Overview

This homework assignment is focused on working with linear models. You will use formulas discussed in lecture to study the important concepts associated with fitting linear models. You will practice fitting, predicting, and assessing linear model performance on a simple example. You will then apply those concepts to a more realistic application. You will train numerous linear basis function models ranging from simple to very complex. You will identify the best model considering training and hold-out test set performance. You will contrast the model selection based on a train/test split with an information metric via the log-Evidence (log marginal likelihood).

IMPORTANT: The RMarkdown assumes you have downloaded the data sets (CSV files) to the same directory you saved the template Rmarkdown file. If you do not have the CSV files in the correct location, the data will not be loaded correctly.

IMPORTANT!!!

Certain 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 free to add more code chunks if you would like.

Load packages

You will use the `tidyverse` in this assignment, as you have done in the previous assignments.

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.2     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.3     v tibble    3.2.1
## v lubridate 1.9.2     v tidyr     1.3.0
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

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

We introduced many important regression topics at the beginning of the semester. Our primary example back then involved a continuous output y and a single continuous input x. The mean trend, μ_n, was related to the input via a quadratic relationship:

$$\mu_n = \beta_0 + \beta_1 x_n + \beta_2 x_n^2$$

We did not go into the mathematical details of the regression problem earlier in the semester. Instead, we conceptually introduced uncertainty and confidence intervals. We also discussed their impact on selecting the best model. However, we have recently covered the necessary mathematical and statistical fundamentals to calculate the uncertainty! You will begin by assuming the likelihood noise, σ, is known in order to focus on the regression coefficients (the $\boldsymbol{\beta}$ parameters). You will therefore study the *scaled* or *relative* coefficient uncertainty. This is a useful starting point because it allows you to focus on the impact the **features** have on learning the regression coefficients.

You will study the *scaled* regression coefficient uncertainty several ways. You will compare the posterior uncertainty based on informative, vague, and no prior specifications with "big" and "small" data applications.

The data from the "quadratic demo" are read in the code chunk below and assigned to the `quad_data_big` object.

```
quad_data_big <- readr::read_csv('quad_demo_data.csv', col_names = TRUE)
```

```
## Rows: 30 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.
```

The glimpse below shows that there are 30 rows for the two continuous variables, the input `x` and the output `y`.

```
quad_data_big %>% glimpse()
```

```
## Rows: 30
## Columns: 2
## $ x <dbl> 1.18689051, 0.74887469, 0.48150224, -0.63588243, -0.10975984, -0.843~
## $ y <dbl> 1.3047004, 6.4105645, -2.5582175, 5.6122242, 0.6981725, -1.3924964, ~
```

The data are labeled as "big" because the 30 observations will serve the role as the "big" or "large" data set in this example. The "small" data version is created for you in the code chunk below by slicing the first 5 rows.

```
quad_data_small <- quad_data_big %>% slice(1:5)
```

The entire small data set is displayed for you below.

```
quad_data_small
```

```
## # A tibble: 5 x 2
##        x      y
##    <dbl>  <dbl>
## 1  1.19   1.30
## 2  0.749  6.41
## 3  0.482 -2.56
## 4 -0.636  5.61
## 5 -0.110  0.698
```

1a)

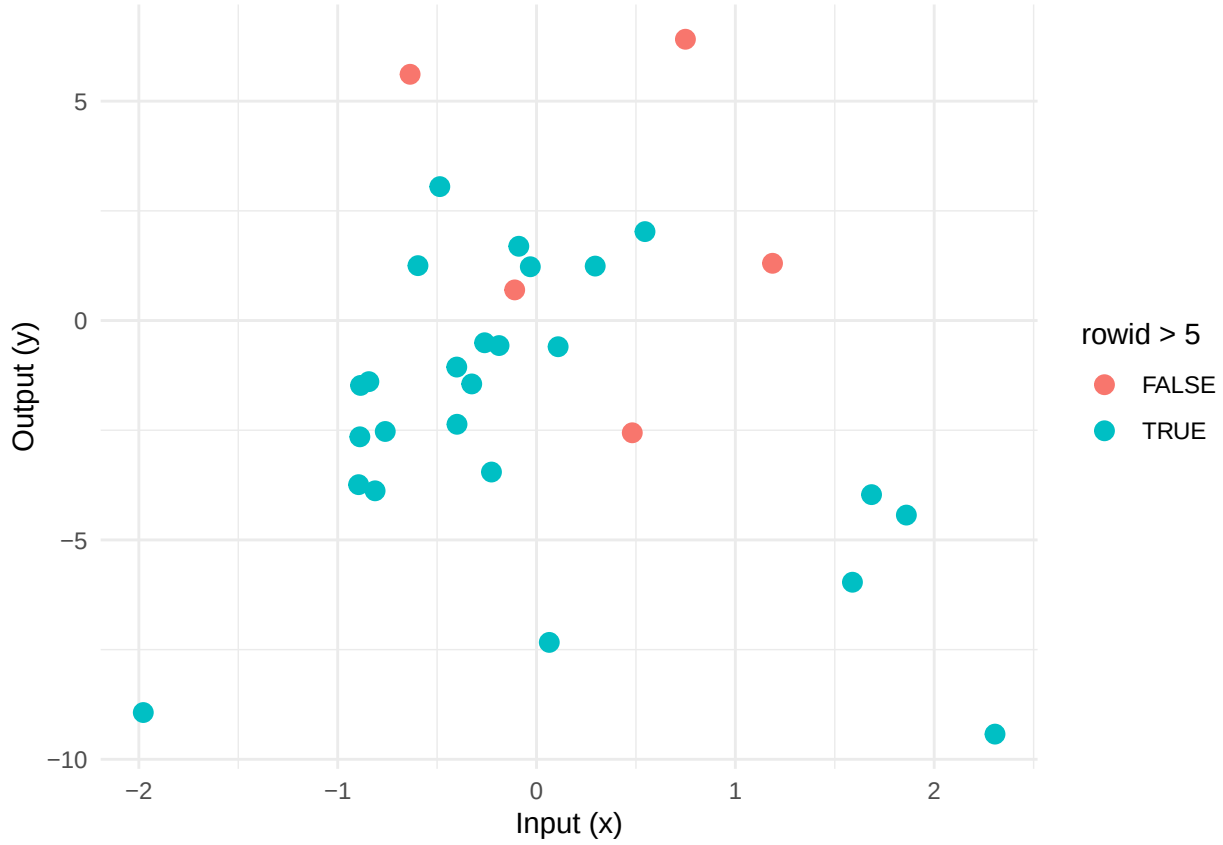
Let's start by visualizing the output to input relationship. You will use the `color` aesthetic to visually identify the first five rows which were separated out to create the small version of the data. You can add a row identifier column via the `tibble::rowid_to_column()` function. By default the new column will be named, `rowid`.

Create a scatter plot using `ggplot2` and the "full" or "big" data set to show the relationship between the output `y` and the input `x`. You should use `tibble::rowid_to_column()` to add the row identifier column. Map the `color` aesthetic within `geom_point()` to `rowid > 5` to identify the first five rows. Set the marker size to 3.

HINT: Don't forget the importance of the `aes()` function!!!!!!!

```
quad_data_big <- rowid_to_column(quad_data_big)

ggplot(quad_data_big, aes(x=x, y=y, color=rowid > 5))+
  geom_point(size =3)+
  labs(x = "Input (x)", y = "Output (y)") +
  theme_minimal()
```



SOLUTION

1b)

As stated in lecture, we need to create the **design matrix** before we can calculate anything associated with linear models. The design matrix contains the features. Those features can be input variables as well as quantities derived from the inputs. Features can therefore be *functions of* the inputs!

You must create **TWO** design matrices in this problem. Both design matrices will use a **quadratic** relationship between the input `x` and the mean trend. The first design matrix, `Xmat_quad_small`, corresponds to the sliced "small" data. The second design matrix, `Xmat_quad_big`, corresponds to the original "big" data.

You are allowed to use the `model.matrix()` function shown in lecture to create the design matrices.

Create two design matrices. Both must use a quadratic relationship between the input and the trend. The comments and variable names in the code chunks below state which data set to use in creating the design matrix.

HINT: You created polynomial features in previous assignments. Please review the syntax from those earlier assignments if you forget how to create the polynomial feature via the formula interface.

```
Xmat_quad_small <- model.matrix (y ~ x + I(x^2), data = quad_data_small)
```

```
Xmat_quad_big <- model.matrix (y ~ x + I(x^2), data = quad_data_big)
```

SOLUTION

1c)

How many rows and columns does EACH design matrix have?

Display the entire small data design matrix to the screen. What values are contained in the FIRST column and why?

SOLUTION The code chunks below show that "Xmat_quad_small" has 5 rows and 3 colums. Also "Xmat_quad_big" has 30 rows and 3 colums.

```
dim(Xmat_quad_small)
```

```
## [1] 5 3
```

```
dim(Xmat_quad_big)
```

```
## [1] 30  3
```

As we can see in the code chunk below, the first column consist of values of 1. This is because it is the "fake variable column" which corresponds to the intercept term.

```
Xmat_quad_small
```

```
##   (Intercept)          x     I(x^2)
## 1           1  1.1868905 1.40870907
## 2           1  0.7488747 0.56081331
## 3           1  0.4815022 0.23184440
## 4           1 -0.6358824 0.40434647
## 5           1 -0.1097598 0.01204722
## attr(,"assign")
## [1] 0 1 2
```

1d)

Let's now begin to study the posterior coefficient uncertainty. As stated previously, you will assume σ is known. Let's keep things simple with an assumed value of $\sigma = 1$. If this bothers you, you can also think of this as examining the *scaled* uncertainty. We discussed in lecture that σ acts as a scaling term on the posterior uncertainty. Thus, the "real" posterior uncertainty will be a multiple of what you are about to calculate. You are thus getting a rough idea on the behavior of the uncertainty due entirely to the inputs!

You will begin by assuming that the prior does not matter and thus are assuming an infinitely diffuse prior or a prior with infinite uncertainty.

Calculate the posterior covariance matrix for the regression coefficients assuming $\sigma = 1$. You must calculate the posterior covariance matrix for both design matrices.

The comments and variable names below state which data set to use.

```
### posterior covariance matrix for the SMALL data
sigma_true = 1
covmat_quad_small_noprior <- sigma_true^2*solve(t(Xmat_quad_small[drop=FALSE]) %*% Xmat_quad_small[drop=
```

```
### posterior covariance matrix for the BIG data
sigma_true = 1
covmat_quad_big_noprior <- sigma_true^2*solve(t(Xmat_quad_big[drop=FALSE]) %*% Xmat_quad_big[drop=FALSE]
```

SOLUTION

1e)

How many rows and columns does EACH posterior covariance matrix have?

SOLUTION Each posterior covariance matrix has 3 rows and 3 colums as it can be seen in the code chunks below.

```
dim(covmat_quad_small_noprior)
```

```
## [1] 3 3
```

```
dim(covmat_quad_big_noprior)
```

```
## [1] 3 3
```

1f)

What is the posterior standard deviation on the quadratic feature for the small data case? What is the posterior standard deviation on the quadratic feature for the big data case?

SOLUTION The code chunks below shows that the posterior standard deviation on the quadratic feature for the small data case is 1.3021277, and the posterior standard deviation on the quadratic feature for the big data case is 0.1566305.

```
post_std_quad_small_noprior <- sqrt(diag(sigma_true^2*solve(t(Xmat_quad_small[drop=FALSE]) %*% Xmat_quad
post_std_quad_small_noprior
```

```
## (Intercept)           x      I(x^2)
##   0.6795325   0.9713975   1.3021277
```

```
post_std_quad_big_noprior <- sqrt(diag(sigma_true^2*solve(t(Xmat_quad_big[drop=FALSE]) %*% Xmat_quad_big
post_std_quad_big_noprior
```

```
## (Intercept)           x      I(x^2)
##   0.2291992   0.2200419   0.1566305
```

1g)

The uncertainty displayed in the previous question does not tell us if a feature is statistically significant or not. We need to know where the coefficient is "located" to answer that. We thus need an *estimate* for the coefficient! We need output measurements to estimate coefficients. Thus, uncertainty can be studied *before* output data are collected but *significance* can only be examined **after** output data are collected.

You will now estimate the coefficients by calculating their posterior mean assuming the prior does not matter. To help you with this question, the output variables are separated from the dataframes and converted to the correct data type for you in the code chunk below. The output column vector for the small data case is displayed below.

```
y_quad_big_col <- quad_data_big %>% pull(y) %>% as.matrix()

y_quad_small_col <- quad_data_small %>% pull(y) %>% as.matrix()
```

```
y_quad_small_col
```

```
##            [,1]
## [1,]  1.3047004
## [2,]  6.4105645
## [3,] -2.5582175
## [4,]  5.6122242
## [5,]  0.6981725
```

You **MUST** continue to assume that the prior does not matter and thus are assuming an infinitely diffuse prior or a prior with infinite uncertainty.

Calculate the regression coefficient posterior mean assuming $\sigma = 1$ and the prior does not matter.

The comments and variable names below state which data set to use.

Display the posterior means for both data cases to the screen.

Would the values change if σ was assumed to equal 5 instead of 1?

```
### posterior mean vector for the SMALL data
postmeans_quad_small_noprior <- (solve(t(Xmat_quad_small[drop=FALSE]) %*% Xmat_quad_small[drop=FALSE])%
postmeans_quad_small_noprior
```

SOLUTION

```
##                  [,1]
## (Intercept)  1.288385
## x           -3.210035
## I(x^2)       3.969612
```

```
### posterior mean vector for the BIG data
postmeans_quad_big_noprior <- (solve(t(Xmat_quad_big[drop=FALSE]) %*% Xmat_quad_big[drop=FALSE])%*%t(Xma
postmeans_quad_big_noprior
```

```
##                   [,1]
## (Intercept)  0.3435989
## x            0.8887829
## I(x^2)      -2.0368238
```

In both cases posterior means does not change if σ is changed because the noise σ does NOT explicitly impact the MLEs.

1h)

Calculate the posterior 95% uncertainty interval on the quadratic feature.

Assuming $\sigma = 1$, is the quadratic feature considered to be statistically significant in the small data case? Assuming $\sigma = 1$, is the quadratic feature considered to be statistically significant in the big data case?

SOLUTION

1) As we calculated in previous questions, for the small data case, the posterior standard deviation of the quadratic feature is 1.3021277, and the posterior mean of the quadratic feature is 3.969612. Thus the 95% uncertainty interval is [1.3653566,6.5738674]. Since the interval does not contain 0, we conclude that the quadratic feature is statistically significant.

2) Again, as we calculated in previous questions, for the big data case, the posterior standard deviation of the quadratic feature is 0.1566305, and the posterior mean of the quadratic feature is -2.0368238. Thus the 95% uncertainty interval is [-2.3500848,-1.7235628]. Since the interval does not contain 0, we conclude that the quadratic feature is statistically significant.

Problem 02

Let's now consider the influence of a prior on the posterior uncertainty. You will work with two priors in this problem. The first is an informative prior and the second is a vague or diffuse prior. Both priors are Multivariate Normal (MVN) distributions. Both assume the prior means are zero and the regression coefficients are independent. However, the prior standard deviations are different between the two priors.

The informative prior assumes a prior standard deviation of 1.4 for each coefficient, while the vague prior assumes a prior standard deviation of 30 for each coefficient.

2a)

You will specify the prior by creating the prior covariance matrix.

Create 2 prior covariance matrices. The first is the prior covariance matrix for the informative prior and the second is the prior covariance matrix for the vague prior. The variable names and comments state which prior you are working with.

Display each prior covariance matrix to the screen.

```
### prior covariance matrix for the INFORMATIVE prior
B0_inform <- ((1.4)^2) * diag(3)
B0_inform
```

SOLUTION

```
##      [,1] [,2] [,3]
## [1,] 1.96 0.00 0.00
## [2,] 0.00 1.96 0.00
## [3,] 0.00 0.00 1.96
```

```
### prior covariance matrix for the VAGUE prior
B0_vague <- ((30)^2) * diag(3)
B0_vague
```

```
##      [,1] [,2] [,3]
## [1,]  900    0    0
## [2,]    0  900    0
## [3,]    0    0  900
```

Insert code chunks to display the covariance matrices.

2b)

You will now calculate the posterior covariance matrix assuming $\sigma = 1$ for the small and "big" data cases.

First, let's focus on the results based on the **informative** prior.

Calculate the posterior covariance matrix based on the informative prior for both the small and big data cases. The variable names and comments state which data set to use.

```
### posterior covariance matrix INFORMATIVE prior and SMALL data
covmat_quad_small_inform <-   solve(solve(B0_inform)+ (t(Xmat_quad_small[drop=FALSE])%*%Xmat_quad_small
```

```
### posterior covariance matrix INFORMATIVE prior and BIG data
covmat_quad_big_inform  <-   solve(solve(B0_inform)+ (t(Xmat_quad_big[drop=FALSE])%*%Xmat_quad_big[drop=
```

SOLUTION

2c)

Next, let's consider the results based on the **vague** prior.

Calculate the posterior covariance matrix based on the vague prior for both the small and big data cases. The variable names and comments state which data set to use.

```
### posterior covariance matrix VAGUE prior and SMALL data
covmat_quad_small_vague <- solve(solve(B0_vague)+ (t(Xmat_quad_small[drop=FALSE])%*%Xmat_quad_small[dro
```

```
### posterior covariance matrix VAGUE prior and BIG data
covmat_quad_big_vague <- solve(solve(B0_vague)+ (t(Xmat_quad_big[drop=FALSE])%*%Xmat_quad_big[drop=FALS
```

SOLUTION

2d)

What is the posterior standard deviation on the quadratic feature based on the INFORMATIVE prior for both data cases? Are the posterior standard deviations similar to the result from 1f)?

SOLUTION

1) As it can be seen in the code chunks below, the posterior standard deviation on the quadratic feature based on the INFORMATIVE prior for small data case is 0.8847677. In problem 1f), the corresponding value was 1.3021277.
2) Again, as it can be seen in the code chunks below, the posterior standard deviation on the quadratic feature based on the INFORMATIVE prior for big data case is 0.1545165. In problem 1f), the corresponding value was 0.1566305.

```
post_std_inform_small <- sqrt(diag(covmat_quad_small_inform))
post_std_inform_small
```

```
## (Intercept)           x      I(x^2)
##   0.5550443   0.7297542   0.8847677
```

```
post_std_inform_big <-sqrt(diag(covmat_quad_big_inform))
post_std_inform_big
```

```
## (Intercept)           x      I(x^2)
##   0.2254803   0.2168627   0.1545165
```

2e)

What is the posterior standard deviation on the quadratic feature based on the VAGUE prior for both data cases? Are the posterior standard deviations similar to the result from 1f)?

SOLUTION

1) As it can be seen in the code chunks below, the posterior standard deviation on the quadratic feature based on the VAGUE prior for small data case is 1.3004233. In problem 1f), the corresponding value was 1.3021277.
2) Again, as it can be seen in the code chunks below, the posterior standard deviation on the quadratic feature based on the VAGUE prior for big data case is 0.1566258. In problem 1f), the corresponding value was 0.1566305.

```
post_std_vague_small <- sqrt(diag(covmat_quad_small_vague))
post_std_vague_small
```

```
## (Intercept)           x      I(x^2)
##   0.6790541   0.9704340   1.3004233
```

```
post_std_vague_big <-sqrt(diag(covmat_quad_big_vague))
post_std_vague_big
```

```
## (Intercept)           x      I(x^2)
##   0.2291909   0.2200348   0.1566258
```

2f)

What is the influence of the prior on the linear model regression coefficients?

SOLUTION

1) Using the results in Problem 2d), we can observe that informative prior affects the posterior standard deviation in the small data case, however it does not have much effect on the posterior standard deviation in the big data case.
2) Using the results in Problem 2e), we can observe that vague prior does not have much effect the posterior standard deviation in the small data case and the big data case.

Problem 03

Now that you have worked though interpreting linear model results, its time to fit a linear model! You will learn the unknown regression coefficients **and** the unknown likelihood noise, σ. From a Bayesian perspective, we no longer have simple formulas for the posterior distribution. We will use the Laplace Approximation to execute the Bayesian analysis.

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 J degrees-of-freedom.

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

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

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

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

3a)

You will fit the Bayesian linear model with the Laplace Approximation by programming the log-posterior function. However, before doing so you will create a list of required information, analogous to those you created in the previous homework assignments. The list will contain the data and the prior parameter specifications. The data consists of two major components. The output vector and the design matrix. You will need these details to execute the log-posterior function.

You will start out by fitting the quadratic trend function which you worked with in the previous two problems. As a reminder, the trend is:

$$\mu_n = \beta_0 + \beta_1 x_n + \beta_2 x_n^2$$

You will create the list of required information for the "big" data case which uses all observations.

Complete the code chunk below by filling in the fields of the `info_quad` list. Assign the output vector to `yobs` and the "big" data design matrix to `design_matrix`. Specify the shared prior mean, `mu_beta`, to be 0, the shared prior standard deviation, `tau_beta`, as 3. The prior rate parameter on the noise, `sigma_rate`, should be assigned to 1.

Please make sure that you assign `yobs` as a "regular" vector and NOT as a matrix data type.

```
info_quad <- list(
  yobs = y_quad_big_col,
  design_matrix = Xmat_quad_big,
  mu_beta = 0,
  tau_beta = 3,
  sigma_rate = 1
)
```

SOLUTION

3b)

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

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 **must** assume that the unknown $\boldsymbol{\beta}$-parameters are 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_quad` list you defined in Problem 3a).

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 β column vector. After you complete the function, test that it works by evaluating the log-posterior at two different sets of parameter values. Try 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 `-1161.363` for the -1 guess values, and a value of `-137.0484` 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()`).

```
lm_logpost <- function(unknowns, my_info)
{
  # specify the number of unknown beta parameters
  length_beta <- length(unknowns)-1

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

  # extract the unbounded noise parameter, varphi
  lik_varphi <- unknowns[length(unknowns)]

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

  # extract design matrix
  X <- my_info$design_matrix

  # calculate the linear predictor
  mu <- X%*%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
}
```

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

```
###
unknowns <- c(-1,-1,-1,-1)
lm_logpost(unknowns, info_quad)
```

```
## [1] -1161.363
```

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

```
###
unknowns <- c(1,1,1,1)
lm_logpost(unknowns, info_quad)
```

```
## [1] -137.0484
```

3c)

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_quad` object. Print the posterior mode and posterior standard deviations to the screen.

Should you be concerned about the initial guess impacting the posterior results?

```
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]))
}
```

SOLUTION Fit the Bayesian linear model.

Since we still have a linear model, there is one unique maximum. So different initial guesses should still give us the same maximum values in the Laplace Approximation. Let's check by trying out two different initial guesses. The first uses zeros for all parameters, and the second uses random initial guess.

```
laplace_quad <- my_laplace(rep(0, 4), lm_logpost, info_quad)
### set the seed for reproducibility
```

```
set.seed(71231)
second_guess <- rnorm(4)
laplace_quad_2 <- my_laplace(second_guess, lm_logpost, info_quad)
```

Let's now compare the identified posterior modes between the two different starting guesses. As shown in the print out below, although they are not exactly the same, for all practical purposes they are similar.

```
###
laplace_quad$mode
```

```
## [1]  0.2881074  0.8281532 -1.9829448  0.9813469
```

```
laplace_quad_2$mode
```

```
## [1]  0.2881776  0.8282522 -1.9827215  0.9813011
```

```
laplace_quad
```

```
## $mode
## [1]  0.2881074  0.8281532 -1.9829448  0.9813469
##
## $var_matrix
##               [,1]         [,2]         [,3]         [,4]
## [1,]  0.355808794  0.092705235 -0.144553819 -0.001571759
## [2,]  0.092705235  0.329804668 -0.108074637 -0.001735026
## [3,] -0.144553819 -0.108074637  0.167680211  0.001555013
## [4,] -0.001571759 -0.001735026  0.001555013  0.015168191
##
## $log_evidence
## [1] -81.55952
##
## $converge
## [1] "YES"
##
## $iter_counts
## [1] 44
```

Let's now compare the posterior covariance matrices. Again, they are not exactly the same, but for all practical purposes they are identical. So, we should not be concerned about the initial guess impacting the posterior results.

```
###
laplace_quad$var_matrix
```

```
##               [,1]         [,2]         [,3]         [,4]
## [1,]  0.355808794  0.092705235 -0.144553819 -0.001571759
## [2,]  0.092705235  0.329804668 -0.108074637 -0.001735026
## [3,] -0.144553819 -0.108074637  0.167680211  0.001555013
## [4,] -0.001571759 -0.001735026  0.001555013  0.015168191
```

```
laplace_quad_2$var_matrix
```

```
##               [,1]         [,2]         [,3]         [,4]
## [1,]  0.355777343  0.092697000 -0.144541926 -0.001569513
## [2,]  0.092697000  0.329775090 -0.108065884 -0.001731937
## [3,] -0.144541926 -0.108065884  0.167666846  0.001561607
## [4,] -0.001569513 -0.001731937  0.001561607  0.015166920
```

Finally, we can see posterior standard deviations for both initial cases below.

```
###
sqrt(diag(laplace_quad$var_matrix))
```

```
## [1] 0.5964971 0.5742862 0.4094877 0.1231592
```

```
sqrt(diag(laplace_quad_2$var_matrix))
```

```
## [1] 0.5964707 0.5742605 0.4094714 0.1231541
```

3d)

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 `post_samples_quad` object. You will then use the posterior samples to study the posterior distribution on the slope on the quadratic feature, β_2. The naming convention provided by the `generate_lm_post_samples()` function names β_2 as the `beta_02` column.

Complete the generate_lm_post_samples() function below. After completing the function, generate 2500 posterior samples from your `post_samples_quad` model. Create a histogram with 55 bins using `ggplot2` for the slope `beta_02`. Calculate the probability that the slope is positive.

```
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")) %>%
    # back transform varphi to sigma!!!
    mutate(sigma = exp(varphi) )
}
```

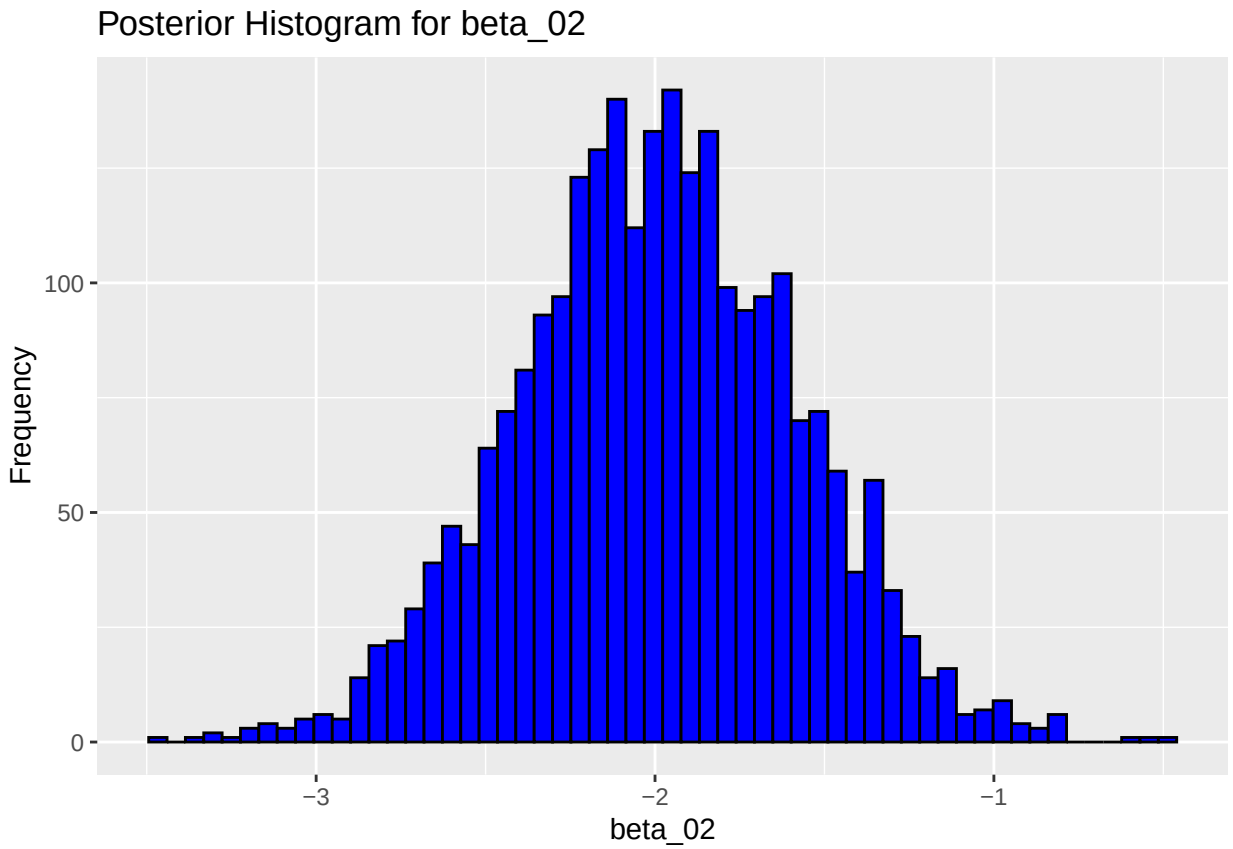
SOLUTION Generate posterior samples.

```
set.seed(87123)
post_samples_quad <- generate_lm_post_samples(laplace_quad,nrow(laplace_quad$var_matrix)-1,2500)
summary(post_samples_quad)
```

```
##     beta_00            beta_01           beta_02            varphi
##  Min.   :-1.81469   Min.   :-0.9905   Min.   :-3.4759   Min.   :0.6036
##  1st Qu.:-0.07724   1st Qu.: 0.4302   1st Qu.:-2.2602   1st Qu.:0.8968
##  Median : 0.31608   Median : 0.8492   Median :-1.9916   Median :0.9791
##  Mean   : 0.31548   Mean   : 0.8453   Mean   :-1.9928   Mean   :0.9799
##  3rd Qu.: 0.70240   3rd Qu.: 1.2425   3rd Qu.:-1.7133   3rd Qu.:1.0621
##  Max.   : 2.26012   Max.   : 2.7172   Max.   :-0.4963   Max.   :1.4156
##      sigma
##  Min.   :1.829
##  1st Qu.:2.452
##  Median :2.662
##  Mean   :2.685
##  3rd Qu.:2.892
##  Max.   :4.119
```

Create the posterior histogram on β_2, calculate the posterior 95% uncertainty interval, and calculate the probability that β_2 is greater than zero.

```
ggplot(data = post_samples_quad, aes(x = beta_02)) +
  geom_histogram(binwidth = (max(post_samples_quad$beta_02)-min(post_samples_quad$beta_02))/55, fill =
  labs(title = "Posterior Histogram for beta_02", x= "beta_02", y= "Frequency")
```



The posterior 95% uncertainty interval on β_2 is: [-2.816641,-1.1690116].

The probability that β_2 is greater than zero is 0 as calculated in the code chunk below.

```
count_g_0 <- sum(post_samples_quad$beta_02 > 0)
total_sampl <- nrow(post_samples_quad)
prob_sigma_g_0 <- count_g_0 / total_sampl
prob_sigma_g_0
```

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

Problem 04

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 we have discussed RMSE and R-squared throughout the semester, you will work with the Mean Absolute Error (MAE) metric.

4a)

The `post_lm_pred_trend_samples()` function is started in the code chunk below. This function generates posterior mean trend predictions. The first argument, `Xnew`, is a potentially new or test design matrix that we wish to make predictions for. The second argument, `Bmat`, is a matrix of posterior samples of the

β-parameters. 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`.

The `post_lm_pred_trend_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`.

```
post_lm_pred_trend_samples <- function(Xnew, Bmat)
{
  # matrix of posterior trend predictions
  Umat <- Xnew %*% t(Bmat)

  # return Umat, completed for you
  Umat
}
```

SOLUTION

4b)

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 β-parameter posterior samples and converts the object to a matrix. It also extracts the posterior samples on σ and converts to a vector. We will use the posterior samples on σ is the next assignment, but the line of code is included below for demonstration purposes.

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

  sigma_vector <- post %>% pull(sigma)

  post_lm_pred_trend_samples(Xnew, Bmat)
}
```

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

Make posterior predictions of the trend on the "big" quadratic demo training set. Assign the posterior trend predictions to the `post_trend_samples_quad` object. What are the dimensions of `post_trend_samples_quad`? What do the rows and columns correspond to?

SOLUTION Make posterior predictions on the training set.

```
post_trend_samples_quad <- make_post_lm_pred(Xmat_quad_big,post_samples_quad)
```

The dimensionality of the posterior predicted mean trend matrix is:

```
dim(post_trend_samples_quad)
```

```
## [1]   30 2500
```

What do the rows and columns correspond to?

Rows correspond to the number of prediction point (30), and columns correspond to the number of posterior samples (2500).

4c)

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! However, 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.

```
### error of the first posterior sample
error_quad_post_01 <- y_quad_big_col - post_trend_samples_quad[ ,1]

### error of the second posterior sample
error_quad_post_02 <- y_quad_big_col - post_trend_samples_quad[ ,2]

### error of the third posterior sample
error_quad_post_03 <- y_quad_big_col - post_trend_samples_quad[ ,3]
```

SOLUTION

4d)

You will now calculate the Mean Absolute Error (MAE) associated with each of the three error samples calculated in Problem 4c). However, before calculating the MAE, let's first consider the dimensions of the `error_quad_post_01`. What is the length of the `error_quad_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_quad_post_01` vector? Calculate the MAE associated with each of the 3 error vectors you calculated in Problem 4c).

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 ?

The dimension of the `error_quad_post_01` vector is 30. When we take the absolute value and then average across all elements in that vector, we are you averaging over the sum of the absolute values of the components of the `error_quad_post_01` vector. The MAE associated with each of the 3 error vectors are displayed below. The values are not the same. The MAE values may not be the same because they represent the average absolute error for different sets of predictions, and each posterior sample has its own set of model parameters, leading to variations in the errors.

```
###
```

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

```
mae_quad_post_01 <- mean(abs(error_quad_post_01))
mae_quad_post_01
```

```
## [1] 2.069975
```

```
mae_quad_post_02 <- mean(abs(error_quad_post_02))
mae_quad_post_02
```

```
## [1] 2.337694
```

```
mae_quad_post_03 <- mean(abs(error_quad_post_03))
mae_quad_post_03
```

```
## [1] 2.169175
```

?

4e)

In Problem 4d), 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 matches between the two and repeat the action over the other dimension (this repetition is referred to *broadcasting*). 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.

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

```
absError_quad_mat <- abs(post_trend_samples_quad - as.vector(y_quad_big_col))

dim(absError_quad_mat)
```

SOLUTION

```
## [1]   30 2500
```

4f)

You must now summarize the absolute errors values 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, `colMeans()` or `rowMeans()`, 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_quad` object. Print the data type (the class) of the `MAE_quad` to the screen and display its length. Check your result is consistent with the MAEs you previously calculated in Problem 4d).

SOLUTION Which function?

We should average down the columns, so we must use colMeans().

```
MAE_quad <- colMeans(absError_quad_mat)
class(MAE_quad)
```

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

```
### data type?
class(MAE_quad)
```

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

```
### length?
length(MAE_quad)
```

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

Check with the results you calculated previously.

```
MAE_quad[1]
```

```
## [1] 2.069975
```

```
MAE_quad[2]
```

```
## [1] 2.337694
```

```
MAE_quad[3]
```

```
## [1] 2.169175
```

Mean Absolute Error (MAE) associated with each of the first three error samples calculated in Problem 4d) and the corresponding values calculated above are equal.

4g)

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

Use the `quantile()` function to print out summary statistics associated with the MAE. You can use the default arguments, and thus pass in `MAE_quad` 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.

```
MAE_quantiles <- quantile(MAE_quad)

print(MAE_quantiles)
```

```
##       0%      25%      50%      75%     100%
## 1.933779 2.003383 2.072490 2.161738 2.734656
```

Why is the MAE uncertain?

The MAE depends on the model's predictions, and these predictions are based on the model parameters. In our case posterior samples are used to make predictions. This leads to variations in the MAE.

Problem 05

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_df`, will serve as the training set, and the second, `test_df`, will serve as a hold-out test set. You will only fit models based on the training set.

```
file_for_train <- "hw07_train_data.csv"

train_df <- readr::read_csv(file_for_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.
```

```
file_for_test <- "hw07_test_data.csv"

test_df <- readr::read_csv(file_for_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.
```

Glimpses are provided for each data set below.

```
train_df %>% glimpse()
```

```
## Rows: 150
## Columns: 2
## $ x <dbl> -2.7644452, -2.7250301, -2.9950975, -0.3434680, -1.7734161, 0.900513~
## $ y <dbl> -0.08761209, -0.51416878, 0.27839713, -0.13669238, -0.33689752, 0.06~
```

```
test_df %>% glimpse()
```

```
## Rows: 50
## Columns: 2
```

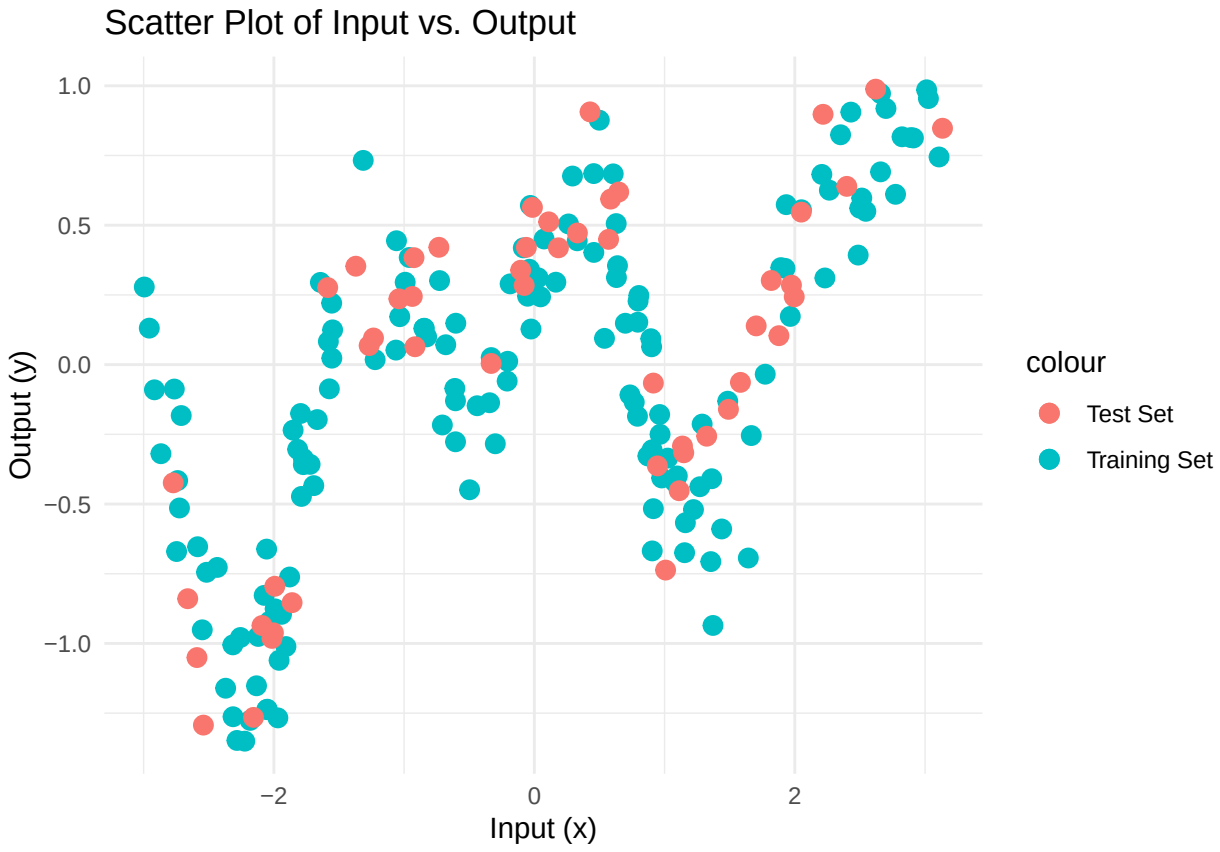
```
## $ x <dbl> 1.87813263, 1.49042058, -2.15634996, 1.97407736, -0.92403985, 1.1373~
## $ y <dbl> 0.10379726, -0.16038056, -1.26524958, 0.28497875, 0.38360882, -0.291~
```

5a)

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.

```
ggplot() +
  geom_point(data = train_df, aes(x= x, y= y, color ="Training Set"), size =3) +
  geom_point(data = test_df, aes(x =x, y = y, color ="Test Set"), size =3)+
  labs(title = "Scatter Plot of Input vs. Output",
       x = "Input (x)",
       y = "Output (y)")+
  theme_minimal()
```



SOLUTION

5b)

Hopefully you can tell from the previous visualization that the response is non-linearly related to the input. You will use basis functions to try and model this non-linear relationship. You could manually try out various models of different levels of complexity. However, let's build off the functions you created in the previous questions to programmatically train and assess many models of varying complexity.

Specifically, you will fit 25 different models to the training set. You will consider a one degree of freedom

natural spline up to 49 degrees of freedom natural spline. 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 would be tedious to define the necessary information by hand, manually train each model, generate the 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 begin 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 in the third code chunk and 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$design_matrix` variable are the training design matrices, while the matrices contained in `spline_matrices$test_matrix` are the corresponding 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 first 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 interior and boundary knots are extracted and saved to the `interior_knots` and `boundary_knots` objects, respectively. Create the training design matrix by calling the `model.matrix()` function with the `splines::ns()` function to create the basis for the `x` variable. The `interior_knots` object must be assigned to the `knots` argument and the `boundary_knots` object must be assigned to the `Boundary.knots` argument. 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?

NOTE: The `make_spline_basis_mats()` function **ASSUMES** the `train_data` and `test_data` arguments contain a column named `x`.

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)

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

  boundary_knots <- as.vector(attributes(train_basis)$Boundary.knots)

  train_matrix <- model.matrix( ~ splines::ns(x, knots = interior_knots, Boundary.knots = boundary_knots

  test_matrix <- model.matrix( ~ splines::ns(x, knots = interior_knots, Boundary.knots = boundary_knots)

  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_df,
                                  test_data = test_df)
```

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 x 2
##    design_matrix    test_matrix
##    <list>           <list>
##  1 <dbl [150 x 2]>  <dbl [50 x 2]>
##  2 <dbl [150 x 4]>  <dbl [50 x 4]>
##  3 <dbl [150 x 6]>  <dbl [50 x 6]>
##  4 <dbl [150 x 8]>  <dbl [50 x 8]>
##  5 <dbl [150 x 10]> <dbl [50 x 10]>
##  6 <dbl [150 x 12]> <dbl [50 x 12]>
##  7 <dbl [150 x 14]> <dbl [50 x 14]>
##  8 <dbl [150 x 16]> <dbl [50 x 16]>
##  9 <dbl [150 x 18]> <dbl [50 x 18]>
## 10 <dbl [150 x 20]> <dbl [50 x 20]>
## # i 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
```

How many rows are the training design matrix and how many rows are in the test matrix?

There are 150 rows in the training design matrix and 50 rows in the test design matrix.

5c)

Each element in the `spline_matrices$design_matrix` object is a separate design matrix. You will use this structure to programmatically train the models. The first code chunk creates a list of information which

stores the training set responses and defines the prior parameters. 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 β 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 trained natural spline which has 3 degrees-of-freedom 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_splines_train` list. Specify the prior mean and prior standard deviation on the β-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 β-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 β-parameters if you know the training design matrix?

Please make sure that you assign `yobs` as a "regular" vector and NOT as a matrix data type.

SOLUTION Assemble the list of required information.

```
info_splines_train <- list(
  yobs = train_df$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

  num_beta_params <- ncol(Xtrain)

  init_beta <- rnorm(num_beta_params)

  init_varphi <- log(rexp(1))

  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`. Please note the code chunk below may take a few minutes to complete.

```
set.seed(724412)
all_spline_models <- purrr::map(spline_matrices$design_matrix,
```

```
                                  manage_spline_fit,
                                  logpost_func = lm_logpost,
                                  my_settings = info_splines_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  1.988768e-06
## [2,] -2.947568e-03  2.062475e-02 -2.310636e-03 -2.388290e-03  4.967740e-08
## [3,] -3.960273e-02 -2.310636e-03  1.070993e-01  1.545448e-02 -5.267515e-06
## [4,] -4.056984e-03 -2.388290e-03  1.545448e-02  1.853818e-02 -1.040898e-06
## [5,]  1.988768e-06  4.967740e-08 -5.267515e-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"
```

5d)

With all 25 spline models fit, it is time to assess their performance and select the best one. 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 4. You will now define a function which performs all necessary 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 output 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, and predict the test set. 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 output

observations, `y_train`, the test set output 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.

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_samples)
{
  # generate posterior samples from the approximate MVN posterior
  post <- generate_lm_post_samples(mvn_result,nrow(mvn_result$var_matrix)-1,2500)
  Bmatx <- post %>% select(starts_with("beta_")) %>% as.matrix()

  # make posterior predictions on the training set
  pred_train <- Xtrain %*% t(Bmatx)

  # make posterior predictions on the test set
  pred_test <- Xtest %*% t(Bmatx)

  # calculate the error between the training set predictions
  # and the training set observations
  error_train <- abs(y_train - pred_train)

  # calculate the error between the test set predictions
  # and the test set observations
  error_test <- abs(y_test - pred_test)

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

  # calculate the MAE on the test set
  mae_test <- colMeans(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 = nrow(mvn_result$var_matrix)-2)
}
```

Apply the `calc_mae_from_laplace()` function to all 25 spline models. **Complete several of the arguments to the `purrr` function.**

Please make sure that you assign `yobs` as a "regular" vector and NOT as a matrix data type.

```
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 = as.vector(train_df$y),
                                          y_test = as.vector(test_df$y),
                                          num_samples = 2500)
```

A glimpse of the `all_spline_mae_results` object is provided below.

```
all_spline_mae_results %>% glimpse()
```

```
## Rows: 125,000
## Columns: 4
## $ post_id <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,~
## $ mae     <dbl> 0.3850699, 0.3923348, 0.3803612, 0.3985039, 0.3832649, 0.38627~
## $ dataset <chr> "training", "training", "training", "training", "training", "t~
## $ J       <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~
```

5e)

If you completed the `calc_mae_from_laplace()` function correctly, you should have an object with 125000 rows and 4 columns. Each split consists of the same number of rows, thus there are 62500 rows associated with the training split and 62500 rows associated with the test split. The object was structured in a "tall" or **long-format** with both splits and all degrees-of-freedom combined together and is thus a TIDY data object. We can easily summarize TIDY data objects with 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"`.

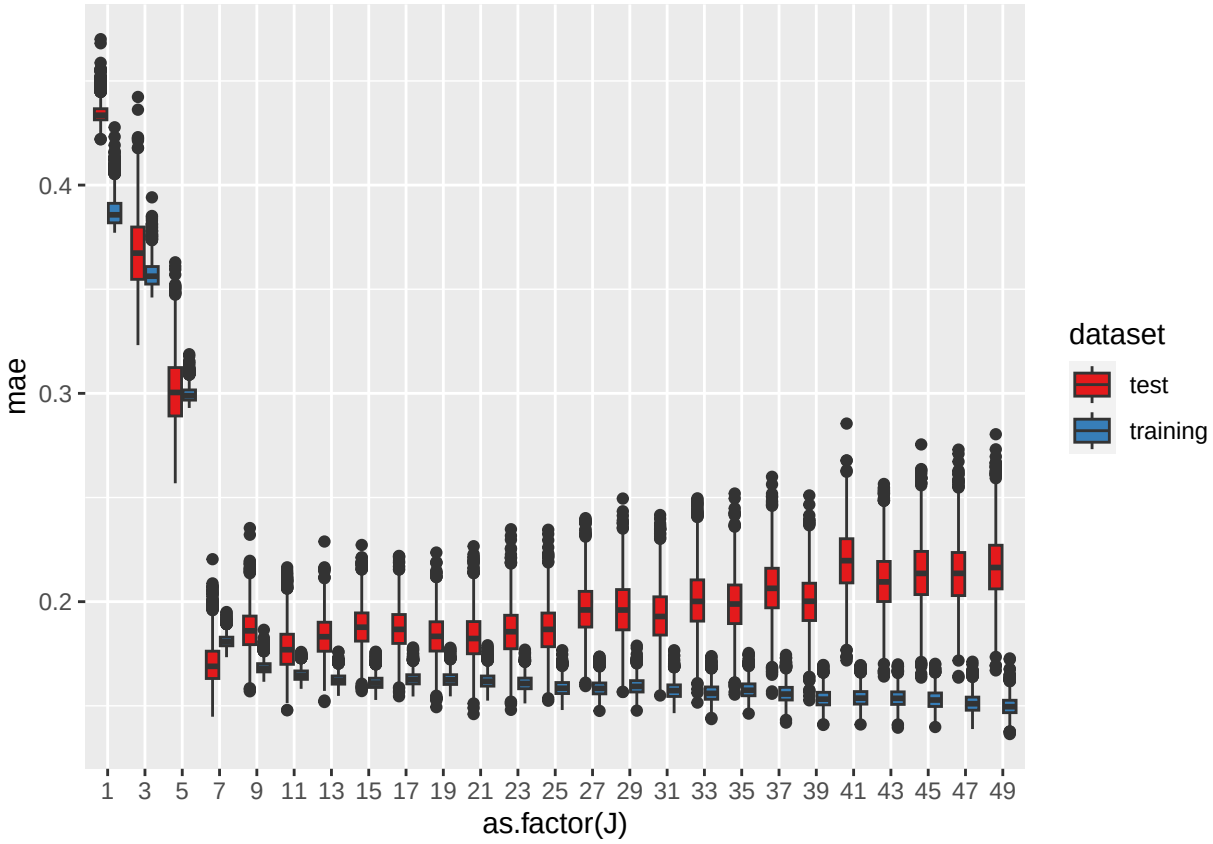
In the second code chunk, pipe the `all_spline_mae_results` object into `ggplot()` where you set the `x` aesthetic to `J` and the `y` aesthetic to `mae`. Use the `stat_summary()` function to calculate the median MAE for each spline order. You must set the `geom` argument within `stat_summary()` to be `'line'` and the `fun` argument to be `'median'`. Map the `color` aesthetic to the `dataset` variable and use the `scale_color_brewer()` scale with the `palette` argument set to `"Set1"`.

Please remember the importance of the `aes()` function when mapping the aesthetics!

Based on these visualizations which degrees-of-freedom appear to overfit the training data?

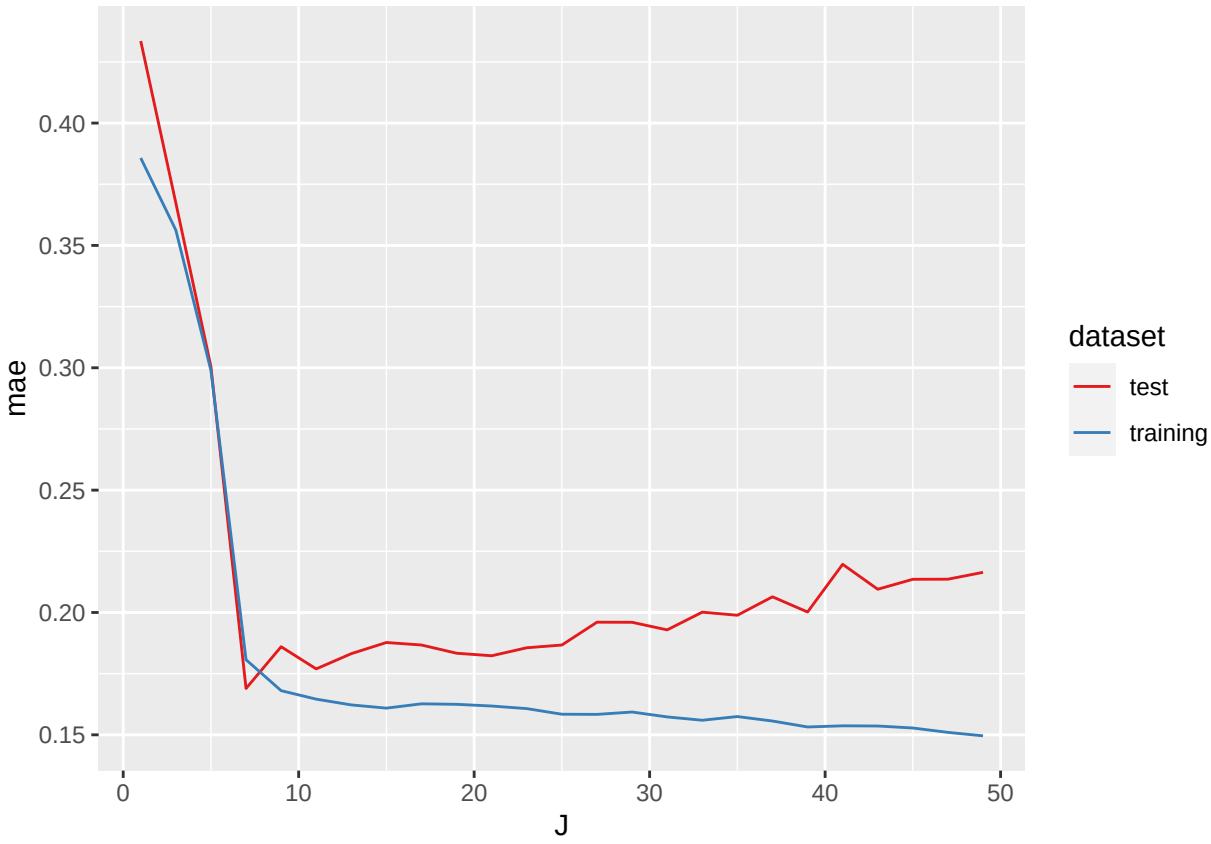
SOLUTION Summarize the posterior samples on the MAE on the training and test sets with boxplots.

```
ggplot(all_spline_mae_results, aes(x = as.factor(J), y = mae)) +
  geom_boxplot(aes(fill = dataset)) +
  scale_fill_brewer(palette = "Set1")
```



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

```
ggplot(all_spline_mae_results, aes(x = J, y = mae, color = dataset)) +
  stat_summary(geom = 'line', fun = 'median') +
  scale_color_brewer(palette = "Set1")
```



Which degrees-of-freedom overfit?

Overfitting typically occurs when the model performs very well on the training data but poorly on the test data. Also, if there is a clear trend where training MAE continues to decrease while test MAE starts to rise, it's an indication of overfitting. So, based of the last plot above we can conclude that, for DOF greater than 15, overfitting occurs.

5f)

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. However, 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?

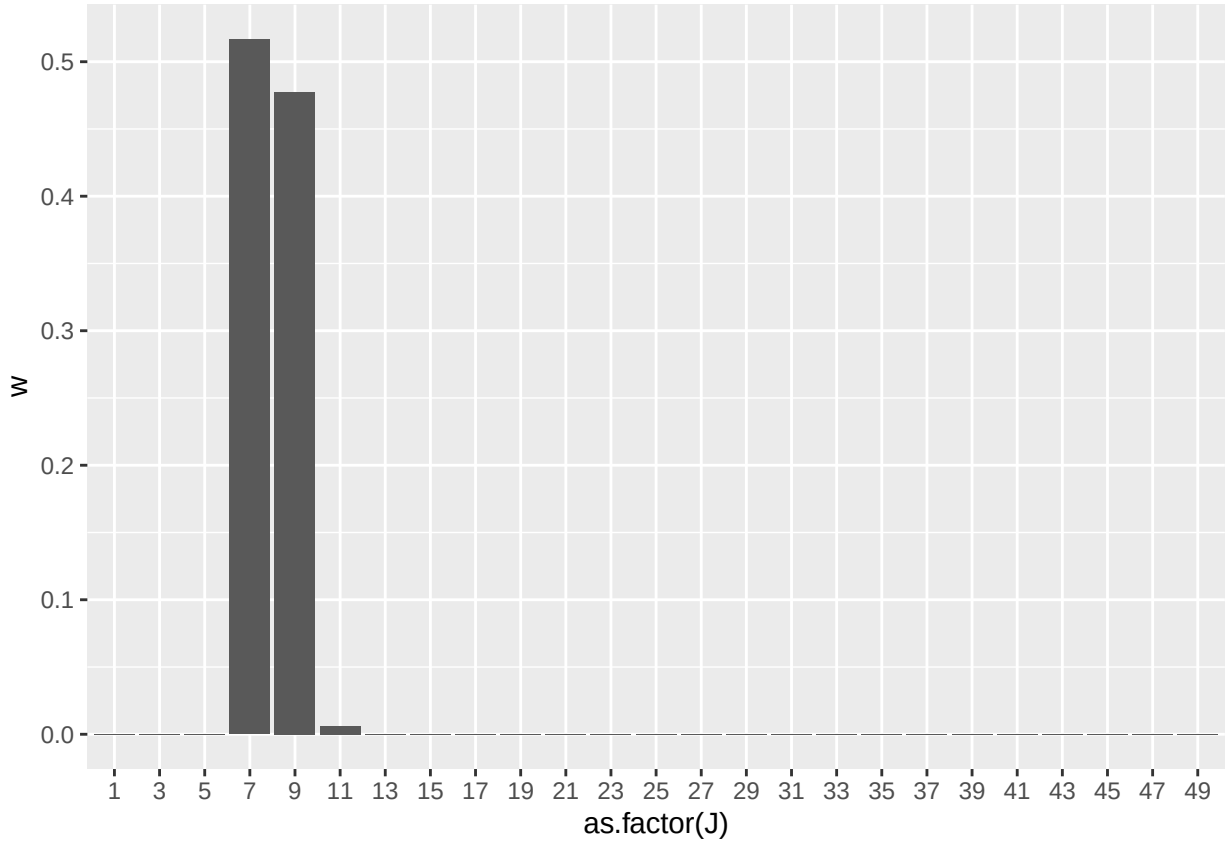
```
spline_evidence <- purrr::map_dbl(all_spline_models, "log_evidence")
```

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

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

result <- tibble::tibble(
  w = spline_weights
) %>%
  mutate(J = spline_dof)

  ggplot(result, aes(x = as.factor(J), y = w)) +
  geom_bar(stat ="identity")
```



SOLUTION

Which model is the best?

The 7 DOF spline has the highest weight, considered to be the most plausible. The 9 DOF spline has the second highest weight.

5g)

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

How well do the assessments from the single train/test split compare to the Evidence-based assessment? If the conclusions are different, why would they be different?

SOLUTION What do you think?

1) In the single train/test split case, the best model seems to be the one with DOF 7, according to the results in Problem 5e).

2) According to the Evidence-based assessment, the best model seems to be again the one with DOF 7, according to the results in Problem 5f).

If the conclusions were different, the reasons would be:

a) In a single train/test split, the performance assessment is highly dependent on the specific random partitioning of the data into training and test sets. This can lead to variations in the performance metrics. On the other hand, the Evidence-based assessment considers the entire dataset and provides a more stable estimate of model performance.

b) In a single train/test split, the training and test sets may not be representative of the overall data distribution. If there is a bias in the split, the assessment results can be skewed. The Evidence-based assessment considers the entire dataset and is less affected by potential bias.