Nonparametric Statistics Final Assignment

Danyu Zhang & Daniel Alonso

March 30, 2021

Exercises

Category A: Problem 6

- **Exercise 5.11**. The *challenger.txt* dataset contains dataset contains information regarding the state of the solidrocket boosters after launch for 23 shuttle flights prior the Challenger launch. Each row has, among others, the variables *fail.field* (indicator of whether there was an incident with the O-rings), *nfail.field* (number of incidents with the O-rings), and *temp* (temperature in the day of launch,measured in degrees Celsius).

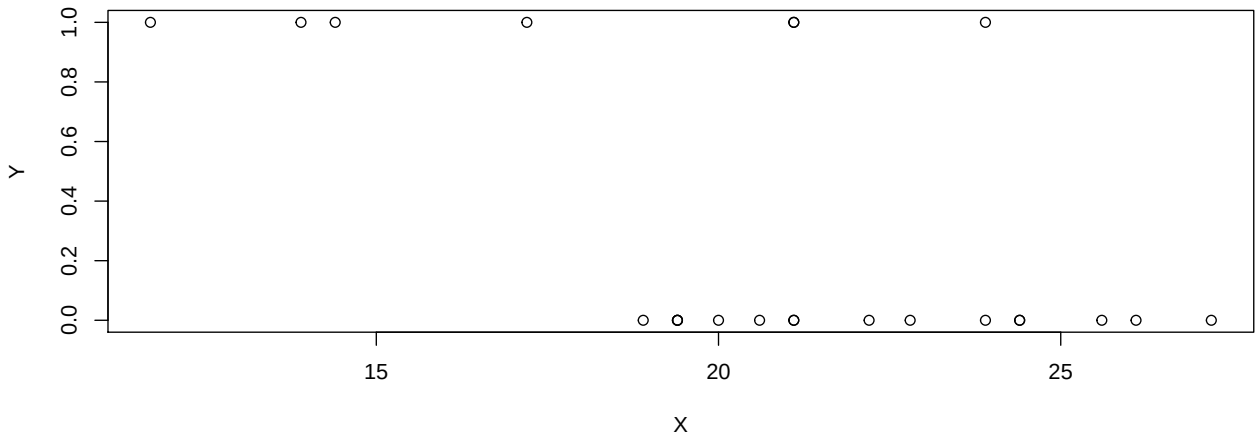
a) Fit a local logistic regression (first degree) for *fails.field* ~ *temp*, for three choices of bandwidths: one that oversmooths, another that is somehow adequate, and another that undersmooths. Do the effects of *temp* on *fails.field* seem to be significant?

We first read the challenger data:

```
challenger <- read.table('https://raw.githubusercontent.com/egarpor/handy/master/datasets/challenger.txt', header = TRUE)
```

We can roughly observe that the lower the temperature, the higher the probability of having a incident with the O-rings as most of the observations that where temperatures are below 20 have had an incident; while the majority of the observations with temperatures higher than 20 have not had any incidents.

```
X = challenger$temp # Assign temperature column to X
Y = challenger$fail.field # Assign fail.field column to X
plot(X, Y) # plotting a scatterplot of temp vs fail.field
```



First of all, we will define a logistic function which transforms β_0 into probability:

```
logistic <- function(x) 1 / (1 + exp(-x)) # logistic function
```

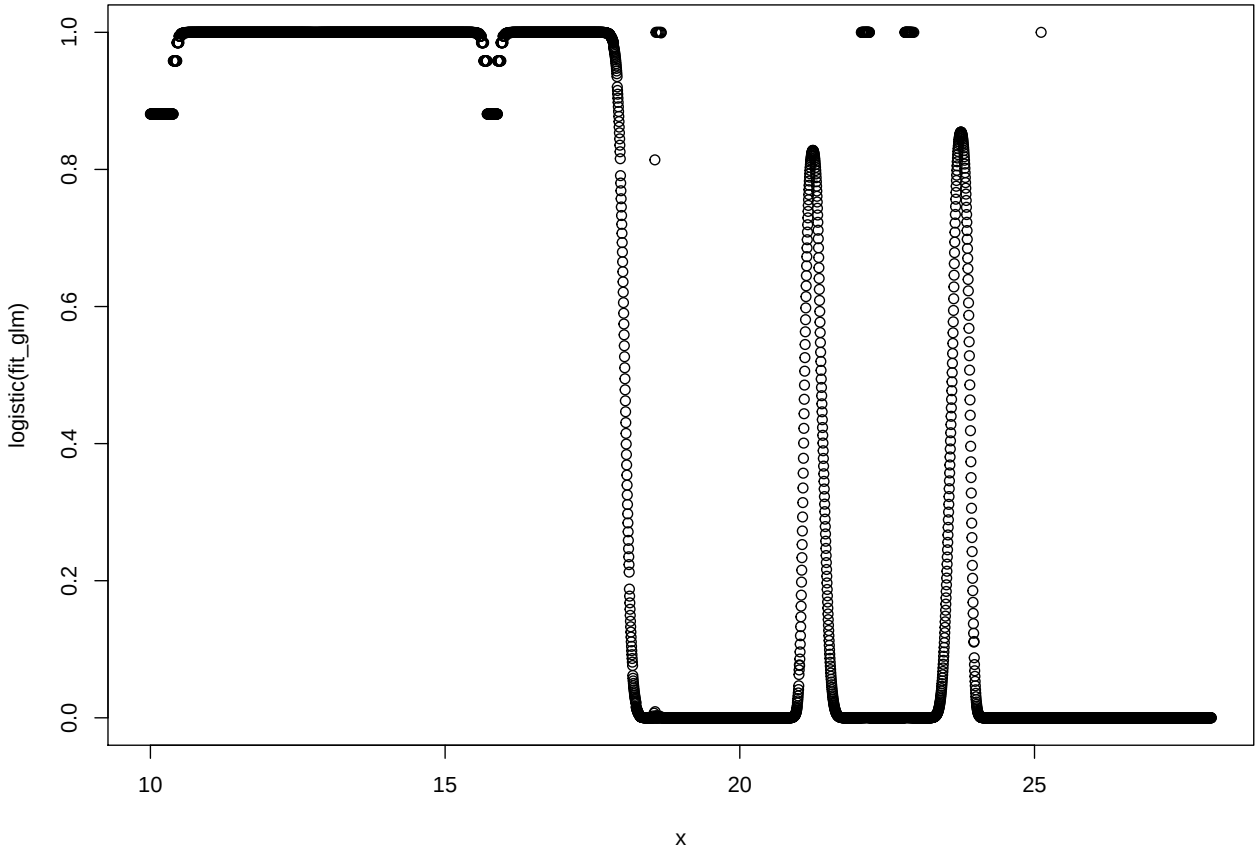
- **Bandwidth that undersmooths**: $h = 0.2$

When the bandwidths equal to 0.2, we can notice that the local logistic regression is roughly interpolating the data points that we have as samples which means that, the estimator is overfitting the samples.

```r
h <- 0.2 # bandwidth
x <- seq(10, 28, l = 5000) # x grid of points to cover

suppressWarnings(
  fit_glm <- sapply(x, function(x) {
    K <- dnorm(x = x, mean = X, sd = h)
    glm.fit(x = cbind(1, X - x), y = Y, weights = K,
            family = binomial())$coefficients[1]
  })
)

plot(x, logistic(fit_glm))
```



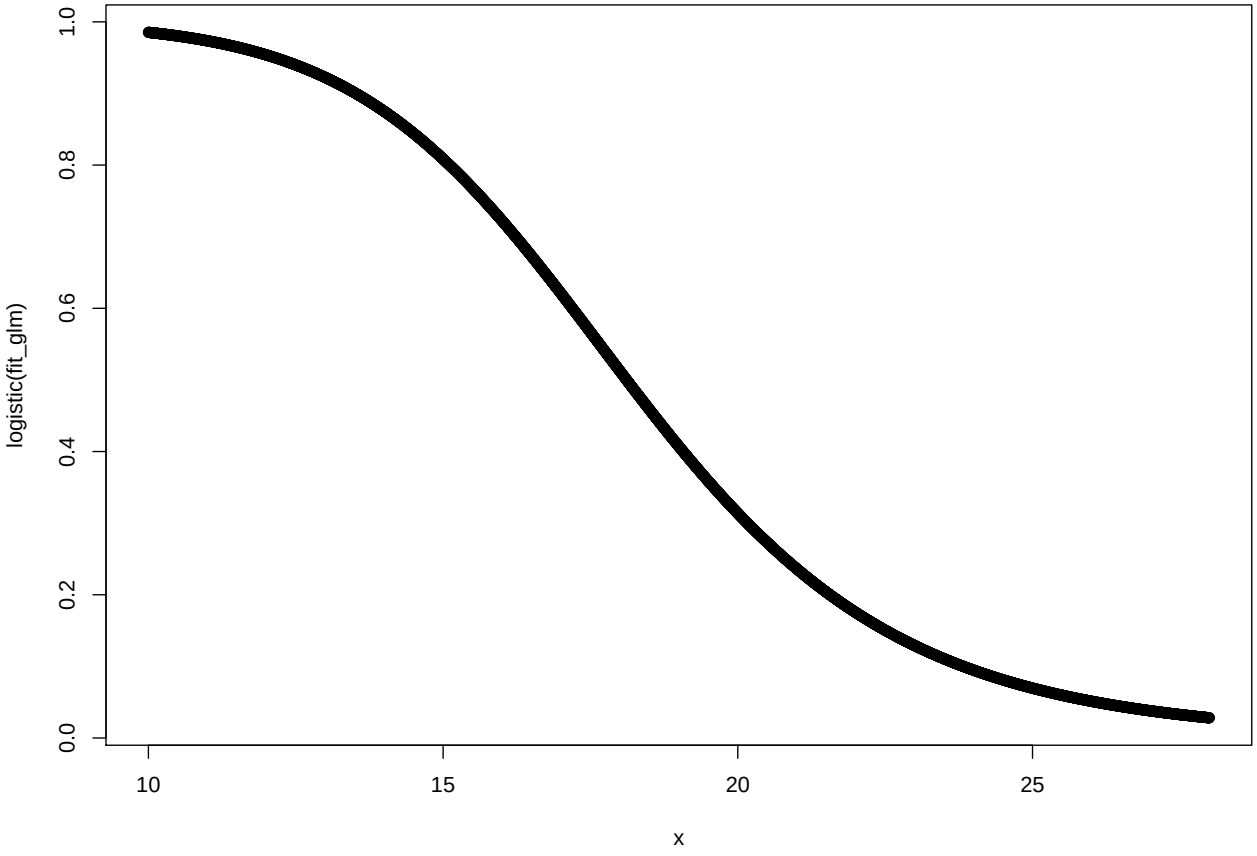
- **Bandwidth that oversmooths**: $h = 10$

If we set the bandwidth to 10, it is somehow very clear to see that the local likelihood estimator oversmooths the samples, which makes that the samples lose their characteristics.

```
h <- 10
x <- seq(10, 28, l = 5000)

suppressWarnings(
  fit_glm <- sapply(x, function(x) {
    K <- dnorm(x = x, mean = X, sd = h)
    glm.fit(x = cbind(1, X - x), y = Y, weights = K,
            family = binomial())$coefficients[1]
  })
)

plot(x, logistic(fit_glm))
```



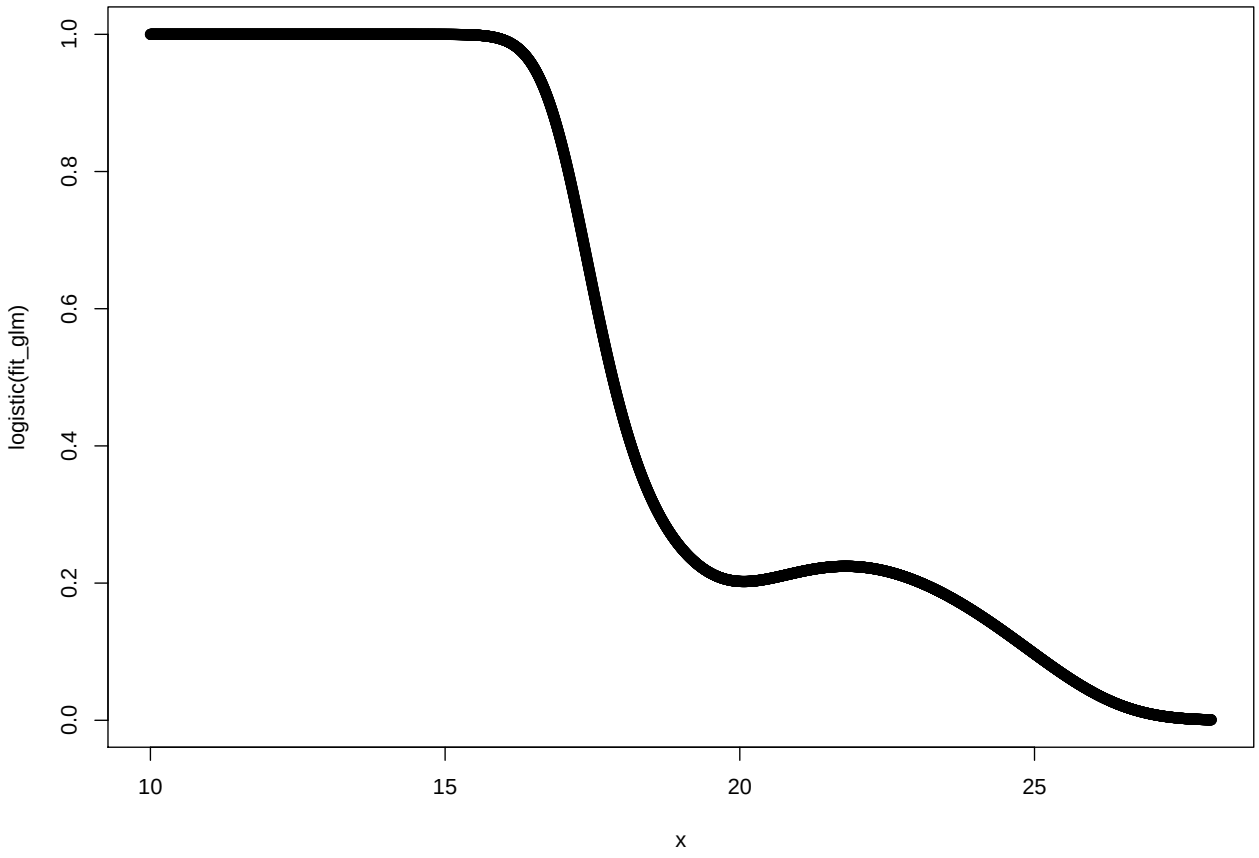
- **Adequate Bandwidth**: $h = 2$

By setting the bandwidth to 2, we can observe that the regression is more or less smooth, and it fits the data set that we have without interpolating it or oversmoothing it.

```
h <- 2
x <- seq(10, 28, l = 5000)

suppressWarnings(
  fit_glm <- sapply(x, function(x) {
    K <- dnorm(x = x, mean = X, sd = h)
    glm.fit(x = cbind(1, X - x), y = Y, weights = K,
            family = binomial())$coefficients[1]
  })
)

plot(x, logistic(fit_glm))
```



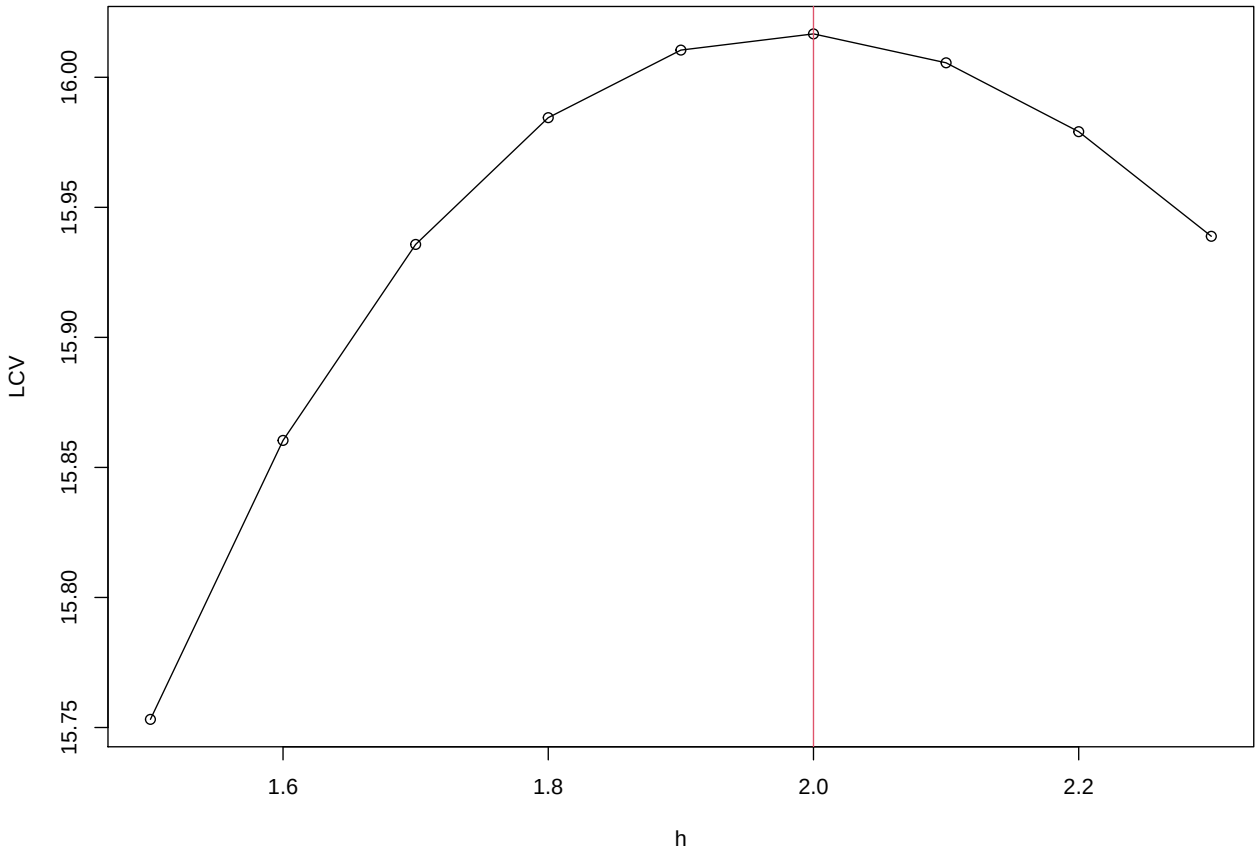
b) Obtain $\hat{h}_{LCV}$ and plot the LCV function with a reasonable accuracy.

In this part, the aim is to obtain the best bandwidth by maximazing the likelihood function. As seen before, we know that the optimal bandwidth value will be approximately around 2, so it is reasonable to try the different values around it.

By plotting the likelihood in function of the possible values of bandwidths, we can see that, 2 is the optimal bandwidth value for the data set.

```
n <- length(Y)
h <- seq(1.5, 2.3, by=0.1)

suppressWarnings(
  LCV <- sapply(h, function(h) {
  sum(sapply(1:n, function(i) {
    K <- dnorm(x = X[i], mean = X[-i], sd = h)
    nlm(f = function(beta) {
      -sum(K * (Y[-i] * (beta[1] + beta[2] * (X[-i] - X[i])) -
                  log(1 + exp(beta[1] + beta[2] * (X[-i] - X[i])))))
      }, p = c(0, 0))$minimum
    }))
  })
)
plot(h, LCV, type = "o")
abline(v = h[which.max(LCV)], col = 2)
```

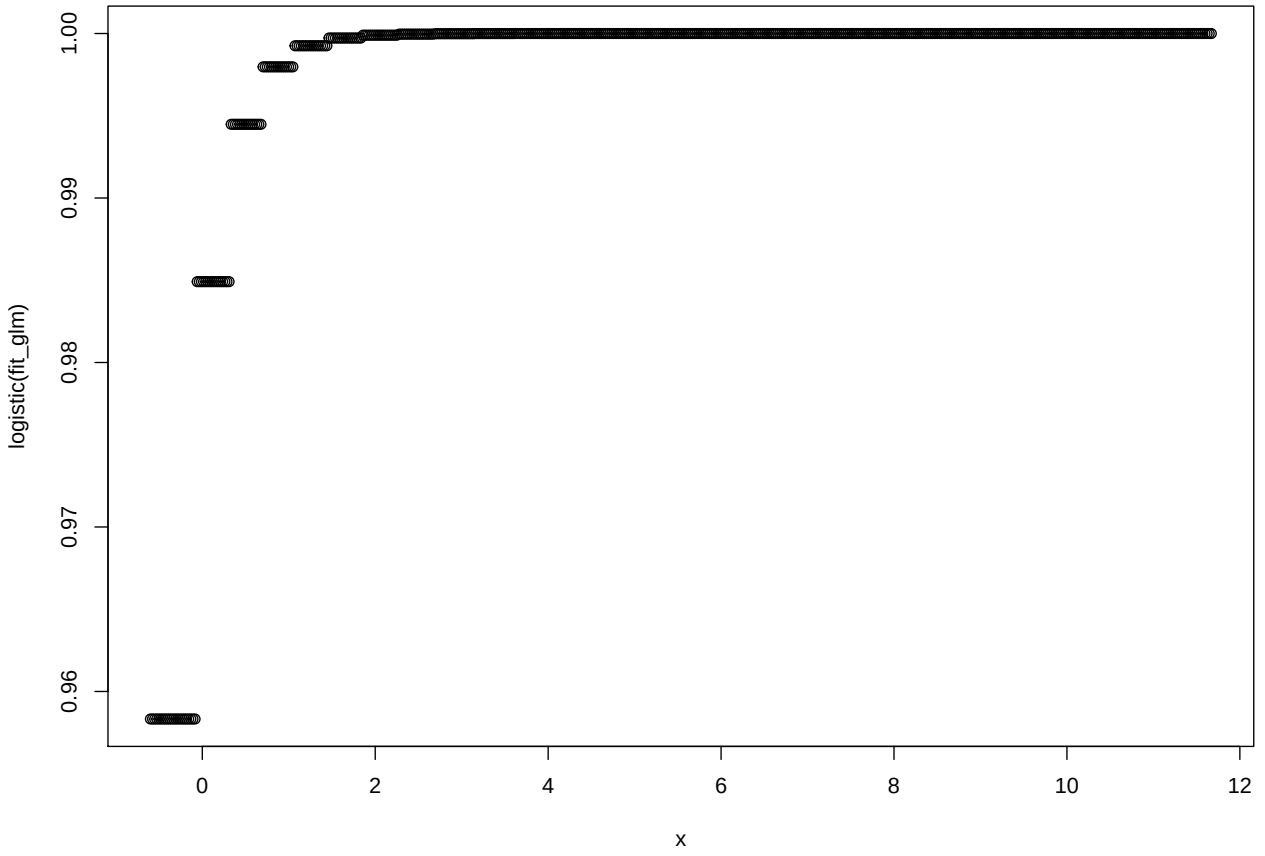


c) Using $\hat{h}_{LCV}$, predict the probability of an incident at temperatures -0.6 (launch temperature of the Challenger) and 11.67 (specific recommendation by the vice president of engineers).

```
h <- 2
x <- seq(-0.6, 11.67, l = 500)

suppressWarnings(
  fit_glm <- sapply(x, function(x) {
    K <- dnorm(x = x, mean = X, sd = h)
    glm.fit(x = cbind(1, X - x), y = Y, weights = K,
            family = binomial())$coefficients[1]
  })
)

plot(x, logistic(fit_glm))
```




```
# Find probability at x=-0.6 and x=11.67
pred0.6 = logistic(fit_glm)[1]
pred11.67 = logistic(fit_glm)[length(fit_glm)]

pred0.6
#> [1] 0.958327
pred11.67
#> [1] 1
```

d) What are the local odds at -0.6 and 11.67? Show the local logistic models about these points, in spirit of Figure 5.1, and interpret the results.

If we fit the estimator utilizing exclusively the point -0.6, we get the estimator value in the neighborhood of -0.6

```
h <- 2
x <- -0.6

suppressWarnings(
  fit_glm <- sapply(x, function(x) {
    K <- dnorm(x = x, mean = X, sd = h)
    glm.fit(x = cbind(1, X - x), y = Y, weights = K,
            family = binomial())$coefficients[1]
  })
)
```

And the result of the fit as follows:

```
#> [1] 3.135335
```

Category B: Problem 4

- **Exercise 4.9**. Perform the following tasks:

a) Code your own implementation of the local cubic estimator. The function must take as input the vector of evaluation points x, the sample *data*, and the bandwidth h. Use the normal kernel. The result must be a vector of the same length as x containing the estimator evaluated at x.

We have implemented the local polynomial estimator for any $0 \leq p < 8$ (however, $p = 3$ is the default parameter as we are asked to implement the local cubic estimator).

Unfortunately, for large datasets + evaluation points (assuming both are of the same size as we are asked in part b of this problem) the function is quite inefficient ($> O(n^3)$).

Given that our focus was to simply implement it and not necessarily be particularly efficient, we have gone for the less efficient form of the local polynomial estimator (as described in this section of the Nonparametric Statistics notes).

Parameters

- **x**: vector of evaluation points
- **data**: sample dataset where the first column (*data[,1]* in R) are the *predictors* and the second column (*data[,2]* in R) are the *response*.
- **h**: bandwidth

Algorithm

The function goes through the following process in order to output the estimation:

1- Values are initialized

- The resulting vector (*result*, initialized as an empty vector)
- The predictors (*predictors*, taken from the first column of the *data* matrix)
- The response (Y, taken from the second column of the *data* matrix)
- A $(p+1) \times 1$ vector with its first entry as 1 and the rest as zero (*e_1*)

The X matrix is also initialized in this step and corresponds to:

$$\mathbf{X} := \begin{pmatrix} 1 & X_1 - x & \cdots & (X_1 - x)^p \\ \vdots & \vdots & \ddots & \vdots \\ 1 & X_n - x & \cdots & (X_n - x)^p \end{pmatrix}_{n \times (p+1)}$$

This is initialized as an empty list, where each entry in the list corresponds to the matrix X corresponding to the k-th value of the input vector x (values the user wants to predict).

2- Main loop steps

- k iterations are initiated (one per element in the input x vector)
- The element k of the X list of matrices X is created as an empty matrix with dimensions $n \times (p+1)$, where $n =$ amount of values in the predictor column of data and $p =$ degree (by default 3).
- i iterations are initiated (one iteration per row in $X[[k]]$) and inside j iterations (one iteration per degree + 1), the dimensions of $X[[k]]$ are used to determine the amount of iterations

- Inside the innermost loop (j-loop) we subtract the k-th element in the input vector from the i-th element in the predictor vector ($predictors[i]$) and we raise the result to the power of $j-1$ (as loops go, this corresponds to $p-3,\ldots,p$, for the case p=3)

3- Weights loop

- For the weights loop we use the normal kernel (*dnorm* in R) and apply it to the difference of the i-th predictor ($predictors[i]$) and the k-th element in the input vector ($x[k]$)

4- Result

- We apply the *diag* R function to the *weights* vector to make it a diagonal matrix with the values of *weights* in the main diagonal.
- We finally add the k-th value (result of $\mathbf{e}_1'(\mathbf{X'WX})^{-1}\mathbf{X'WY}$ for the k-th x)
- Return the result vector

Function code with annotations

```r
lce <- function(x, data, h, p=3) {
    ## INITIALIZING VARIABLES
    # Resulting vector initilization
    result <- c()

    # Predictors initialization (copied from data col 1)
    predictors <- data[,1]

    # Response initialization (copied from data col 2)
    Y <- data[,2]

    # e_1 (vector of size p+1 x 1 of zeros with the exception of the first entry)
    e_1 <-  matrix(c(c(1), rep(0,p)),nrow=p+1,ncol=1)

    # X matrix
    X <- list()

    ## MAIN LOOP
    for (k in 1:length(x)){
        # k-th matrix of the list of X, whose size will correspond with the length
        # of the values the user wants to predict (length of x)
        X[[k]] <- matrix(,nrow=length(predictors),ncol=p+1)

        # Filling up the k-th X matrix entry by entry
        for (i in 1:dim(X[[k]])[1]) {
            for (j in 1:dim(X[[k]])[2]) {
                # the k-th X matrix's i,j-th positions are filled
                # subtracting the i-th predictor's value with the k-th
                # element in the values the user wants to evaluate
                # all to the power j-1, where the first value obtained
                # is a 0 in the exponent, rendering the first column of each
                # X matrix a column of ones, and the last one to the power
                # of the p defined by the user (default p=3)
                X[[k]][i,j] <- (predictors[i] - x[k])^(j-1)
            }
        }

        # Weights, these change in every loop, therefore must be initialized
        # in every loop.
        # Each weight corresponds to one element of the x vector provided
        # by the user
        weights <- c()
        for (i in 1:length(predictors)) {
            # We use the K_h normal kernel of the difference between
            # the i-th predictor and the k-th element of x
            weights[i] <- pnorm((predictors[i] - x[k])/h)/h
        }
        # We create a matrix where the entries of the weights vector
        # are in the diagonal of a matrix (where all other values are 0)
        weights <- diag(weights)

        # We calculate W_i^P (x[i]), that is the matrix product of:
        # the transpose of e_1 times the inverse of the matrix product
        # of the transpose of the k-th X times the weights times the k-th X
        # times the transpose of the k-th X times the weights times the response
        # We then assign this to the k-th element of the result vector initialized before
        result[k] <- t(e_1) %*% solve(t(X[[k]]) %*% weights %*% X[[k]]) %*% t(X[[k]]) %*% weights %*% Y
    }

    ## RETURNING VALUES
    # We return the estimator evaluated at the vector x
    return(result)
}
```

b) Test the implementation by estimating the regression function in the location model $Y = m(X) + \epsilon$, where $m(x) = (x-1)^2$, $X \sim N(1,1)$, and $\epsilon \sim N(0, 0.5)$. Do it for a sample of size $n = 500$.

We assign m to the function $m(x) = (x-1)^2$

```
# function to estimate
m = function(x) ((x-1)^2)
```

We select a bandwidth h:

```
# bandwidth
h = 0.5
```

We generate *pred* (predictor variable values) and *resp* (response variable values) and we add the ϵ for each response entry in a loop after applying the $m(x)$ function defined earlier to the values of *pred*

```
# data simulation
pred <- rnorm(500, mean=1, sd=1)
resp <- c()
for (i in 1:length(pred)) {
    resp <- c(resp, m(pred[i])) + rnorm(1, mean=0, sd=0.5)
}
```

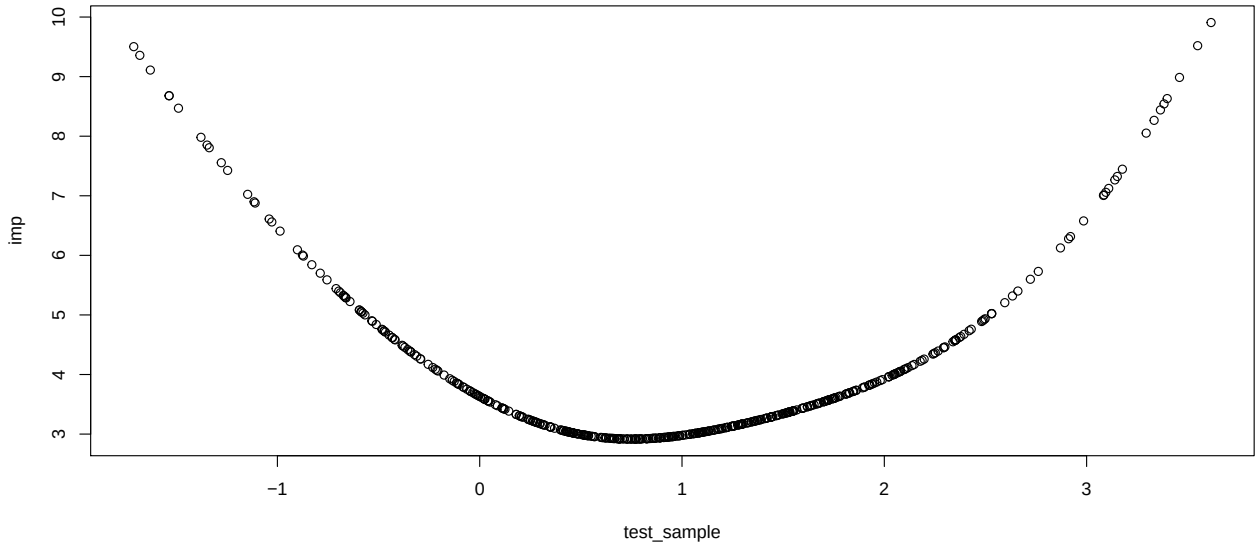
We create the variable *simulated_dataset* as a matrix with dimensions length(pred) $\times$ 2, the rows being amount of terms in *pred* and *resp* and columns being: 1- predictors and 2- response. This will facilitate the input to our *lce* function.

```
# appending to list object
simulated_dataset <- matrix(,nrow=length(pred),ncol=2)
simulated_dataset[,1] <- pred
simulated_dataset[,2] <- resp
```

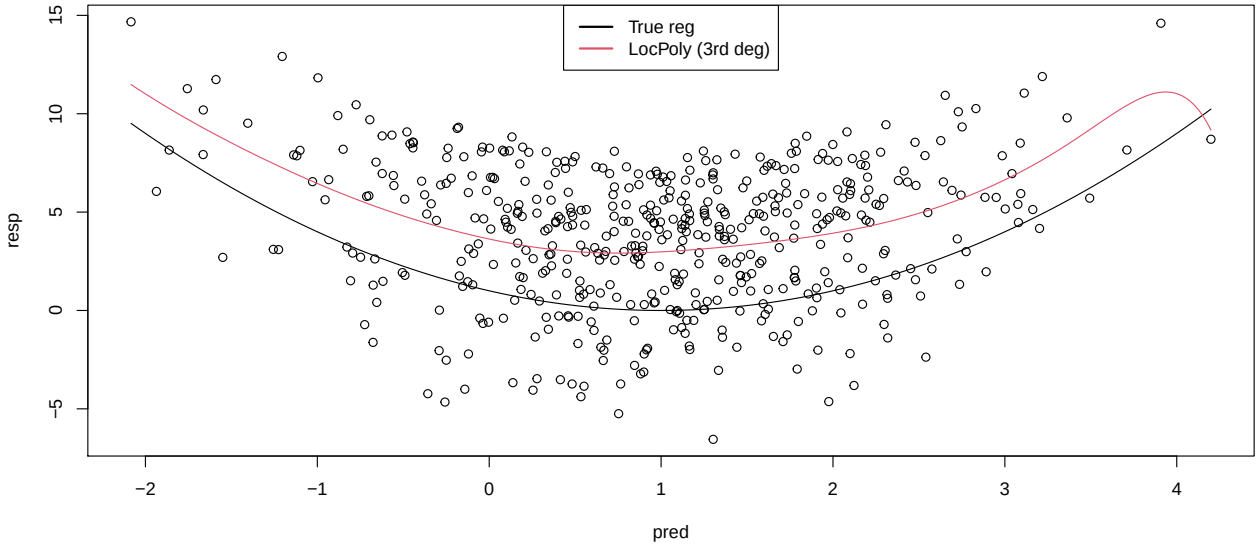
We run our custom implementation of the local cubic estimator for a sample size of $n = 500$, we generate the values from a normal distribution using *rnorm* with $\mu = 1$ and $\sigma = 1$.

```
# Running the custom implementation
test_sample <- rnorm(500, mean=1, sd=1)
imp <- lce(x=test_sample, data=simulated_dataset, h=h)
```

We plot the points obtained by our implementation vs the original values as follows



We run our implementation vs the true regression for a grid within the range of *pred*.



We can see that our implementation traces the true regression reasonably accurately with our chosen bandwidth $h = 0.5$

Category C: Problem 4

- **Exercise 3.30**. Load the *ovals.RData* file.

a) Split the dataset into the training sample, comprised of the first 2,000 observations, and the test sample (rest of the sample). Plot the dataset with colors for its classes. What can you say about the classification problem?

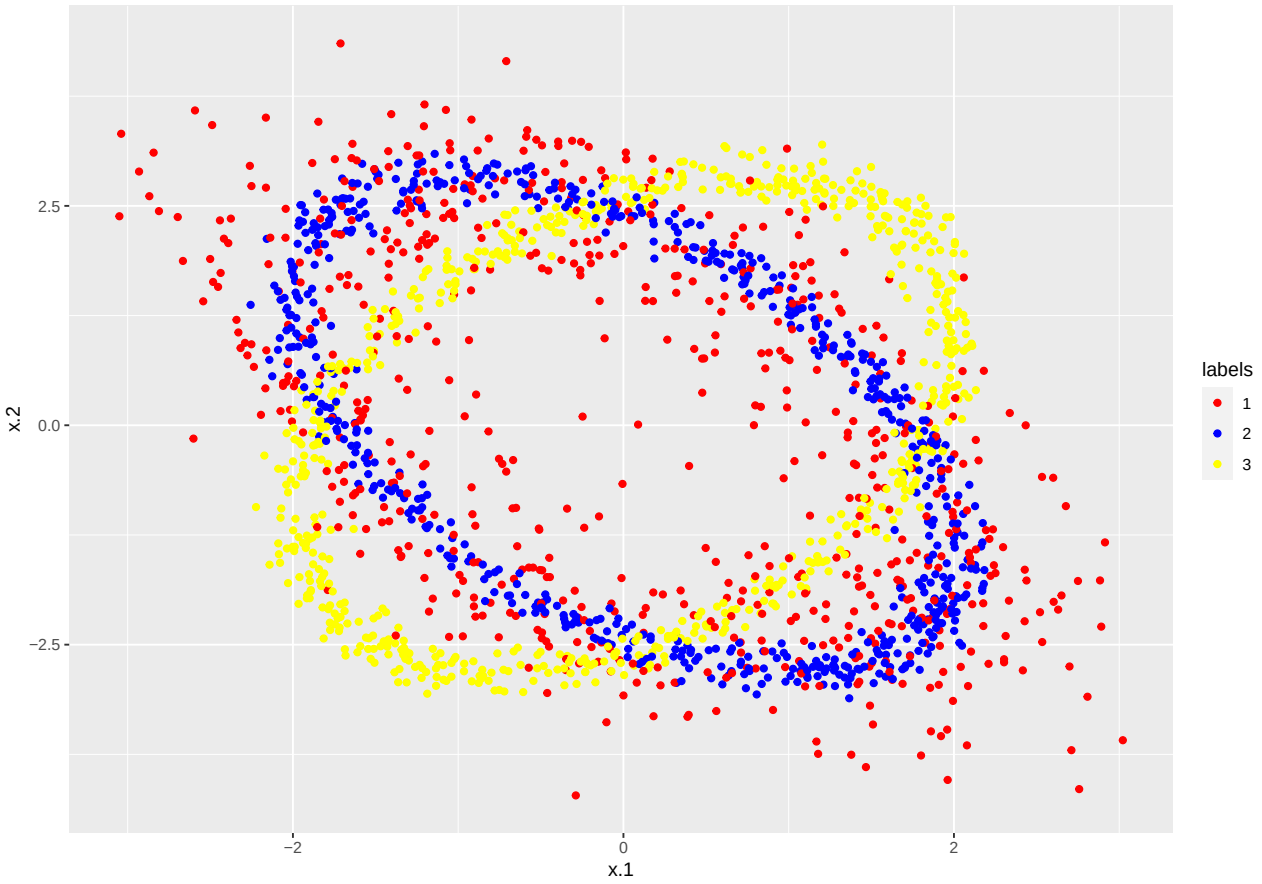
Importing the necessary libraries for the task:

```
# Importing necessary libraries
library(ks)
library(dplyr)
library(ggplot2)
```

We can roughly see that the distribution of the observations of the third class and the first/second class are very different, which means that the classification problem should technically be relatively easy (splitting the third class data from the first or the second data). But by looking at our resulting plot, the distribution of the first class and the second class are very similar, only that the distribution of the first class observations is more dispersed, the shape seems to be the same, so the classification problem between those two classes will probably be more difficult.

```
# loading the data
load(url('https://raw.githubusercontent.com/egarpor/handy/master/datasets/ovals.RData'))

ovals$labels = as.factor(ovals$labels) # converting the labels column to factor
train <- ovals[1:2000,] # splitting the dataset into train
test <- ovals[2001:3000,] # and test set
```


2.5
0.0
−2.5
x.2
−2
0
2
x.1
labels
1
2
3

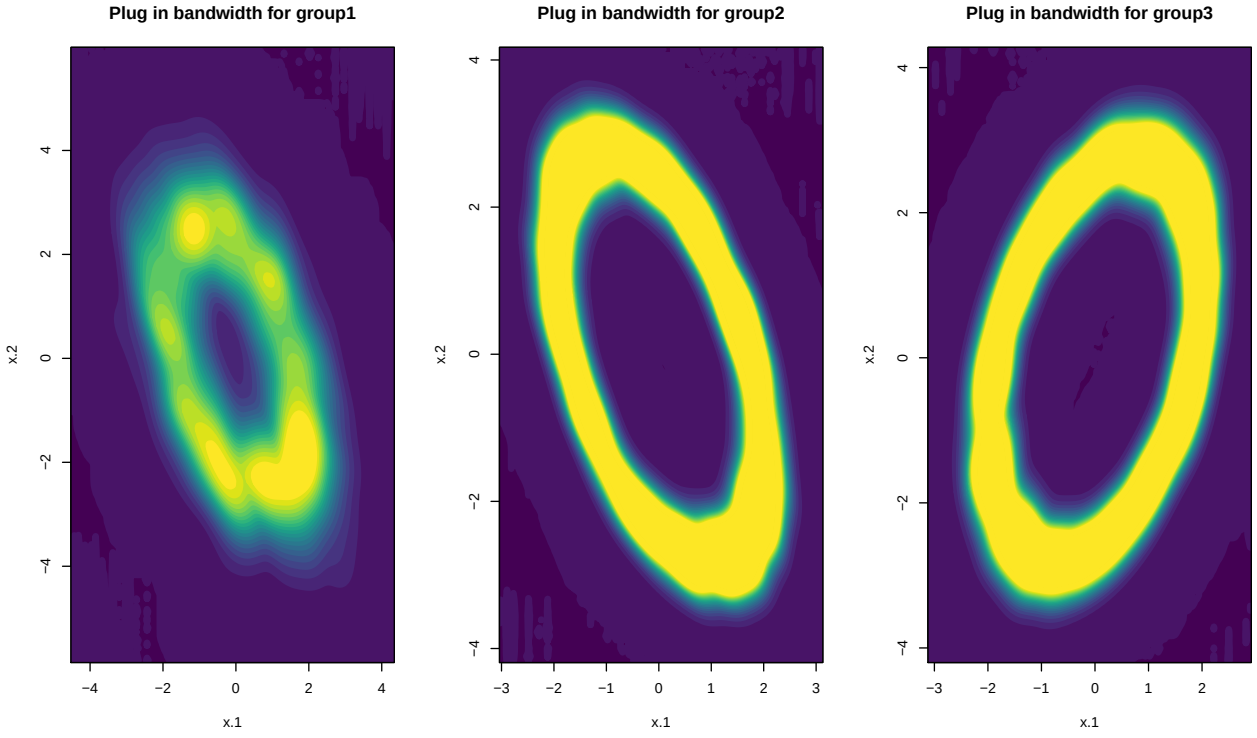
b) Using the training sample, compute the plug-in bandwidth matrices for all the classes.

```
# Filtering x1, x2 and x3 by their respective grouping (determined by the labels column)
x1 <- ovals %>% filter(labels=="1") %>%
  select(x.1,x.2)
x2 <- ovals %>% filter(labels=="2") %>%
  select(x.1,x.2)
x3 <- ovals %>% filter(labels=="3") %>%
  select(x.1,x.2)

# Computing the plug in bandwidth for each subset (x1, x2 and x3)
h1 <- ks::Hpi(x1)
h2 <- ks::Hpi(x2)
h3 <- ks::Hpi(x3)
```

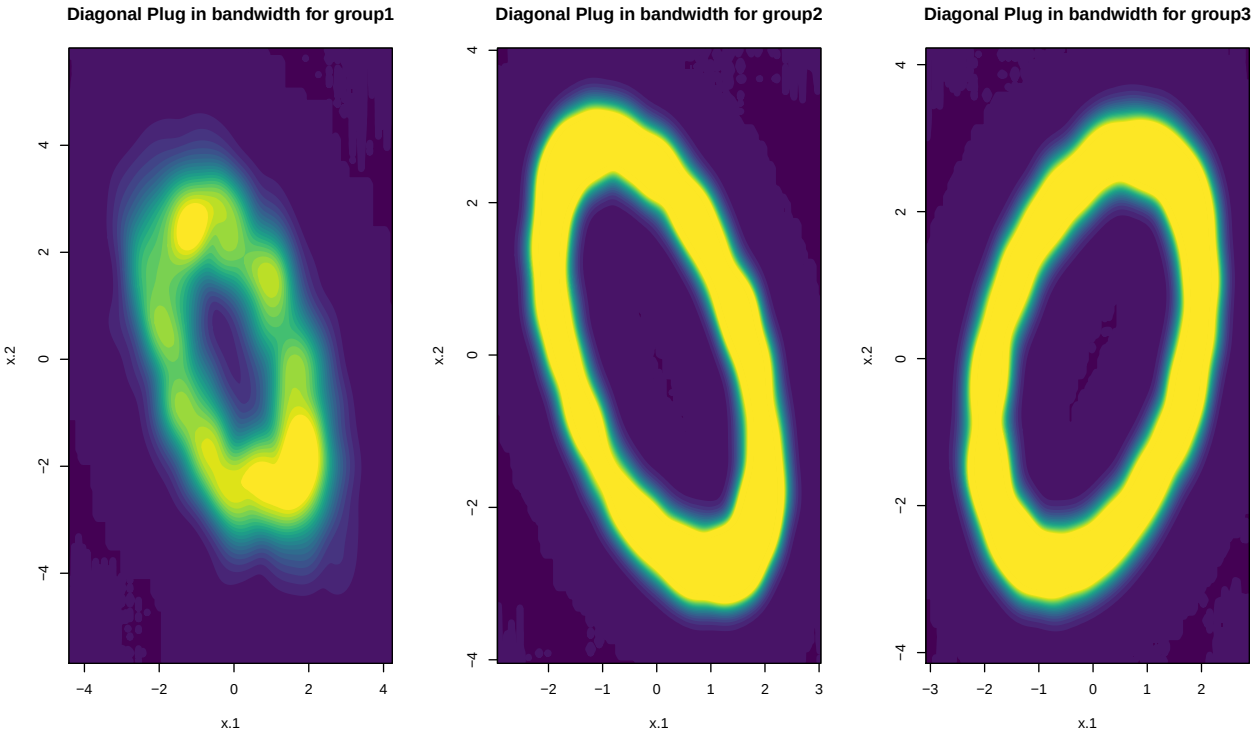
As previously mentioned, we can notice that the border of the first class if blurrier and harder to define than that of the other 2 classes due to the fact that the observations are noticeably more dispersed; also, the shape of the estimated distributions of group 1 and group 2 are very similar while the estimated distribution of the third group is very different.

```
cont <- seq(0, 0.05, l = 20) # Defining a grid of 20 elements between 0 and 0.05
col <- viridis::viridis
```



We can observe that the estimation of the distribution with diagonal bandwidths is rather likely the same as that of non-diagonal bandwidths.

```
# Obtaining diagonal plug in bandwidths
h1_diag <- ks::Hpi.diag(x1)
h2_diag <- ks::Hpi.diag(x2)
h3_diag <- ks::Hpi.diag(x3)
```



c) Use these plug-in bandwidths to perform kernel discriminant analysis.

First we split the predictors from the response in order to fit the model.

```
x <- train[,1:2]
groups<-train$labels
```

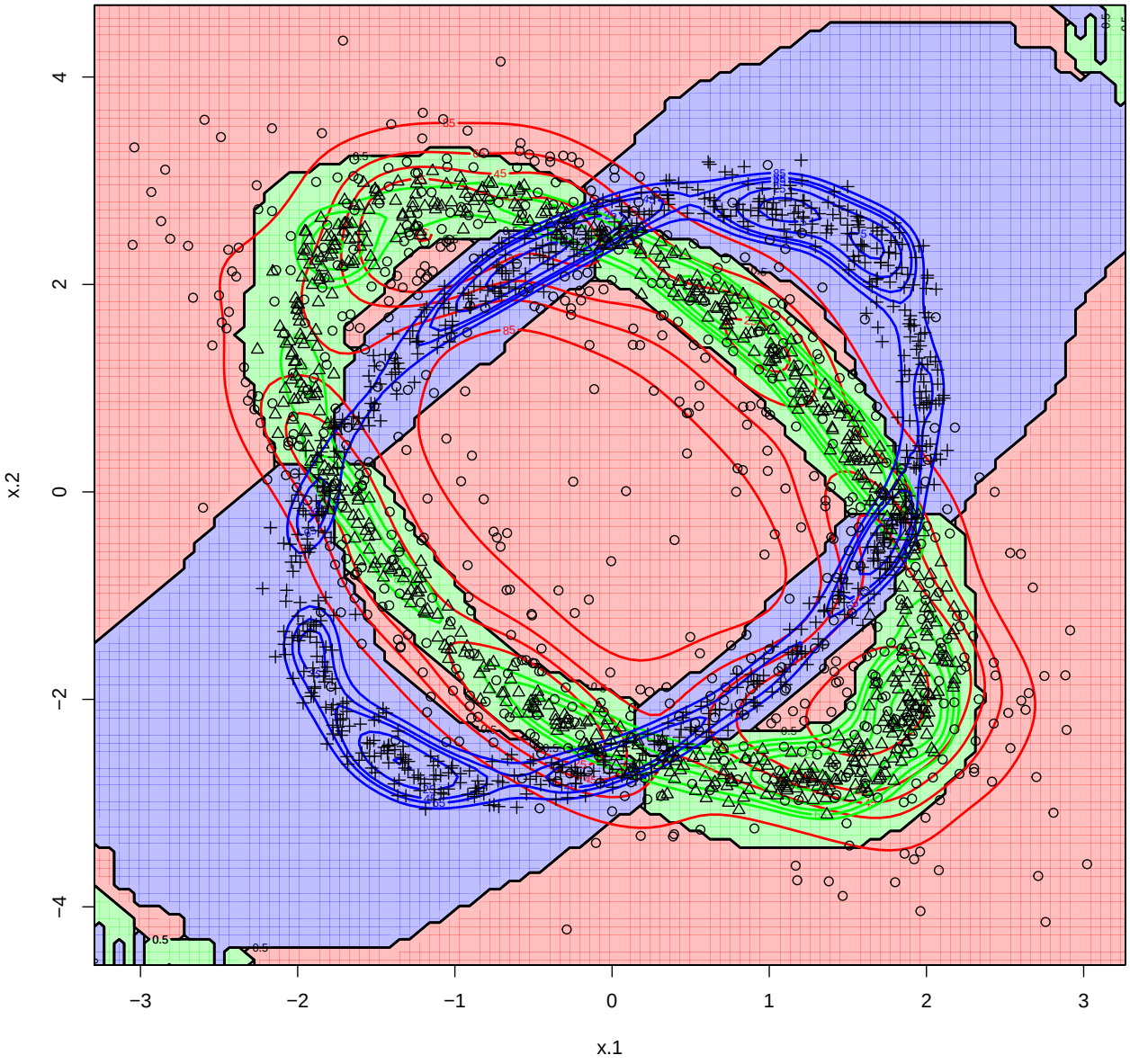
Then we use *kda* function from package *ks* which computes automatically the model selecting the plug in bandwidths.

```
# Hpi bandwidths are computed
kda <- ks::kda(x=x, x.group = groups)
```

d) Plot the contours of the kernel density estimator of each class and the classes partitions. Use coherent colors between contours and points.

We can see that the areas classified as red/green coincide, as the highest density red/green areas correspond to the same groups of points, however, most of these were classified in the green group, as the density is significantly more concentrated in the green area.

The remaining points around the green/blue areas and the middle part are classified inside the red group, as most of its points are dispersed around the areas corresponding to the green/blue sections. Obviously, red points that fall into the highest density sections of our green/blue groups will be misclassified, but the classification errors for this group should be rather small.



e) Predict the class for the test sample and compare with the true classes. Then report the successful classification rate.

From the report generated by *compare* from package *ks*, we can see that the kda classifier has a general accuracy of **71.7%**, it classifies specially well the third group (**93.86%) and very formidably well the 2nd group (92.67%**), it seems to have special difficulty with the first group, the most dispersed one, as we expected, this is understandable, as it is based on KDE.

The method works impressively for unusually shaped datasets like this one, given the ambiguous and less strong concentration/density of points in the first group, we only classify well the points belonging to this group that are scattered around, outside of high density regions belonging to the two other groups. The final accuracy for this specific group is ~**46.71%**, which is basically guessing, however, given the strength at predicting the rest of the groups, depending on the nature of a project, we could more granularly tune the model to predict members of this group slightly better.

A combination of KDA and other models might be suggested solely for this group. What this tells us is that we can confidently trust its prediction as long as the density for a group is somewhat concentrated. With those, the prediction is outstandingly accurate.

```
new_data=test[1:2] # dividing the test set to predict
```

```
pred_kda <- predict(kda, x = new_data) # predicting over the new_data section of test
ks::compare(x.group = test$labels, est.group = pred_kda)$cross # confusion matrix for the KDA model
#>           1 (est.) 2 (est.) 3 (est.) Total
#> 1 (true)        96      159       79   334
#> 2 (true)         1      316       24   341
#> 3 (true)         0       20      305   325
#> Total           97      495      408  1000
```

f) Compare the successful classification rate with the one given by LDA. Is it better than kernel discriminant analysis?

And this is where KDA shines versus other models, our results for LDA prove that for a dataset like this one, LDA is quite weak at predicting, where even when purely guessing, general accuracy is incredibly low (**33.6%**), and indiviudal group accuracy is in the same ballpark.

```
lda_model = MASS::lda(x, grouping = groups, data=train) # running LDA model
pred_lda = predict(lda_model, newdata = new_data)$class # predicting using LDA
ks::compare(test$labels, pred_lda) # confusion matrix for LDA
#> $cross
#>           1 (est.) 2 (est.) 3 (est.) Total
#> 1 (true)        42      139      153   334
#> 2 (true)        34      150      157   341
#> 3 (true)        88       93      144   325
#> Total          164      382      454  1000
#>
#> $error
#> [1] 0.664
```

g) Repeat f with QDA.

QDA yields a similar result to LDA, albeit only slightly better, with a general accuracy of ~**43.4%** the model is basically guessing and doing so quite innacurately as well. Individual group accuracy is also quite bad. Therefore for this dataset, neither LDA or QDA are appropriate.

```
qda_model = MASS::qda(x, grouping = groups, data=train) # running QDA model
pred_qda = predict(qda_model, newdata =new_data)$class # predicting using QDA
ks::compare(test$labels, pred_qda) # confusion matrix for QDA
#> $cross
#>           1 (est.) 2 (est.) 3 (est.) Total
#> 1 (true)        97      121      116   334
#> 2 (true)       102      128      111   341
#> 3 (true)        43       73      209   325
#> Total          242      322      436  1000
#>
#> $error
#> [1] 0.566
```