Homework 4 - Classifiers

Charles Liu (304804942)

5/19/2020

Homework 4 Requirements

There is no separate instruction file. This file is the instructions and you will modify it for your submission.

You will submit two files.

The files you submit will be:

1. `102b_hw4_output_First_Last.Rmd` Take this R Markdown file and make the necessary edits so that it generates the requested output.
2. `102b_hw4_output_First_Last.pdf` OR `102b_hw_4_output_First_Last.html` Your output file. This can be a PDF or an HTML file. This is the primary file that will be graded. **Make sure all requested output is visible in the output file.**

Failure to submit all files will result in an automatic 40 point penalty.

Academic Integrity

Modifying the following statement with your name.

"By including this statement, I, Charles Liu, declare that all of the work in this assignment is my own original work. At no time did I look at the code of other students nor did I search for code solutions online. I understand that plagiarism on any single part of this assignment will result in a 0 for the entire assignment and that I will be referred to the dean of students."

If you collaborated verbally with other students, please also include the following line to credit them.

"At no point did I show another student my code, nor did I look at another student's code."

Part 1 Naive Bayes Classifier for Iris data

Task: Write a function that performs Naive Bayes classification for the iris data. The function will output probabiity estimates of the species for a test case.

The function will accept three inputs: a row matrix for the x values of the test case, a matrix of x values for the training data, and a vector of class labels for the training data.

The function will create the probability estimates based on the training data it has been provided.

Within the function use a Gaussian model and estimate the mean and standard deviation of the Gaussian populations based on the training data provided. (Hint: You have 24 parameters to estimate: the mean and standard deviation of each of the 4 variables for each of the three species. With the naive assumption, you do not have to estimate any covariances.)

```r
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
iris_nb <- function(testx, trainx, trainy){
  a <- table(trainy)[1]
  b <- table(trainy)[2]
  c <- table(trainy)[3]

  x1 <- trainx[1:a, ]
  x2 <- trainx[(a+1):(a+b), ]
  x3 <- trainx[(a+b+1):(a+b+c), ]

  # we will use dnorm(...)
  # Construct the Likelihoods (aka "lh_...")
  lh_A1 <- dnorm(testx[ ,1], mean(x1[ ,1]), sd(x1[ ,1]))
  lh_A2 <- dnorm(testx[ ,2], mean(x1[ ,2]), sd(x1[ ,2]))
  lh_A3 <- dnorm(testx[ ,3], mean(x1[ ,3]), sd(x1[ ,3]))
  lh_A4 <- dnorm(testx[ ,4], mean(x1[ ,4]), sd(x1[ ,4]))
  lh_A_all <- (lh_A1 * lh_A2 * lh_A3 * lh_A4)

  lh_B1 <- dnorm(testx[ ,1], mean(x2[ ,1]), sd(x2[ ,1]))
  lh_B2 <- dnorm(testx[ ,2], mean(x2[ ,2]), sd(x2[ ,2]))
  lh_B3 <- dnorm(testx[ ,3], mean(x2[ ,3]), sd(x2[ ,3]))
  lh_B4 <- dnorm(testx[ ,4], mean(x2[ ,4]), sd(x2[ ,4]))
  lh_B_all <- (lh_B1 * lh_B2 * lh_B3 * lh_B4)

  lh_C1 <- dnorm(testx[ ,1], mean(x3[ ,1]), sd(x3[ ,1]))
  lh_C2 <- dnorm(testx[ ,2], mean(x3[ ,2]), sd(x3[ ,2]))
  lh_C3 <- dnorm(testx[ ,3], mean(x3[ ,3]), sd(x3[ ,3]))
  lh_C4 <- dnorm(testx[ ,4], mean(x3[ ,4]), sd(x3[ ,4]))
  lh_C_all <- (lh_C1 * lh_C2 * lh_C3 * lh_C4)

  # Construct the Priors (aka "prior_...")
  prior_1 <- (a) / (a+b+c)
  prior_2 <- (b) / (a+b+c)
  prior_3 <- (c) / (a+b+c)

  # Construct the Marginal (aka "marginal")
  marginal <- (lh_A_all * prior_1) + (lh_B_all * prior_2) + (lh_C_all * prior_3)

  # Construct the Posteriors (aka "posterior_...")
  posterior_1 <- (lh_A_all * prior_1) / (marginal)
  posterior_2 <- (lh_B_all * prior_2) / (marginal)
  posterior_3 <- (lh_C_all * prior_3) / (marginal)

  # put it all in a dataframe for ease of reading
  results <- data.frame(setosa = posterior_1, versicolor = posterior_2,
                        virginica = posterior_3)
  rownames <- NULL
  return(results)
}
```

```
### output should be a named vector that looks something like this:
## [these numbers are completely made up btw]
    setosa versicolor  virginica
 0.9518386  0.0255936  0.0225678
```

```
set.seed(1)
training_rows <- sort(c(sample(1:50, 40), sample(51:100, 40), sample(101:150, 40)))
training_x <- as.matrix(iris[training_rows, 1:4])
training_y <- iris[training_rows, 5]

# test cses
test_case_a <- as.matrix(iris[24, 1:4]) # true class setosa
test_case_b <- as.matrix(iris[73, 1:4]) # true class versicolor
test_case_c <- as.matrix(iris[124, 1:4]) # true class virginica

# class predictions of test cases
iris_nb(test_case_a, training_x, training_y)
```

Testing it out

```
##        setosa   versicolor    virginica
## setosa      1 1.029887e-13 4.098385e-18
```

```
iris_nb(test_case_b, training_x, training_y)
```

```
##               setosa versicolor  virginica
## setosa 2.980587e-115  0.9034742 0.09652578
```

```
iris_nb(test_case_c, training_x, training_y)
```

```
##               setosa versicolor virginica
## setosa 6.078393e-136 0.09540725 0.9045928
```

```
# should work and produce slightly different estimates based on new training data
set.seed(10)
training_rows2 <- sort(c(sample(1:50, 25), sample(51:100, 25), sample(101:150, 25)))
```

```r
training_x2 <- as.matrix(iris[training_rows2, 1:4])
training_y2 <- iris[training_rows2, 5]

iris_nb(test_case_a, training_x2, training_y2)
```

```
##        setosa  versicolor    virginica
## setosa      1 2.60018e-10 4.610354e-17
```

```r
iris_nb(test_case_b, training_x2, training_y2)
```

```
##               setosa versicolor  virginica
## setosa 1.735964e-131  0.9195738 0.08042615
```

```r
iris_nb(test_case_c, training_x2, training_y2)
```

```
##               setosa versicolor virginica
## setosa 3.400709e-149  0.1887116 0.8112884
```

Naive Bayes with R

While instructive and education (I hope) to write your own NaiveBayes function, in practical settings, I recommend using the production ready code from some time-tested packages.

I've included some code for using the `naiveBayes()` function that is part of the `e1071` package. No need to modify anything. The results prediced by `naiveBayes()` should match the results from the function you wrote.

```r
# code provided. no need to edit. These results should match your results above.
library(e1071)
```

```
## Warning: package 'e1071' was built under R version 3.6.3
```

```r
nb_model1 <- naiveBayes(training_x, training_y)
predict(nb_model1, newdata = test_case_a, type = 'raw')
```

```
##      setosa   versicolor    virginica
## [1,]      1 1.029887e-13 4.098385e-18
```

```r
predict(nb_model1, newdata = test_case_b, type = 'raw')
```

```
##             setosa versicolor  virginica
## [1,] 2.980587e-115  0.9034742 0.09652578
```

```r
predict(nb_model1, newdata = test_case_c, type = 'raw')
```

```
##             setosa versicolor virginica
## [1,] 6.078393e-136 0.09540725 0.9045928
```

```r
nb_model2 <- naiveBayes(training_x2, training_y2)
predict(nb_model2, newdata = test_case_a, type = 'raw')
```

```
##      setosa  versicolor    virginica
## [1,]      1 2.60018e-10 4.610354e-17
```

```r
predict(nb_model2, newdata = test_case_b, type = 'raw')
```

```
##             setosa versicolor  virginica
## [1,] 1.735964e-131  0.9195738 0.08042615
```

```r
predict(nb_model2, newdata = test_case_c, type = 'raw')
```

```
##             setosa versicolor virginica
## [1,] 3.400709e-149  0.1887116 0.8112884
```

Part 2: K-nearest neighbors Classifier for the Iris data

Task: Write a classifier using the K-nearest neighbors algorithm for the iris data set.

First write a function that will calculate the euclidean distance from a vector A (in 4-dimensional space) to another vector B (also in 4-dimensional space).

Use that function to find the k nearest neighbors to then make a classification.

The function will accept four inputs: a row matrix for the x values of the test case, a matrix of x values for the training data, a vector of class labels for the training data, and the k parameter.

The function will return a single label.

```
# Euclidean Distance
distance <- function(a, b){
  sum_diff <- sum((a-b)^2)
  Euclid_results <- sqrt(sum_diff)
  return(Euclid_results)
}

# KNN
iris_knn <- function(testx, trainx, trainy, k){
  dist <- rep(NA, dim(trainx)[1])
  for(i in 1:dim(trainx)[1]){
    dist[i] <- distance(testx, trainx[i, ])
  }
  knn_index = order(dist)[1:(k+1)]
  knn = table(trainy[knn_index])
  max_knn_names <- names(knn)[which.max(knn)]
  return(max_knn_names)
}
```

```
iris_knn(test_case_a, training_x, training_y, 5)
```

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

```
iris_knn(test_case_b, training_x, training_y, 5) # will incorrectly label as virginica with this traini
```

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

```
iris_knn(test_case_c, training_x, training_y, 5)
```

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

```
iris_knn(test_case_a, training_x2, training_y2, 5)
```

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

```
iris_knn(test_case_b, training_x2, training_y2, 5)
```

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

```
iris_knn(test_case_c, training_x2, training_y2, 5) # will incorrectly label as versicolor with this tra
```

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

KNN with R

Again, if you plan on using KNN in real-life, use a function from a package.

I've included some code for using the `knn()` function that is part of the `class` package. No need to modify anything. The results prediced by `knn()` should match the results from the function you wrote, including the misclassification of some of the test cases based on the training data.

```
library(class)
knn(train = training_x, cl = training_y, test = test_case_a, k = 5)
```

```
## [1] setosa
## Levels: setosa versicolor virginica
```

```
knn(train = training_x, cl = training_y, test = test_case_b, k = 5) # will incorrectly label as virgini
```

```
## [1] virginica
## Levels: setosa versicolor virginica
```

```
knn(train = training_x, cl = training_y, test = test_case_c, k = 5)
```

```
## [1] virginica
## Levels: setosa versicolor virginica
```

```
knn(train = training_x2, cl = training_y2, test = test_case_a, k = 5)
```

```
## [1] setosa
## Levels: setosa versicolor virginica
```

```
knn(train = training_x2, cl = training_y2, test = test_case_b, k = 5)
```

```
## [1] versicolor
## Levels: setosa versicolor virginica
```

```
knn(train = training_x2, cl = training_y2, test = test_case_c, k = 5) # will incorrectly label as versi
```

```
## [1] virginica
## Levels: setosa versicolor virginica
```

Part 3: SVM

Manual implementation of SVM is a bit of a pain (quadratic programming is hard), and I will not include it in the hw.

For the interested student, I refer them to the code from book's companion github repository: https://github.com/sdrogers/fcmlcode/blob/master/R/chapter5/svmhard.R and this post on stackexchange: https://stats.stackexchange.com/questions/179900/optimizing-a-support-vector-machine-with-quadratic-programming

Instead, I will use an example of a mixture model that can be separated via SVM.

The mixture model comes from the excellent (but advanced) textbook, *The Elements of Statistical Learning*, which is made to be freely available by the authors at: https://web.stanford.edu/~hastie/ElemStatLearn/

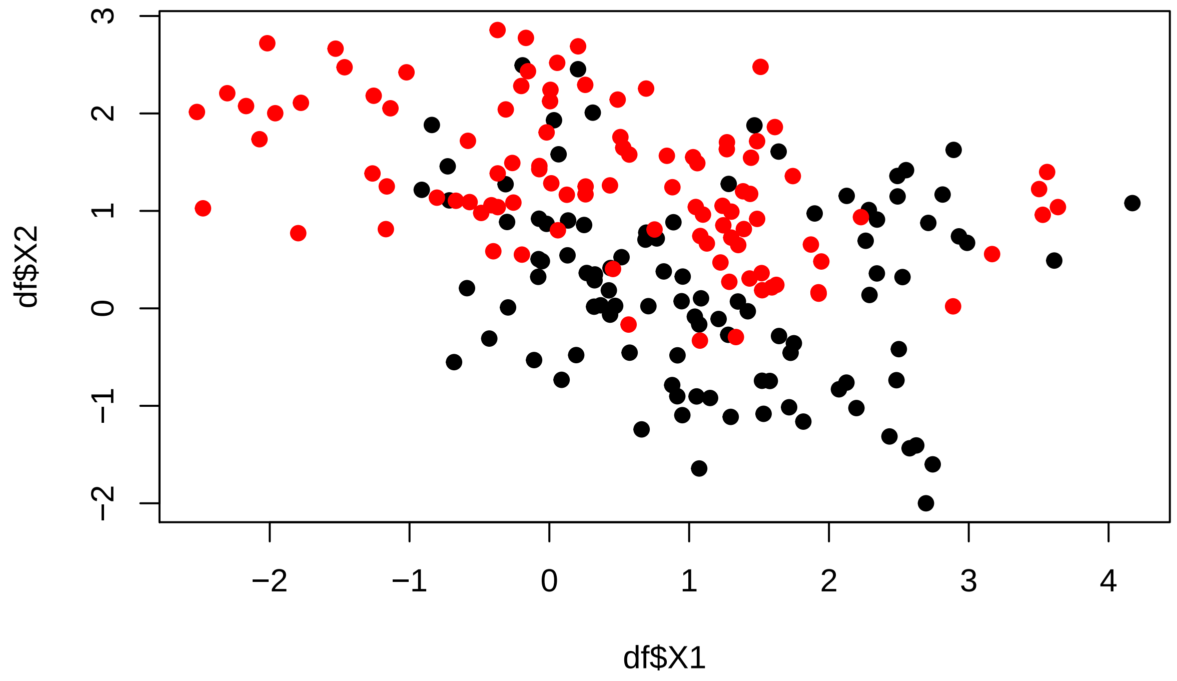
```
library(devtools) # needed to install the "ElemStatLearn" package
```

```
## Warning: package 'devtools' was built under R version 3.6.3
```

```
## Loading required package: usethis
```

```
## Warning: package 'usethis' was built under R version 3.6.3
```

```r
library(ElemStatLearn)
data(mixture.example)
df <- data.frame(mixture.example$x, y = as.factor(mixture.example$y)) # turn the data into a dataframe
plot(df$X1,df$X2, col = df$y, pch = 19) # create a plot of the mixture
```



We will use the `svm()` function available in package `e1071`.

Read the documentation on the function `svm()`.

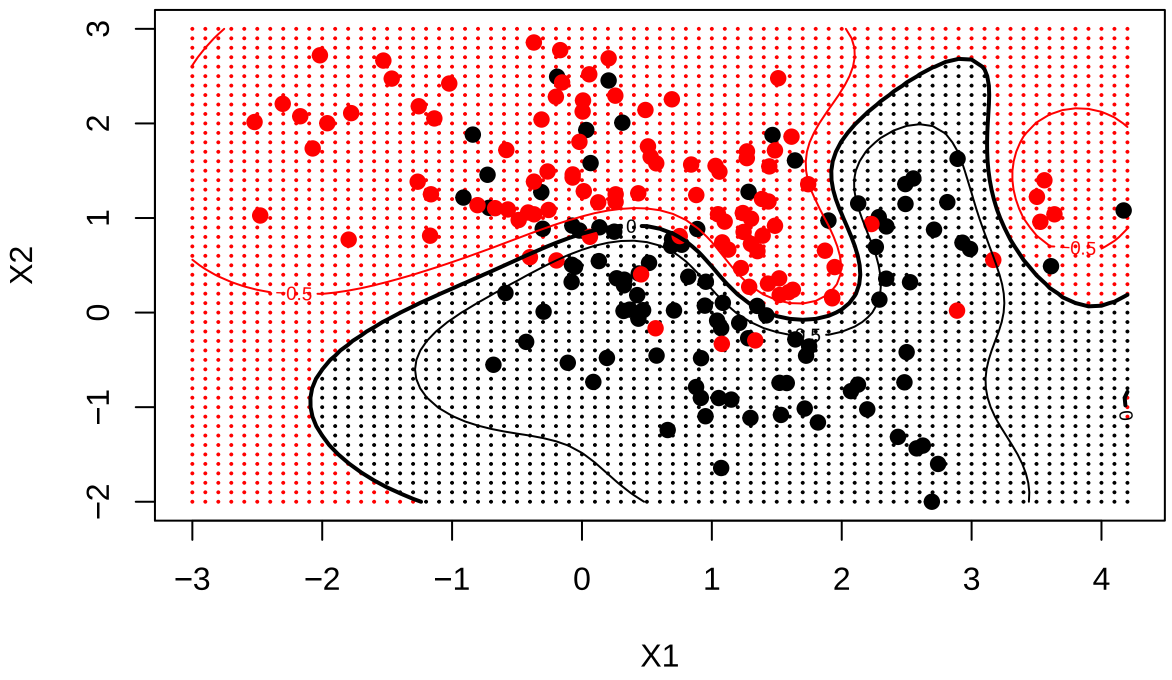
For the following models, we will use a **radial-basis function**, which is equivalent to using a Gaussian Kernel function. (The Gaussian Kernel function projects the 2-dimensional data into infinite dimensional space and takes the inner product of these infinite dimensional vectors. It doesn't actually do this, but the resulting inner product can be found and used to draw a decision boundary.)

The svm function allows for multiple arguments, but we will focus on the effect of the arguments for gamma and cost.

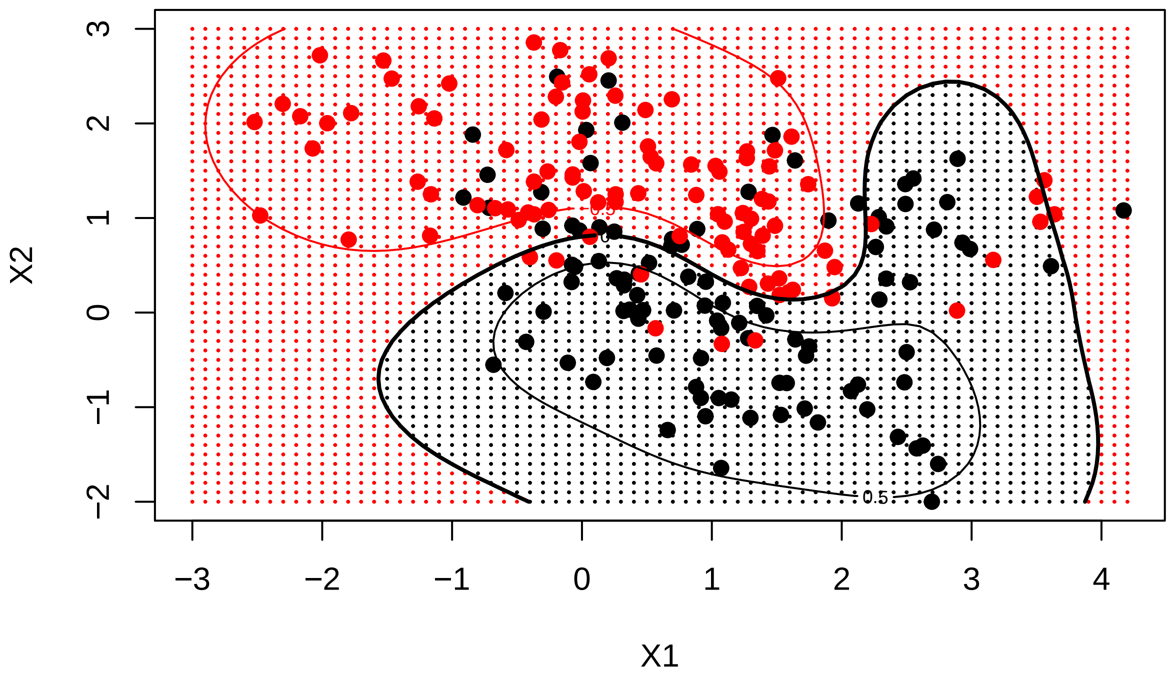
I have created 9 classification models using SVM and different values of gamma and cost.

Pay attention to the values of `gamma` and `cost`. At the very end comment on the effect of each parameter on the resulting model.

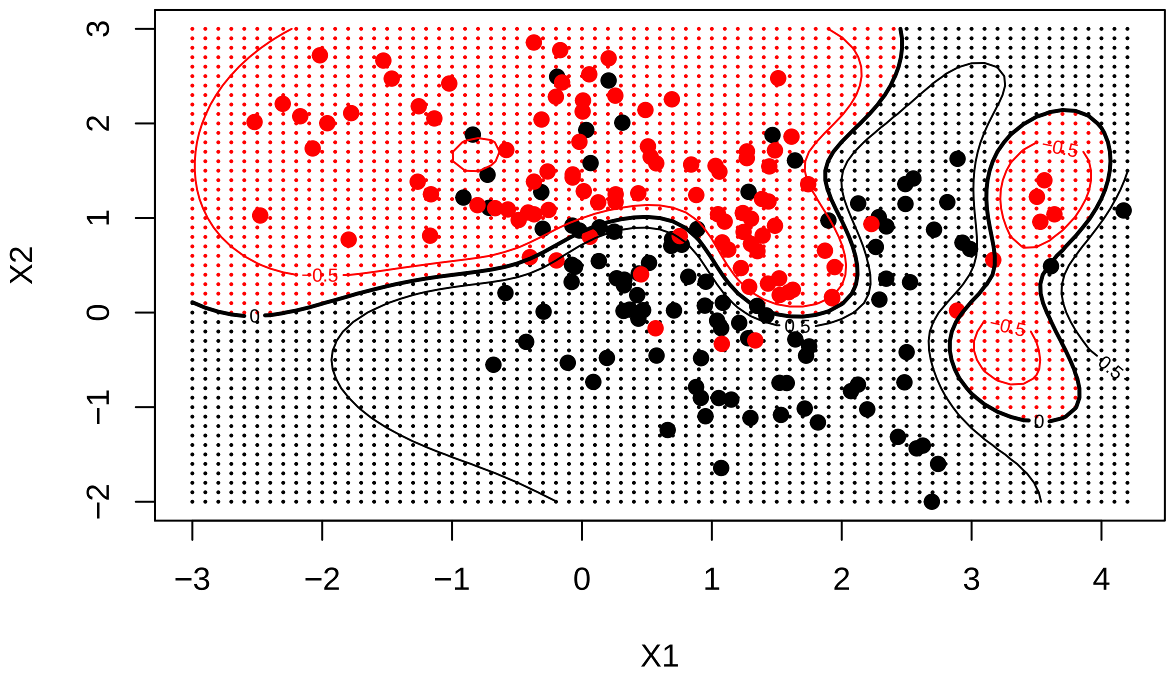
```r
library(e1071)
model <- svm(y ~ . , data = df, scale = FALSE, kernel = "radial", gamma = 1, cost = 1)
```



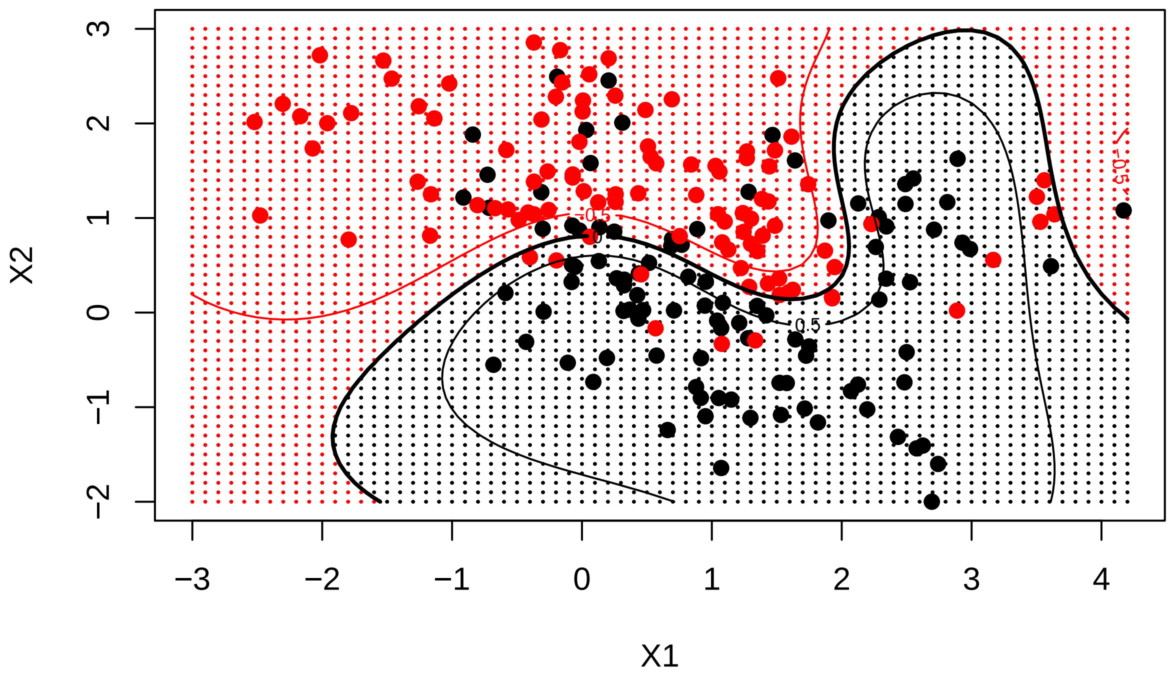
```r
model <- svm(y ~ . , data = df, scale = FALSE, kernel = "radial", gamma = 1, cost = 0.1)
```



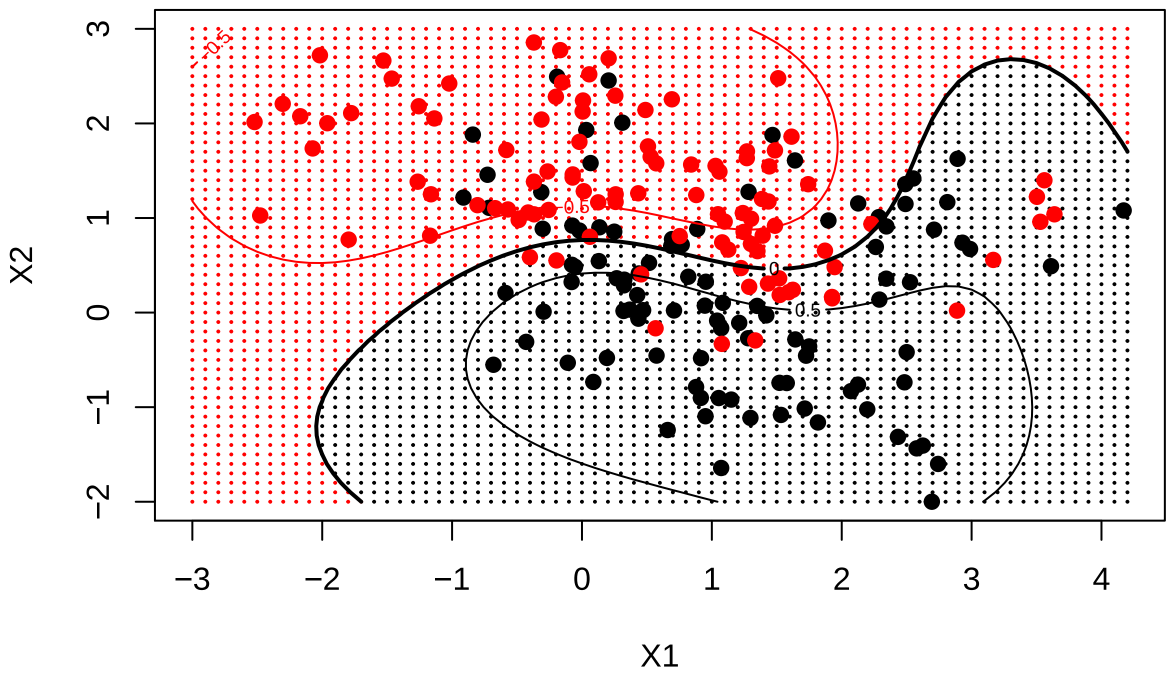
```
model <- svm(y ~ . , data = df, scale = FALSE, kernel = "radial", gamma = 1, cost = 10)
```



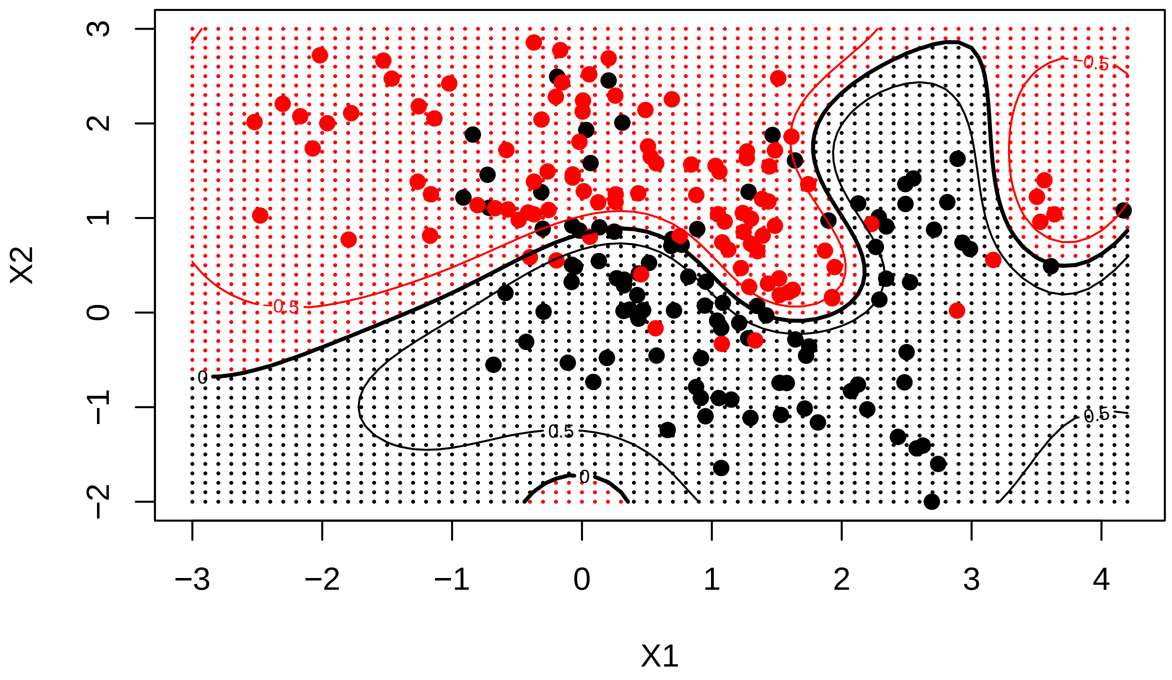
```
model <- svm(y ~ . , data = df, scale = FALSE, kernel = "radial", gamma = 0.5, cost = 1)
```



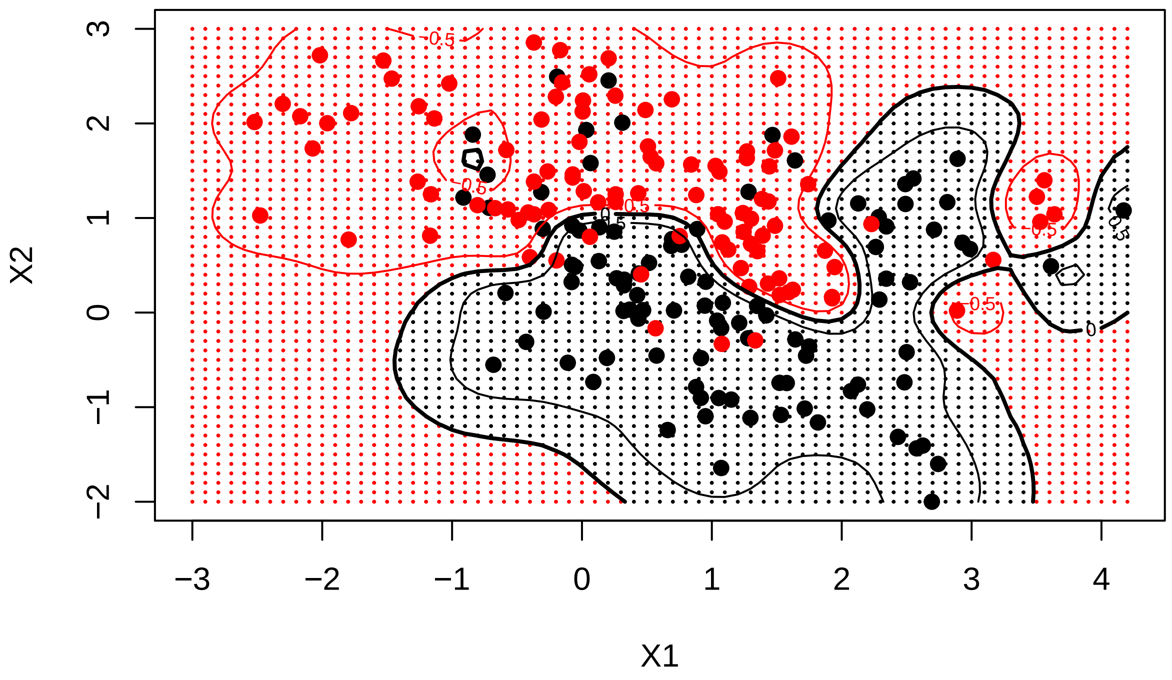
```
model <- svm(y ~ . , data = df, scale = FALSE, kernel = "radial", gamma = 0.5, cost = 0.10)
```



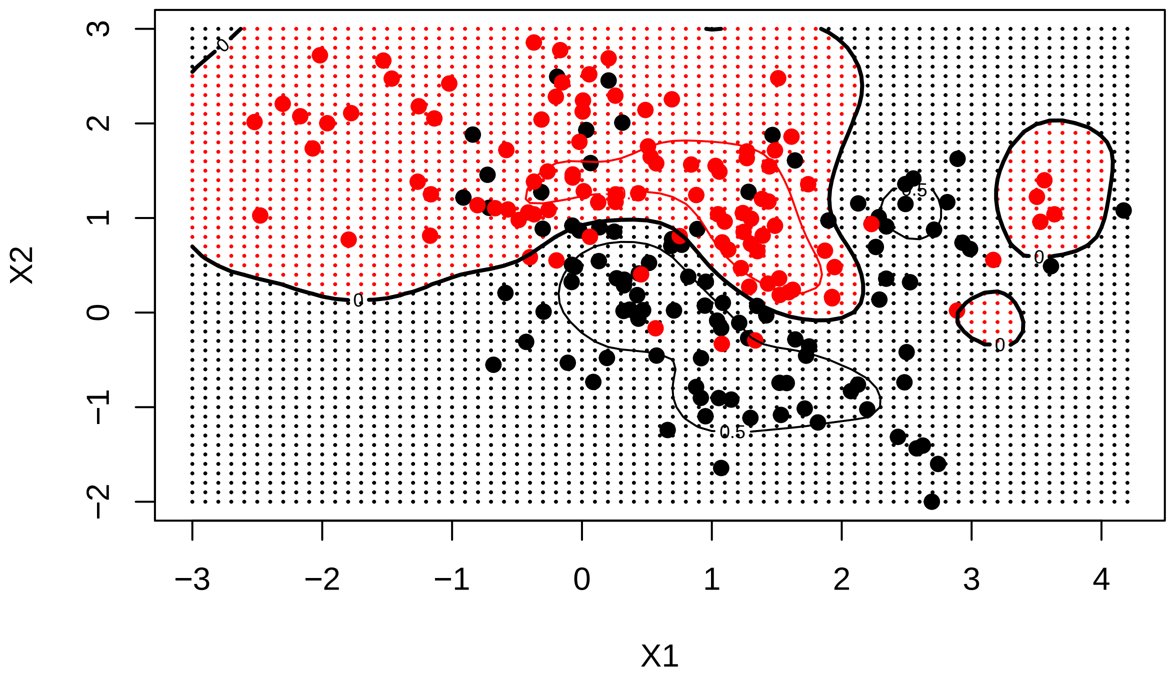
```r
model <- svm(y ~ . , data = df, scale = FALSE, kernel = "radial", gamma = 0.5, cost = 10)
```



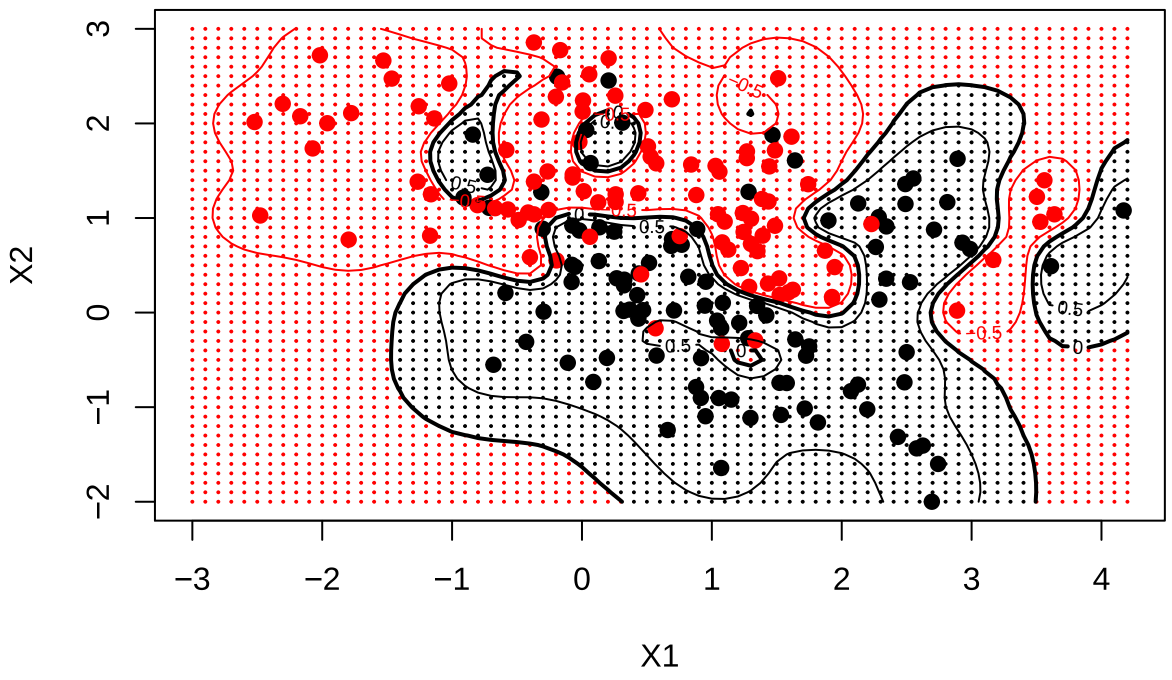
```
model <- svm(y ~ . , data = df, scale = FALSE, kernel = "radial", gamma = 5, cost = 1)
```



```r
model <- svm(y ~ . , data = df, scale = FALSE, kernel = "radial", gamma = 5, cost = 0.1)
```



```r
model <- svm(y ~ . , data = df, scale = FALSE, kernel = "radial", gamma = 5, cost = 10)
```



Write about the effect of the cost paramter: **Answer:** A large Cost (C) gives you low bias and high variance, while a small Cost (C) gives you a higher bias and low variance. The Cost (C) measures the missclassification of the training dataset.

Write about the effect of the gamma parameter: **Answer:** A small Gamma will give you low bias and high variance, while a large Gamma will give you higher bias and low variance. Gamma defines how far the influence of a single training value reaches point(s), with low values meaning ‘far’ and high values meaning ‘close’.