INFSCI 2595 Homework: 04

Assigned: February 22, 2020; Due: March 8, 2020

Dustin (Ting-Hsuan) Ma

Submission time: March 8, 2020 at 9:00PM

Collaborators Stofanak, Patrick John

Overview

This assignment is focused on fitting and evaluating linear and generalized linear models. You will work with simple linear relationships, as well as basis functions. You will evaluate model performance on training and test sets, and see how those comparisons relate to assessing model performance using the Evidence (marginal likelihood).

Completing this assignment requires filling in missing pieces of information from existing code chunks, programming complete code chunks from scratch, typing discussions about results, and working with LaTeX style math formulas. A template .Rmd file is available to use as a starting point for this homework assignment. The template is available on CourseWeb.

IMPORTANT: Please pay attention to the `eval` flag within the code chunk options. Code chunks with `eval=FALSE` will **not** be evaluated (executed) when you Knit the document. You **must** change the `eval` flag to be `eval=TRUE`. This was done so that you can Knit (and thus render) the document as you work on the assignment, without worrying about errors crashing the code in questions you have not started. Code chunks which require you to enter all of the required code do not set the `eval` flag. Thus, those specific code chunks use the default option of `eval=TRUE`.

Load packages

This assignment uses the `dplyr` and `ggplot2` packages, which are loaded in the code chunk below. The assignment also uses the `tibble` package to create tibbles, and the `readr` package to load CSV files, and the `purrr` package for functional programming. All of the listed packages are part of the `tidyverse` and so if you downloaded and installed the `tidyverse` already, you will have all of these packages. This assignment will use the `MASS` package to generate random samples from a MVN distribution. The `MASS` package should be installed with base R, and is listed with the System Library set of packages.

```
library(ggplot2)
library(dplyr)
```

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 order 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.

```
train_01 <- readr::read_csv("https://raw.githubusercontent.com/jyurko/INFSCI_2595_Spring_2020/master/hw
```

```
## Parsed with column specification:
## cols(
##   x = col_double(),
##   y = col_double()
## )
```

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

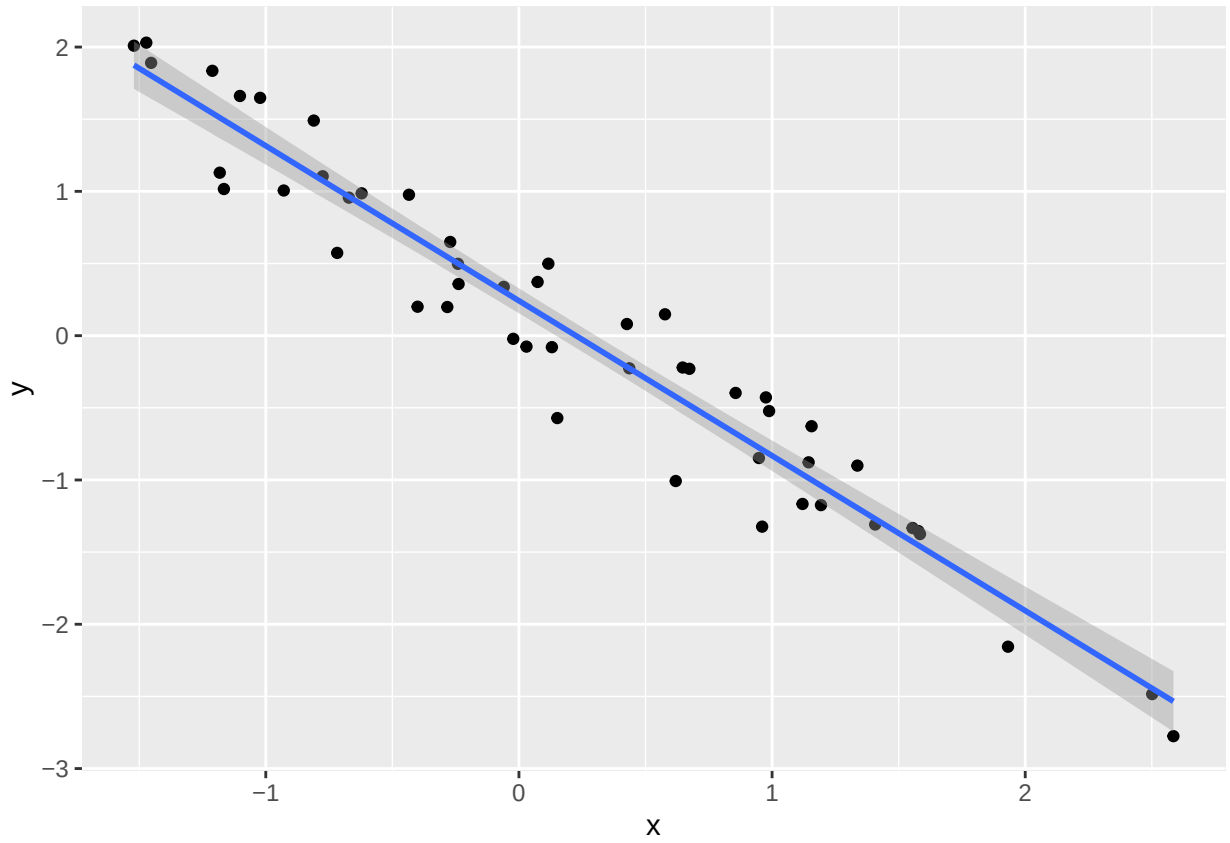
```
## Observations: 50
## Variables: 2
## $ x <dbl> -1.10210335, -0.71797339, 1.19319957, 1.55483625, 0.98823320, 1.1...
## $ y <dbl> 1.66107092, 0.57348497, -1.17436302, -1.33294329, -0.52248022, -1...
```

PROBLEM **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 Based on the plot, a linear relationship is most likely the relationship between the input and the responsed.

```
train_01 %>% ggplot(mapping = aes(x = x,y = y))+
  geom_point()+
  geom_smooth(method='lm')
```

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



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:

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

PROBLEM 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.

```
Xmat_01 <- glm(formula = y ~ x, data = train_01, family = "gaussian")

info_01 <- list(
  yobs = train_01$y,
  design_matrix = model.matrix(y ~ x, data = train_01),
```

```
  mu_beta = 0,
  tau_beta = 5,
  sigma_rate = 1
)
```

SOLUTION

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 β-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 β-parameters are listed before the unknown φ parameter in the `unknowns` vector. 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).

PROBLEM 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 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 `-296.5826` for the -1 guess values, and a value of `-123.9015` for the +1 guess values.

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

```
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
}
```

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

SOLUTION

```
## [1] -296.5826
```

```
lm_logpost(c(1,1,1),info_01)
```

```
## [1] -123.9015
```

1d)

The `my_laplace()` function is started for you in the code chunk below. You will complete the missing parts and then you will fit the Bayesian linear model from a startig guess of zero for all parameters.

PROBLEM Complete the `my_laplace()` function below and then fit the Bayesian linear model using a starting guess of zero for all parameters. Print the posterior mode and posterior standard deviations to the screen. Should you be concerned about the initial guess impacted 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$par
  h <- -solve(fit$hessian)
  p <- length(mode)
  int <- p/2 * log(2*pi) + 0.5 * log(det(h)) + logpost_func(mode, ...)

  list(mode = mode,
       var_matrix = h,
       log_evidence = int,
       converge = ifelse(fit$convergence == 0,
                         "YES",
                         "NO"),
       iter_counts = fit$counts[1])
}
```

SOLUTION Fit the Bayesian linear model.

```
laplace_01 <- my_laplace(c(0,0,0),lm_logpost,info_01)
```

Display the posterior modes and posterior standard deviations.

```
laplace_01$mode
```

```
## [1]  0.2417635 -1.0737303 -1.2259596
```

```
sqrt(diag(laplace_01$var_matrix))
```

```
## [1] 0.04243158 0.03979166 0.10056448
```

No, the initial guess should not matter. Since the posterior is also a gaussian distribution, the response surface is unimodal. Therefore, any initial guess within a reasonable range will converge towards the solution.

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 last argument, `num_samples`, specifies the total number of posterior samples to generate. The function is nearly complete, except you must back-transform from the unbounded φ parameter to the noise σ. After completing the function, generate 2500 posterior samples from your model stored in the `laplace_01` object. Be careful to specify the number of β-parameters correctly!

After generating the posterior samples you will study the posterior distribution on the slope, β_1.

PROBLEM Complete the `generate_lm_post_samples()` function by back-transforming from `varphi` to `sigma`. 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`, and then 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() %>% tbl_df() %>%
    purrr::set_names(c(sprintf("beta_%02d", (1:length_beta) - 1), "varphi")) %>%
```

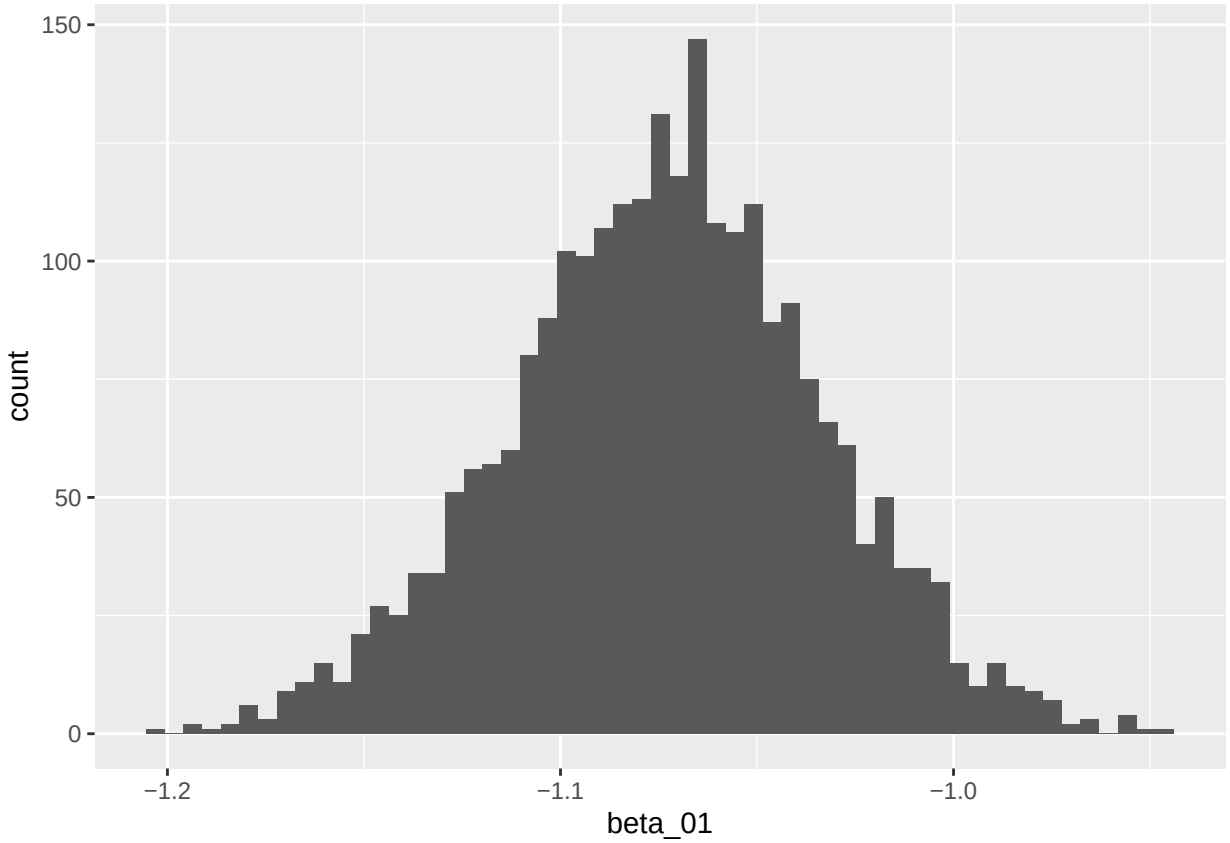
```
    mutate(sigma = exp(varphi))
}
```

SOLUTION Generate posterior samples.

```
set.seed(87123)
sample_size = 2500
post_samples_01 <- generate_lm_post_samples(laplace_01, 2, sample_size)
```

Create the posterior histogram on β_1.

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



The posterior probability that the slope is greater than zero is:

```
post_samples_01 %>% filter(beta_01 < 0) %>% mutate(prob = n()) %>% summarize(1-unique(prob/sample_size))
```

```
## # A tibble: 1 x 1
##   `1 - unique(prob/sample_size)`
##                            <dbl>
## 1                              0
```

The probability of the slope being grater than zero is zero.

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 discused 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 test design matrix. The second argument, `Bmat`, is a matrix of posterior samples of the β-parameters, and the third arugment, `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.

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

```
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
  Rmat <- matrix(rep(sigma_vector, M), M, byrow = TRUE)

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

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

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

SOLUTION

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 β-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 %>% dplyr::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.

PROBLEM 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(info_01$design_matrix, data.frame(post_samples_01))
```

The dimensionality of the posterior predicted mean trend matrix is:

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

```
## [1]   50 2500
```

The dimensionality of the posterior predicted response matrix is:

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

```
## [1]   50 2500
```

Both the posterior predicted mean trend and response are 50 rows by 2500 columns. This does match the number of predicted 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.

PROBLEM 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 <- train_01$y-post_pred_samples_01$Umat[,1]

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

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

We are considering the mean trend when calculating the error because we want to see how it fits to the trainning data.

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 dimensionality 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?

PROBLEM 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 The length of 'error_01_post_01' is 50 elements.

```
len_err01 <- length(error_01_post_01)
len_err02 <- length(error_01_post_02)
len_err03 <- length(error_01_post_03)
```

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

```
mae_01_post_01 <- sum(abs(error_01_post_01))/len_err01
mae_01_post_01
```

```
## [1] 0.2533644
```

```
mae_01_post_02 <- sum(abs(error_01_post_02))/len_err02
mae_01_post_02
```

```
## [1] 0.242996
```

```
mae_01_post_03 <- sum(abs(error_01_post_03))/len_err03
mae_01_post_03
```

```
## [1] 0.2432436
```

The three MAE are not the same. They are different because of the 2500 randomly generated predictions.

2e)

In Problem 2d) you calculated the MAE associated with the first 3 posterior samples. However, you can calculate 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.

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

SOLUTION The dimensionality of 'absE01mat' should be 50 rows by 2500 columns.

```
absE01mat <- abs(train_01$y - post_pred_samples_01$Umat)

dim(absE01mat)
```

```
## [1]   50 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?

PROBLEM 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 Average across the col, 'colMeans()', because you want to average across 50 elements and end up with 2500 predictions.

```
MAE_01 <- colMeans(absE01mat)

class(MAE_01)
```

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

```
length(MAE_01)
```

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

Check with the results you calculated previously.

```
MAE_01[1]
```

```
## [1] 0.2533644
```

```
MAE_01[2]
```

```
## [1] 0.242996
```

```
MAE_01[3]
```

```
## [1] 0.2432436
```

The first 3 values are consistent with the values calculated from problem 2d).

2g)

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

PROBLEM **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%
## 0.2383362 0.2417837 0.2459046 0.2524688 0.2892289
```

The MAE is uncertain because it is only trying to minimize error as far as the prior will allow.

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.

```
train_02 <- readr::read_csv("https://raw.githubusercontent.com/jyurko/INFSCI_2595_Spring_2020/master/hw
```

```
## Parsed with column specification:
## cols(
##   x = col_double(),
##   y = col_double()
## )
```

```
test_02 <- readr::read_csv("https://raw.githubusercontent.com/jyurko/INFSCI_2595_Spring_2020/master/hw_
```

```
## Parsed with column specification:
## cols(
##   x = col_double(),
```

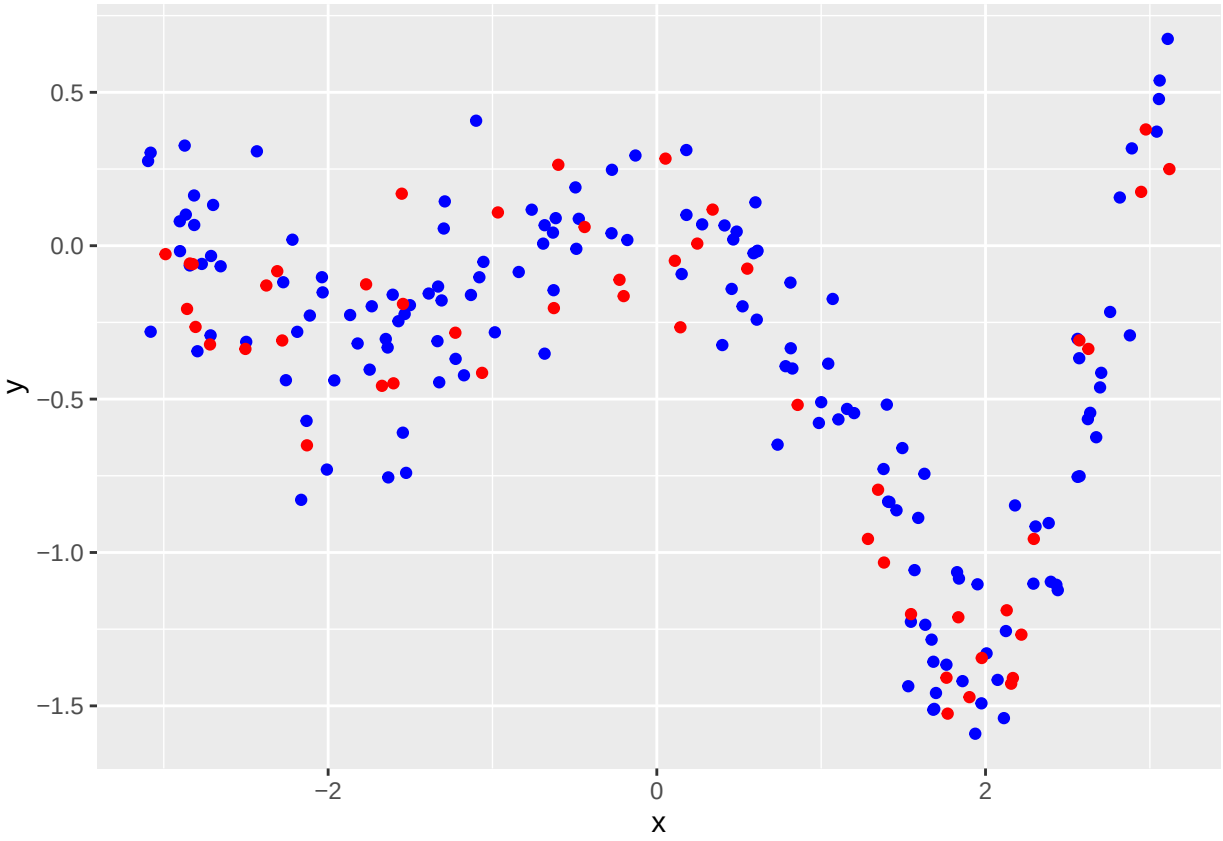
```
##   y = col_double()
## )
```

3a)

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

PROBLEM 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_02, aes(x = x, y = y), color = "blue") +
  geom_point(data = test_02, aes(x = x, y = y), color = "red")
```



SOLUTION

3b)

You will fit 25 different models to the training set. You will consider a first-order spline up to a 25th order spline. 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 order, `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 12 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.

PROBLEM Complete the code chunk below. You must specify the `splines::ns()` function call correctly such that the degrees-of-freedom, `df` argument equals the desired spline basis order 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(1:25, make_spline_basis_mats,
                                  train_data = train_02,
                                  test_data = test_02)
```

Get a glimpse of the structure of `spline_matrices`.

```
glimpse(spline_matrices)
```

```
## Observations: 25
## Variables: 2
## $ design_matrix <list> [<matrix[150 x 2]>, <matrix[150 x 3]>, <matrix[150 x...
## $ test_matrix   <list> [<matrix[50 x 2]>, <matrix[50 x 3]>, <matrix[50 x 4]...
```

Display the elements of `spline_matrices` to the screen.

```
spline_matrices
```

```
## # A tibble: 25 x 2
##    design_matrix         test_matrix
##    <list>                <list>
##  1 <dbl[,2] [150 x 2]>   <dbl[,2] [50 x 2]>
##  2 <dbl[,3] [150 x 3]>   <dbl[,3] [50 x 3]>
##  3 <dbl[,4] [150 x 4]>   <dbl[,4] [50 x 4]>
##  4 <dbl[,5] [150 x 5]>   <dbl[,5] [50 x 5]>
##  5 <dbl[,6] [150 x 6]>   <dbl[,6] [50 x 6]>
##  6 <dbl[,7] [150 x 7]>   <dbl[,7] [50 x 7]>
##  7 <dbl[,8] [150 x 8]>   <dbl[,8] [50 x 8]>
##  8 <dbl[,9] [150 x 9]>   <dbl[,9] [50 x 9]>
##  9 <dbl[,10] [150 x 10]> <dbl[,10] [50 x 10]>
## 10 <dbl[,11] [150 x 11]> <dbl[,11] [50 x 11]>
## # ... with 15 more rows
```

Check the dimensionality 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   3
```

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

```
## [1] 50  3
```

There are 150 rows in the traing matrix and 50 in the test matrix.

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.

PROBLEM 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 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?

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.

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

  init_beta <- rnorm(ncol(Xtrain), mean = my_settings$mu_beta, sd = my_settings$tau_beta)

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

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

Train all 25 spline models. 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)
```

Check the Laplace Approximation results of the second order spline.

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

```
## $mode
## [1]  0.009804092 -0.938417436 -0.575939575 -0.778865989
##
## $var_matrix
##               [,1]          [,2]          [,3]          [,4]
## [1,]  1.040097e-02 -2.210034e-02  6.203853e-04 -3.667047e-07
## [2,] -2.210034e-02  5.433992e-02 -2.475891e-03  9.032168e-07
## [3,]  6.203853e-04 -2.475891e-03  1.757128e-02  1.354379e-07
## [4,] -3.667047e-07  9.032168e-07  1.354379e-07  3.340253e-03
##
## $log_evidence
## [1] -114.6635
##
## $converge
## [1] "YES"
```

```
##
## $iter_counts
## function
##      110
```

Check that the optimizations successfully converged for each model.

```
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`.

PROBLEM 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_samples)
{
  # generate posterior samples from the approximate MVN posterior
  post <- generate_lm_post_samples(mvn_result, length(mvn_result$mode)-1, 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 <- y_train - pred_train$Umat

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

  # 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)
```

3e)

If you completed the `calc_mae_from_laplace()` function correctly, you should have an object with 60000 rows and just 4 columns. The object was structured in a "tall" or "long-format" in order to allow summarizing all of the posterior MAE samples with `ggplot2`. 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.y` to specify what function

to apply to the `y` aesthetic.

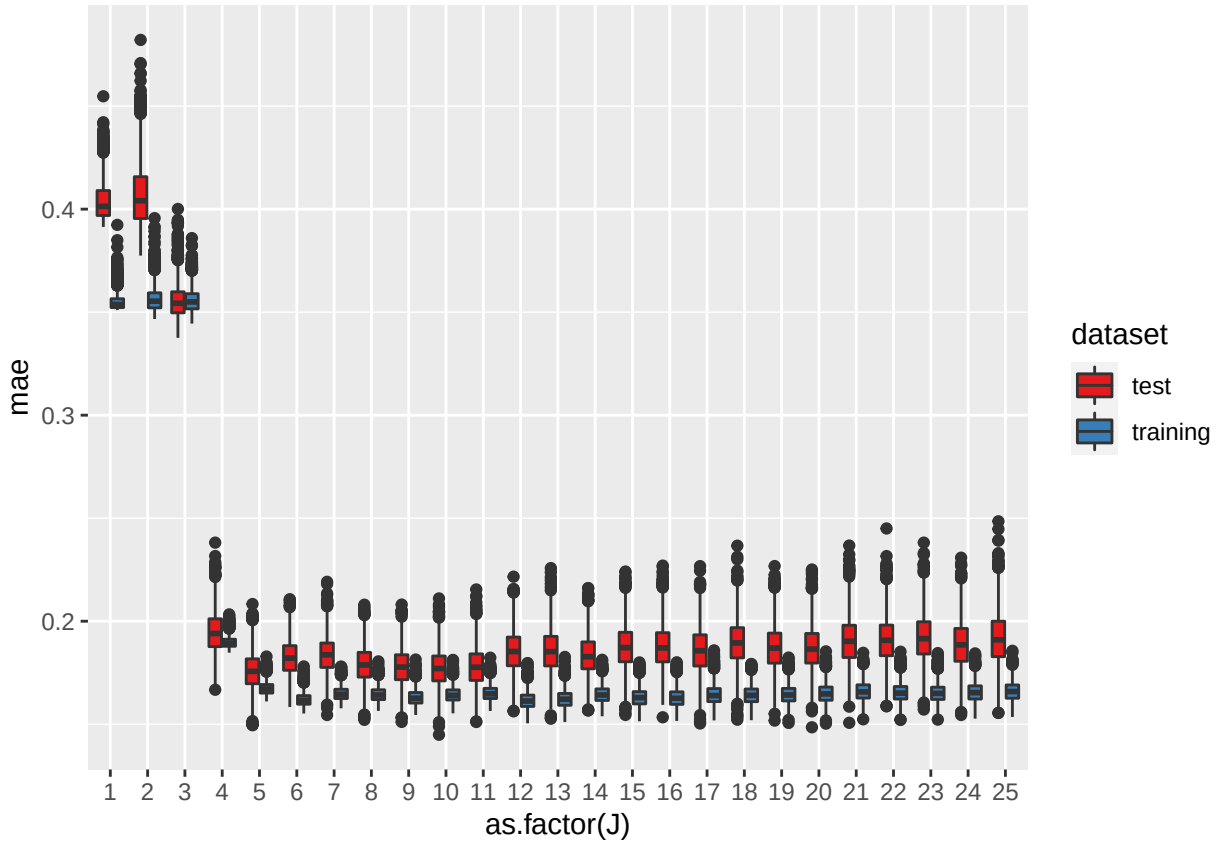
PROBLEM **Complete the two code chunks below. In the first code chunk, pipe the `all_spline_mae_results` object into `ggplot()`. Set the `x` aesthetic to be `as.factor(J)` and the `y` aesthetic to be `mae`. In the `geom_boxplot()` call, map the `fill` aesthetic to the `dataset` variable. Use the scale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

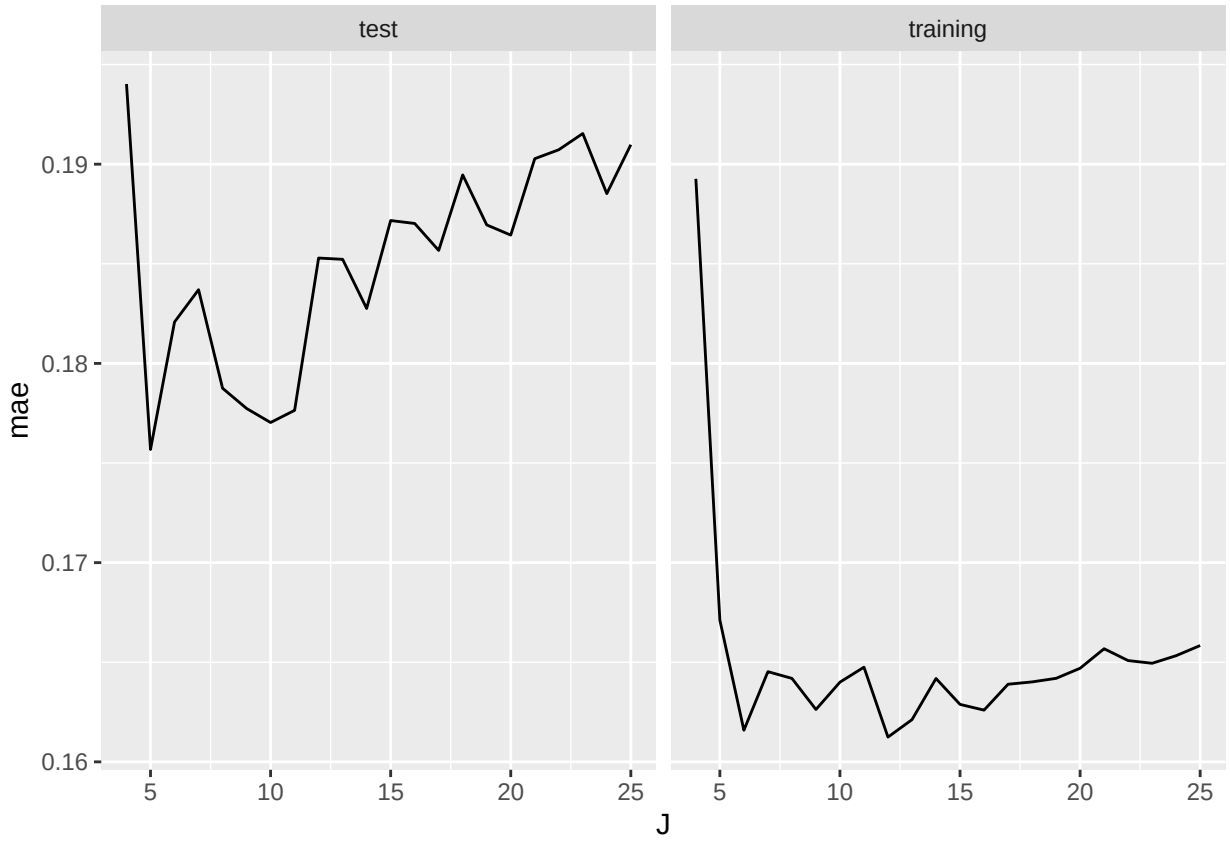


Summarize the posterior samples on the MAE on the training and test sets by focusing on the posterior median MAE values.

```
all_spline_mae_results %>%
  filter(J > 3) %>%
```

```
  ggplot(mapping = aes(x = J, y = mae)) +
  stat_summary(geom = "line", fun.y = "median") +
  facet_wrap("dataset")
```

```
## Warning: `fun.y` is deprecated. Use `fun` instead.
```



Based on the plots above, the fifth order spline is the best model to use for this training and test set.

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.

PROBLEM Calculate the posterior model weights associated with each spline model and assign the weights to the `spline_weights` variable. The `spline_weights` vector is assigned to the `w` variable in a `tibble` and the result is piped into `tibble::rowid_to_column()` where the row identification label is named `J` to correspond to the spline basis order. Pipe the result into a

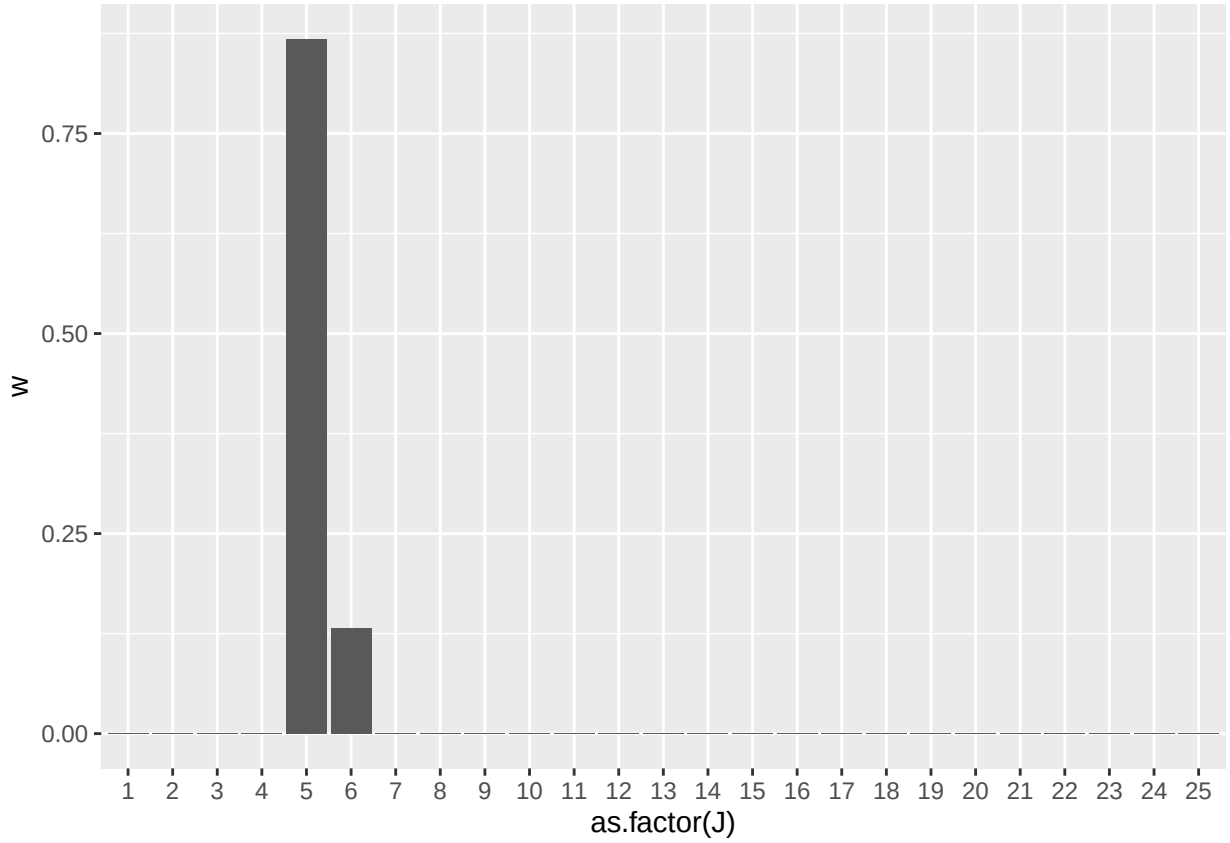
ggplot() call where you set the x aesthetic to as.factor(J) and the y aesthetic to w. Include a geom_bar() geometric object where the stat argument is set to "identity".

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))

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



The fifth order spline model is considered the best.

3g)

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

PROBLEM 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 Both methods of comparison agreed. Data split comparison showed that the test set for the 5th order spline had the least MAE, and the evidence based comparison showed that the 5th order spline had the most amount of weight.

Problem 04

In lecture we discussed how basis functions allow extending the linear model to handle non-linear relationships. It was also discussed how to generalize the linear modeling approach to binary outcomes with logistic regression. In this problem you will define the log-posterior function for logistic regression. By doing so, you will be able to directly contrast what you did to define the log-posterior function for the linear model in previous problems within this assignment.

4a)

The complete probability model for logistic regression consists of the likelihood of the response y_n given the event probability μ_n, the inverse link function between the probability of the event, μ_n, and the linear predictor, η_n, and the prior on all linear predictor model coefficients $\boldsymbol{\beta}$.

As in lecture, you will assume that the $\boldsymbol{\beta}$-parameters are a-priori independent Gaussians with a shared prior mean μ_β and a shared prior standard deviation τ_β.

PROBLEM Write out complete probability model for logistic regression. You must write out the n-th observation's linear predictor using the inner product of the n-th row of a design matrix $\mathbf{x}_{n,:}$ and the unknown $\boldsymbol{\beta}$-parameter column vector. You can assume that the number of unknown coefficients is equal to $D + 1$.

You are allowed to separate each equation into its own equation block.

HINT: The "given" sign, the vertical line, |, is created by typing `\mid` in a latex math expression. The product symbol (the giant PI) is created with `prod_{}^{}`.

SOLUTION The complete model with N individual likelihoods:

$$p(\beta \mid \mathbf{y_n}, \mathbf{x_{n,:}}) \propto \mathbf{p}(\mathbf{y_n} \mid \mu_\mathbf{n})\mathbf{p}(\beta)$$

$$\mu_n = logit^{-1}(\eta_n)$$
$$\eta_n = x_{n,:}\beta$$

The likelihood is a bernoulli distribution,

$$p(y_n \mid \mu_n) = Bernoulli(y_n \mid \mu_n) = Bernoulli(y_n \mid logit^{-1}(x_{n,:}\beta))$$

The probability of β-prior is dependent on the prior mean and standard deviation,

$$p(\beta) = \prod_{n=0}^{N}(normal(\beta_d \mid \mu_\beta, \tau_\beta))$$

4b)

The code chunk below loads in a data set consisting of two variables. A continuous input `x` and a binary outcome `y`. The binary outcome is encoded as 0 if the event does not occur and 1 if the event does occur. The `count()` function is used to count the number of observations associated with each binary class in the second code chunk. As shown below, the classes are roughly balanced.

```
train_03 <- readr::read_csv("https://raw.githubusercontent.com/jyurko/INFSCI_2595_Spring_2020/master/hw
```

```
## Parsed with column specification:
## cols(
##   x = col_double(),
##   y = col_double()
## )
```

```
train_03 %>% glimpse()
```

```
## Observations: 100
## Variables: 2
## $ x <dbl> -0.99886904, -1.35011298, 0.95776645, -0.60482885, 1.90268647, -0...
## $ y <dbl> 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1,...
```

```
train_03 %>% count(y)
```

```
## # A tibble: 2 x 2
##       y     n
##   <dbl> <int>
## 1     0    54
## 2     1    46
```

You will fit two logistic regression models to the dataset. The first will be a linear relationship for the linear predictor and the second will be a cubic relationship. The linear predictor expressions are written for you in the equation blocks below.

$$\eta_n = \beta_0 + \beta_1 x_n$$

$$\eta_n = \beta_0 + \beta_1 x_n + \beta_2 x_n^2 + \beta_3 x_n^3$$

Before creating the log-posterior function for logistic regression, you will create lists of necessary information in the same style that you did in the previous regression problems. You must create the design matrices associated with the linear and cubic relationships, `Xmat_03_line` and `Xmat_03_cube`. You must then complete the lists `info_03_line` and `info_03_cube` by setting the observed responses to the `yobs` variable, the design matrices to the `design_matrix` variable, and also set the $\boldsymbol{\beta}$ prior hyperparameters.

PROBLEM Create the design matrices for the linear and cubic relationships. Specify the lists of required information for both models. Specify the prior mean to be 0 and the prior standard deviation to be 5.

SOLUTION Create each design matrix.

```
Xmat_03_line <- model.matrix(y ~ x, data = train_03)

Xmat_03_cube <- model.matrix(y ~ I(x)+I(x^2)+I(x^3), data = train_03)
```

Create the lists of required information.

```
info_03_line <- list(
  yobs = train_03$y,
  design_matrix = Xmat_03_line,
  mu_beta = 0,
  tau_beta = 5
)

info_03_cube <- list(
```

```
  yobs = train_03$y,
  design_matrix = Xmat_03_cube,
  mu_beta = 0,
  tau_beta = 5
)
```

4c)

You will now define the log-posterior function for logistic regression, `glm_logpost()`. The first argument to `glm_logpost()` is the vector of unknowns and the second argument is the list of required information. You will assume that the variables within the `my_info` list are those contained in the `info_03_line` list you defined previously.

PROBLEM Complete the code chunk to define the `glm_logpost()` function. The comments describe what you need to fill in. Do you need to separate out the β-parameters from the vector of `unknowns`?

After you complete `glm_logpost()`, test it by setting the `unknowns` vector to be a vector of -1's for the linear relationship case. If you have succesfully programmed the function you should get a result equal to `-118.3933`.

SOLUTION No, you do not need to seperate out the β-parameters from the unknown vector. We can utillize the `as.matrix()` functions to turn the vector into a matrix, and do a dot product multiplication within R to return us the eta values.

```
glm_logpost <- function(unknowns, my_info)
{
  # extract the design matrix and assign to X
  X <- my_info$design_matrix

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

  # calculate the event probability
  mu <- boot::inv.logit(eta)

  # evaluate the log-likelihood
  log_lik <- sum(dbinom(x = my_info$yobs,
                        size = 1,
                        prob = mu,
                        log = TRUE))

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

  # sum together
  log_lik + log_prior
}
```

Test out your function using the linear relationship information and setting the unknowns to a vector of -1's.

```
glm_logpost(c(-1,-1),info_03_line)
```

```
## [1] -118.3933
```

4d)

Execute the Laplace Approximation for the linear and cubic models. Use initial guess values of zero for both models. After fitting both models, calculate the middle 95% uncertainty intervals for the cubic model parameters.

PROBLEM Perform the Laplace Approximation for the linear and cubic models. Set the initial guesses to vectors of zero for both models. Should you be concerned that the initial guess will impact the results?

After fitting, calculate the middle 95% uncertainty interval on the β parameters for both models. Which parameters contain zero in the middle 95% uncertainty interval?

SOLUTION No, for laplace approximation, as long as the hessian and gradient exists for the model, the initial guess should not matter.

```
laplace_03_line <- my_laplace(c(0, 0), glm_logpost, info_03_line)

laplace_03_cube <- my_laplace(c(0, 0, 0, 0), glm_logpost, info_03_cube)
```

Calculate the posterior middle 95% uncertainty interval on the parameters associated with each model.

```
sd_line <- diag(laplace_03_line$var_matrix)
sd_cube <- diag(laplace_03_cube$var_matrix)

beta_quantile_line <- matrix(, nrow = 2, ncol = 2)
for (count in 1:2){
  beta_quantile_line[count,1] <- qnorm(p = 0.025,
                                       mean = laplace_03_line$mode[count], sd = sqrt(sd_line[count]))
  beta_quantile_line[count,2] <- qnorm(p = 0.975,
                                       mean = laplace_03_line$mode[count], sd = sqrt(sd_line[count]))
}
beta_quantile_line
```

```
##            [,1]      [,2]
## [1,] -0.5483238 0.4109734
## [2,]  0.9622198 2.2942987
```

```
beta_quantile_cube <- matrix(, nrow = 4, ncol = 2)
for (count in 1:4){
  beta_quantile_cube[count,1] <- qnorm(p = 0.025,
                                       mean = laplace_03_cube$mode[count], sd = sqrt(sd_cube[count]))
  beta_quantile_cube[count,2] <- qnorm(p = 0.975,
                                       mean = laplace_03_cube$mode[count], sd = sqrt(sd_cube[count]))
}
beta_quantile_cube
```

```
##            [,1]      [,2]
## [1,] -0.5542633 0.8108316
## [2,] -1.0503106 2.1117873
## [3,] -2.0097870 0.8239758
## [4,] -0.7004492 3.2075328
```

Each row is associated with the β values, and the columns is associated with the value of percentile.

4e)

Let's compare the performance of the models using the Evidence-based assessment. Since we only have two models, calculate the Bayes Factor between the linear model and the cubic model. Based on the Bayes Factor which model do you think is better?

PROBLEM Calculate the Bayes Factor with the linear model in the numerator and the cubic model in the denominator. Which model is supported more from the data?

```
exp(laplace_03_line$log_evidence)/exp(laplace_03_cube$log_evidence)
```

SOLUTION

```
## [1] 4.513366
```

Since value is positive, and larger than 1. the linear model is a better fit for this dataset.

4f)

You will now spend a little more time with the cubic relationship results. Calculate the posterior correlation matrix between the unknown $\boldsymbol{\beta}$ parameters. Are any of the parameter highly correlated or anti-correlated?

PROBLEM Calculate the posterior correlation matrix between the parameters in the cubic model.

SOLUTION From the outputs below, β_1 and β_3 are strongly anti-correlated.

```
cov2cor(laplace_03_cube$var_matrix)
```

```
##              [,1]         [,2]       [,3]        [,4]
## [1,]  1.000000000 -0.001536955 -0.6956232  0.09505111
## [2,] -0.001536955  1.000000000  0.1156659 -0.84995947
## [3,] -0.695623192  0.115665948  1.0000000 -0.31555305
## [4,]  0.095051113 -0.849959468 -0.3155531  1.00000000
```

Problem 05

In Problem 04, you compared the linear and cubic relationships based on the Evidence. Your assessment considered how well the model "fit" the data via the likelihood, based on the constraints imposed by the prior. The likelihood examines how likely the binary outcome is given the event probability. Thus, the Evidence is considering if the observations are consistent with the modeled event probability. In this problem, you will consider point-wise error metrics by calculating the confusion matrix associated with the training set. Confusion matrices are useful because the accuracy and errors are in the same "units" of the data.

However, the logistic regression model predicts the event probability via the log-odds ratio. In order to move from the probability to the binary outcome a decision must be made. As discussed during the Applied Machine Learning portion of the course, the decision consists of comparing the predicted probability to a threshold value. If the predicted probability is greater than the threshold, classify the outcome as the event. Otherwise, classify the outcome as the non-event.

In order to classify the training points, you must make posterior predictions with the logistic regression models you fit in Problem 04.

5a)

Although you were able to apply the `my_laplace()` function to both the regression and logistic regression settings, you cannot directly apply the `generate_lm_post_samples()` function. You will therefore adapt `generate_lm_post_samples()` and define `generate_glm_post_samples()`. The code chunk below starts the function for you and uses just two input arguments, `mvn_result` and `num_samples`. You must complete the function.

PROBLEM Why can you not directly use the `generate_lm_post_samples()` function? Since the `length_beta` argument is NOT provided to `generate_glm_post_samples()`, how can you determine the number of β-parameters? Complete the code chunk below by first assigning the number of β-parameters to the `length_beta` variable. Then generate the random samples from the MVN distribution. You do not have to name the variables, you only need to call the correct random number generator.

SOLUTION Since `mvn_result` returns a list including the mode of the samples. I can use that to determine the degree of the linear model.

```
generate_glm_post_samples <- function(mvn_result, num_samples)
{
  # specify the number of unknown beta parameters
  length_beta <- length(mvn_result$mode)

  # generate the random samples
  beta_samples <- MASS::mvrnorm(n = num_samples,
                mu = mvn_result$mode,
                Sigma = mvn_result$var_matrix)

  # change the data type and name
  beta_samples %>%
    as.data.frame() %>% tbl_df() %>%
    purrr::set_names(sprintf("beta_%02d", (1:length_beta) - 1))
}
```

5b)

You will now define a function which calculates the posterior prediction samples on the linear predictor and the event probability. The function, `post_glm_pred_samples()` is started for you in the code chunk below. It consists of two input arguments `Xnew` and `Bmat`. `Xnew` is a test design matrix where rows correspond to prediction points. The matrix `Bmat` stores the posterior samples on the β-parameters, where each row is a posterior sample.

PROBLEM Complete the code chunk below by using matrix math to calculate the linear predictor at every posterior sample. Then, calculate the event probability for every posterior sample.

The `eta_mat` and `mu_mat` matrices are returned within a list, similar to how the `Umat` and `Ymat` matrices were returned for the regression problems.

HINT: The `boot::inv.logit()` can take a matrix as an input. When it does, it returns a matrix as a result.

```
post_glm_pred_samples <- function(Xnew, Bmat)
{
  # calculate the linear predictor at all prediction points and posterior samples
```

```
  eta_mat <- Xnew %*% t(Bmat)

  # calculate the event probability
  mu_mat <- boot::inv.logit(eta_mat)

  # book keeping
  list(eta_mat = eta_mat, mu_mat = mu_mat)
}
```

SOLUTION

5c)

The code chunk below defines a function `summarize_glm_pred_from_laplace()` which manages the actions necessary to summarize posterior predictions of the event probability. The first argument, `mvn_result`, is the Laplace Approximation object. The second object is the test design matrix, `Xtest`, and the third argument, `num_samples`, is the number of posterior samples to make. You must follow the comments within the function in order to generate posterior prediction samples of the linear predictor and the event probability, and then to summarize the posterior predictions of the event probability.

The result from `summarize_glm_pred_from_laplace()` summarizes the posterior predicted event probability with the posterior mean, as well as the 5th and 95th quantiles. If you have completed the `post_glm_pred_samples()` function correctly, the dimensions of the `mu_mat` matrix should be consistent with those from the `Umat` matrix from the regression problems. The posterior summary statistics summarize over all posterior samples. You must therefore choose between `colMeans()` and `rowMeans()` as to how to calculate the posterior mean event probability for each prediction point. The posterior quantiles are calculated for you.

PROBLEM Follow the comments in the code chunk below to complete the definition of the `summarize_glm_pred_from_laplace()` function. You must generate posterior samples, make posterior predictions, and then

HINT: The result from `post_glm_pred_samples()` is a list.

```
summarize_glm_pred_from_laplace <- function(mvn_result, Xtest, num_samples)
{
  # generate posterior samples of the beta parameters
  betas <- generate_glm_post_samples(mvn_result, num_samples)

  # data type conversion
  betas <- as.matrix(betas)

  # make posterior predictions on the test set
  pred_test <- post_glm_pred_samples(Xtest, betas)

  # calculate summary statistics on the posterior predicted probability
  # summarize over the posterior samples

  # posterior mean, should you summarize along rows (rowMeans) or
  # summarize down columns (colMeans) ???
  mu_avg = rowMeans(pred_test$mu_mat)

  # posterior quantiles
  mu_q05 = apply(pred_test$mu_mat, 1, stats::quantile, probs = 0.05)
```

```
  mu_q95 = apply(pred_test$mu_mat, 1, stats::quantile, probs = 0.95)

  # book keeping
  tibble::tibble(
    mu_avg = mu_avg,
    mu_q05 = mu_q05,
    mu_q95 = mu_q95
  ) %>%
    tibble::rowid_to_column("pred_id")
}
```

SOLUTION

5d)

Summarize the posterior predicted event probability associated with the training set for both the linear and cubic relationships. After making the predictions, a code chunk is provided for you which generates a figure showing how the posterior predicted probability summaries compare with the observed binary outcomes. Which of the two models appears to capture the trends in the binary outcomes better?

PROBLEM Call `summarize_glm_pred_from_laplace()` for the linear and cubic relationships on the training set. Specify the number of posterior samples to be 2500. Print the dimensions of the resulting objects to the screen. How many rows are in each data set?

The third code chunk below uses the prediction summaries to visualize the posterior predicted event probability on the training set. Which relationship seems more in line with the observations?

SOLUTION Execute the prediction summaries in the code chunk below.

```
post_pred_summary_03_line <- summarize_glm_pred_from_laplace(laplace_03_line,Xmat_03_line,2500)

post_pred_summary_03_cube <- summarize_glm_pred_from_laplace(laplace_03_cube,Xmat_03_cube,2500)
```

Print the dimensions of the objects to the screen.

```
dim(post_pred_summary_03_line)
```

```
## [1] 100   4
```

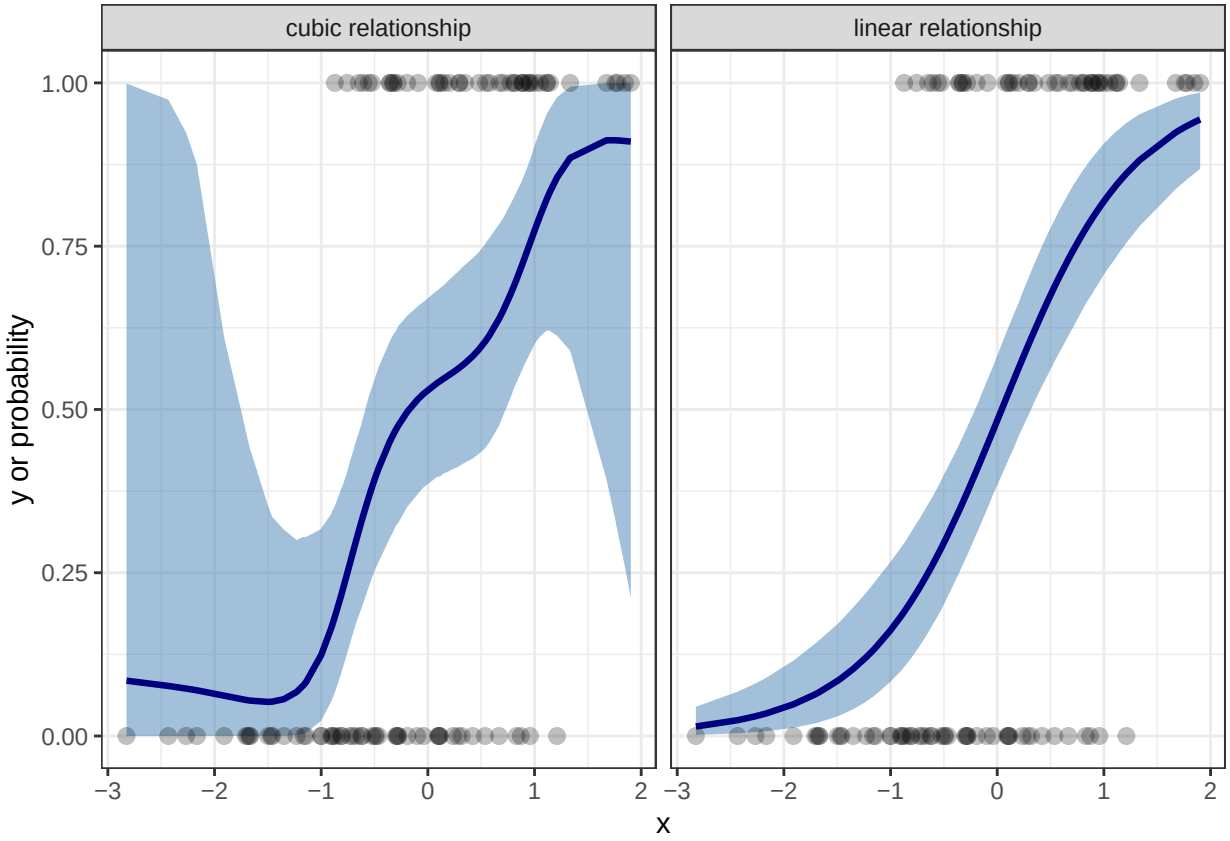
```
dim(post_pred_summary_03_cube)
```

```
## [1] 100   4
```

The figure below is created for you. The posterior predicted mean event probabilities are displayed by the navyblue curves. The posterior predicted middle 90% uncertainty intervals on the event probabilities are shown by the light blue ribbon.

```
post_pred_summary_03_line %>%
  mutate(type = "linear relationship") %>%
  bind_rows(post_pred_summary_03_cube %>%
              mutate(type = "cubic relationship")) %>%
  left_join(train_03 %>% tibble::rowid_to_column("pred_id"),
            by = "pred_id") %>%
  ggplot(mapping = aes(x = x)) +
  geom_ribbon(mapping = aes(ymin = mu_q05,
                            ymax = mu_q95,
```

```
                          group = type),
            fill = "steelblue", alpha = 0.5) +
geom_line(mapping = aes(y = mu_avg,
                        group = type),
          color = "navyblue", size = 1.15) +
geom_point(mapping = aes(y = y),
           size = 2.5, alpha = 0.25) +
facet_grid( . ~ type) +
labs(y = "y or probability") +
theme_bw()
```



The linear relationship is more inline with the observations.

5e)

You will now consider classifying the predictions based upon a threshold value of 0.5. You will compare that threshold value to the posterior predicted event probabilities associated with the training set. Although the Bayesian model provides a full posterior predictive distribution, you will work just with the posterior mean value. Thus, you will create a single confusion matrix, rather than considering the uncertainty in the confusion matrix.

Creating the confusion matrix is rather simple compared to some of the previous tasks in this assignment. The first step is to classify the prediction as event or non-event, which can be accomplished with an if-statement. The `ifelse()` function provides an "Excel-like" conditional statement, and is a simple way to perform the classification task. The syntax for `ifelse()` consists of three arguments, shown below:

```
ifelse(<conditional test>, <return if condition is TRUE>, <return if condition is FALSE>)
```

The first argument is the conditional test you wish to apply. The second argument is what will be returned if the condition is true, and the third argument is what will be returned if the condition is false.

You will use the `ifelse()` function to compare the posterior predicted mean event probability to the assumed threshold value of 0.5.

PROBLEM Pipe the `post_pred_summary_03_line` object into a `mutate()` call and create a new variable `pred_y` which is the result of an `ifelse()` operation. For the conditional test, return a value of `1` if the posterior predicted mean event probability is greater than 0.5, and return `0` otherwise. Repeat the process for the `post_pred_summary_03_cube` object.

```
post_pred_summary_03_line_b <- post_pred_summary_03_line %>%
  mutate(pred_y = ifelse(mu_avg > 0.5, 1, 0))

post_pred_summary_03_cube_b <- post_pred_summary_03_cube %>%
  mutate(pred_y = ifelse(mu_avg > 0.5, 1, 0))
```

SOLUTION

5f)

The code chunk below uses the `left_join()` function to merge the training data set, `train_03` with each of the posterior prediction summary objects. The results, `post_pred_summary_03_line_c` and `post_pred_summary_03_cube_c` both now have predicted classifications, `pred_y`, and observed outcomes `y`.

```
post_pred_summary_03_line_c <- post_pred_summary_03_line_b %>%
  left_join(train_03 %>% tibble::rowid_to_column("pred_id"),
            by = "pred_id")

post_pred_summary_03_cube_c <- post_pred_summary_03_cube_b %>%
  left_join(train_03 %>% tibble::rowid_to_column("pred_id"),
            by = "pred_id")
```

You now have everything you need to calculate the confusion matrix for the linear and cubic relationship models. A simple way to do this is with the `count()` function from `dplyr`, which counts the unique combinations of the variables you provide to it. `count()` returns a data.frame with the combination of the columns used to perform the grouping and counting operation, as well as a new column `n` which stores the number of rows associated with each combination.

PROBLEM Use the `count()` function to determine the confusion matrix associated with each relationship. How many true-positives, true-negatives, false-positives, and false-negatives does each relationship have?

SOLUTION The linear model has 33 true-positive, 40 true-negative, 14 false-positive, and 13 false-negative.

```
dplyr::count(post_pred_summary_03_line_c, y, pred_y)
```

```
## # A tibble: 4 x 3
##       y pred_y     n
##   <dbl>  <dbl> <int>
## 1     0      0    40
## 2     0      1    14
## 3     1      0    13
## 4     1      1    33
```

The cubic model has 35 true-positive, 37 true-negative, 17 false-positive, and 11 false-negative.

```
dplyr::count(post_pred_summary_03_cube_c, y, pred_y)
```

```
## # A tibble: 4 x 3
##       y pred_y     n
##   <dbl>  <dbl> <int>
## 1     0      0    38
## 2     0      1    16
## 3     1      0    12
## 4     1      1    34
```