Stat 432 Homework 6

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?

```
# K(x) = 1/sqrt(2*pi) * exp(-x^2/2)
gaussian_kernel <- function(x) {
  return((1/sqrt(2 * pi)) * exp(-0.5 * x^2))
}

nadaraya_watson <- function(Xtrain, Ytrain, Xtest, h) {
  weights <- sapply(1:nrow(Xtrain), function(i) {
    kernel_x1 <- gaussian_kernel((Xtest[1] - Xtrain[i, 1]) / h)
    kernel_x2 <- gaussian_kernel((Xtest[2] - Xtrain[i, 2]) / h)
    return(kernel_x1 * kernel_x2)
  })

  weights <- weights / sum(weights)
  predicted_value <- sum(weights * Ytrain)
  return(predicted_value)
}

h <- 0.07
predicted_value <- nadaraya_watson(Xtrain, Ytrain, Xtest[1,], h)
predicted_value
```

```
## [1] 4.198552
```

Using a two-dimensional Nadaraya-Watson kernel regression estimator using the Gaussian kernel function with a bandwidth $h = 0.07$. The predicted value for the first testing observation is 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)
simulations <- 5000
h <- 0.07
beta <- c(1.5, 1.5)
```

```r
# True value for Xtest
Xtest <- matrix(runif(2), ncol = 2) # generate new test data
true_value <- exp(Xtest %*% beta)

# Initialize storage for predictions
predictions <- numeric(simulations)

# Run the simulation
for (i in 1:simulations) {
  # Generate new training data
  Xtrain <- matrix(runif(200 * 2), ncol = 2)
  Ytrain <- exp(Xtrain %*% beta) + rnorm(200)

  # Predict for the first testing observation
  predictions[i] <- nadaraya_watson(Xtrain, Ytrain, Xtest[1,], h)
}

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

# Compute Variance
variance <- mean((predictions - average_prediction)^2)

# Compute MSE
mse <- bias_squared + variance

# Output the results
cat("Bias^2:", bias_squared, "\n")
```

```
## Bias^2: 0.002130784
```

```r
cat("Variance:", variance, "\n")
```

```
## Variance: 0.09827509
```

```r
cat("MSE:", mse, "\n")
```

```
## MSE: 0.1004059
```

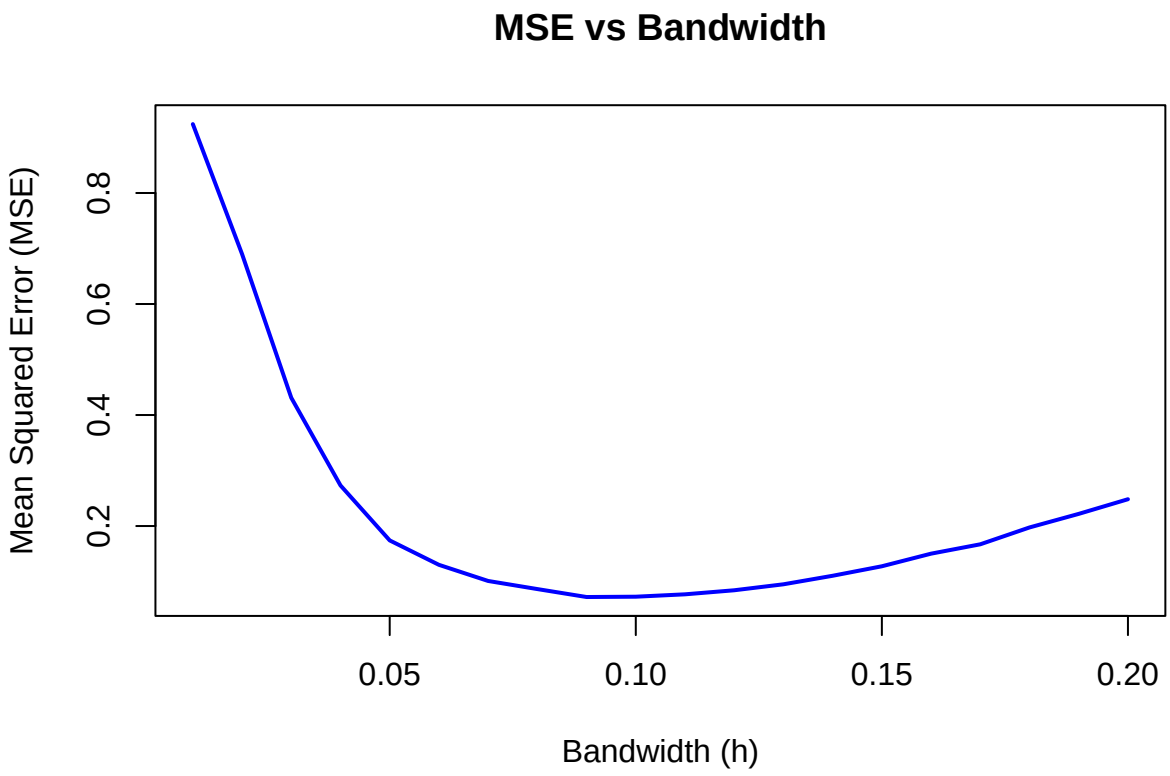
Yes, the MSE matches the theoretical understanding of the bias-variance trade-off. The MSE is the sum of Bias^2 and Variance, and in this case, the variance is significantly larger than the bias. This indicates that the model is slightly underfitting (low bias, higher variance), which is expected in kernel regression when using a smaller bandwidth.

Based on the bias-variance trade-off, to reduce the MSE, we should increase the bandwidth h. A smaller bandwidth results in higher variance (as seen in the simulation), while a larger bandwidth smooths the prediction, reducing variance but potentially increasing bias. Given that the current Bias^2 is small and the Variance is relatively large, increasing h would help reduce the MSE by lowering the variance more significantly than the potential increase in bias.

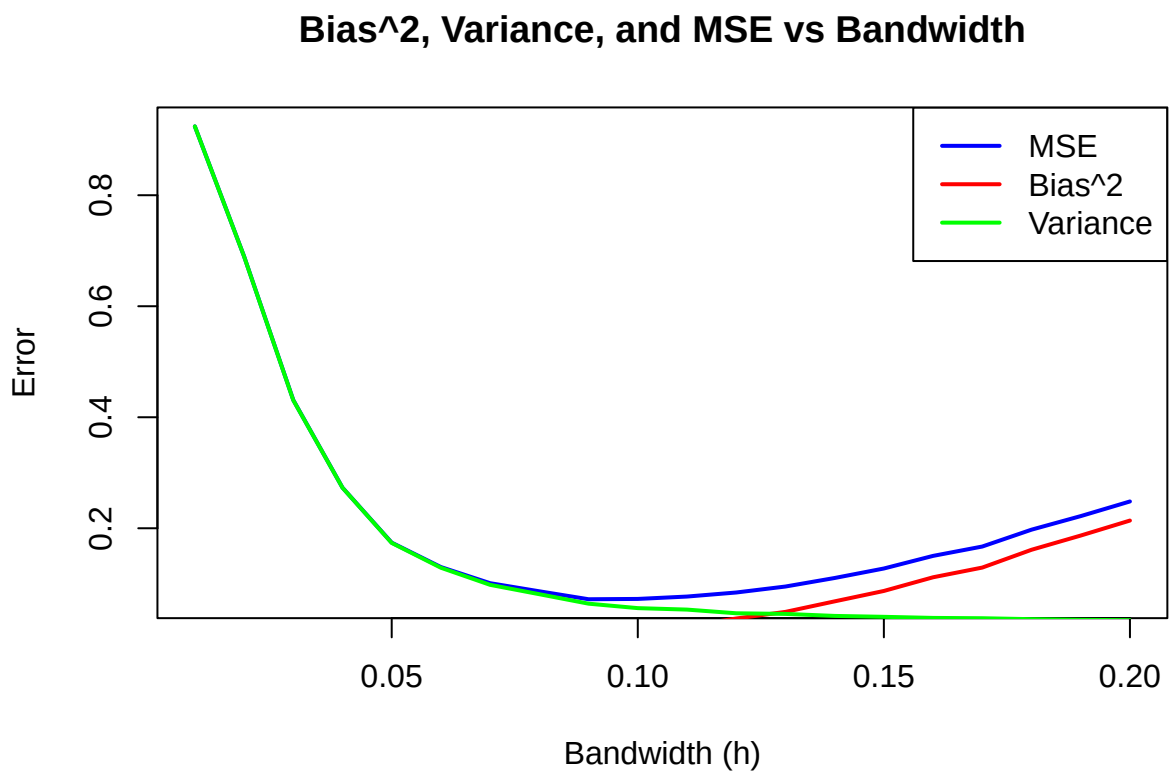
c. [15 pts] In practice, we will have to use cross-validation to select the optimal bandwidth. However, if you have the power of simulating as many datasets as you can, and you also know the true model, how would you find the optimal bandwidth for the bias-variance trade-off for this particular model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

```
set.seed(2)
simulations <- 5000
beta <- c(1.5, 1.5)
bandwidths <- seq(0.01, 0.2, by = 0.01)  # try different bandwidths


# True value for Xtest
Xtest <- matrix(runif(2), ncol = 2) # generate new test data
true_value <- exp(Xtest %*% beta)

# Initialize storage for results
results <- data.frame(bandwidth = bandwidths, Bias2 = numeric(length(bandwidths)),
                      Variance = numeric(length(bandwidths)), MSE = numeric(length(bandwidths)))

# Loop over different bandwidths
for (b in 1:length(bandwidths)) {
  h <- bandwidths[b]

  # Initialize storage for predictions
  predictions <- numeric(simulations)

  # Run the simulation
  for (i in 1:simulations) {
    # Generate new training data
    Xtrain <- matrix(runif(200 * 2), ncol = 2)
    Ytrain <- exp(Xtrain %*% beta) + rnorm(200)

    # Predict for the first testing observation
    predictions[i] <- nadaraya_watson(Xtrain, Ytrain, Xtest[1,], h)
  }

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

  # Compute Variance
  variance <- mean((predictions - average_prediction)^2)

  # Compute MSE
  mse <- bias_squared + variance

  # Store results
  results$Bias2[b] <- bias_squared
  results$Variance[b] <- variance
  results$MSE[b] <- mse
}

# Plot MSE vs. bandwidth
plot(results$bandwidth, results$MSE, type = "l", col = "blue", lwd = 2,
```

```
     xlab = "Bandwidth (h)", ylab = "Mean Squared Error (MSE)", main = "MSE vs Bandwidth")
```




```
# Plot Bias^2, Variance, and MSE on the same plot
plot(results$bandwidth, results$MSE, type = "l", col = "blue", lwd = 2,
     xlab = "Bandwidth (h)", ylab = "Error", main = "Bias^2, Variance, and MSE vs Bandwidth")
lines(results$bandwidth, results$Bias2, col = "red", lwd = 2)  # Add Bias^2 curve
lines(results$bandwidth, results$Variance, col = "green", lwd = 2)  # Add Variance curve

# Add a legend to differentiate the lines
legend("topright", legend = c("MSE", "Bias^2", "Variance"), col = c("blue", "red", "green"), lty = 1, lw
```




```r
# Find the optimal bandwidth
optimal_bandwidth <- results$bandwidth[which.min(results$MSE)]
cat("Optimal bandwidth:", optimal_bandwidth, "\n")
```

```
## Optimal bandwidth: 0.09
```

The optimal bandwidth h for the Nadaraya-Watson kernel regression estimator was found by simulating multiple datasets and calculating the bias, variance, and mean squared error (MSE) for a range of bandwidth values. Based on the simulation results, the bandwidth that minimized the MSE was $h = 0.09$. This result aligns with the theoretical understanding of the bias-variance trade-off, where smaller bandwidths tend to have lower bias but higher variance, and larger bandwidths have the opposite effect. By finding the point where the MSE is minimized, we ensure the best balance between bias and variance for this model.

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.

```r
  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.

```r
# Define the Gaussian kernel function for multivariate data
gaussian_kernel <- function(x0, x, h) {
  # x0: test point (vector)
  # x: matrix of training points
  # h: bandwidth
  distances <- sqrt(rowSums((x - matrix(x0, nrow = nrow(x), ncol = length(x0), byrow = TRUE))^2))
  weights <- exp(- (distances^2) / (2 * h^2)) / (h * sqrt(2 * pi))
  return(weights)
}

# Bandwidth
# h <- 0.2

# Test point
x0 <- Xtest[1, ]

# Compute the kernel weights for the test point x0 against all training points Xtrain
weights <- gaussian_kernel(x0, Xtrain, h)

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

```
##        25%        50%        75%
## 0.06303223 0.25435682 0.57483065
```

To calculate the kernel weights of x_0 against all the training data points, we used a Gaussian kernel function. This function computes the distance between the test point x_0 and each training point, then applies a Gaussian weighting with a bandwidth of $h = 0.2$. The 25th, 50th, and 75th percentiles of the calculated weights are 0.06303223, 0.25435682, and 0.57483065, respectively. These percentiles help assess the spread of the kernel weights across the training data.

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.

$$(y - X\beta_{x_0})^T W (y - X\beta_{x_0}),$$

where:

- y is the vector of training outputs.
- X is the matrix of training inputs (augmented with polynomial terms if needed for local polynomial regression).
- β_{x_0} is the local regression coefficient at point x_0.
- W is the diagonal matrix of kernel weights, with the i-th diagonal element being $K_h(x_0, x_i)$.

The objective function can be rewritten as:

$$L(\beta_{x_0}) = (y - X\beta_{x_0})^T W (y - X\beta_{x_0})$$

Differentiating $L(\beta_{x_0})$ with respect to β_{x0} gives:

$$\frac{\partial L(\beta_{x_0})}{\partial \beta_{x_0}} = -2X^T W (y - X\beta_{x_0})$$

Setting the derivative to zero gives the normal equation:

$$X^T W X \beta_{x_0} = X^T W y$$

Then try to solve for β_{x_0}:

$$\beta_{x_0} = (X^T W X)^{-1} X^T W y$$

This is the normal equation for estimating β_{x_0} in local polynomial regression coefficients at the target 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.

```
# Bandwidth
h <- 0.1

# Select the test point x0
x0 <- Xtest[1, ]

# Augment the design matrices to include the intercept (column of 1's)
X_train <- cbind(1, Xtrain)
X_test <- cbind(1, Xtest[1, , drop=FALSE])  # Test point as a row vector

# Compute the kernel weights using the Gaussian kernel function
gaussian_kernel <- function(x0, x, h) {
  # Calculate the distance between x0 and each point in x
  distance <- sqrt(rowSums((x - matrix(x0, nrow = nrow(x), ncol = length(x0), byrow = TRUE))^2))
  # Compute the Gaussian kernel weights
  weights <- exp(- (distance^2) / (2 * h^2)) / (h * sqrt(2 * pi))
  return(weights)
}

# Compute the kernel weights for the test point x0 against all training points
```

```r
W_diag <- gaussian_kernel(x0, Xtrain, h)
W <- diag(W_diag)

# Compute X^T W X and X^T W Y
XtWX <- t(X_train) %*% W %*% X_train
XtWy <- t(X_train) %*% W %*% Ytrain

# Solve for beta_x0 (the local regression coefficients at x0)
beta_x0 <- solve(XtWX, XtWy)

# Make the prediction at x0 using the augmented test point matrix
y_hat_x0 <- X_test %*% beta_x0

# Compute the true expectation at x0
true_y_x0 <- exp(sum(x0 * beta))

# Print results
print(list(beta_x0 = beta_x0, prediction = y_hat_x0, true_expectation = true_y_x0))
```

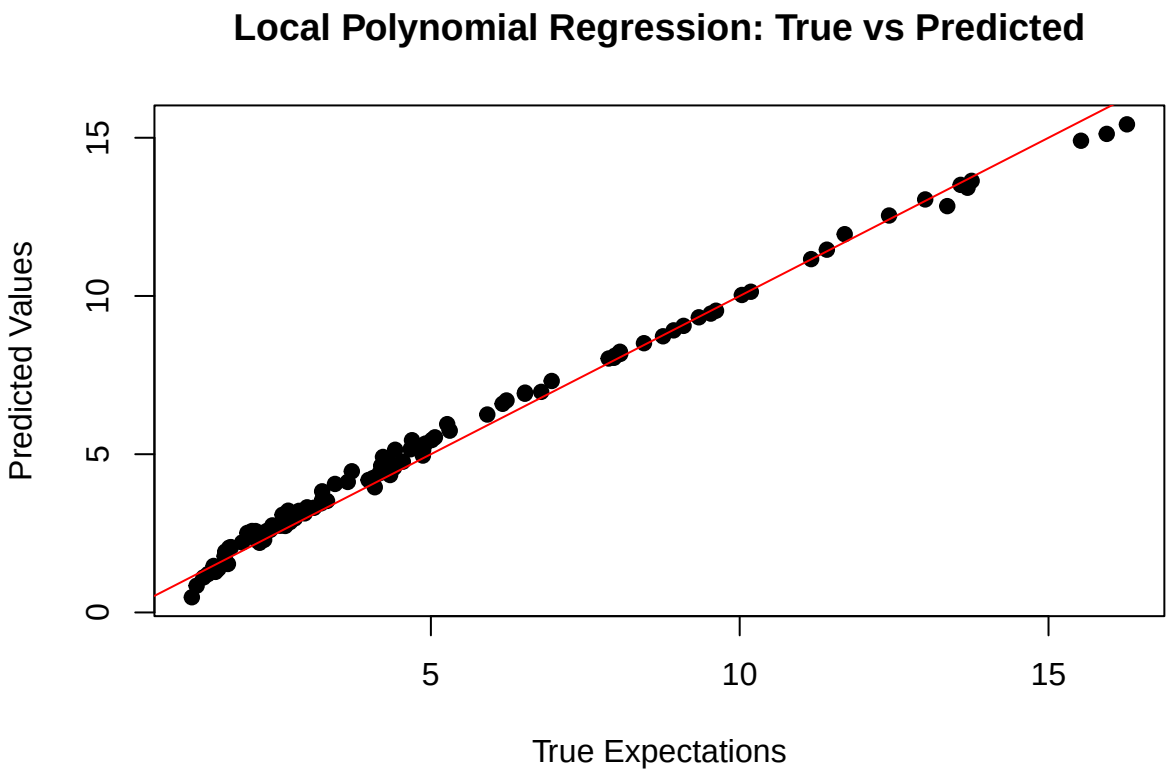
```
## $beta_x0
##           [,1]
## [1,] -2.190675
## [2,]  7.374825
## [3,]  6.497026
##
## $prediction
##          [,1]
## [1,] 4.324448
##
## $true_expectation
## [1] 4.137221
```

The estimated β_{x_0} for the testing point X_{test} was calculated using the provided data and a bandwidth of $h = 0.1$. The resulting estimates for β_{x_0} were -2.190675, 7.374825 and 6.497026. Using these estimated coefficients, the predicted value at the test point x_0 was computed to be 4.324448. In comparison, the true expectation at the test point, based on the known model, was 4.137221. The prediction is close to the true expectation, indicating that the local polynomial regression model performs well in approximating the target output.

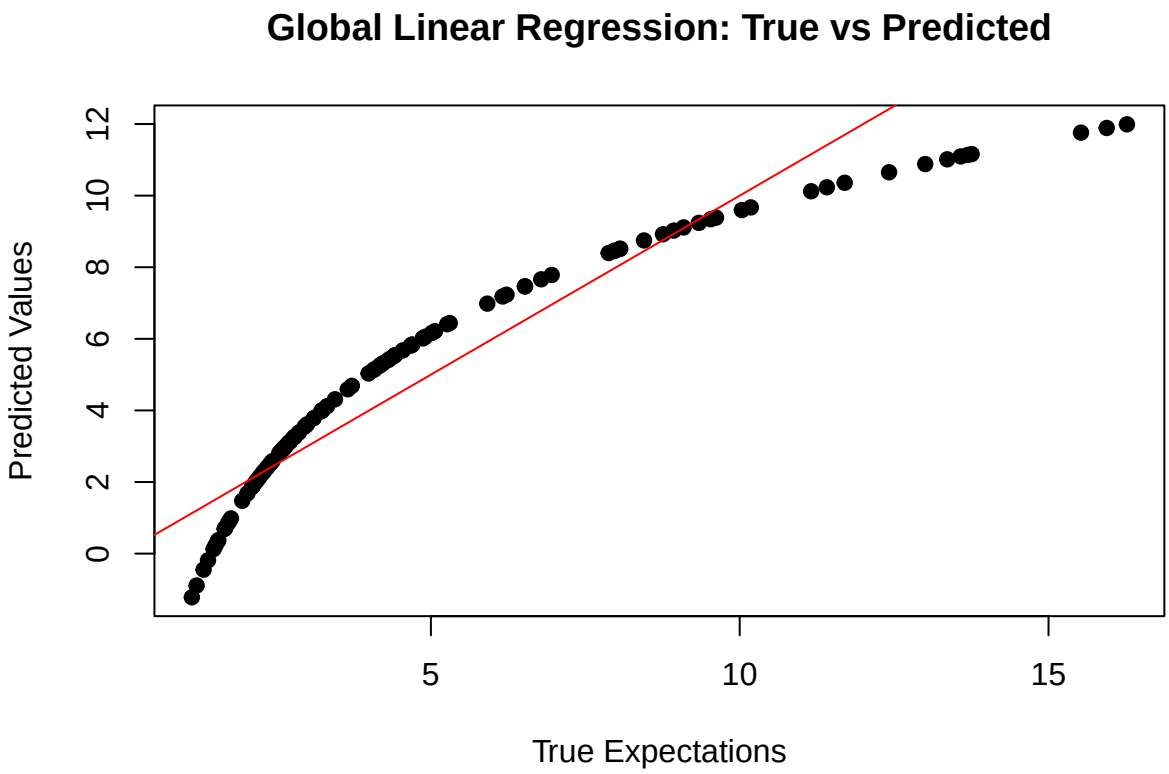
d. [20 pts] Now, let's use this model to predict the following 100 testing points. After you fit the model, provide a scatter plot of the true expectation versus the predicted values on these testing points. Does this seem to be a good fit? As a comparison, fit a global linear regression model to the 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.

```r
# Bandwidth
# h <- 0.1

testn <- 100
Xtest <- matrix(runif(testn * p), ncol = p)

# Initialize vectors for predicted values and true expectations
```

```r
predicted_values <- numeric(testn)
true_expectations <- numeric(testn)

# Loop over each testing point to apply the local regression model
for (i in 1:testn) {
  x0 <- Xtest[i, ]

  # Compute the kernel weights for each test point
  W_diag <- gaussian_kernel(x0, Xtrain, h)
  W <- diag(W_diag)

  # Compute X^T W X and X^T W y
  XtWX <- t(Xtrain) %*% W %*% Xtrain
  XtWy <- t(Xtrain) %*% W %*% Ytrain

  # Solve for beta_x0
  beta_x0 <- solve(XtWX, XtWy)

  # Make the prediction at x0
  predicted_values[i] <- sum(x0 * beta_x0)

  # Compute the true expectation for x0
  true_expectations[i] <- exp(sum(x0 * beta))
}

# Fit a global linear regression model using training data
Xtrain_df <- as.data.frame(Xtrain)
Xtest_df <- as.data.frame(Xtest)
Ytrain_df <- data.frame(Ytrain = Ytrain)

global_model <- lm(Ytrain ~ ., data = cbind(Ytrain_df, Xtrain_df))

# Predict the values using the global model
global_predictions <- predict(global_model, newdata = Xtest_df)

# Plot true expectations vs. predicted values for local regression
plot(true_expectations, predicted_values,
     main = "Local Polynomial Regression: True vs Predicted",
     xlab = "True Expectations", ylab = "Predicted Values", pch = 19, col = "black")
abline(0, 1, col = "red")
```




```r
# Plot true expectations vs. predicted values for global linear model
plot(true_expectations, global_predictions,
     main = "Global Linear Regression: True vs Predicted",
     xlab = "True Expectations", ylab = "Predicted Values", pch = 19, col = "black")
abline(0, 1, col = "red")
```



In comparing the performance of the local polynomial regression model and the global linear regression model for predicting 100 test points, the first plot illustrates the predicted values from the local polynomial regression against the true expectations. The points align closely along the red line, indicating a strong agreement between the predictions and the true values. This suggests that the local model provides a good fit to the data.

In contrast, the second plot displays the results of the global linear regression model. Here, the predictions deviate significantly from the true values, particularly for lower and higher expectations, leading to a curved pattern. This indicates that the global linear model does not fit the data as well as the local polynomial regression, which better captures the underlying relationships. Overall, the local linear model outperforms the global model, as evidenced by its closer alignment with the true expectations.