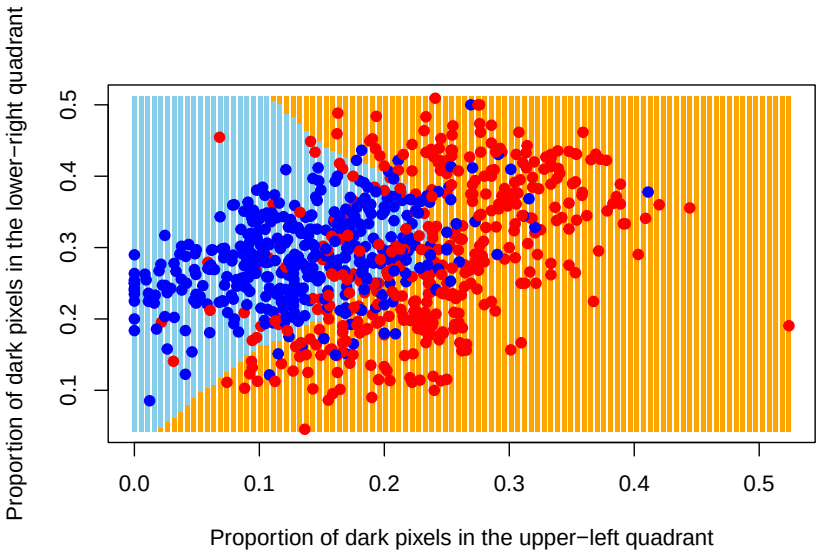
```
##           Reference
## Prediction   2   7
##          2 335  63
##          7  66 336
## Accuracy    Kappa
## 0.838750 0.677504
```



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.

From the cross validation we get that a k value of 83 is the best value for this data set with an accuracy of 83.38% on the training dataset. After training such model, we observe the 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.

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



Finally, lets compare the peformance of our two best models on the test dataset. The RBF testing accuracy is 83.5% (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. Meanwhile, the KNN model got a testing accuracy of 82%. This is also a good testing accuracy. However, the RBF model is clearly less overfit than the KNN model.

```
## [1] Testing Dataset RBF Confusion Matrix:
##           Reference
## Prediction  2  7
##          2 82 19
##          7 14 85
##  Accuracy     Kappa
## 0.8350000 0.6701319
## [1] Testing Dataset KNN Confusion Matrix:
##           Reference
## Prediction  2  7
##          2 83 23
##          7 13 81
##  Accuracy     Kappa
## 0.8200000 0.6408619
```

In conclusion, this analysis has shown 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.