CS 1675 Spring 2022 Homework: 07

Assigned March 16, 2022; Due: March 23, 2022

Sameera Boppana

Submission time: March 23, 2022 at 11:00PM EST

Collaborators

Include the names of your collaborators here.

Jeffery Janotka

Overview

This homework assignment is focused on working with linear models. You will fit, make predictions with, and assess linear model performance. You will calculate errors on training and test sets, as well as evaluate performance using the log-Evidence (log marginal likelihood).

IMPORTANT: code chunks are created for you. Each code chunk has `eval=FALSE` set in the chunk options. You **MUST** change it to be `eval=TRUE` in order for the code chunks to be evaluated when rendering the document.

You are allowed to add as many code chunks as you see fit to answer the questions.

Load packages

This assignment will use packages from the `tidyverse` suite.

```
library(tidyverse)
```

```
## ── Attaching packages ─────────────────────────────────────── tidyverse 1.3.1 ──
```

```
## ✓ ggplot2 3.3.5     ✓ purrr   0.3.4
## ✓ tibble  3.1.6     ✓ dplyr   1.0.8
## ✓ tidyr   1.2.0     ✓ stringr 1.4.0
## ✓ readr   2.1.2     ✓ forcats 0.5.1
```

```
## Warning: package 'tidyr' was built under R version 4.0.5
```

```
## Warning: package 'readr' was built under R version 4.0.5
```

```
## Warning: package 'dplyr' was built under R version 4.0.5
```

```
## ── Conflicts ─────────────────────────────────────────── tidyverse_conflicts() ──
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

This assignment also uses the `splines` and `MASS` packages. Both are installed with base `R` and so you do not need to download any additional packages to complete the assignment.

Problem 01

This problem is focused on setting up the log-posterior function for a linear model. You will program the function using matrix math, such that you can easily scale your code from a linear relationship with a single input up to complex linear basis function models. You will assume independent Gaussian priors on all $\boldsymbol{\beta}$-parameters with a shared prior mean μ_β and shared prior standard deviation, τ_β. An Exponential prior with rate parameter λ will be assumed for the likelihood noise, σ. The complete probability model for the response, y_n , is shown below using the linear basis notation. The n-th row of the basis design matrix, $\boldsymbol{\Phi}$ is denoted as $\boldsymbol{\phi}_{n,:}$. It is assumed that the basis is of degree-of-freedom J.

$$y_n \mid \mu_n, \sigma \sim \text{normal}\left(y_n \mid \mu_n, \sigma\right)$$

$$\mu_n = \boldsymbol{\phi}_{n,:}\boldsymbol{\beta}$$

$$\boldsymbol{\beta} \mid \mu_\beta, \tau_\beta \sim \prod_{j=0}^{J} \left(\text{normal}\left(\beta_j \mid \mu_\beta, \tau_\beta\right)\right)$$

$$\sigma \mid \lambda \sim \text{Exp}\left(\sigma \mid \lambda\right)$$

1a)

The code chunk below reads in a data set consisting of two variables, an input `x` and a response `y` . As shown by the `glimpse()` of the data set, there are 50 observations of the two continuous variables.

```
url_01 <- "https://raw.githubusercontent.com/jyurko/CS_1675_Spring_2022/main/HW/07/tr
ain_01.csv"
train_01 <- readr::read_csv(url_01, col_names = TRUE)
```

```
## Rows: 21 Columns: 2
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: ","
## dbl (2): x, y
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

As shown by the glimpse in the code chunk below there are 21 observations of the 2 continuous variables.

```
train_01 %>% glimpse()
```

```
## Rows: 21
## Columns: 2
## $ x <dbl> -3.0, -2.7, -2.4, -2.1, -1.8, -1.5, -1.2, -0.9, -0.6, -0.3, 0.0, 0.3…
## $ y <dbl> 3.30501327, 3.03876578, 2.53813946, 2.46437294, 1.72665889, 1.997619…
```

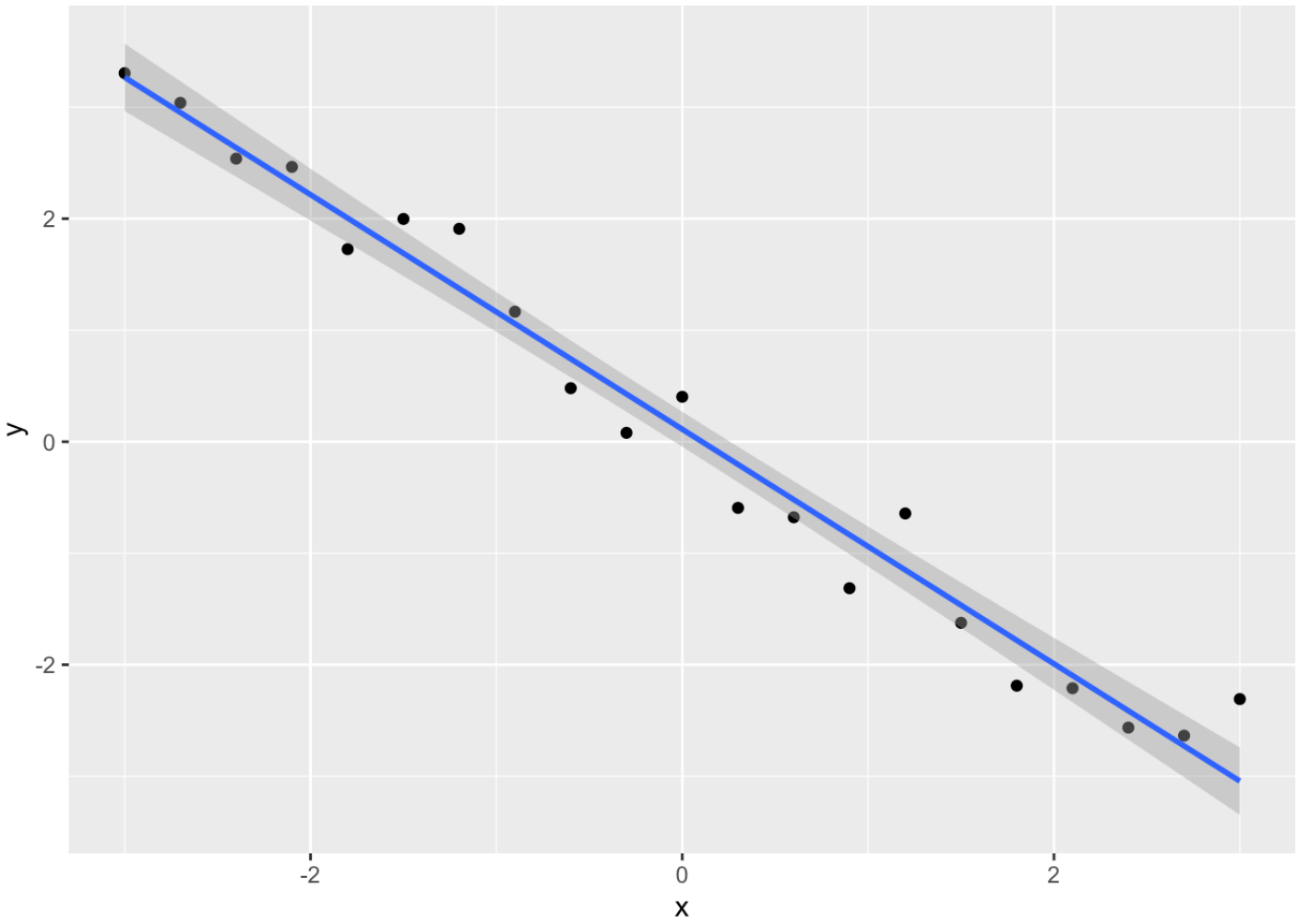
Create a scatter plot between the response and the input using `ggplot()`. In addition to using `geom_point()`, include a `geom_smooth()` layer to your graph. Set the `method` argument to `'lm'` in the call to `geom_smooth()`.

Based on the figure what type of relationship do you think exists between the response and the input?

SOLUTION

```
ggplot(data = train_01) + geom_point(mapping = aes(x = x, y = y)) + geom_smooth(mappi
ng = aes(x, y), method = 'lm')
```

```
## `geom_smooth()` using formula 'y ~ x'
```



There appears to be a negative linear relationship between the response and the input. As the input increases, the response decreases.

1b)

In your response to Problem 1a), you should see a “best fit line” and it’s associated confidence interval displayed with the scatter plot. Behind the scenes, `ggplot2()` fits a linear model between the response and the input with Maximum Likelihood Estimation, and plots the result on the figure. You will now work through a full Bayesian linear model. Before coding the log-posterior function, you will start out by creating the list of required information, `info_01` , which defines the data and hyperparameters that you will ultimately pass into the log-posterior function.

You will need to create a design matrix assuming a linear relationship between the input and the response. The mean trend function is written for you below:

$$\mu_n = \beta_0 + \beta_1 x_n$$

Create the design matrix assuming a linear relationship between the input and the response, and assign the object to the `Xmat_01` variable. Complete the `info_01` list by assigning the response to `yobs` and the design matrix to `design_matrix`. Specify the shared prior mean, `mu_beta`, to be 0, the shared prior standard deviation, `tau_beta`, as 5, and the rate parameter on the noise, `sigma_rate`, to be 1.

SOLUTION

```
Xmat_01 <- model.matrix(y ~ x, data = train_01)

info_01 <- list(
  yobs = train_01$y,
  design_matrix = Xmat_01,
  mu_beta = 0,
  tau_beta = 5,
  sigma_rate = 1
)
```

1c)

You will now define the log-posterior function `lm_logpost()`. You will continue to use the log-transformation on σ, and so you will actually define the log-posterior in terms of the mean trend $\boldsymbol{\beta}$-parameters and the unbounded noise parameter, $\varphi = \log[\sigma]$.

The comments in the code chunk below tell you what you need to fill in. The unknown parameters to learn are contained within the first input argument, `unknowns`. You will assume that the unknown $\boldsymbol{\beta}$-parameters are listed before the unknown φ parameter in the `unknowns` vector. You must specify the number of $\boldsymbol{\beta}$ parameters programmatically to allow scaling up your function to an arbitrary number of unknowns. You will assume that all variables contained in the `my_info` list (the second argument to `lm_logpost()`) are the same fields in the `info_01` list you defined in Problem 1b).

Define the log-posterior function by completing the code chunk below. You must calculate the mean trend, `mu`, using matrix math between the design matrix and the unknown $\boldsymbol{\beta}$ column vector. After you complete the function, test that it out by evaluating the log-posterior at two different sets of parameter values. Try out values of -1 for all parameters, and then try out values of 1 for all parameters.

HINT: If you have successfully completed the log-posterior function, you should get a value of `-109.5809` for the -1 guess values, and a value of `-68.12651` for the +1 guess values.

HINT: Don't forget about useful data type conversion functions such as `as.matrix()` and `as.vector()` (or `as.numeric()`).

SOLUTION

```
lm_logpost <- function(unknowns, my_info)
{
  # specify the number of unknown beta parameters
  length_beta <- ncol(my_info$design_matrix)

  # extract the beta parameters from the `unknowns` vector
  beta_v <- unknowns[1:length_beta]

  # extract the unbounded noise parameter, varphi
  lik_varphi <- unknowns[length_beta + 1]

  # back-transform from varphi to sigma
  lik_sigma <- exp(lik_varphi)

  # extract design matrix
  X <- my_info$design_matrix

  # calculate the linear predictor
  mu <- as.vector(X %*% as.matrix(beta_v))

  # evaluate the log-likelihood
  log_lik <- sum(dnorm(x = my_info$yobs,
                       mean = mu,
                       sd = lik_sigma,
                       log = TRUE))

  # evaluate the log-prior
  log_prior_beta <- sum(dnorm(x = beta_v,
                              mean = my_info$mu_beta,
                              sd = my_info$tau_beta,
                              log = TRUE))

  log_prior_sigma <- dexp(x = lik_sigma,
                          rate = my_info$sigma_rate,
                          log = TRUE)

  # add the mean trend prior and noise prior together
  log_prior <- log_prior_beta + log_prior_sigma

  # account for the transformation
  log_derive_adjust <- lik_varphi

  # sum together
  (log_lik + log_prior + log_derive_adjust)

}
```

Test out `lm_logpost()` with guess values of -1 for all parameters.

```
unknowns <- c(-1,-1,-1)
lm_logpost(unknowns, info_01)
```

```
## [1] -109.5809
```

Test out `lm_logpost()` with guess values of 1 for all parameters.

```
unknowns <- c(1,1,1)
lm_logpost(unknowns, info_01)
```

```
## [1] -68.12651
```

1d)

The `my_laplace()` function is started for you in the code chunk below. You must fill in the portion after the optimization is executed. Although you have filled in these elements before it is good to review to make sure you remember how the posterior covariance matrix is calculated! Also, you must complete the calculation of the `int` variable which stands for "integration" and equals the Laplace Approximation's estimate to the Evidence.

Complete the `my_laplace()` function below and then fit the Bayesian linear model using a starting guess of zero for all parameters. Assign your result to the `laplace_01` object. Print the posterior mode and posterior standard deviations to the screen. Should you be concerned about the initial guess impacting the posterior results?

SOLUTION

```
my_laplace <- function(start_guess, logpost_func, ...)
{
  # code adapted from the `LearnBayes`` function `laplace()`
  fit <- optim(start_guess,
               logpost_func,
               gr = NULL,
               ...,
               method = "BFGS",
               hessian = TRUE,
               control = list(fnscale = -1, maxit = 1001))

  mode <- fit$par
  post_var_matrix <- -solve(fit$hessian)
  p <- length(mode) # number of unknown parameters
  int <- p/2 * log(2 * pi) + 0.5 * log(det(post_var_matrix)) + logpost_func(mode, ...
)
  # package all of the results into a list
  list(mode = mode,
       var_matrix = post_var_matrix,
       log_evidence = int,
       converge = ifelse(fit$convergence == 0,
                         "YES",
                         "NO"),
       iter_counts = as.numeric(fit$counts[1]))
}
```

Fit the Bayesian linear model.

```
laplace_01 <- my_laplace(rep(0,ncol(Xmat_01) + 1), lm_logpost, info_01)
```

Display the posterior mode and posterior standard deviations.

```
laplace_01$mode
```

```
## [1]  0.1118562 -1.0517750 -1.1065825
```

```
laplace_01$var_matrix
```

```
##               [,1]          [,2]          [,3]
## [1,]  5.206248e-03 -1.518981e-10 -1.142399e-06
## [2,] -1.518981e-10  1.577880e-03  3.243614e-06
## [3,] -1.142399e-06  3.243614e-06  2.439495e-02
```

1e)

The `generate_lm_post_samples()` function is started for you in the code chunk below. The first argument, `mvn_result`, is the Laplace Approximation result object returned from the `my_laplace()` function. The second argument, `length_beta`, specifies the number of mean trend $\boldsymbol{\beta}$-parameters to the model. The naming of the variables is taken care of for you. This function should look quite similar to the previous assignment…

After completing the function, generate 2500 posterior samples of the parameters in your model and assign the result to the `laplace_01` object. You will then use the posterior samples to study the posterior distribution on the slope β_1.

Complete the generate_lm_post_samples() function below. After completing the function, generate 2500 posterior samples from your `laplace_01` model. Create a histogram with 55 bins using `ggplot2` for the slope `beta_01`. Calculate the probability that the slope is positive.

SOLUTION

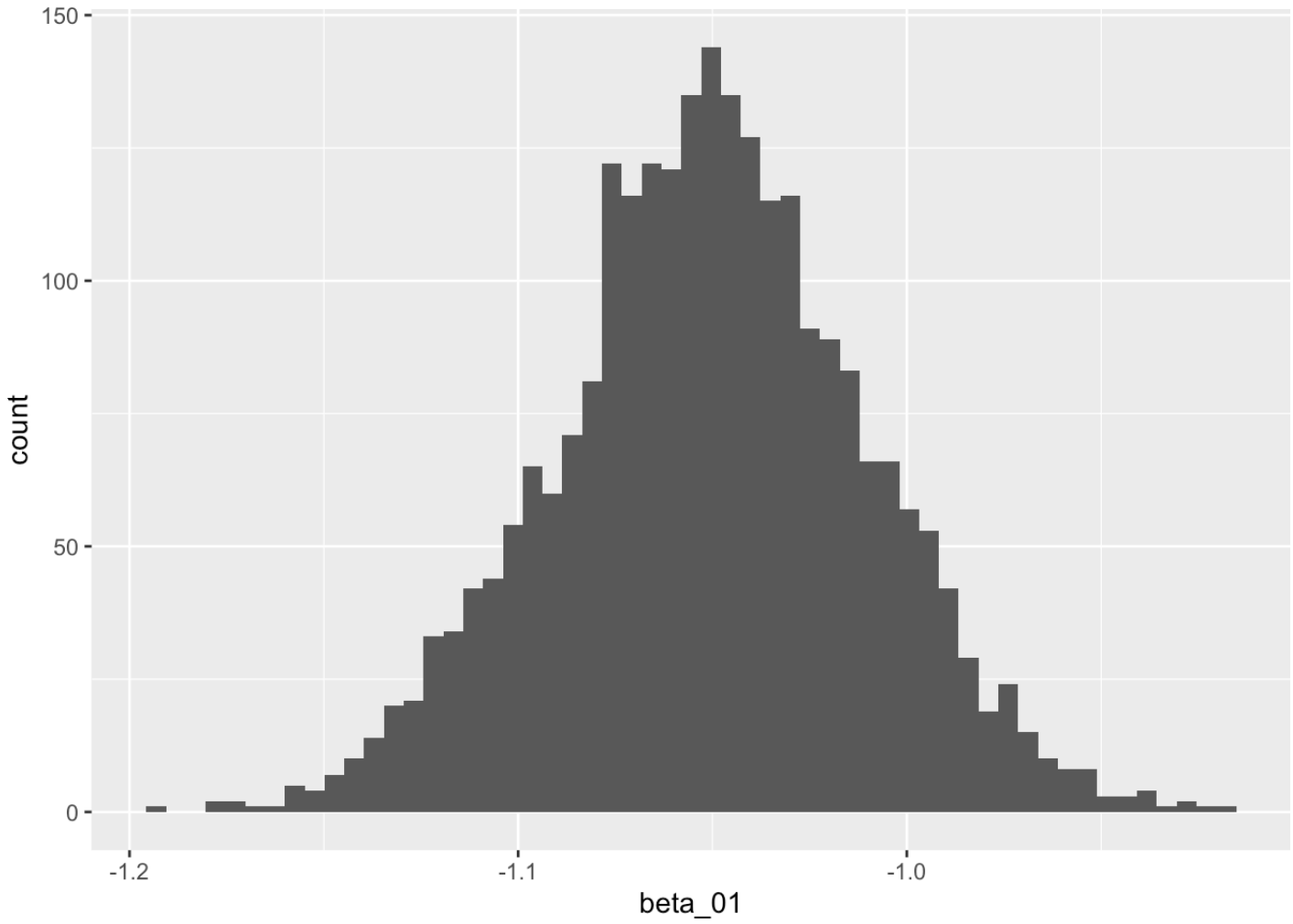
```
generate_lm_post_samples <- function(mvn_result, length_beta, num_samples)
{
  MASS::mvrnorm(n = num_samples,
                mu = mvn_result$mode ,
                Sigma = mvn_result$var_matrix ) %>%
    as.data.frame() %>% tibble::as_tibble() %>%
    purrr::set_names(c(sprintf("beta_%02d", 0:(length_beta-1)), "varphi")) %>%
    mutate(sigma = exp(varphi))
}
```

Generate posterior samples.

```
set.seed(87123)
post_samples_01 <- generate_lm_post_samples(laplace_01, ncol(Xmat_01), 2500)
```

Create the posterior histogram on β_1 and calculate the probability that β_1 is greater than zero.

```
post_samples_01 %>%
  ggplot(mapping = aes(x = beta_01)) +
  geom_histogram(bins = 55)
```




```
mean(post_samples_01$beta_01 > 0)
```

```
## [1] 0
```

Problem 02

Now that you can fit a Bayesian linear model, it's time to work with making posterior predictions from the model. You will use those predictions to calculate and summarize the errors of the model relative to observations. Since RMSE and R-squared have been discussed throughout lecture, you will work with the Mean Absolute Error (MAE) metric.

2a)

The code chunk below starts the `post_lm_pred_samples()` function. This function generates posterior mean trend predictions and posterior predictions of the response. The first argument, `Xnew`, is a potentially new or test design matrix that we wish to make predictions at. The second argument, `Bmat`, is a matrix of

posterior samples of the $\boldsymbol{\beta}$-parameters, and the third argument, `sigma_vector`, is a vector of posterior samples of the likelihood noise. The `Xnew` matrix has rows equal to the number of predictions points, `M`, and the `Bmat` matrix has rows equal to the number of posterior samples `S`.

You must complete the function by performing the necessary matrix math to calculate the matrix of posterior mean trend predictions, `Umat`, and the matrix of posterior response predictions, `Ymat`. You must also complete missing arguments to the definition of the `Rmat` and `Zmat` matrices. The `Rmat` matrix replicates the posterior likelihood noise samples the correct number of times. The `Zmat` matrix is the matrix of randomly generated standard normal values. You must correctly specify the required number of rows to the `Rmat` and `Zmat` matrices.

The `post_lm_pred_samples()` returns the `Umat` and `Ymat` matrices contained within a list.

Perform the necessary matrix math to calculate the matrix of posterior predicted mean trends `Umat` and posterior predicted responses `Ymat`. You must specify the number of required rows to create the `Rmat` and `Zmat` matrices.

SOLUTION

```
post_lm_pred_samples <- function(Xnew, Bmat, sigma_vector)
{
  # number of new prediction locations
  M <- nrow(Xnew)
  # number of posterior samples
  S <- nrow(Bmat)

  # matrix of linear predictors
  Umat <- Xnew %*% t(Bmat)

  # assmeble matrix of sigma samples, set the number of rows
  Rmat <- matrix(rep(sigma_vector, M), M, byrow = TRUE)

  # generate standard normal and assemble into matrix
  # set the number of rows
  Zmat <- matrix(rnorm(M*S), M, byrow = TRUE)

  # calculate the random observation predictions
  Ymat <- Umat + Rmat * Zmat

  # package together
  list(Umat = Umat, Ymat = Ymat)
}
```

2b)

The code chunk below is completed for you. The function `make_post_lm_pred()` is a wrapper which calls the `post_lm_pred_samples()` function. It contains two arguments. The first, `Xnew`, is a test design matrix. The second, `post`, is a data.frame of posterior samples. The function extracts the $\boldsymbol{\beta}$-parameter posterior samples and converts the object to a matrix. It also extracts the posterior samples on σ and converts to a vector.

```
make_post_lm_pred <- function(Xnew, post)
{
  Bmat <- post %>% select(starts_with("beta_")) %>% as.matrix()

  sigma_vector <- post %>% pull(sigma)

  post_lm_pred_samples(Xnew, Bmat, sigma_vector)
}
```

You now have enough pieces in place to generate posterior predictions from your model.

Make posterior predictions on the training set. What are the dimensions of the returned `Umat` and `Ymat` matrices? Do the columns correspond to the number of prediction points?

HINT: The `make_post_lm_pred()` function returns a list. To access the variables or fields of a list use the `$` operator.

SOLUTION

Make posterior predictions on the training set.

```
post_pred_samples_01 <- make_post_lm_pred(Xmat_01, post_samples_01)
```

The dimensionality of the posterior predicted mean trend matrix is:

```
dim(post_pred_samples_01$Umat)
```

```
## [1]   21 2500
```

The dimensionality of the posterior predicted response matrix is:

```
dim(post_pred_samples_01$Ymat)
```

```
## [1]   21 2500
```

For both Umat and Ymat the dimensions are 21x2500. This number does correspond to the number of prediction points as we did 2500 predictions points.

2c)

You will now use the model predictions to calculate the error between the model and the training set observations. Since you generated 2500 posterior samples, you have 2500 different sets of predictions! So, to get started you will focus on the first 3 posterior samples.

Calculate the error between the predicted mean trend and the training set observations for each of the first 3 posterior predicted samples. Assign the errors to separate vectors, as indicated in the code chunk below.

Why are you considering the mean trend when calculating the error with the response, and not the predicted response values?

SOLUTION

The error between the first 3 posterior predicted mean trend samples and the training set observations are calculated below.

```
### error of the first posterior sample
error_01_post_01 <- post_pred_samples_01$Umat[,1] - train_01$y

### error of the second posterior sample
error_01_post_02 <- post_pred_samples_01$Umat[,2] - train_01$y

### error of the third posterior sample
error_01_post_03 <- post_pred_samples_01$Umat[,3] - train_01$y
```

We use the mean trend when calculating the error with the response and not the predicted response because we want to see how far off the posterior predicted sample is to the overall predicted average mean. This represents the uncertainty in the mean trend.

2d)

You will now calculate the Mean Absolute Error (MAE) associated with each of the three error samples calculated in Problem 2c). However, before calculating the MAE, first consider the dimensions of the `error_01_post_01`. What is the length of the `error_01_post_01` vector? When you take the absolute value and then average across all elements in that vector, what are you averaging over?

What is the length of the `error_01_post_01` vector? Calculate the MAE associated with each of the 3 error vectors you calculated in Problem 2c. What are you averaging over when you calculate the mean absolute error? Are the three MAE values the same? If not, why would they be different?

HINT: The absolute value can be calculated with the `abs()` function.

SOLUTION

```
length(error_01_post_01)
```

```
## [1] 21
```

The length of error_01_post_01 is 21.

Now calculate the MAE associated with each of the first three posterior samples.

```
mae_01_post_01 <- sum(abs(error_01_post_01)) / length(error_01_post_01)

mae_01_post_02 <- sum(abs(error_01_post_02)) / length(error_01_post_02)

mae_01_post_03 <- sum(abs(error_01_post_03)) / length(error_01_post_03)

mae_01_post_01
```

```
## [1] 0.2718539
```

```
mae_01_post_02
```

```
## [1] 0.3064965
```

```
mae_01_post_03
```

```
## [1] 0.2857988
```

When calculating the mean absolute error we are averaging the sum of all the error with the response dividing by the the length of each of the vectors. The three MAE values are not the same because the posterior predicted value is different amongst the first three samples and when computing the MAE against the same training set observations, they will produce different values.

2e)

In Problem 2d) you calculated the MAE associated with the first 3 posterior samples. You will now work through calculating the MAE associated with every posterior sample. Although it might seem like you need to use a for-loop to do so, `R` will simplify the operation for you. If you perform an addition or subtraction between a matrix and a vector, `R` will find the dimension that that matches between the two and then repeat the action over the other dimension. Consider the code below, which has a vector, `a_numeric`, subtracted from a matrix `a_matrix`:

```
a_matrix - a_numeric
```

Assuming that `a_matrix` has 10 rows and 25 columns and `a_numeric` is length 10, `R` will subtract `a_numeric` from each column in `a_matrix`. The result will be another matrix with the same dimensionality as `a_matrix`. To confirm this is the case, consider the example below where a vector of length 2 is subtracted from a matrix of 2 rows and 4 columns. The resulting dimensionality is 2 rows by 4 columns.

```
### a 2 x 4 matrix
matrix(1:8, nrow = 2, byrow = TRUE)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
```

```
### a vector length 2
c(1, 2)
```

```
## [1] 1 2
```

```
### subtracting the two yields a matrix
matrix(1:8, nrow = 2, byrow = TRUE) - c(1, 2)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    1    2    3
## [2,]    3    4    5    6
```

You will use this fact to calculate the error associated with each training point and each posterior sample all at once.

Calculate the absolute value of the error between the mean trend matrix and the training set response. Print the dimensions of the `absE01mat` matrix to screen.

SOLUTION

```
absE01mat <- abs(post_pred_samples_01$Umat - train_01$y)
### dimensions?
dim(absE01mat)
```

```
## [1]   21 2500
```

The dimensions of the matrix is 21 x 2500.

2f)

You must now summarize the absolute value errors by averaging them appropriately. Should you average across the rows or down the columns? In `R` the `colMeans()` will calculate the average value associated with each column in a matrix and returns a vector. Likewise, the `rowMeans()` function calculates the average value along each row and returns a vector. Which function should you use to calculate the MAE associated with each posterior sample?

Calculate the MAE associated with each posterior sample and assign the result to the `MAE_01` object. Print the data type (the class) of the `MAE_01` to the screen and display its length. Check your result is consistent with the MAEs you previously calculated in Problem 2d).

SOLUTION

```
MAE_01 <- colMeans(absE01mat)
### data type?
class(MAE_01)
```

```
## [1] "numeric"
```

```
### length?
length(MAE_01)
```

```
## [1] 2500
```

Check with the results you calculated previously.

```
### your code here
MAE_01[1]
```

```
## [1] 0.2718539
```

```
mae_01_post_01
```

```
## [1] 0.2718539
```

```
MAE_01[2]
```

```
## [1] 0.3064965
```

```
mae_01_post_02
```

```
## [1] 0.3064965
```

```
MAE_01[3]
```

```
## [1] 0.2857988
```

```
mae_01_post_03
```

```
## [1] 0.2857988
```

The MAEs calculated the two different ways match up.

2g)

You have calculated the MAE associated with each posterior sample, and thus represented the uncertainty in the MAE! Why is the MAE uncertain?

Use the `quantile()` function to print out summary statistics associated with the MAE. You can use the default arguments, and thus pass in `MAE_01` into `quantile()` without setting any other argument. Why is the MAE uncertain? Or put another way, what causes the MAE to be uncertain?

SOLUTION

Calculate the quantiles of the MAE below.

```
quantile(MAE_01)
```

```
##        0%       25%       50%       75%      100%
## 0.2553225 0.2688640 0.2774844 0.2918598 0.3780777
```

The uncertainty in the Beta-parameters causes uncertainty in the mean trend. Since we used the predicted mean trend in our MAEs there is uncertainty in the Beta-parameters and therefore, uncertainty within the MAE.

Problem 03

You will now make use of the model fitting and prediction functions you created in the previous problems to study the behavior of a more complicated non-linear modeling task. The code chunk below reads in two data sets. Both consist of two continuous variables, an input `x` and a response `y`. The first, `train_02`, will serve as the training set, and the second, `test_02`, will serve as a hold-out test set. You will only fit models based on the training set.

```
url_02_train <- "https://raw.githubusercontent.com/jyurko/CS_1675_Spring_2022/main/HW
/07/train_02.csv"

train_02 <- readr::read_csv(url_02_train, col_names = TRUE)
```

```
## Rows: 150 Columns: 2
## ── Column specification ────────────────────────────────────────────────────
## Delimiter: ","
## dbl (2): x, y
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
url_02_test <- "https://raw.githubusercontent.com/jyurko/CS_1675_Spring_2022/main/HW/
07/test_02.csv"

test_02 <- readr::read_csv(url_02_test, col_names = TRUE)
```

```
## Rows: 50 Columns: 2
## ── Column specification ────────────────────────────────────────────────────
## Delimiter: ","
## dbl (2): x, y
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

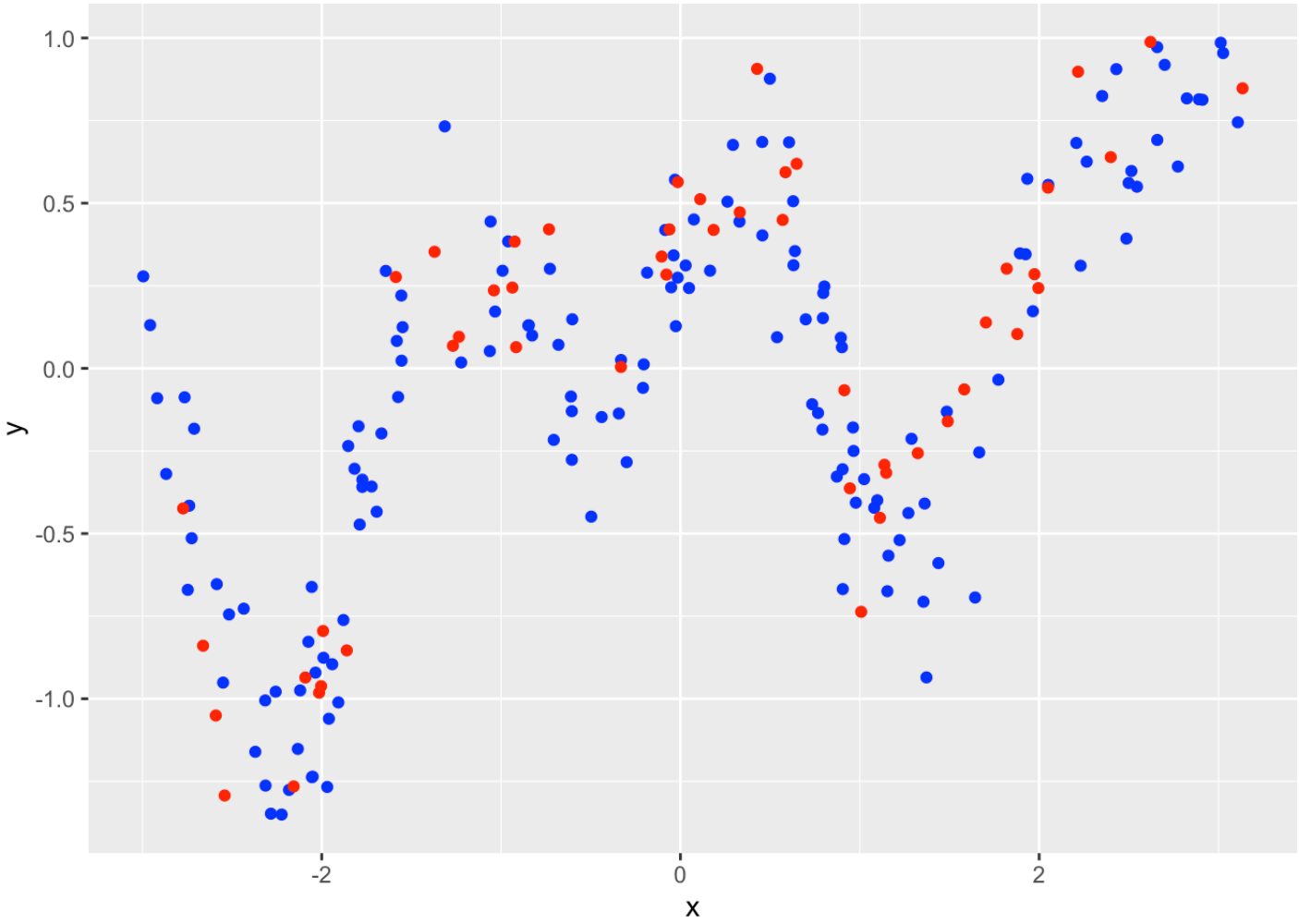
3a)

It's always a good idea to start out by visualizing the data before modeling.

Create a scatter plot between the response and the input with `ggplot2`. Include both the training and test sets together in one graph. Use the marker color to distinguish between the two.

SOLUTION

```
ggplot(aes(x, y ), data = train_02) +  geom_point(col = "blue") +
  geom_point(mapping = aes(x, y), data = test_02, col = "red")
```



3b)

You will fit 25 different models to the training set. You will consider a one degree of freedom spline up to 49 degrees of freedom. You will consider only odd values for the degrees of freedom. Your goal will be to find which spline is the "best" in terms of generalizing from the training set to the test set. To do so, you will calculate the MAE on the training set and on the test set for each model. It will be rather tedious to set up all of the necessary information by hand, manually train each model, generate posterior samples, and make predictions from each model. Therefore, you will work through completing functions that will enable you to programmatically loop over each candidate model.

You will start out by completing a function to create the training set and test set for a desired spline basis. The function `make_spline_basis_mats()` is started for you in the first code chunk below. The first argument is the desired spline basis degrees of freedom, `J`. The second argument is the training set, `train_data`, and the third argument is the hold-out test set, `test_data`.

The second through fifth code chunks below are provided to check that you completed the function correctly. The second code chunk uses `purrr::map_dfr()` to create the training and test matrices for all 25 models. A glimpse of the resulting object, `spline_matrices`, is displayed to the screen for you in the third code chunk. It is printed to the screen in the fourth code chunk. You should see a `tibble` consisting of two

variables, `design_matrix` and `test_matrix`. Both variables are lists containing matrices. The matrices contained in the `spline_matrices$design_matrix` variable are the different training design matrices, while the matrices contained in `spline_matrices$test_matrix` are the associated hold-out test basis matrices.

The fifth code chunk below prints the dimensionality of the 1st and 2nd order spline basis matrices to the screen. It shows that to access a specific matrix, you need to use the `[[]]` notation.

Complete the code chunk below. You must specify the `splines::ns()` function call correctly such that the degrees-of-freedom, `df`, argument equals the user specified degrees of freedom, `J`, and that the basis is applied to the `x` variable within the user supplied `train_data` argument. The knots are extracted for you and saved to the `knots_use_basis` object. Create the training design matrix by calling the `model.matrix()` function with the `splines::ns()` function to create the basis for the `x` variable with `knots` equal to `knots_use_basis`. Make sure you assign the data sets correctly to the `data` argument of `model.matrix()`.

How many rows are in the training matrices and how many rows are in the test matrices?

SOLUTION

Define the `make_spline_basis_mats()` function.

```
make_spline_basis_mats <- function(J, train_data, test_data)
{
  train_basis <- splines::ns(train_data$x, df = J)

  knots_use_basis <- as.vector(attributes(train_basis)$knots)

  train_matrix <- model.matrix(~ splines::ns (x, knots = knots_use_basis),
                               data = train_data )

  test_matrix <- model.matrix(~ splines::ns (x, knots = knots_use_basis),
                              data = test_data)

  tibble::tibble(
    design_matrix = list(train_matrix),
    test_matrix = list(test_matrix)
  )
}
```

Create each of the training and test basis matrices.

```
spline_matrices <- purrr::map_dfr(seq(1, 50, by = 2),
                                  make_spline_basis_mats,
                                  train_data = train_02,
                                  test_data = test_02)
```

Get a glimpse of the structure of `spline_matrices`.

```
glimpse(spline_matrices)
```

```
## Rows: 25
## Columns: 2
## $ design_matrix <list> <<matrix[150 x 2]>>, <<matrix[150 x 4]>>, <<matrix[150 …
## $ test_matrix   <list> <<matrix[50 x 2]>>, <<matrix[50 x 4]>>, <<matrix[50 x 6…
```

Display the elements of `spline_matrices` to the screen.

```
spline_matrices
```

```
## # A tibble: 25 × 2
##    design_matrix    test_matrix
##    <list>           <list>
##  1 <dbl [150 × 2]>  <dbl [50 × 2]>
##  2 <dbl [150 × 4]>  <dbl [50 × 4]>
##  3 <dbl [150 × 6]>  <dbl [50 × 6]>
##  4 <dbl [150 × 8]>  <dbl [50 × 8]>
##  5 <dbl [150 × 10]> <dbl [50 × 10]>
##  6 <dbl [150 × 12]> <dbl [50 × 12]>
##  7 <dbl [150 × 14]> <dbl [50 × 14]>
##  8 <dbl [150 × 16]> <dbl [50 × 16]>
##  9 <dbl [150 × 18]> <dbl [50 × 18]>
## 10 <dbl [150 × 20]> <dbl [50 × 20]>
## # … with 15 more rows
```

The code chunk below is created for you. It shows how to check the dimensions of several training and test matrices.

```
dim(spline_matrices$design_matrix[[1]])
```

```
## [1] 150   2
```

```
dim(spline_matrices$test_matrix[[1]])
```

```
## [1] 50  2
```

```
dim(spline_matrices$design_matrix[[2]])
```

```
## [1] 150   4
```

```
dim(spline_matrices$test_matrix[[2]])
```

```
## [1] 50  4
```

```
nrow(spline_matrices$design_matrix[[1]])
```

```
## [1] 150
```

```
nrow(spline_matrices$test_matrix[[1]])
```

```
## [1] 50
```

The training matrix has 150 rows and the test matrix has 50 rows.

3c)

Each element in the `spline_matrices$design_matrix` object is a separate design matrix. You will use this structure to programmatically train each model. The first code chunk creates a list of information which stores the training set responses and defines the prior hyperparameters. The second code chunk below defines the `manage_spline_fit()` function. The first argument is a design matrix `Xtrain`, the second argument is the log-posterior function, `logpost_func`, and the third argument is `my_settings`. `manage_spline_fit()` sets the initial starting values to the $\boldsymbol{\beta}$ parameters by generating random values from a standard normal. The initial value for the unbounded φ parameter is set by log-transforming a random draw from the prior on σ. After creating the initial guess values, the `my_laplace()` function is called to fit the model.

You will complete both code chunks in order to programmatically train all 25 spline models. After completing the first two code chunks, the third code chunk performs the training for you. The fourth code chunk below shows how to access training results associated with the second-order spline by using the `[[]]` operator. The fifth code chunk checks that each model converged.

Complete the first two code chunks below. In the first code chunk, assign the training responses to the `yobs` variable within the `info_02_train` list. Specify the prior mean and prior standard deviation on the $\boldsymbol{\beta}$-parameters to be 0 and 20, respectively. Specify the rate parameter on the unknown σ to be 1.

Complete the second code chunk by generating a random starting guess for all $\boldsymbol{\beta}$-parameters from a standard normal. Create the random initial guess for φ by generating a random number from the Exponential prior distribution on σ and log-transforming the variable. Complete the call the `my_laplace()` function by passing in the initial values as a vector of correct size.

HINT: How can you determine the number of unknown $\boldsymbol{\beta}$-parameters if you know the training design matrix?

SOLUTION

Assemble the list of required information.

```
info_02_train <- list(
  yobs = train_02$y,
  mu_beta = 0,
  tau_beta = 20,
  sigma_rate = 1
)
```

Complete the function which manages the execution of the Laplace Approximation to each spline model. Note that `Xtrain` is assumed to be a design matrix in the `manage_spline_fit()` function.

```
manage_spline_fit <- function(Xtrain, logpost_func, my_settings)
{
  my_settings$design_matrix <- Xtrain

  init_beta <- rnorm(ncol(Xtrain))

  init_varphi <- log(rexp(n = 1, rate = my_settings$sigma_rate))

  my_laplace( c(init_beta, init_varphi), logpost_func, my_settings)
}
```

The next three code chunks are completed for you. All 25 spline models are trained in the first code chunk below. Notice that because the training design matrices have already been created, we just need to loop over each element of `spline_matrices$design_matrix`.

```
set.seed(724412)
all_spline_models <- purrr::map(spline_matrices$design_matrix,
                                manage_spline_fit,
                                logpost_func = lm_logpost,
                                my_settings = info_02_train)
```

The code chunk below shows how to access a single model fitting result. Specifically, it extracts the result of the second trained model, the 3 degree of freedom spline.

```
all_spline_models[[2]]
```

```
## $mode
## [1] -1.0128021  0.3502016  2.5063039  1.1685141 -0.8202799
##
## $var_matrix
##               [,1]          [,2]          [,3]          [,4]          [,5]
## [1,]  1.674058e-02 -2.947568e-03 -3.960273e-02 -4.056984e-03  1.988769e-06
## [2,] -2.947568e-03  2.062475e-02 -2.310636e-03 -2.388290e-03  4.967719e-08
## [3,] -3.960273e-02 -2.310636e-03  1.070993e-01  1.545448e-02 -5.267518e-06
## [4,] -4.056984e-03 -2.388290e-03  1.545448e-02  1.853818e-02 -1.040898e-06
## [5,]  1.988769e-06  4.967719e-08 -5.267518e-06 -1.040898e-06  3.340949e-03
##
## $log_evidence
## [1] -113.1528
##
## $converge
## [1] "YES"
##
## $iter_counts
## [1] 32
```

The code chunk below shows how to check that all optimizations converged.

```
purrr::map_chr(all_spline_models, "converge")
```

```
##  [1] "YES" "YES" "YES" "YES" "YES" "YES" "YES" "YES" "YES" "YES" "YES" "YES"
## [13] "YES" "YES" "YES" "YES" "YES" "YES" "YES" "YES" "YES" "YES" "YES" "YES"
## [25] "YES"
```

3d)

With all 25 spline models fit, it is time to assess which model is the best. Several different approaches have been discussed in lecture for how to identify the “best” model. You will start out by calculating the MAE on the training set and the test set. You went through the steps to generate posterior samples, make posterior predictions and to calculate the posterior MAE distribution in Problem 2. You will now define a function which performs all of those steps together.

The function `calc_mae_from_laplace()` is started for you in the first code chunk below. The first argument, `mvn_result`, is the result of the `my_laplace()` function for a particular model. The second and third arguments, `Xtrain` and `Xtest`, are the training and test basis matrices associated with the model, respectively. The fourth and fifth arguments, `y_train` and `y_test`, are the observations on the training set and test set, respectively. The last argument, `num_samples`, is the number of posterior samples to generate.

You will complete the necessary steps to generate posterior samples from the model, predict the training set, predict the test. Then you will calculate the training set MAE and test set MAE, associated with each posterior sample. The last portion of the `calc_mae_from_laplace()` function is mostly completed for you.

After you complete the `calc_mae_from_laplace()`, the second code chunk applies the function to all 25 spline models. It is nearly complete. You must specify the arguments which define the training set observations, `y_train`, the test set observations, `y_train`, and the number of posterior samples, `num_samples`.

Complete all steps to calculate the MAE on the training set and test sets in the first code chunk below. Complete the lines of code in order to: generate posterior samples from the supplied `mvn_result` object, make posterior predictions on the training set, make posterior predictions on the test, and then calculate the MAE associated with each posterior sample on the training and test sets. In the book keeping portion of the function, you must specify the order of the spline model.

You must specify the training set and test set observed responses correctly in the second code chunk. You must specify the number of posterior samples to be 2500.

HINT: Remember that the result of the `make_post_lm_pred()` function is a list!

SOLUTION

Complete all steps to define the `calc_mae_from_laplace()` function below.

```
calc_mae_from_laplace <- function(mvn_result, Xtrain, Xtest, y_train, y_test, num_sam
ples)
{
  # generate posterior samples from the approximate MVN posterior
  post <- generate_lm_post_samples(mvn_result, ncol(Xtrain), num_samples)

  # make posterior predictions on the training set
  pred_train <- make_post_lm_pred(Xtrain, post)

  # make posterior predictions on the test set
  pred_test <- make_post_lm_pred(Xtest, post)

  # calculate the error between the training set predictions
  # and the training set observations
  error_train <- pred_train$Umat - y_train

  # calculate the error between the test set predictions
  # and the test set observations
  error_test <- pred_test$Umat - y_test

  # calculate the MAE on the training set
  mae_train <- colMeans(abs(error_train))

  # calculate the MAE on the test set
  mae_test <- colMeans(abs(error_test))

  # book keeping, package together the results
  mae_train_df <- tibble::tibble(
    mae = mae_train
  ) %>%
    mutate(dataset = "training") %>%
    tibble::rowid_to_column("post_id")

  mae_test_df <- tibble::tibble(
    mae = mae_test
  ) %>%
    mutate(dataset = "test") %>%
    tibble::rowid_to_column("post_id")

  # you must specify the order, J, associated with the spline model
  mae_train_df %>%
    bind_rows(mae_test_df) %>%
    mutate(J = length(mvn_result$mode) - 2)
}
```

Apply the `calc_mae_from_laplace()` function to all 25 spline models.

```
set.seed(52133)
all_spline_mae_results <- purrr::pmap_dfr(list(all_spline_models,
                                               spline_matrices$design_matrix,
                                               spline_matrices$test_matrix),
                                          calc_mae_from_laplace,
                                          y_train = train_02$y,
                                          y_test = test_02$y ,
                                          num_samples = 2500)

all_spline_mae_results
```

```
## # A tibble: 125,000 × 4
##    post_id   mae dataset      J
##      <int> <dbl> <chr>    <dbl>
##  1       1 0.385 training     1
##  2       2 0.392 training     1
##  3       3 0.380 training     1
##  4       4 0.399 training     1
##  5       5 0.383 training     1
##  6       6 0.386 training     1
##  7       7 0.396 training     1
##  8       8 0.378 training     1
##  9       9 0.378 training     1
## 10      10 0.384 training     1
## # … with 124,990 more rows
```

3e)

If you completed the `calc_mae_from_laplace()` function correctly, you should have an object with 62500 rows for each split and just 4 columns. The object was structured in a “tall” or “long-format” and is thus a “tidy data” object. We can therefore easily summarize the posterior MAE samples with `ggplot2` (if you are a Python user this is the same philosophy of the Seaborn statistical visualization package). You will summarize the MAE posterior distributions with boxplots and by focusing on the median MAE. To focus on the median MAE, you will use the `stat_summary()` function. This is a flexible function capable of creating many different geometric objects. It consists of arguments such as `geom` to specify the type of geometric object to display and `fun` to specify what function to apply to the `y` aesthetic.

Complete the two code chunks below. In the first code chunk, pipe the `all_spline_mae_results` object into `ggplot()`. Set the `x` aesthetic to be `as.factor(J)` and the `y` aesthetic to be `mae`. In the `geom_boxplot()` call, map the `fill` aesthetic to the `dataset` variable. Use the scale `scale_fill_brewer()` with the `palette` argument set equal to `"Set1"`.

In the second code chunk, pipe the `all_spline_mae_results` object into a `filter()` call. Keep only the models with the spline order greater than 3. Pipe the result into a `ggplot()` call where you set the `x` aesthetic to `J` and the `y` aesthetic to `mae`. Use the `stat_summary()` call to calculate the median MAE for each spline order. You must set the `geom` argument within `stat_summary()` to be `'line'` and the `fun` argument to be `'median'`. Rather than coloring by `dataset`, use `facet_wrap()` to specify separate facets (subplots) for each `dataset`.

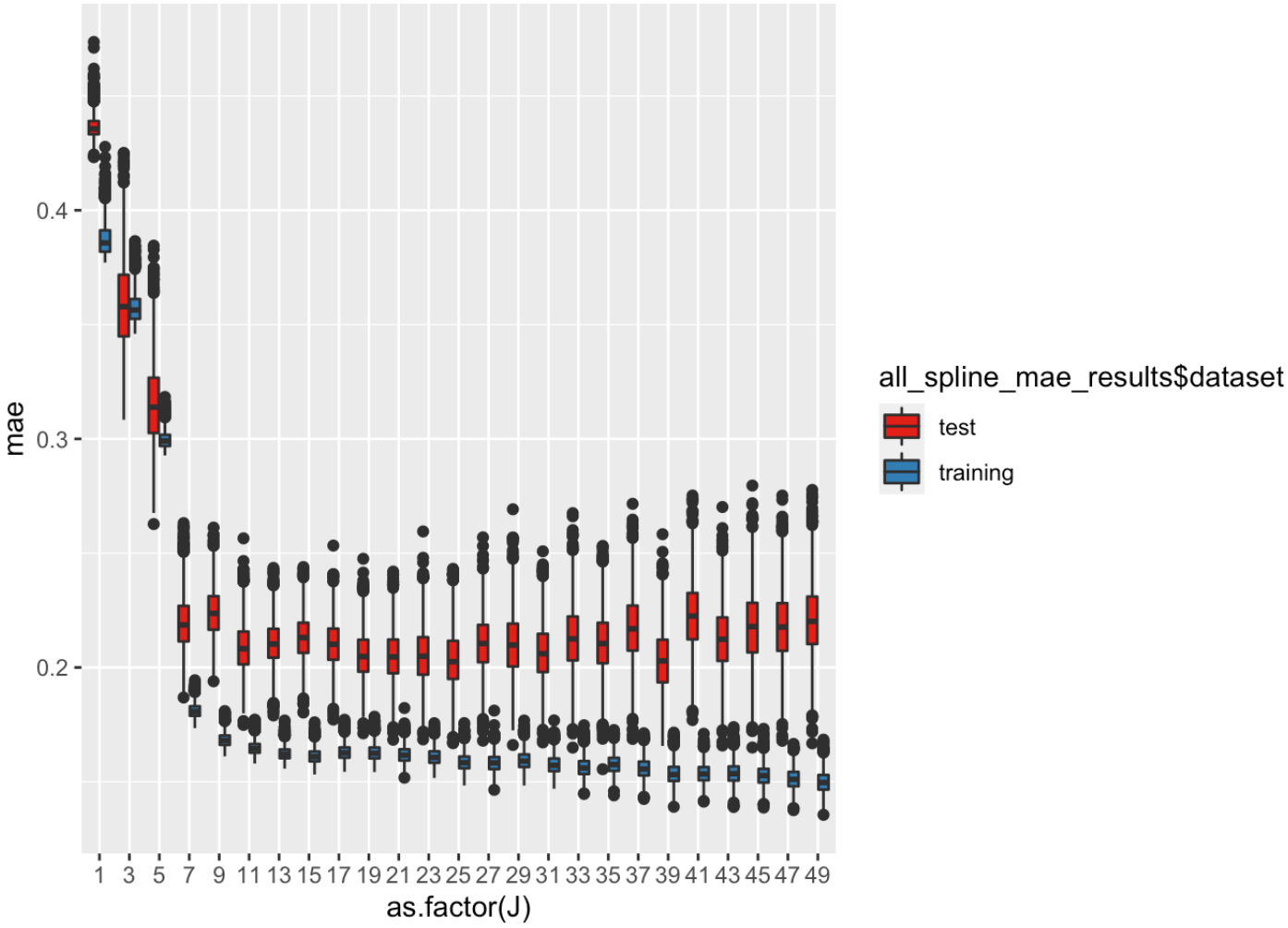
Based on these visualizations which models appear to overfit the training data?

SOLUTION

Summarize the posterior samples on the MAE on the training and test sets with boxplots.

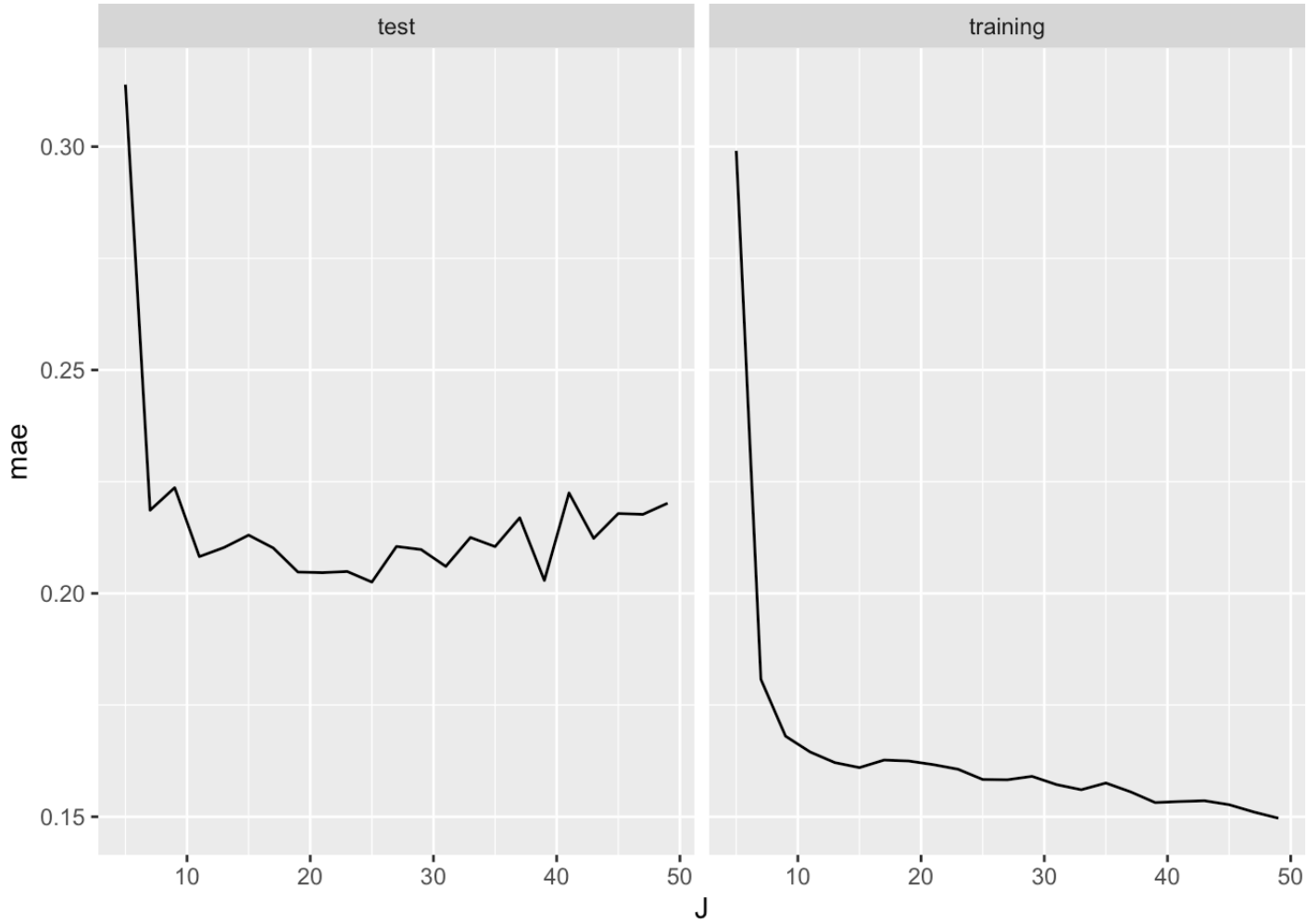
```
all_spline_mae_results %>%
  ggplot(mapping = aes(x = as.factor(J), y = mae)) + geom_boxplot(mapping = aes(x = a
s.factor(J), fill = all_spline_mae_results$dataset)) +
  scale_fill_brewer(palette = "Set1")
```

```
## Warning: Use of `all_spline_mae_results$dataset` is discouraged. Use `dataset`
## instead.
```



Summarize the posterior samples on the MAE on the training and test sets by focusing on the posterior median MAE values.

```
all_spline_mae_results %>%
  filter(J > 3) %>%
  ggplot(x = J, y = mae) +
  stat_summary(mapping = aes(x = J, y = mae), geom = "line", fun = "median") +
  facet_wrap( ~dataset)
```



The highest order of splines seem to overfit the model. On the training set, they contain very low MAE values, but on the training set they contain slightly higher MAE values than lower order spline models.

3f)

By comparing the posterior MAE values on the training and test splits you are trying to assess how well the models generalize to new data. As discussed in lecture, other metrics exist for trying to assess how well a model generalizes, based just on the training set performance. The Evidence or marginal likelihood is attempting to evaluate generalization by integrating the likelihood over all a-priori allowed parameter combinations. If the Evidence can be calculated, it can be used to weight all models relative to each other. Thereby allowing you to assess which model appears to be “most probable”.

You will now calculate the posterior model weights associated with each model. The first code chunk below is completed for you, by extracting the log-evidence associated with model into the `numeric` vector `spline_evidence`. You will use the log-evidence to calculate the posterior model weights and visualize the results with a bar graph.

Calculate the posterior model weights associated with each spline model and assign the weights to the `spline_weights` variable. The `spline_dof` vector is created for you which stores the degrees of freedom considered in this problem. The `spline_weights` vector is assigned to the `w` variable in a `tibble` and the result is piped into `mutate()` where you must specify the `J` variable to be the degrees of freedom values. Pipe the result into a `ggplot()` call where you set the `x` aesthetic to `as.factor(J)` and the `y` aesthetic to `w`. Include a `geom_bar()` geometric object where the `stat` argument is set to `"identity"`.

Based on your visualization, which model is considered the best?

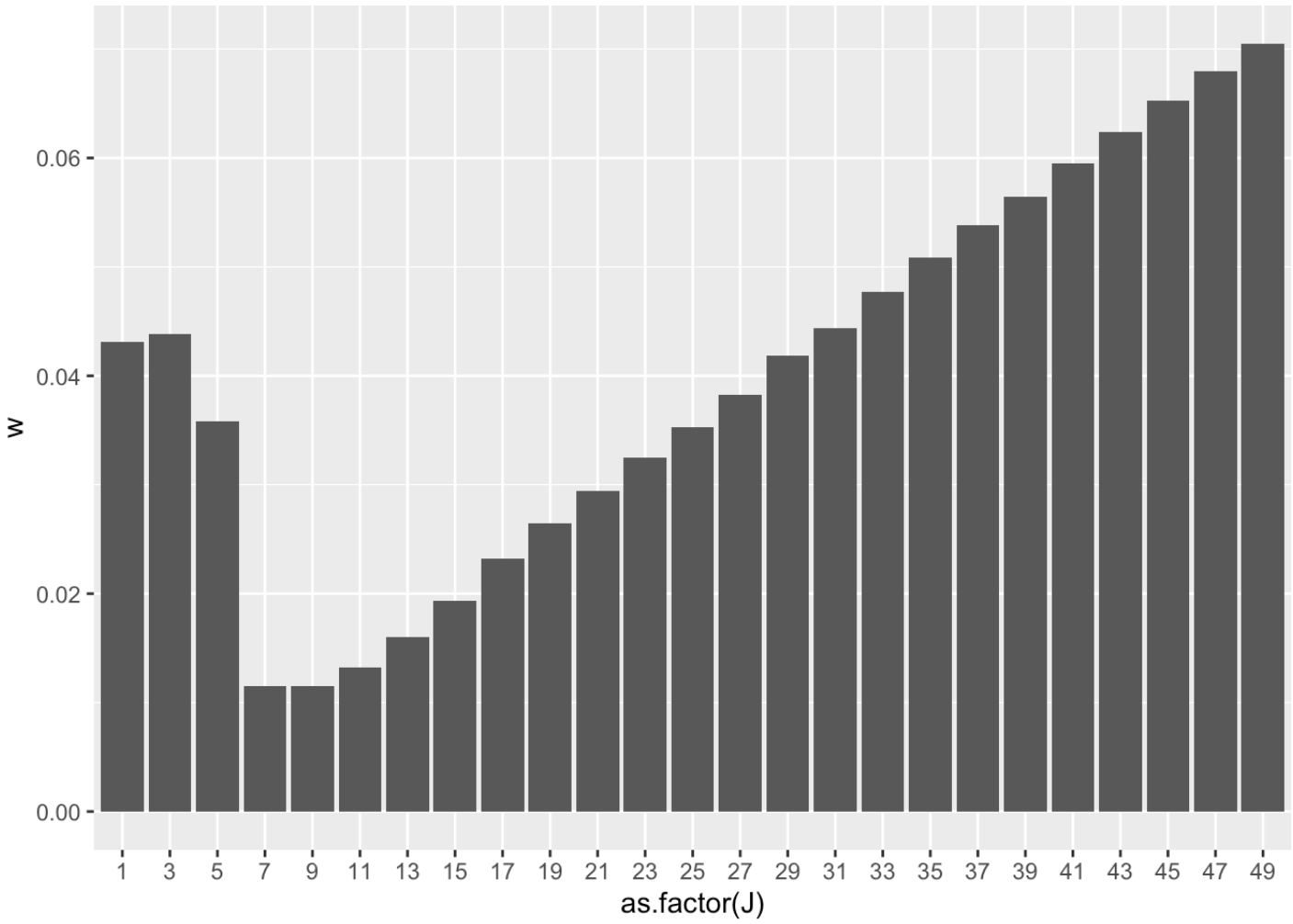
SOLUTION

```
spline_evidence <- purrr::map_dbl(all_spline_models, "log_evidence")
```

```
spline_weights <- spline_evidence / sum(spline_evidence)

spline_dof <- seq(1, 50, by = 2)

tibble::tibble(
  w = spline_weights
) %>%
  mutate(J = spline_dof) %>%
  ggplot(x = as.factor(J), y = w) + geom_bar(mapping = aes( x = as.factor(J), y = w),
stat = "identity")
```



It appears model with order 49 is the best model because it has the highest weight. Since it has the highest weight, it has the largest impact on the mean trend.

3g)

You have compared the models several different ways, are your conclusions the same?

How well do the assessments from the data split comparison compare to the Evidence-based assessment? If the conclusions are different, why would they be different?

SOLUTION

The data split comparison compared to the Evidence-based assessment allows for much more comprehensive assessment of the models. The conclusions are different because the data split comparison allows for to prevent against overfitting while it appears that the Evidence-based assessment looks at which model has the lowest MAE on the training set.