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 with full Gaussian kernel
nadaraya_watson <- function(Xtrain, Ytrain, Xtest, h) {
  gaussian_kernel <- function(x, xi, h) {
    return((1 / (h * sqrt(2 * pi))) * 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 <- gaussian_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 the kernel regression with the full Gaussian kernel formula
simulate_kernel_regression <- function(trainn, testn, p, beta, h, B) {
  bias_squared <- 0
  variance <- 0
```

```r
  mse <- 0
  true_value <- exp(Xtest %*% beta)

  # Gaussian kernel function
  gaussian_kernel <- function(x, xi, h) {
    return((1 / (h * sqrt(2 * pi))) * exp(-sum((x - xi)^2) / (2 * h^2)))
  }

  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 using Gaussian kernel
    num <- 0
    denom <- 0
    for (i in 1:nrow(Xtrain)) {
      weight <- gaussian_kernel(Xtest, Xtrain[i, ], h)
      num <- num + weight * Ytrain[i]
      denom <- denom + weight
    }
    predictions[b] <- num / denom
  }

  # 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))
}

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

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

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

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

```
## Variance:  0.09560785
```

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

```
## MSE:  0.09787273
```

```r
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.
- 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 behaviour.

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.

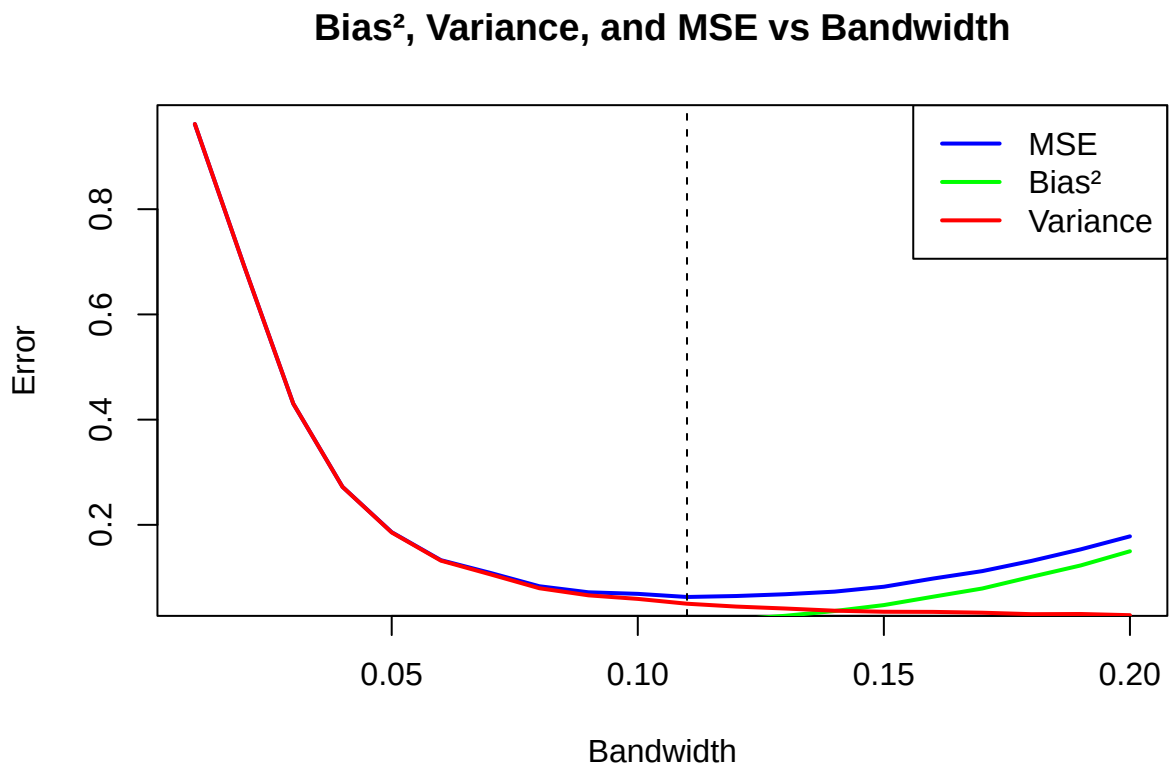
```r
# Extend the simulate_kernel_regression function to return bias_squared and variance
find_optimal_bandwidth_with_decomposition <- function(h_values, trainn, testn, p, beta, B) {
  mse_values <- numeric(length(h_values))
  bias_squared_values <- numeric(length(h_values))
  variance_values <- numeric(length(h_values))

  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$mse
    bias_squared_values[i] <- res$bias_squared
    variance_values[i] <- res$variance
  }

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

# Test a range of bandwidths
h_values <- seq(0.01, 0.2, by = 0.01)
optimal_bandwidth_decomposition <- find_optimal_bandwidth_with_decomposition(h_values, trainn = 200,
                                                                             testn = 1, p = 2, beta = be

# Output the optimal bandwidth
cat("Optimal Bandwidth:", optimal_bandwidth_decomposition$optimal_h, "\n")
```

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

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

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

```r
# Plot MSE vs Bandwidth with Bias^2 and Variance
plot(h_values, optimal_bandwidth_decomposition$mse_values, type = "l", col = "blue", lwd = 2,
     xlab = "Bandwidth", ylab = "Error", main = "Bias², Variance, and MSE vs Bandwidth")
lines(h_values, optimal_bandwidth_decomposition$bias_squared_values, col = "green", lwd = 2)
lines(h_values, optimal_bandwidth_decomposition$variance_values, col = "red", lwd = 2)
legend("topright", legend = c("MSE", "Bias²", "Variance"), col = c("blue", "green", "red"), lty = 1, lwd
# Add a vertical line for the optimal bandwidth
abline(v = optimal_bandwidth_decomposition$optimal_h, col = "black", lty = 2)
```



Based on the provided output and analysis of the bias-variance trade-off, the optimal bandwidth for this kernel regression model is 0.11, which minimizes the mean squared error (MSE) to 0.06293818.

The optimal bandwidth was found by balancing bias and variance. Small bandwidths lead to high variance and low bias, as the model becomes too sensitive to noise and overfits the data. Larger bandwidths result in high bias and low variance, where the model oversmooths the data and fails to capture important patterns (underfitting).

At the optimal bandwidth of 0.11, the MSE is minimized, providing the best trade-off between underfitting and overfitting. This demonstrates that the model at this bandwidth captures the true data patterns well without being overly sensitive to noise.

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 Gaussian kernel function
gaussian_kernel <- function(x0, xi, h) {
  return((1 / (h * sqrt(2 * pi))) * exp(-((x0 - xi)^2) / (2 * h^2)))
}

# Define bandwidth
h <- 0.07

# Calculate kernel weights for each training data point against the test point
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)
}
# 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))
print(quantiles)
```

```
##          25%          50%          75%
## 8.141912e-07 8.735899e-03 1.028260e+00
```

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.

```r
# Define Gaussian kernel function
gaussian_kernel <- function(x0, xi, h) {
  return((1 / (h * sqrt(2 * pi))) * exp(-sum((x0 - xi)^2) / (2 * h^2)))
}

# Calculate the kernel weights for the training points relative to the test point
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)
}

# Define the bandwidth
h <- 0.07

# Generate the design matrix with an intercept term
Xtrain_augmented <- cbind(1, Xtrain)  # Adding a column of 1s for the intercept

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

# Create the diagonal weight matrix W
W <- diag(weights)

# Estimate beta_x0 using the normal equation
beta_x0 <- solve(t(Xtrain_augmented) %*% W %*% Xtrain_augmented) %*%
  t(Xtrain_augmented) %*% W %*% Ytrain

# Report the estimated beta_x0
cat("Estimated beta_x0: ", beta_x0, "\n")
```

```
## Estimated beta_x0:  -1.749993 5.690594 6.870116
```

```
# Predict the response at the test point
Xtest_augmented <- c(1, Xtest)  # Add an intercept term for the test point
predicted_value <- sum(Xtest_augmented * beta_x0)

# Calculate the true expectation for the test point
true_value <- exp(Xtest %*% beta)

# Compare the predicted value with the true expectation
cat("Predicted value: ", predicted_value, "\n")
```

```
## Predicted value:  4.264039
```

```
cat("True value: ", true_value, "\n")
```

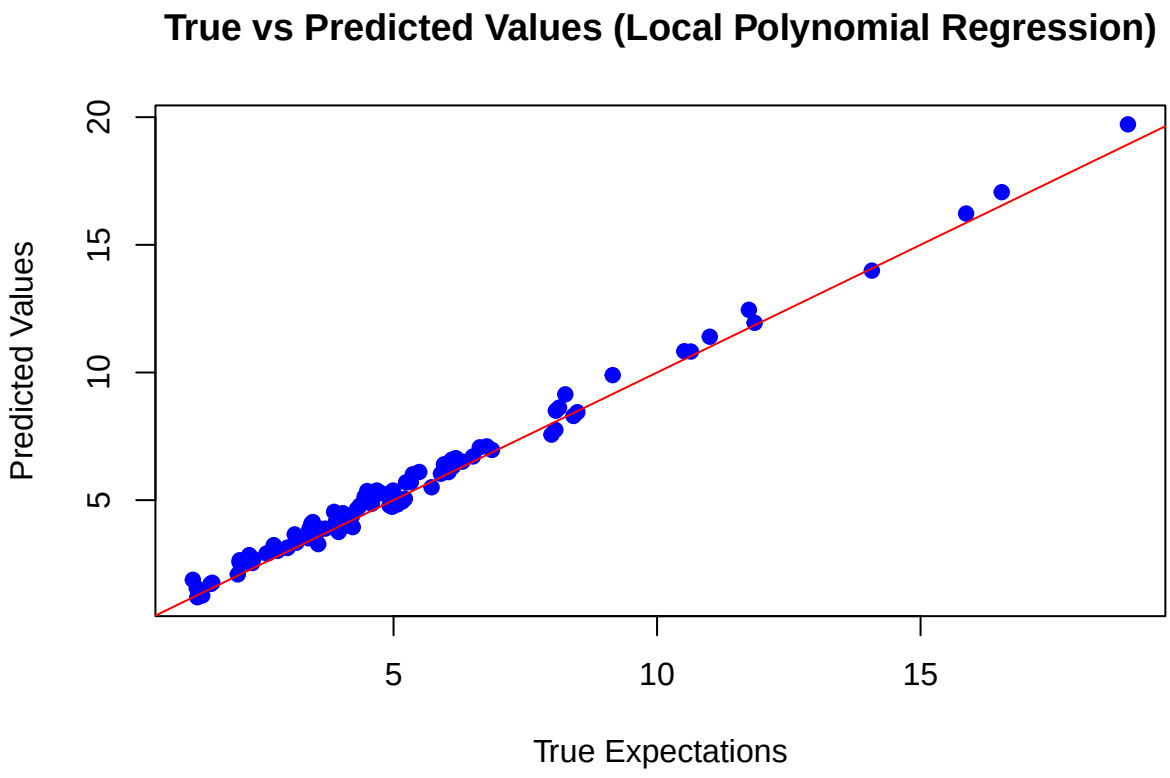
```
## True value:  4.137221
```

The predicted value 4.264039 is very close to the true value 4.137221, with a small difference of approximately: Difference=4.264039-4.137221=0.126818. This small difference suggests that the local polynomial regression model is performing well in predicting the response at the test point x_0. The slight difference can be attributed to the approximation errors inherent in the kernel regression process as well as the influence of the noise ϵ in the observed data.

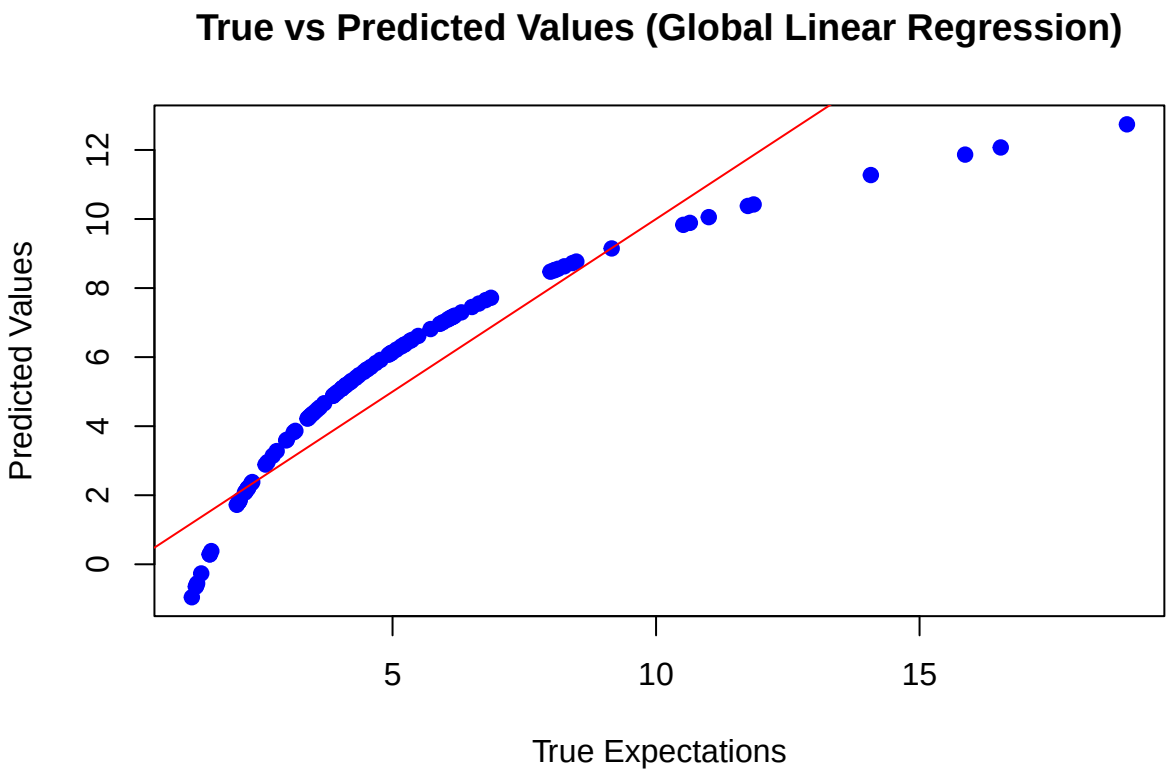
d. [20 pts] Now, let's use this model to predict the following 100 testing points. After you fit the model, provide a scatter plot of the true expectation versus the predicted values on these testing points. Does

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
set.seed(432)
testn <- 100
Xtest <- matrix(runif(testn * p), ncol = p)
# Function to predict using local polynomial regression
predict_local_polynomial <- function(Xtrain, Ytrain, Xtest, h) {
  predictions <- numeric(nrow(Xtest))
  for (i in 1:nrow(Xtest)) {
    # Calculate kernel weights for the i-th test point
    weights <- calculate_kernel_weights(Xtrain, Xtest[i, ], h)
    W <- diag(weights)
    Xtrain_augmented <- cbind(1, Xtrain)  # Augment training data with intercept
    beta_x0 <- solve(t(Xtrain_augmented) %*% W %*% Xtrain_augmented) %*%
      t(Xtrain_augmented) %*% W %*% Ytrain
    Xtest_augmented <- c(1, Xtest[i, ])  # Add intercept for test point
    predictions[i] <- sum(Xtest_augmented * beta_x0)  # Predict using beta
  }
  return(predictions)
}

# Predict using the local polynomial regression model for 100 test points
h <- 0.07
local_predictions <- predict_local_polynomial(Xtrain, Ytrain, Xtest, h)

# Calculate the true expectations for the 100 test points
true_values <- exp(Xtest %*% beta)

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

# Predict the responses for the 100 test points using the global model
Xtest_df <- data.frame(Xtest)
global_predictions <- predict(global_model, newdata = Xtest_df)

# Scatter plot of true vs predicted values (Local Polynomial Regression)
plot(true_values, local_predictions, xlab = "True Expectations", 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 of true vs predicted values (Global Linear Regression)
plot(true_values, global_predictions, xlab = "True Expectations", ylab = "Predicted Values",
     main = "True vs Predicted Values (Global Linear Regression)", col = "blue", pch = 19)
abline(0, 1, col = "red")  # Line of perfect prediction
```




```
# Compare model performance by calculating the Mean Squared Error (MSE)
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
```

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

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

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

The local polynomial regression model clearly outperforms the global linear regression model in terms of prediction accuracy, as indicated by the significantly lower MSE (0.1534 compared to 1.7513).

Additionally, from the scatter plots:

- The local polynomial regression plot shows points closely aligned along the red line of perfect prediction, indicating that the local model makes accurate predictions across a wide range of values.
- The global linear regression plot, however, shows a clear curvature, meaning that the global model struggles to capture the nonlinear relationship in the data, particularly for larger values of the true expectation. Thus, the local polynomial regression model is more suitable for this dataset and performs better in comparison to the global linear regression model.