Stat 432 Homework 6

Qianhua Zhou

Assigned: Sep 29, 2024; Due: 11:59 PM CT, Oct 10, 2024

Contents

Question 1: Multivariate Kernel Regression Simulation (45 pts)

Similar to the previous homework, we will use simulated datasets to evaluate a kernel regression model. You should write your own code to complete this question. We use two-dimensional data generator:

$$Y = \exp(\beta^T x) + \epsilon$$

where $\beta = c(1, 1)$, X is generated uniformly from $[0, 1]^2$, and ϵ follows i.i.d. standard Gaussian. Use the following code to generate a set of training and testing data:

```
set.seed(2)
trainn <- 200
testn <- 1
p = 2
beta <- c(1.5, 1.5)

# generate data

Xtrain <- matrix(runif(trainn * p), ncol = p)
Ytrain <- exp(Xtrain %*% beta) + rnorm(trainn)
Xtest <- matrix(runif(testn * p), ncol = p)

# the first testing observation
Xtest
```

```
##           [,1]      [,2]
## [1,] 0.4152441 0.5314388
```

```
# the true expectation of the first testing observation
exp(Xtest %*% beta)
```

```
##          [,1]
## [1,] 4.137221
```

a. [10 pts] For this question, you need to **write your own code** for implementing a two-dimensional Nadaraya-Watson kernel regression estimator, and predict **just the first testing observation**. For this task, we will use independent Gaussian kernel function introduced during the lecture. Use the same bandwidth h for both dimensions. As a starting point, use $h = 0.07$. What is your predicted value?

```
# Nadaraya-Watson kernel regression estimator
nadaraya_watson <- function(Xtrain, Ytrain, Xtest, h) {
  kernel <- function(x, xi, h) {
    exp(-sum((x - xi)^2) / (2 * h^2))
  }

  # Calculate the numerator and denominator for the kernel regression estimator
  num <- 0
  denom <- 0
  for (i in 1:nrow(Xtrain)) {
    weight <- kernel(Xtest, Xtrain[i, ], h)
    num <- num + weight * Ytrain[i]
    denom <- denom + weight
  }

  # Return the predicted value
  return(num / denom)
}

# Apply the estimator to predict the first testing observation
h <- 0.07
predicted_value <- nadaraya_watson(Xtrain, Ytrain, Xtest, h)
cat("predicted_value: ", predicted_value)
```

```
## predicted_value:  4.198552
```

b. [20 pts] Based on our previous understanding the bias-variance trade-off of KNN, do the same simulation analysis for the kernel regression model. Again, you only need to consider the predictor of this one testing point. Your simulation needs to be able to calculate the following quantities:

- Bias^2
- Variance
- Mean squared error (MSE) of prediction

Use at least 5000 simulation runs. Based on your simulation, answer the following questions:

- Does the MSE matches our theoretical understanding of the bias-variance trade-off?
- Comparing the bias and variance you have, should we increase or decrease the bandwidth h to reduce the MSE?

```
set.seed(2)
simulate_kernel_regression <- function(trainn, testn, p, beta, h, B) {
  bias_squared <- 0
  variance <- 0
  mse <- 0
  true_value <- exp(Xtest %*% beta)
```

```
  predictions <- numeric(B)
  for (b in 1:B) {
    # Generate new training data
    Xtrain <- matrix(runif(trainn * p), ncol = p)
    Ytrain <- exp(Xtrain %*% beta) + rnorm(trainn)

    # Predict the test observation
    predictions[b] <- nadaraya_watson(Xtrain, Ytrain, Xtest, h)
  }

  # Bias^2
  bias_squared <- (mean(predictions) - true_value)^2

  # Variance
  variance <- var(predictions)

  # MSE
  mse <- mean((predictions - true_value)^2)

  return(list(bias_squared = bias_squared, variance = variance, mse = mse))
}

B <- 5000
results <- simulate_kernel_regression(trainn = 200, testn = 1, p = 2, beta = beta, h = 0.07, B = B)
```

```
## Warning in predictions - true_value: Recycling array of length 1 in vector-array arithmetic is depre
##   Use c() or as.vector() instead.
```

```
cat("Bias^2: ", results$bias_squared, "\n")
```

```
## Bias^2:  0.002284004
```

```
cat("Variance: ", results$variance, "\n")
```

```
## Variance:  0.09560785
```

```
cat("MSE: ", results$mse, "\n")
```

```
## MSE:  0.09787273
```

```
cat("Bias^2 + Variance: ", results$bias_squared + results$variance, "\n")
```

```
## Bias^2 + Variance:  0.09789185
```

-Yes, the MSE aligns well with the theoretical understanding of the bias-variance trade-off. The MSE is very close to the sum of $Bias^2$ and Variance, which confirms the relationship MSE= $Bias^2$ + Variance + noise. Given that the variance dominates the MSE, this suggests that the model's variability in predictions is the primary source of error, with very little contribution from bias.

- In this case, the variance is much larger than the bias, suggesting that the model is overfitting due to a small bandwidth h. To reduce the variance and, consequently, the MSE, we should increase the bandwidth h. Increasing h will smooth the kernel regression model, reducing its sensitivity to the individual training points and thus lowering the variance. However, increasing h will also increase the bias slightly, but given the current low bias, this trade-off should improve the overall MSE.

c. [15 pts] In practice, we will have to use cross-validation to select the optimal bandwidth. However, if you have the power of simulating as many datasets as you can, and you also know the true model, how would you find the optimal bandwidth for the bias-variance trade-off for this particular model and sample size? Provide enough evidence to claim that your selected bandwidth is (almost) optimal.

```r
# Simulate the kernel regression and calculate MSE, Bias^2, and Variance
simulate_kernel_regression <- function(trainn, testn, p, beta, h, B) {
  true_value <- exp(Xtest %*% beta)  # True value for the test point

  predictions <- numeric(B)  # Store predictions from each simulation

  for (b in 1:B) {
    # Generate new training data
    Xtrain <- matrix(runif(trainn * p), ncol = p)
    Ytrain <- exp(Xtrain %*% beta) + rnorm(trainn)

    # Predict the test observation using kernel regression
    predictions[b] <- nadaraya_watson(Xtrain, Ytrain, Xtest, h)
  }

  # Bias^2
  bias_squared <- (mean(predictions) - true_value)^2

  # Variance
  variance <- var(predictions)

  # MSE
  mse <- mean((predictions - true_value)^2)

  return(list(bias_squared = bias_squared, variance = variance, mse = mse))
}

# Function to find the optimal bandwidth
find_optimal_bandwidth <- function(h_values, trainn, testn, p, beta, B) {
  mse_values <- numeric(length(h_values))

  for (i in 1:length(h_values)) {
    h <- h_values[i]
    res <- simulate_kernel_regression(trainn = trainn, testn = testn, p = p, beta = beta, h = h, B = B)
    mse_values[i] <- res$mse
  }

  optimal_h <- h_values[which.min(mse_values)]
  return(list(optimal_h = optimal_h, mse_values = mse_values))
}

# Test a range of bandwidths
```

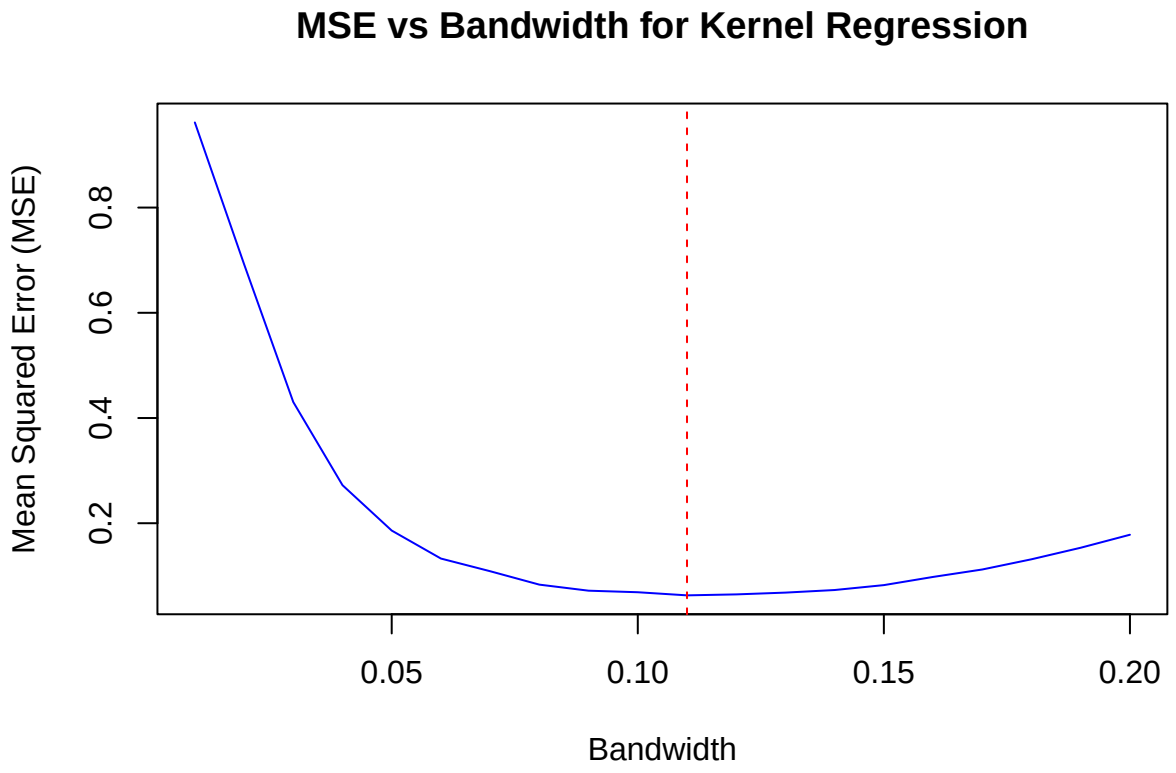
```r
h_values <- seq(0.01, 0.2, by = 0.01)
optimal_bandwidth <- find_optimal_bandwidth(h_values, trainn = 200, testn = 1, p = 2, beta = beta, B = 5

# Output the optimal bandwidth and minimum MSE
cat("Optimal Bandwidth:", optimal_bandwidth$optimal_h, "\n")
```

```
## Optimal Bandwidth: 0.11
```

```r
cat("Minimum MSE:", min(optimal_bandwidth$mse_values), "\n")
```

```
## Minimum MSE: 0.06293818
```

```r
# Plot the MSE vs Bandwidth
plot(h_values, optimal_bandwidth$mse_values, type = "l", col = "blue",
     xlab = "Bandwidth", ylab = "Mean Squared Error (MSE)",
     main = "MSE vs Bandwidth for Kernel Regression")

# Add a vertical line for the optimal bandwidth
abline(v = optimal_bandwidth$optimal_h, col = "red", lty = 2)
```



Question 2: Local Polynomial Regression (55 pts)

We introduced the local polynomial regression in the lecture, with the objective function for predicting a target point x_0 defined as

$$(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}_{x_0})^{\mathrm{T}}\mathbf{W}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}_{x_0}),$$

where W is a diagonal weight matrix, with the ith diagonal element defined as $K_h(x_0, x_i)$, the kernel distance between x_i and x_0. In this question, we will write our own code to implement this model. We will use the same simulated data provided at the beginning of Question 1.

```
  set.seed(2)
  trainn <- 200
  testn <- 1
  p = 2
  beta <- c(1.5, 1.5)

  # generate data

  Xtrain <- matrix(runif(trainn * p), ncol = p)
  Ytrain <- exp(Xtrain %*% beta) + rnorm(trainn)
  Xtest <- matrix(runif(testn * p), ncol = p)
```

a. [10 pts] Using the same kernel function as Question 1, calculate the kernel weights of x_0 against all observed training data points. Report the 25th, 50th and 75th percentiles of the weights so we can check your answer.

```
# Define the Gaussian kernel function
gaussian_kernel <- function(x0, x, h) {
  exp(-sum((x0 - x)^2) / (2 * h^2))
}

# Calculate the kernel weights for the testing point Xtest
calculate_kernel_weights <- function(Xtrain, Xtest, h) {
  weights <- numeric(nrow(Xtrain))
  for (i in 1:nrow(Xtrain)) {
    weights[i] <- gaussian_kernel(Xtest, Xtrain[i, ], h)
  }
  return(weights)
}

# Set bandwidth h
h <- 0.07

# Calculate the kernel weights
weights <- calculate_kernel_weights(Xtrain, Xtest, h)

# Report the 25th, 50th, and 75th percentiles of the weights
quantiles <- quantile(weights, probs = c(0.25, 0.5, 0.75))
quantiles
```

```
##          25%          50%          75%
## 5.774659e-13 4.999524e-08 3.882088e-05
```

b. [15 pts] Based on the objective function, derive the normal equation for estimating the local polynomial regression in matrix form. And then define the estimated $\boldsymbol{\beta}_{x_0}$. Write your answer in latex.

Derivation of the Normal Equation

Given the objective function for local polynomial regression:

$$(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}_{x_0})^{\mathrm{T}}\mathbf{W}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}_{x_0}),$$

where:

- $\mathbf{y}$ is the vector of observed responses,
- $\mathbf{X}$ is the design matrix of the observed data points (with the intercept and covariates),
- $\boldsymbol{\beta}_{x_0}$ is the vector of regression coefficients at the point x_0,
- $\mathbf{W}$ is the diagonal weight matrix, where the i-th element is $K_h(x_0, x_i)$, representing the kernel distance between x_i and x_0.

To estimate $\boldsymbol{\beta}_{x_0}$, we differentiate the objective function with respect to $\boldsymbol{\beta}_{x_0}$ and set it equal to zero.

The normal equation is derived as follows:

$$\frac{\partial}{\partial \boldsymbol{\beta}_{x_0}} \left((\mathbf{y} - \mathbf{X}\boldsymbol{\beta}_{x_0})^{\mathrm{T}}\mathbf{W}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}_{x_0})\right) = 0.$$

Simplifying the derivative:

$$-2\mathbf{X}^{\mathrm{T}}\mathbf{W}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}_{x_0}) = 0,$$

which leads to the following equation:

$$\mathbf{X}^{\mathrm{T}}\mathbf{W}\mathbf{y} = \mathbf{X}^{\mathrm{T}}\mathbf{W}\mathbf{X}\boldsymbol{\beta}_{x_0}.$$

Solving for $\boldsymbol{\beta}_{x_0}$, we get the normal equation:

$$\boldsymbol{\beta}_{x_0} = (\mathbf{X}^{\mathrm{T}}\mathbf{W}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{W}\mathbf{y}.$$

This is the closed-form solution for the local polynomial regression coefficients at the point x_0.

c. [10 pts] Based on the observed data provided in Question 1, calculate the estimated $\boldsymbol{\beta}_{x_0}$ for the testing point `Xtest` using the formula you derived. Report the estimated $\boldsymbol{\beta}_{x_0}$. Calculate the prediction on the testing point and compare it with the true expectation.

```
# Local polynomial regression estimation
estimate_beta <- function(Xtrain, Ytrain, Xtest, h) {
  # Add intercept column to Xtrain
  Xtrain_aug <- cbind(1, Xtrain)
  Xtest_aug <- c(1, Xtest)

  # Calculate the kernel weights
  W <- diag(calculate_kernel_weights(Xtrain, Xtest, h))

  # Compute beta using the normal equation
  beta <- solve(t(Xtrain_aug) %*% W %*% Xtrain_aug) %*% t(Xtrain_aug) %*% W %*% Ytrain
```

```
  return(beta)
}

# Estimate beta for Xtest
beta_estimated <- estimate_beta(Xtrain, Ytrain, Xtest, h)
beta_estimated
```

```
##           [,1]
## [1,] -1.749993
## [2,]  5.690594
## [3,]  6.870116
```

```
# Predict the value at Xtest using the estimated beta
prediction <- c(1, Xtest) %*% beta_estimated
prediction
```

```
##          [,1]
## [1,] 4.264039
```

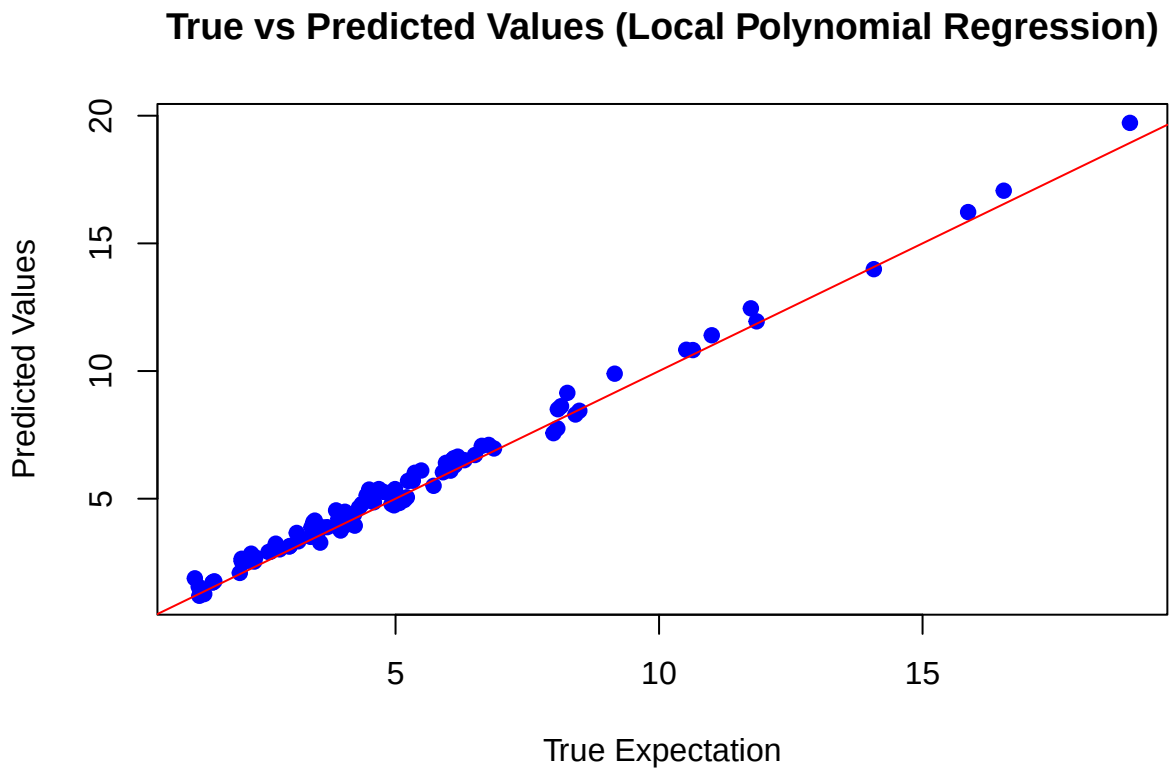
```
# Calculate the true expectation at Xtest
true_value <- exp(Xtest %*% beta)
true_value
```

```
##          [,1]
## [1,] 4.137221
```

d. [20 pts] Now, let's use this model to predict the following 100 testing points. After you fit the model, provide a scatter plot of the true expectation versus the predicted values on these testing points. Does this seem to be a good fit? As a comparison, fit a global linear regression model to the training data and predict the testing points. Does your local linear model outperforms the global linear mode? Note: this is not a simulation study. You should use the same training data provided previously.

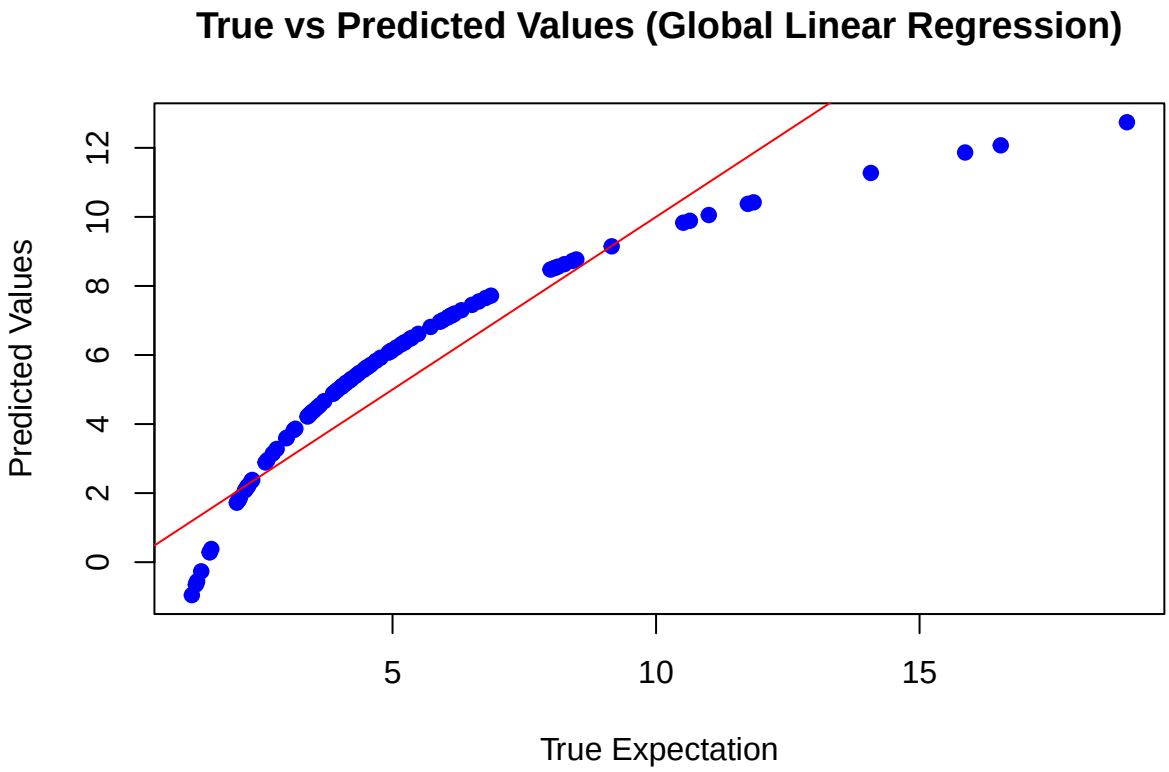
```
set.seed(432)

# Generate 100 testing points
testn <- 100
Xtest <- matrix(runif(testn * p), ncol = p)

# True expectation for the test points
true_values <- exp(Xtest %*% beta)

# Local Polynomial Regression Function (from previous parts)
predict_local_polynomial <- function(Xtrain, Ytrain, Xtest, h) {
  predictions <- numeric(nrow(Xtest))
  for (i in 1:nrow(Xtest)) {
    beta_est <- estimate_beta(Xtrain, Ytrain, Xtest[i, ], h)
    predictions[i] <- c(1, Xtest[i, ]) %*% beta_est
  }
  return(predictions)
}
```

```r
# Set bandwidth h
h <- 0.07

# Predict using local polynomial regression for the 100 test points
local_predictions <- predict_local_polynomial(Xtrain, Ytrain, Xtest, h)

# Global Linear Regression Model
# We need to adjust the Xtrain for the global model
Xtrain_df <- data.frame(Xtrain)

# Fit global linear regression model
global_model <- lm(Ytrain ~ ., data = Xtrain_df)

# Make predictions for the 100 test points
Xtest_df <- data.frame(Xtest)  # Match the structure of Xtrain
global_predictions <- predict(global_model, newdata = Xtest_df)

# Scatter plot for Local Polynomial Regression
plot(true_values, local_predictions, xlab = "True Expectation", ylab = "Predicted Values",
     main = "True vs Predicted Values (Local Polynomial Regression)", col = "blue", pch = 19)
abline(0, 1, col = "red")  # Line of perfect prediction
```




```r
# Scatter plot for Global Linear Regression
plot(true_values, global_predictions, xlab = "True Expectation", ylab = "Predicted Values",
     main = "True vs Predicted Values (Global Linear Regression)", col = "blue", pch = 19)
```

```r
abline(0, 1, col = "red")  # Line of perfect prediction
```




```r
# Compare the performance of local and global models
local_mse <- mean((local_predictions - true_values)^2)
global_mse <- mean((global_predictions - true_values)^2)

cat("Local Polynomial Regression MSE:", local_mse, "\n")
```

```
## Local Polynomial Regression MSE: 0.1534425
```

```r
cat("Global Linear Regression MSE:", global_mse, "\n")
```

```
## Global Linear Regression MSE: 1.751315
```

```r
if (local_mse < global_mse) {
  cat("The local polynomial regression model performs better.\n")
} else {
  cat("The global linear regression model performs better.\n")
}
```

```
## The local polynomial regression model performs better.
```

The local polynomial regression seems to be a goo fit. And it outperforms the global linear model.