HW06_zilinw3

Zilin Wang (zilinw3)

2024-10-08

Contents

Question 1: Multivariate Kernel Regression Simulation (45 pts) 1
Question 2: Local Polynomial Regression (55 pts) 5

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:

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

# 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
```

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

```
##          [,1]
## [1,] 2.577147
```

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?

Answer:

```r
# Define the Gaussian kernel function
gaussian_kernel <- function(x, x_i, h) {
  sum(exp(-0.5 * (x - x_i)^2 / h^2) / sqrt(2 * pi) / h)
}

# Nadaraya-Watson estimator function
nadaraya_watson <- function(x, X, Y, h) {
  weights <- apply(X, 1, gaussian_kernel, x = x, h = h)
  weighted_Y <- sum(weights * Y)
  sum_weights <- sum(weights)
  if (sum_weights == 0) {
    return(0)
  } else {
    return(weighted_Y / sum_weights)
  }
}

# Predicting the first test observation
h <- 0.07
predicted_value <- nadaraya_watson(Xtest, Xtrain, Ytrain, h)
cat("Prediction of the first testing observation:", predicted_value, "\n")
```

```
## Prediction of the first testing observation: 3.027713
```

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?

Answer:

```r
set.seed(2)

# Simulation parameters
n_sim <- 5000
predictions <- numeric(n_sim)
true_values <- exp(Xtest %*% beta)

# Running the simulation
for (i in 1:n_sim) {
  Xtrain <- matrix(runif(trainn * p), ncol = p)
  Ytrain <- exp(Xtrain %*% beta) + rnorm(trainn)

  predictions[i] <- nadaraya_watson(Xtest, Xtrain, Ytrain, h)
}

# Calculating bias^2, variance, and MSE
bias2 <- (mean(predictions) - true_values)^2
variance <- var(predictions)
true_values_scalar <- as.numeric(true_values)
mse <- mean((predictions - true_values_scalar)^2)

list(bias_squared = bias2, variance = variance, MSE = mse)
```

```
## $bias_squared
##            [,1]
## [1,] 0.03645077
##
## $variance
## [1] 0.01972173
##
## $MSE
## [1] 0.05616855
```

```r
cat("Bias^2 + Variance:", bias2 + variance, "\n")
```

```
## Bias^2 + Variance: 0.0561725
```

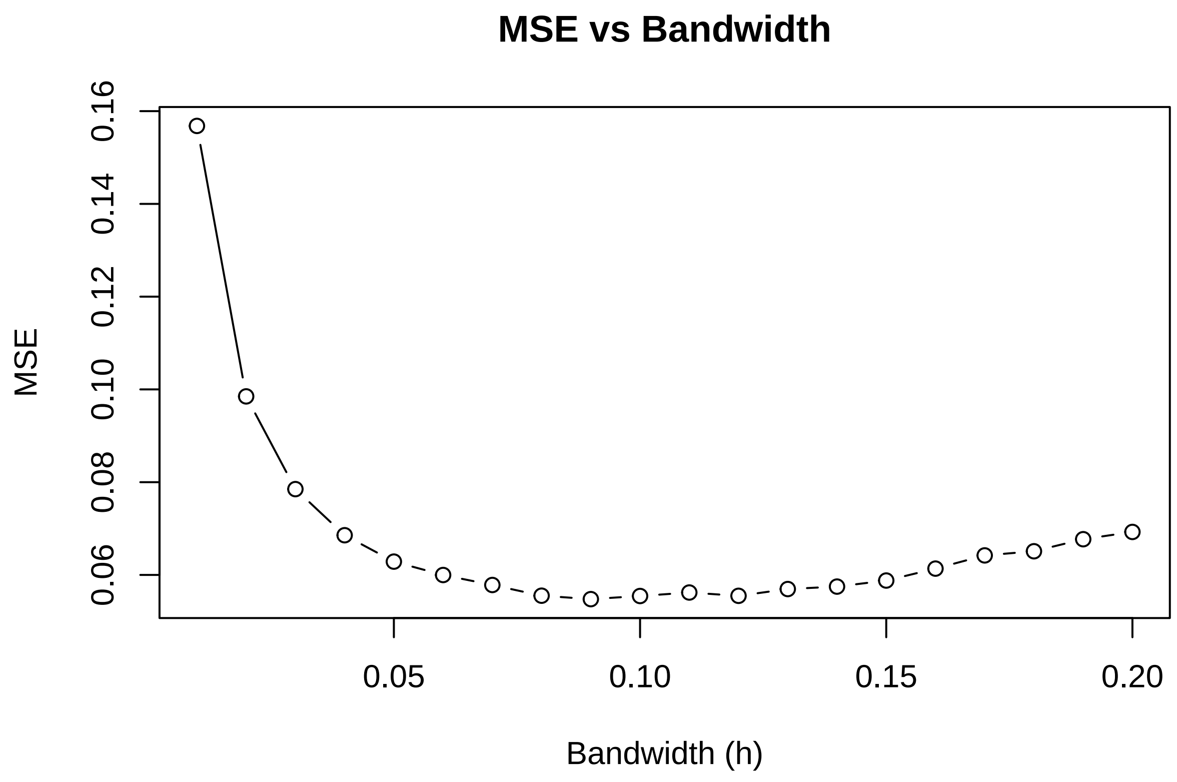
The theoretical understanding of the bias-variance trade-off tells us that MSE can be decomposed into bias squared and variance, where: MSE = bias^2 + variance. From our results above, Bias^2 + Variance = 0.0561725, which is very close to the reported MSE of 0.05616855, suggesting a slight numerical precision issue or rounding but generally confirms our theoretical understanding.

Since the bias squared (0.03645077) is larger than the variance (0.01972173), the model is likely over-smoothing (underfitting). Therefore, decreasing the bandwidth h may reduce MSE. This adjustment should help in reducing the bias component of the MSE, potentially leading to a better overall prediction error, provided the increase in variance does not counterbalance the gain from reduced bias too much.

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.

Answer:

```r
# Define a sequence of bandwidths to test
bandwidths <- seq(0.01, 0.2, by = 0.01)

# Initialize vectors to store the results
mse_results <- numeric(length(bandwidths))

# Number of simulations
n_sim <- 5000
set.seed(2)

# Simulation
for (i in 1:length(bandwidths)) {
  h <- bandwidths[i]
  predictions <- numeric(n_sim)
  for (j in 1:n_sim) {
    Xtrain <- matrix(runif(trainn * p), ncol = p)
    Ytrain <- exp(Xtrain %*% beta) + rnorm(trainn)

    # Predict test point
    predictions[j] <- nadaraya_watson(Xtest, Xtrain, Ytrain, h)
  }

  # Calculate MSE
  mse_results[i] <- mean((predictions - as.numeric(true_values))^2)
}

# Plot MSE vs bandwidth
plot(bandwidths,
     mse_results,
     type = 'b',
     xlab = "Bandwidth (h)",
     ylab = "MSE",
     main = "MSE vs Bandwidth")
```



```r
# Select the optimal bandwidth
optimal_h <- bandwidths[which.min(mse_results)]
cat("The optimal bandwidth:", optimal_h, "\n")
```

```
## The optimal bandwidth: 0.09
```

To find the optimal bandwidth for the bias-variance trade-off, we can begin by defining a range of bandwidths to test. This range should be broad enough to capture the point at which increasing the bandwidth begins to detrimentally increase the bias more than it reduces the variance. Then, for each bandwidth, simulate a large number of datasets. Also for each simulated dataset, use the Nadaraya-Watson model to predict values and calculate the MSE. Then, plot the calculated MSE against the bandwidth values.

Finally, choose the bandwidth with the lowest MSE across all tested bandwidths, which balances the bias-variance trade-off effectively. Also, it is the minimum points in the MSE vs Bandwidth plot. Therefore, the selected bandwidth(h = 0.09) is (almost) optimal.

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

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

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.

Answer:

```r
# Define the Gaussian kernel function
gaussian_kernel <- function(x, x_i, h) {
  sum(exp(-0.5 * (x - x_i)^2 / h^2) / sqrt(2 * pi) / h)
}

# Bandwidth
h <- 0.07

# Calculate weights for Xtest against all Xtrain
weights <- apply(Xtrain, 1, gaussian_kernel, x = Xtest, h = h)

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

```
##         25%         50%         75%
## 0.004307721 0.502557115 3.257525715
```

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 β_{x_0}. Write your answer in latex.

Answer:

For the local polynomial regression model, the objective function we're minimizing is:

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

where W is a diagonal matrix of weights $K_h(x_0, x_i)$ for each training point x_i.

The normal equation for estimating β_{x_0} can be derived as follows, leading to the least squares estimate:

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

Thus, the estimated β_{x_0} is given by:

$$\hat{\beta}_{x_0} = (X^T W X)^{-1} X^T W y$$

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

Answer:

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

# Estimate beta_x0
beta_x0 <- solve(t(Xtrain) %*% W %*% Xtrain) %*% (t(Xtrain) %*% W %*% Ytrain)
cat("The estimate beta_x0:", beta_x0, "\n")
```

```
## The estimate beta_x0: 6.020964 4.644229
```

```
# Prediction for the test point
predicted_value <- as.numeric(Xtest %*% beta_x0)
true_value <- exp(Xtest %*% beta)

# Compare the prediction with the true expectation
list(prediction = predicted_value, 'true expectation' = true_value)
```

```
## $prediction
## [1] 4.968292
##
## $`true expectation`
##          [,1]
## [1,] 4.137221
```

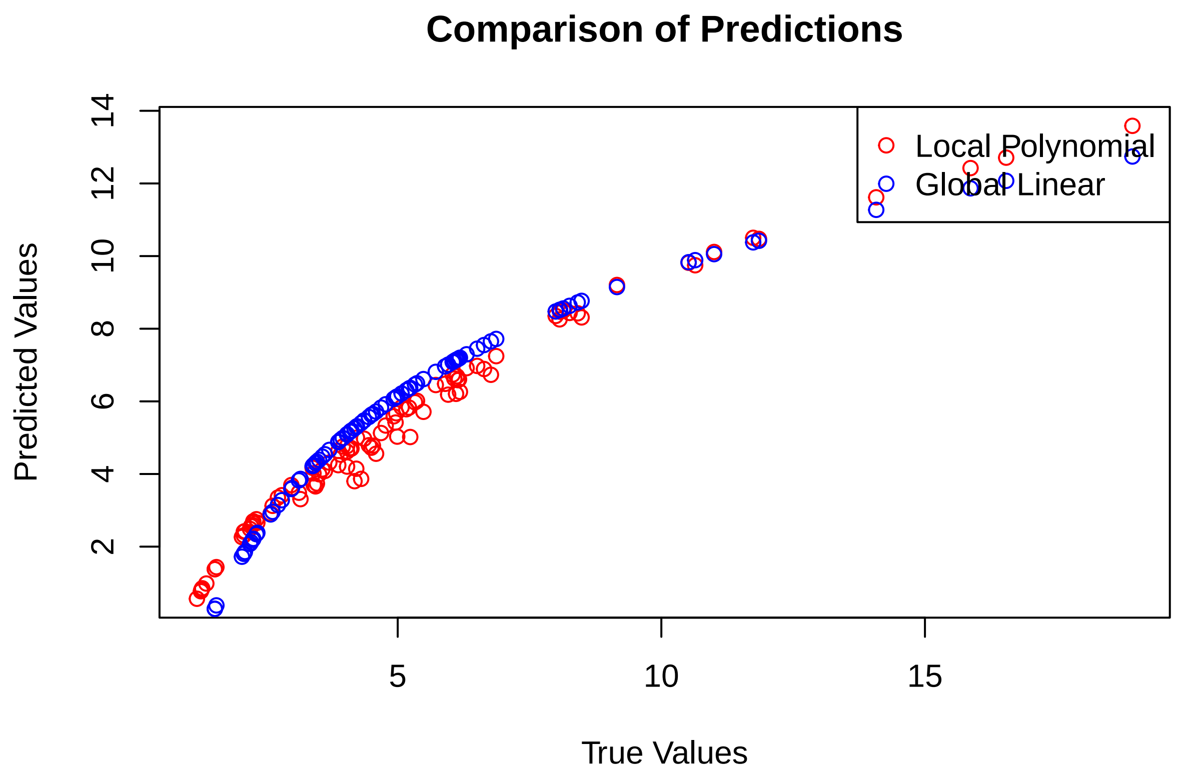
The prediction on the testing point is slightly larger than the true expectation.

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

Answer:

```r
# Predict using local polynomial regression
local_predictions <- apply(Xtest, 1, function(x_new) {
  weights <- apply(Xtrain, 1, gaussian_kernel, x = x_new, h = h)
  W <- diag(weights)
  beta_x0 <- solve(t(Xtrain) %*% W %*% Xtrain) %*% (t(Xtrain) %*% W %*% Ytrain)
  return(as.numeric(x_new %*% beta_x0))
})

# True values
true_values <- exp(Xtest %*% beta)

# Global linear regression model
train_df <- as.data.frame(cbind(Ytrain, Xtrain))
colnames(train_df) <- c("Y", "V1", "V2")
test_df <- as.data.frame(Xtest)
colnames(test_df) <- c("V1", "V2")
global_model <- lm(Y ~ ., data = train_df)
global_predictions <- predict(global_model, newdata = test_df)

# Scatter plot of true expectations vs. predicted values for local model
plot(true_values,
     local_predictions,
     col = 'red',
     main = "Comparison of Predictions",
     xlab = "True Values",
     ylab = "Predicted Values")

# Add global model predictions
points(true_values, global_predictions, col = 'blue')
legend("topright", legend=c("Local Polynomial", "Global Linear"), col=c("red", "blue"), pch=1)
```



```
# Evaluate which model performs better
mse_local <- mean((local_predictions - true_values)^2)
mse_global <- mean((global_predictions - true_values)^2)
cat("The MSE for local model:", mse_local, "\n")
```

```
## The MSE for local model: 0.8663492
```

```
cat("The MSE for global model:", mse_global, "\n")
```

```
## The MSE for global model: 1.751315
```

The local linear model appears to be a good fit, especially for lower values, as indicated by both the visual alignment in the scatter plot and the lower MSE.

The local linear model outperforms the global linear model because the MSE for the local linear model(0.8663492) is smaller than the MSE for the global linear model(1.751315).