BDA - Project

Anonymous

The regression analysis is one of the main statistical analysis techniques that is used to predict an outcome of interest based on a set of covariates or predictors. The regression problem is aimed to estimate the parameters $\beta \in R^p$ using

$$Y \propto \beta_0 + X\beta + \epsilon,$$

where $X \in R^{n\times p}$ matrix is the observed scores on the p predictor variables, β is a p-dimensional parameter vector of regression coefficients and ϵ is a standard normal variable. In the “Age of Big Data”, usually the number of covariates are large. In this setting sparsity means that only a few of these covariates have meaningful correlation with outcome. However, we do not have any prior information that which covariates are relevant and which are irrelevant. Regularized (or penalized) regressions are the statisticl techniques that have the ability to selects few variables of β to predict the outcome instead of using all β’s (Friedman, Hastie, and Tibshirani 2001). In Bayesian regression setting, this could be achieved by using a prior distribution on β that induce sparsity to the problem and performs a function similar to that of the penalty term in classical penalization.

In this projects we implemented different sparsity induce regression using stan and compare it how much it varies against theory frequentist (optimization) version of them.

Data set

Our course project concerns analyzing the data from the University of California, Irvine’s Machine Learning Repository on ‘Crimes and Communities Unnormalized’ (available at :) (Redmond and Baveja 2002) This Dataset combines socio-economic data from the ’90 Census, law enforcement data from the 1990 Law Enforcement Management and Admin Stats survey, and crime data from the 1995 FBI UCR. The dataset contains a large amount of information collected from each community which can be summarized in the broad categories of race, age, employment, marital status, immigration data and home ownership. The per capita violent crimes variable is calculated using per community population. The UCI dataset gave us data on the numbers of different types of crimes (like murder, rape, burglary, etc.) committed annually within each community and the sum of crime variables considered violent crimes in the United States: murder, rape, robbery, and assault in each community.

Using of this dataset, our goal is to build a linear regression model that can identify the correlation between the crime rates and violent per population (ViolentCrimesPerPop) and various socio-economic factors of that community such as population, ethnicty, age, incoeme, education, marital status, housing, etc.

Bayesian linear regression

Bayesian linear regression assums that the responses (outcomes) are sampled from a probability distribution such as normal distribution:

$$y \sim \mathcal{N}(\beta^T X, \sigma^2) = \mathcal{N}(\beta_0 + \sum_{j=1}^{p} x_{ij}\beta_j, \sigma^2)$$

where β_0 represents the intercept, β_j the regression coefficient for predictor j, and σ^2 is the residual variance.

Load the necessary lobraries

First of all we load the libraries that will be used:

```
library(tidyr)
library(rstan)
```

```
## Loading required package: StanHeaders
```

```
## Loading required package: ggplot2
```

```
## rstan (Version 2.19.2, GitRev: 2e1f913d3ca3)
```

```
## For execution on a local, multicore CPU with excess RAM we recommend calling
## options(mc.cores = parallel::detectCores()).
## To avoid recompilation of unchanged Stan programs, we recommend calling
## rstan_options(auto_write = TRUE)
```

```
##
## Attaching package: 'rstan'
```

```
## The following object is masked from 'package:tidyr':
##
##     extract
```

```
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
library(loo)
```

```
## This is loo version 2.1.0.
## **NOTE: As of version 2.0.0 loo defaults to 1 core but we recommend using as many
as possible. Use the 'cores' argument or set options(mc.cores = NUM_CORES) for an ent
ire session. Visit mc-stan.org/loo/news for details on other changes.
```

```
##
## Attaching package: 'loo'
```

```
## The following object is masked from 'package:rstan':
##
##     loo
```

```
library(ggplot2)
library(gridExtra)
library(bayesplot)
```

```
## This is bayesplot version 1.7.0
## - Online documentation and vignettes at mc-stan.org/bayesplot
```

```
## - bayesplot theme set to bayesplot::theme_default()
```

```
##    * Does _not_ affect other ggplot2 plots
```

```
##    * See ?bayesplot_theme_set for details on theme setting
```

```
theme_set(bayesplot::theme_default(base_family = "sans"))
library(shinystan)
```

```
## Loading required package: shiny
```

```
##
## This is shinystan version 2.5.0
```

```
source('stan_utility.R')
library(aaltobda)
SEED <- 48927 # set random seed for reproducability
set.seed(SEED)
library(grid)
library(gridExtra)
library(plotly)    # for 3D plotting
```

```
##
## Attaching package: 'plotly'
```

```
## The following object is masked from 'package:ggplot2':
##
##     last_plot
```

```
## The following object is masked from 'package:stats':
##
##     filter
```

```
## The following object is masked from 'package:graphics':
##
##     layout
```

```
util <- new.env()
source('stan_utility.R', local=util)
```

Reading the data

Next we need to read in data from the link:

```
crimedata <-read.csv("http://archive.ics.uci.edu/ml/machine-learning-databases/00211/
CommViolPredUnnormalizedData.txt",
                            header = FALSE, sep = ",", quote = "\"", dec = ".", fill
 = TRUE, comment.char = "",
                            na.strings="?",strip.white=TRUE,stringsAsFactors = defaul
t.stringsAsFactors())
```

The raw data consists of 147 attributes including 18 depending variables (potential outcomes) and 2215 observations.

preprocessing

The UCI dataset was filled with missing data, noted by ? marks. In order to clean the data, we first replaced all of the ? marks with NAs. There are 5 columns in the data set which are not informative for regreesion problem (e.g. community name, code, state, etc) . Since the explanatory variable of our analysis is ViolentCrimesPerPop, we removed every row in the data set in which the ViolentCrimesPerPop value for that observation was NA.

```
# Unnecessary fields
names(crimedata)[1] <- "communityname"
names(crimedata)[2] <- "state"
names(crimedata)[3] <- "countyCode"
names(crimedata)[4] <- "communityCode"
names(crimedata)[5] <- "fold"

# Potential goals
names(crimedata)[130] <- "murders"
names(crimedata)[131] <- "murdPerPop"
names(crimedata)[132] <- "rapes"
names(crimedata)[133] <- "rapesPerPop"
names(crimedata)[134] <- "robberies"
names(crimedata)[135] <- "robbbPerPop"
names(crimedata)[136] <- "assaults"
names(crimedata)[137] <- "assaultPerPop"
names(crimedata)[138] <- "burglaries"
names(crimedata)[139] <- "burglPerPop"
names(crimedata)[140] <- "larcenies"
names(crimedata)[141] <- "larcPerPop"
names(crimedata)[142] <- "autoTheft"
names(crimedata)[143] <- "autoTheftPerPop"
names(crimedata)[144] <- "arsons"
names(crimedata)[145] <- "arsonsPerPop"
names(crimedata)[146] <- "ViolentCrimesPerPop"
names(crimedata)[147] <- "nonViolPerPop"

possible_targets <- c("murders", "murdPerPop", "rapes", "rapesPerPop", "robberies",
"robbbPerPop",
             "assaults",  "assaultPerPop", "burglaries", "burglPerPop", "larcenies",
"larcPerPop",
             "autoTheft", "autoTheftPerPop", "arsons", "arsonsPerPop", "nonViolPerPo
p")

outcome <- c("ViolentCrimesPerPop")


notneededFeatures <- c(possible_targets, "communityname", "state",
                      "countyCode", "communityCode", "fold")

possible_predictors <- colnames(crimedata)[!(colnames(crimedata) %in%
                                                       notneededFeatures)]
crimedata <- crimedata[, names(crimedata) %in% possible_predictors]
out <- log(crimedata[outcome])

crimedata[outcome] <- out
inf_row_index <- which(grepl(-Inf, out[,1]))
crimedata_No_inf <- crimedata[-c(inf_row_index),]
#
crimedatacleaned <- na.omit(crimedata)
```

Split dataset intro train and test

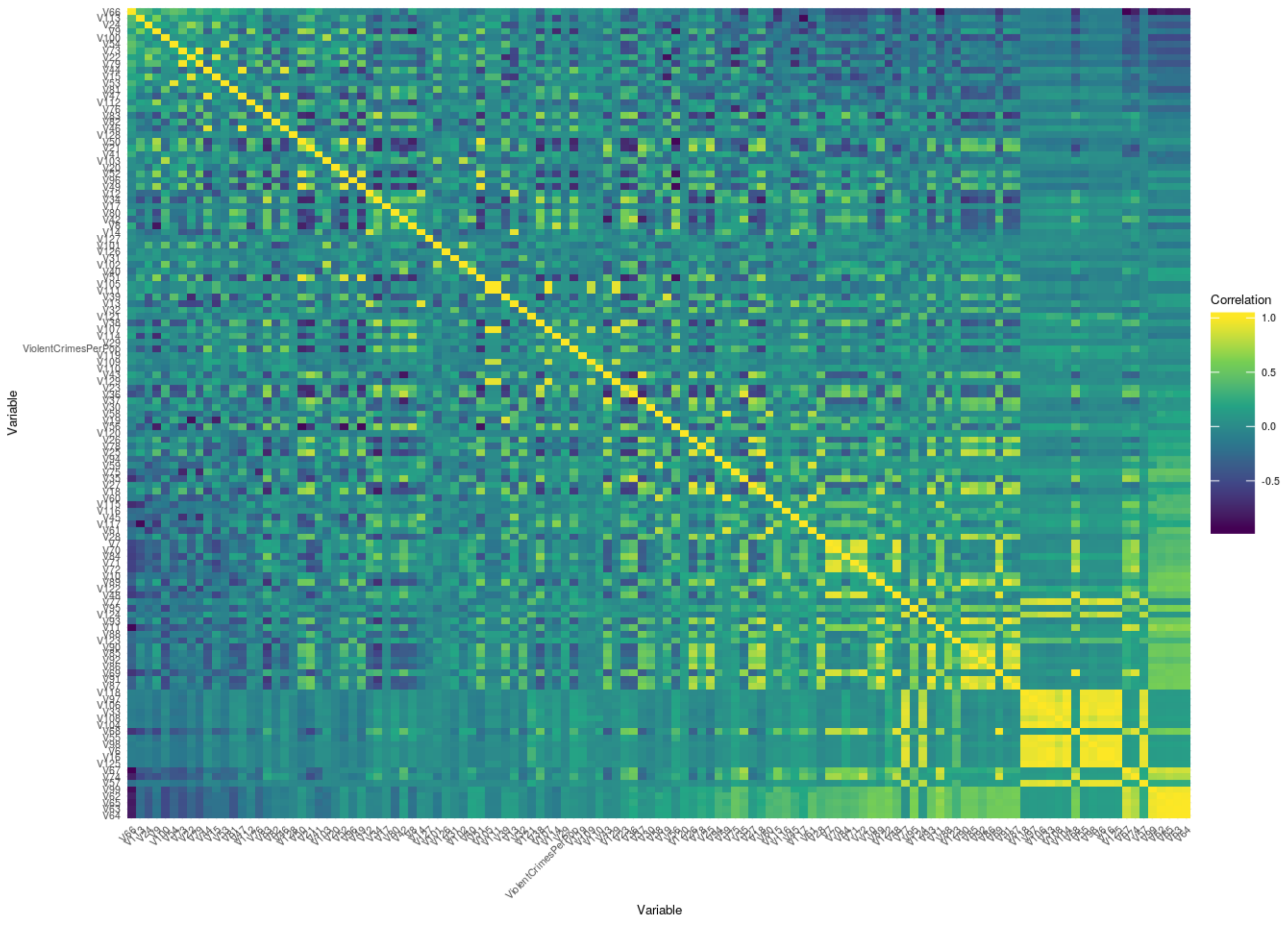
We need to split out the data into train and test sets. We train our model on 80% of data and test it on 20% of data. Next, since the dataset was unnormalized we normalized the data set and log transformed the output variable.

```
train_index <- sample(1:nrow(crimedatacleaned), 0.8 * nrow(crimedatacleaned))
test_index <- setdiff(1:nrow(crimedatacleaned), train_index)

y_crime <- crimedatacleaned[, names(crimedatacleaned) %in% outcome]
y_crime_train <- y_crime[train_index]
y_crime_test <- y_crime[test_index]

X <- colnames(crimedatacleaned)[!(colnames(crimedatacleaned) %in%
                                          outcome)]
X <-  crimedatacleaned[, names(crimedatacleaned) %in% X]
X_crime <- data.matrix(X, rownames.force = NA)

X_crime_train <- X_crime[train_index,]
X_crime_test <- X_crime[test_index,]

# Normalization
X_train <- scale(X_crime_train)

# Normalization: Scale test data using training data summary stats (no cheating!)
X_test <- scale(X_crime_test) #, center = means2, scale = SDs2)


y_train <- (y_crime_train)
y_test <- (y_crime_test)
```

Visualize correlation of train data

It would be useful to visualize the train data to get a sense of information about the data we are working with. We used correlation matrix to show the correlation between covariates and outcome.

```
# Plotting correlation matrix
cor(crimedatacleaned) %>%
  as.data.frame() %>%
  mutate(Var1 = factor(row.names(.), levels=row.names(.))) %>% # For nice order
  gather(Var2, Correlation, 1:125) %>%
  ggplot(aes(reorder(Var2, Correlation), # Reorder to visualize
             reorder(Var1, -Correlation), fill = Correlation)) +
  geom_tile() +
  scale_fill_continuous(type = "viridis") +
  xlab("Variable") +
  ylab("Variable") +
  theme_minimal(base_size = 5) +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



We then prepare data for our stan models:

```
# First, prepare data for Stan
data_crime <- list(N_train = nrow(X_train),
                  N_test  = nrow(X_test),
                  N_pred  = ncol(X_train),
                  y_train = y_train,
                  X_train = X_train,
                  X_test  = X_test)
```

Bayesian linear regression (un-regularized)

Even though we already suspect it won't be a good model for this data, it's still a good idea to start by fitting the simplest linear regression model for the start. We employ a linear regression model without informative prior information (default uniform priors on weights). The simple model in stan is as:

```
writeLines(readLines("uniform.stan"))
```

```
## data{
##     int N_train;             // "# training observations"
##     int N_test;              // "# test observations"
##     int N_pred;              // "# predictor variables"
##     vector[N_train] y_train; // "training outcomes"
##     matrix[N_train, N_pred] X_train; // "training data"
##     matrix[N_test, N_pred] X_test;   // "testing data"
## }
##
## parameters{
##  real beta_0; //intercept
##  real<lower=0> sigma; //error variance
##  vector[N_pred] beta; // regression parameters
## }
##
## model{
##   // No priors on the slopes
##   sigma ~ normal(0, 1);
##  y_train ~ normal(beta_0 + X_train * beta, sigma); //likelihood
## }
##
## generated quantities{
##  real y_test[N_test]; //predicted outputs
##  vector[N_train] log_lik;
##  for(i in 1:N_train){
##     log_lik[i] = normal_lpdf(y_train[i] | beta_0 + X_train[i,] * beta, sigma);
##   }
##  for(i in 1:N_test){
##      y_test[i] = normal_rng(beta_0 + X_test[i,] * beta, sigma);
##  }
## }
```

We fit this model and check the summary of the model.

```
bayes_uniform <- stan_model('uniform.stan')

# Fit the model using Stan's NUTS HMC sampler
fit_bayes_uniform <- sampling(bayes_uniform, data_crime, iter = 2000,
                            warmup = 500, chains = 4, cores = 4)
```

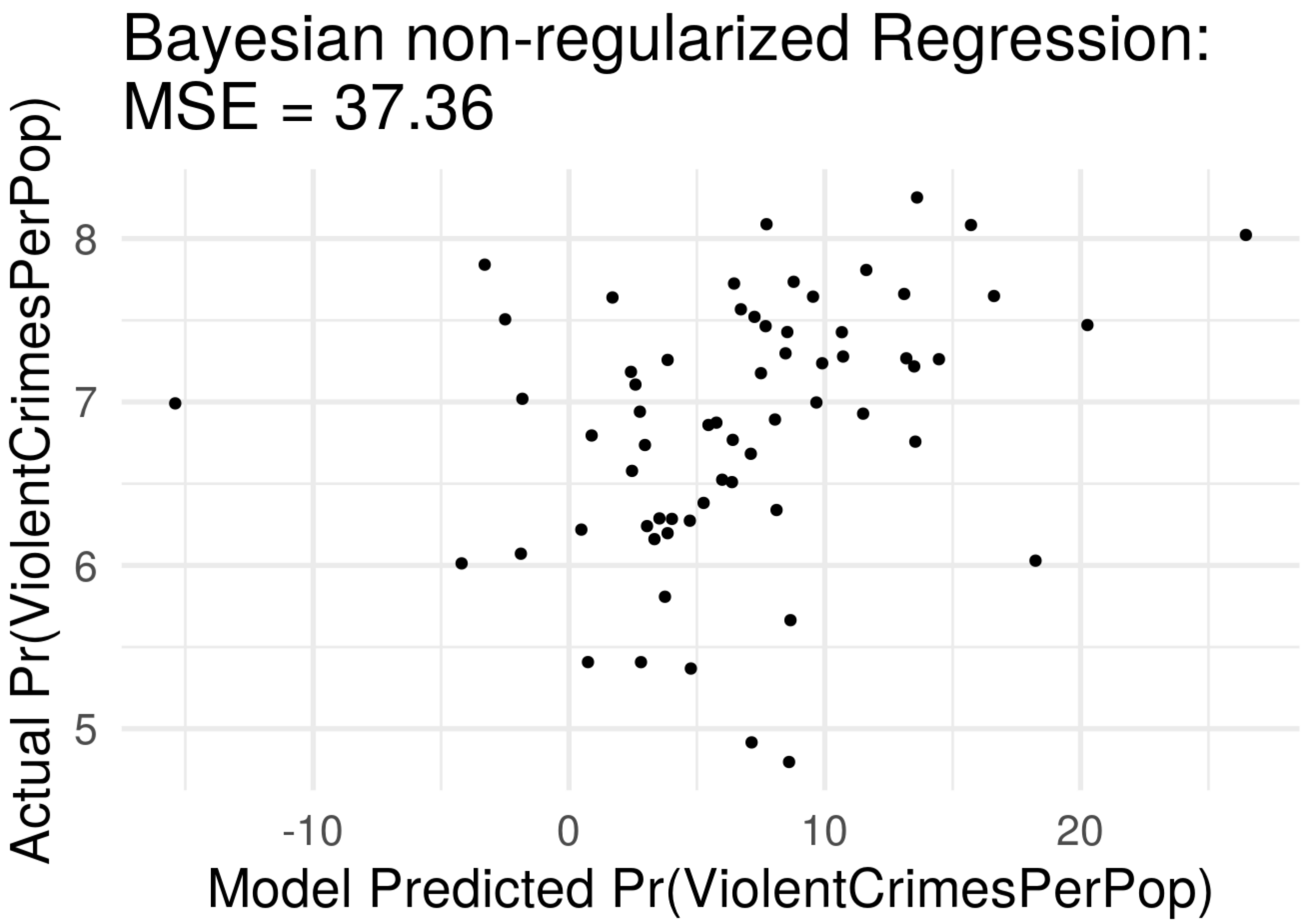
```
## Warning: There were 6000 transitions after warmup that exceeded the maximum treede
pth. Increase max_treedepth above 10. See
## http://mc-stan.org/misc/warnings.html#maximum-treedepth-exceeded
```

```
## Warning: Examine the pairs() plot to diagnose sampling problems
```

```
## Warning: The largest R-hat is 2.9, indicating chains have not mixed.
## Running the chains for more iterations may help. See
## http://mc-stan.org/misc/warnings.html#r-hat
## Warning: Bulk Effective Samples Size (ESS) is too low, indicating posterior means
and medians may be unreliable.
## Running the chains for more iterations may help. See
## http://mc-stan.org/misc/warnings.html#bulk-ess
```

```
## Warning: Tail Effective Samples Size (ESS) is too low, indicating posterior varian
ces and tail quantiles may be unreliable.
## Running the chains for more iterations may help. See
## http://mc-stan.org/misc/warnings.html#tail-ess
```

```
# Extract posterior distribution (parameters and predictions)
post_uniform <- rstan::extract(fit_bayes_uniform)

# Compute mean of the posterior predictive distribution over test set predictors,
# which integrates out uncertainty in parameter estimates
y_pred_bayes_uniform <- apply(post_uniform$y_test, 2, mean)

# Plot correlation between posterior predicted mean and actual Pr(ViolentCrimesPerPo
p)
qplot(x = y_pred_bayes_uniform, y = y_test,
      main = paste0("Bayesian non-regularized Regression:\n",
                    "MSE = ", round(mean((y_test - y_pred_bayes_uniform)^2),2))) +
  xlab("Model Predicted Pr(ViolentCrimesPerPop)") +
  ylab("Actual Pr(ViolentCrimesPerPop)") +
  theme_minimal(base_size = 20)
```



```r
# print(fit_bayes_uniform, pars = "beta")

# Convergence diagnostics (Rhat, divergences, neff)
util$check_all_diagnostics(fit_bayes_uniform)
```

```
## [1] "n_eff / iter for parameter beta[100] is 0.000362326523733212!"
## [1] "n_eff / iter for parameter beta[106] is 0.000362272876678583!"
## [1] "n_eff / iter for parameter y_test[6] is 0.000781875115572645!"
## [1] "n_eff / iter for parameter y_test[9] is 0.000763230449265738!"
## [1] "n_eff / iter for parameter y_test[11] is 0.000976346374148566!"
## [1] "n_eff / iter for parameter y_test[13] is 0.000826635688764178!"
## [1] "n_eff / iter for parameter y_test[17] is 0.00087146318555887!"
## [1] "n_eff / iter for parameter y_test[18] is 0.000866179684759157!"
## [1] "n_eff / iter for parameter y_test[22] is 0.000795182619023829!"
## [1] "n_eff / iter for parameter y_test[27] is 0.00080882456769244!"
## [1] "n_eff / iter for parameter y_test[29] is 0.000860225530310946!"
## [1] "n_eff / iter for parameter y_test[34] is 0.000894482930030252!"
## [1] "n_eff / iter for parameter y_test[39] is 0.000867944306029947!"
## [1] "n_eff / iter for parameter y_test[42] is 0.000885309257989581!"
## [1] "n_eff / iter for parameter y_test[47] is 0.000876705080699229!"
## [1] "n_eff / iter for parameter y_test[57] is 0.000869949750280572!"
## [1] "n_eff / iter for parameter y_test[59] is 0.000908600009522415!"
## [1] "  n_eff / iter below 0.001 indicates that the effective sample size has likel
y been overestimated"
## [1] "Rhat for parameter beta[2] is 1.1882529496912!"
## [1] "Rhat for parameter beta[3] is 1.18719488777441!"
## [1] "Rhat for parameter beta[4] is 1.1268681222548!"
## [1] "Rhat for parameter beta[5] is 1.14774738234682!"
## [1] "Rhat for parameter beta[6] is 1.20324832583475!"
## [1] "Rhat for parameter beta[7] is 1.52283941848164!"
## [1] "Rhat for parameter beta[8] is 1.27824599643973!"
## [1] "Rhat for parameter beta[9] is 1.52990325689591!"
## [1] "Rhat for parameter beta[10] is 1.36172554238875!"
## [1] "Rhat for parameter beta[15] is 1.17404046869036!"
## [1] "Rhat for parameter beta[16] is 1.15704225265082!"
## [1] "Rhat for parameter beta[17] is 1.31688869686809!"
## [1] "Rhat for parameter beta[18] is 1.15653893102333!"
## [1] "Rhat for parameter beta[19] is 1.20511858522481!"
## [1] "Rhat for parameter beta[21] is 1.10836336935001!"
## [1] "Rhat for parameter beta[22] is 1.11646191423376!"
## [1] "Rhat for parameter beta[25] is 1.11836338456946!"
## [1] "Rhat for parameter beta[26] is 1.14407165255744!"
## [1] "Rhat for parameter beta[27] is 1.16830181803305!"
## [1] "Rhat for parameter beta[28] is 1.12029320653942!"
## [1] "Rhat for parameter beta[29] is 1.14410824672121!"
## [1] "Rhat for parameter beta[30] is 1.15694873276427!"
## [1] "Rhat for parameter beta[31] is 1.16993971300941!"
## [1] "Rhat for parameter beta[32] is 1.10805855413783!"
## [1] "Rhat for parameter beta[36] is 1.18167305489235!"
## [1] "Rhat for parameter beta[38] is 1.17357718618175!"
## [1] "Rhat for parameter beta[39] is 1.28423679040309!"
## [1] "Rhat for parameter beta[40] is 1.18441307786083!"
## [1] "Rhat for parameter beta[41] is 1.27198455337109!"
## [1] "Rhat for parameter beta[42] is 1.27702781236985!"
## [1] "Rhat for parameter beta[43] is 1.32150102146939!"
## [1] "Rhat for parameter beta[48] is 1.17175154822406!"
## [1] "Rhat for parameter beta[49] is 1.19469792988642!"
## [1] "Rhat for parameter beta[50] is 1.11763897532901!"
## [1] "Rhat for parameter beta[51] is 1.20488546766508!"
## [1] "Rhat for parameter beta[53] is 1.37779263515512!"
## [1] "Rhat for parameter beta[54] is 1.32707663303758!"
## [1] "Rhat for parameter beta[55] is 1.36432681634986!"
## [1] "Rhat for parameter beta[56] is 1.28707971923043!"
## [1] "Rhat for parameter beta[57] is 1.26418255730783!"
## [1] "Rhat for parameter beta[58] is 1.16238302174931!"
## [1] "Rhat for parameter beta[59] is 1.19126706304137!"
## [1] "Rhat for parameter beta[60] is 1.18782592503708!"
## [1] "Rhat for parameter beta[61] is 1.15038474825183!"
## [1] "Rhat for parameter beta[62] is 1.11747425130266!"
## [1] "Rhat for parameter beta[63] is 1.48637216743653!"
## [1] "Rhat for parameter beta[64] is 1.42220409397448!"
## [1] "Rhat for parameter beta[65] is 1.32130339143806!"
## [1] "Rhat for parameter beta[66] is 1.38388078743124!"
## [1] "Rhat for parameter beta[67] is 1.14926845923642!"
## [1] "Rhat for parameter beta[68] is 1.28567180573361!"
## [1] "Rhat for parameter beta[69] is 1.12059467282872!"
## [1] "Rhat for parameter beta[70] is 1.30300491281719!"
## [1] "Rhat for parameter beta[71] is 1.26449377425417!"
## [1] "Rhat for parameter beta[72] is 1.10534587096675!"
## [1] "Rhat for parameter beta[73] is 1.1255616256262!"
## [1] "Rhat for parameter beta[74] is 1.31389946491151!"
## [1] "Rhat for parameter beta[79] is 1.24065721319613!"
## [1] "Rhat for parameter beta[80] is 2.05060918533931!"
## [1] "Rhat for parameter beta[82] is 2.04960356524792!"
## [1] "Rhat for parameter beta[83] is 2.0492120613669!"
## [1] "Rhat for parameter beta[84] is 1.57260419890798!"
## [1] "Rhat for parameter beta[85] is 1.13538971840639!"
## [1] "Rhat for parameter beta[86] is 1.57636970991159!"
## [1] "Rhat for parameter beta[87] is 1.57491357759195!"
## [1] "Rhat for parameter beta[88] is 1.16900917866745!"
## [1] "Rhat for parameter beta[89] is 1.1737573394615!"
## [1] "Rhat for parameter beta[92] is 1.14857364484347!"
## [1] "Rhat for parameter beta[94] is 1.16047248387168!"
## [1] "Rhat for parameter beta[95] is 1.14615321848781!"
## [1] "Rhat for parameter beta[96] is 1.13084631524492!"
## [1] "Rhat for parameter beta[98] is 1.11072996994734!"
## [1] "Rhat for parameter beta[99] is 1.26107426845704!"
## [1] "Rhat for parameter beta[100] is 4.0805858648595!"
## [1] "Rhat for parameter beta[101] is 1.18990485609967!"
## [1] "Rhat for parameter beta[102] is 1.17614411148192!"
## [1] "Rhat for parameter beta[106] is 4.08189742405248!"
## [1] "Rhat for parameter beta[108] is 1.10373786303656!"
## [1] "Rhat for parameter beta[109] is 1.33455606674017!"
## [1] "Rhat for parameter beta[110] is 1.3062431370868!"
## [1] "Rhat for parameter beta[111] is 1.18157234872519!"
## [1] "Rhat for parameter beta[112] is 1.2827768823838!"
## [1] "Rhat for parameter beta[115] is 1.19549951958436!"
## [1] "Rhat for parameter beta[117] is 1.21635314277539!"
## [1] "Rhat for parameter beta[118] is 1.16928254213572!"
## [1] "Rhat for parameter beta[120] is 1.2410558522885!"
## [1] "Rhat for parameter beta[121] is 1.13255918651966!"
## [1] "Rhat for parameter beta[124] is 1.16434833634172!"
## [1] "Rhat for parameter y_test[1] is 1.97983503177707!"
## [1] "Rhat for parameter y_test[2] is 1.74816617089976!"
## [1] "Rhat for parameter y_test[3] is 2.05460792541041!"
## [1] "Rhat for parameter y_test[4] is 2.00904994016534!"
## [1] "Rhat for parameter y_test[5] is 2.13642334150059!"
## [1] "Rhat for parameter y_test[6] is 1.50344097580091!"
## [1] "Rhat for parameter y_test[7] is 2.07319333658542!"
## [1] "Rhat for parameter y_test[8] is 2.10475537149923!"
## [1] "Rhat for parameter y_test[9] is 1.53173507854748!"
## [1] "Rhat for parameter y_test[10] is 2.09455640342271!"
## [1] "Rhat for parameter y_test[11] is 1.56872288097253!"
## [1] "Rhat for parameter y_test[12] is 1.97981933878516!"
## [1] "Rhat for parameter y_test[13] is 1.46189178491015!"
## [1] "Rhat for parameter y_test[14] is 1.91365839877009!"
## [1] "Rhat for parameter y_test[15] is 1.89993759618842!"
## [1] "Rhat for parameter y_test[16] is 2.06006524247911!"
## [1] "Rhat for parameter y_test[17] is 1.52809789814149!"
## [1] "Rhat for parameter y_test[18] is 1.81803329802752!"
## [1] "Rhat for parameter y_test[19] is 2.01356108815545!"
## [1] "Rhat for parameter y_test[20] is 1.7210191291547!"
## [1] "Rhat for parameter y_test[21] is 1.5733685281073!"
## [1] "Rhat for parameter y_test[22] is 1.48593963634171!"
## [1] "Rhat for parameter y_test[23] is 1.95831659815423!"
## [1] "Rhat for parameter y_test[24] is 1.45594420819531!"
## [1] "Rhat for parameter y_test[25] is 1.91516890395885!"
## [1] "Rhat for parameter y_test[26] is 1.29520548869164!"
## [1] "Rhat for parameter y_test[27] is 1.68778088544353!"
## [1] "Rhat for parameter y_test[28] is 1.5347593175005!"
## [1] "Rhat for parameter y_test[29] is 1.63726237427377!"
## [1] "Rhat for parameter y_test[30] is 2.04802193309747!"
## [1] "Rhat for parameter y_test[31] is 1.9527142772992!"
## [1] "Rhat for parameter y_test[32] is 1.94056084399702!"
## [1] "Rhat for parameter y_test[33] is 2.12680298792831!"
## [1] "Rhat for parameter y_test[34] is 1.42368520850045!"
## [1] "Rhat for parameter y_test[35] is 1.72368831156855!"
## [1] "Rhat for parameter y_test[36] is 1.68940856149004!"
## [1] "Rhat for parameter y_test[37] is 1.97304095426128!"
## [1] "Rhat for parameter y_test[38] is 2.06586298929425!"
## [1] "Rhat for parameter y_test[39] is 1.42732201853224!"
## [1] "Rhat for parameter y_test[40] is 1.91615353275288!"
## [1] "Rhat for parameter y_test[41] is 2.02778148125691!"
## [1] "Rhat for parameter y_test[42] is 1.72487632934671!"
## [1] "Rhat for parameter y_test[43] is 1.95423864241604!"
## [1] "Rhat for parameter y_test[44] is 1.81965783940559!"
## [1] "Rhat for parameter y_test[45] is 1.70070539614536!"
## [1] "Rhat for parameter y_test[46] is 1.89962961108317!"
## [1] "Rhat for parameter y_test[47] is 1.75360998112579!"
## [1] "Rhat for parameter y_test[48] is 2.17126866086129!"
## [1] "Rhat for parameter y_test[49] is 1.9693365066789!"
## [1] "Rhat for parameter y_test[50] is 1.98831394298087!"
## [1] "Rhat for parameter y_test[51] is 2.0092437214118!"
## [1] "Rhat for parameter y_test[52] is 1.90538518516214!"
## [1] "Rhat for parameter y_test[53] is 2.12048666073019!"
## [1] "Rhat for parameter y_test[54] is 1.96691830018012!"
## [1] "Rhat for parameter y_test[55] is 1.77664020255254!"
## [1] "Rhat for parameter y_test[56] is 1.52070068791406!"
## [1] "Rhat for parameter y_test[57] is 1.53886440792368!"
## [1] "Rhat for parameter y_test[58] is 1.81353653258029!"
## [1] "Rhat for parameter y_test[59] is 1.41199215345478!"
## [1] "Rhat for parameter y_test[60] is 2.00635004211766!"
## [1] "Rhat for parameter y_test[61] is 1.96759589700731!"
## [1] "Rhat for parameter y_test[62] is 2.06110826635149!"
## [1] "Rhat for parameter y_test[63] is 1.47789845578106!"
## [1] "Rhat for parameter y_test[64] is 2.08354657948475!"
## [1] "Rhat for parameter log_lik[11] is 1.12930606912716!"
## [1] "Rhat for parameter log_lik[41] is 1.10652379559373!"
## [1] "Rhat for parameter log_lik[88] is 1.31042142418898!"
## [1] "Rhat for parameter log_lik[97] is 1.25332430893733!"
## [1] "Rhat for parameter log_lik[124] is 1.12901297984788!"
## [1] "Rhat for parameter log_lik[146] is 1.1188448566775!"
## [1] "Rhat for parameter log_lik[152] is 1.13870477383779!"
## [1] "Rhat for parameter log_lik[158] is 1.10592097372044!"
## [1] "Rhat for parameter log_lik[171] is 1.10788768505205!"
## [1] "Rhat for parameter log_lik[189] is 1.14869480639579!"
## [1] "Rhat for parameter lp__ is 1.10200977916069!"
## [1] "  Rhat above 1.1 indicates that the chains very likely have not mixed"
## [1] "0 of 6000 iterations ended with a divergence (0%)"
## [1] "6000 of 6000 iterations saturated the maximum tree depth of 10 (100%)"
## [1] "  Run again with max_depth set to a larger value to avoid saturation"
## [1] "E-BFMI indicated no pathological behavior"
```

The results indicate a really bad model. Rhat > 1.1 is usually indicative of problems in the fit. Both large split R^ and low effective samples per transition are consequences of poorly mixing Markov chains. Improving the mixing of the Markov chains almost always requires tweaking the model specification, for example with a reparameterization or stronger priors (Betancourt 2017). A uniform prior on β denotes no penalty at all, and we are left with traditional, non-regularized regression. If we assume that $\beta \sim U(-\infty, +\infty)$ and can take on any real-valued number, and every value is equally likely (uniform distributon), the mode of the posterior distribution on each β weight will be equivalent to the maximum likelihood estimate of the respective β weight. An unbounded uniform distribution on β produces the same behavior as traditional linear regression and allows us to maximally learn from the data. However, we can use a prior distribution that pulls the β weights toward 0 (unlike the unbounded uniform distribution). In the following we will check some of these priors.

Bayesian Regularized regression

Regularized linear regression models are aimed to have a more conservative estimation of weights (β's) in the model. The central idea of penalized regression approaches is to add a penalty term to the minimization of the sum of squared residuals, with the goal of shrinking small coefficients towards zero while leaving large coefficients large. In the Bayesian world, we can capture such an effect in the form of a prior distribution over our β weights.

In Bayesian analysis, a prior distribution is specified for each parameter as follows (Van Erp, Oberski, and Mulder 2019):

$$p(\beta_0, \beta, \sigma^2, \lambda) = p(\beta_0)p(\beta|\sigma^2, \lambda)p(\sigma^2)p(\lambda)$$

Bayesian models view estimation as a problem of integrating prior information with information gained from data, which we formalize using probability distributions. These models require us to specify a prior distribution for each parameter we seek to estimate. Therefore, we need to specify a prior on the slopes (β), and error variance (σ). Our choice of prior distribution on β is what determines how much information we learn from the data, analagous to the penalty term λ used for frequentist regularization. Smaller and larger values of λ parameter leads us to more and less learning from data, respectively. Therefore, λ is called a hyperparameter. There are many ways to regularize the estimation procedures including ridge, LASSO (Laplace), horeshoe regression (Carvalho, Polson, and Scott 2010).

Ridge regression

The normal distribution for prior on β is mathematically equivalent in expectation to using the ridge penalty in the frequentist model (Figueiredo 2002):

$$\beta \sim \mathcal{N}(0, \sigma_\beta)$$

The normal distribution places very little prior probability on large-magnitude β weights (i.e. far from 0), while placing high prior probability on small-magnitude weights (i.e. near 0). On the other hand, σ_β controls how wide the normal distribution is, thus controlling the specific amount of prior probability placed on small- to large-magnitude β weights. Below is the Stan code that specifies the Bayesian variant of ridge regression:

```
writeLines(readLines("ridge.stan"))
```

```
## data{
##   int N_train;             // training observations
##   int N_test;              // test observations
##   int N_pred;              // predictor variables
##   vector[N_train] y_train; // training outcomes
##   matrix[N_train, N_pred] X_train; // training data
##   matrix[N_test, N_pred] X_test;   // testing data
## }
## parameters{
##   real beta_0; // intercept
##   real<lower=0> sigma;   // error SD
##   real<lower=0> sigma_B; // SD across betas
##   vector[N_pred] beta;   // regression beta weights
## }
## model{
##   beta_0 ~ normal(0, 1); // prior on intercept
##   sigma ~ normal(0, 1); // model error SD
##   beta ~ normal(0, sigma_B); // beta prior : ridge
##   y_train ~ normal(beta_0 + X_train * beta, sigma); // model likelihood
## }
## generated quantities{
##   real y_test[N_test]; // test data predictions
##   vector[N_train] log_lik;
##   for(i in 1:N_train){
##   log_lik[i] = normal_lpdf(y_train[i] | beta_0 + X_train[i,] * beta, sigma);
##   }
##   for(i in 1:N_test){
##       y_test[i] = normal_rng(beta_0 + X_test[i,] * beta, sigma);
##   }
## }
```

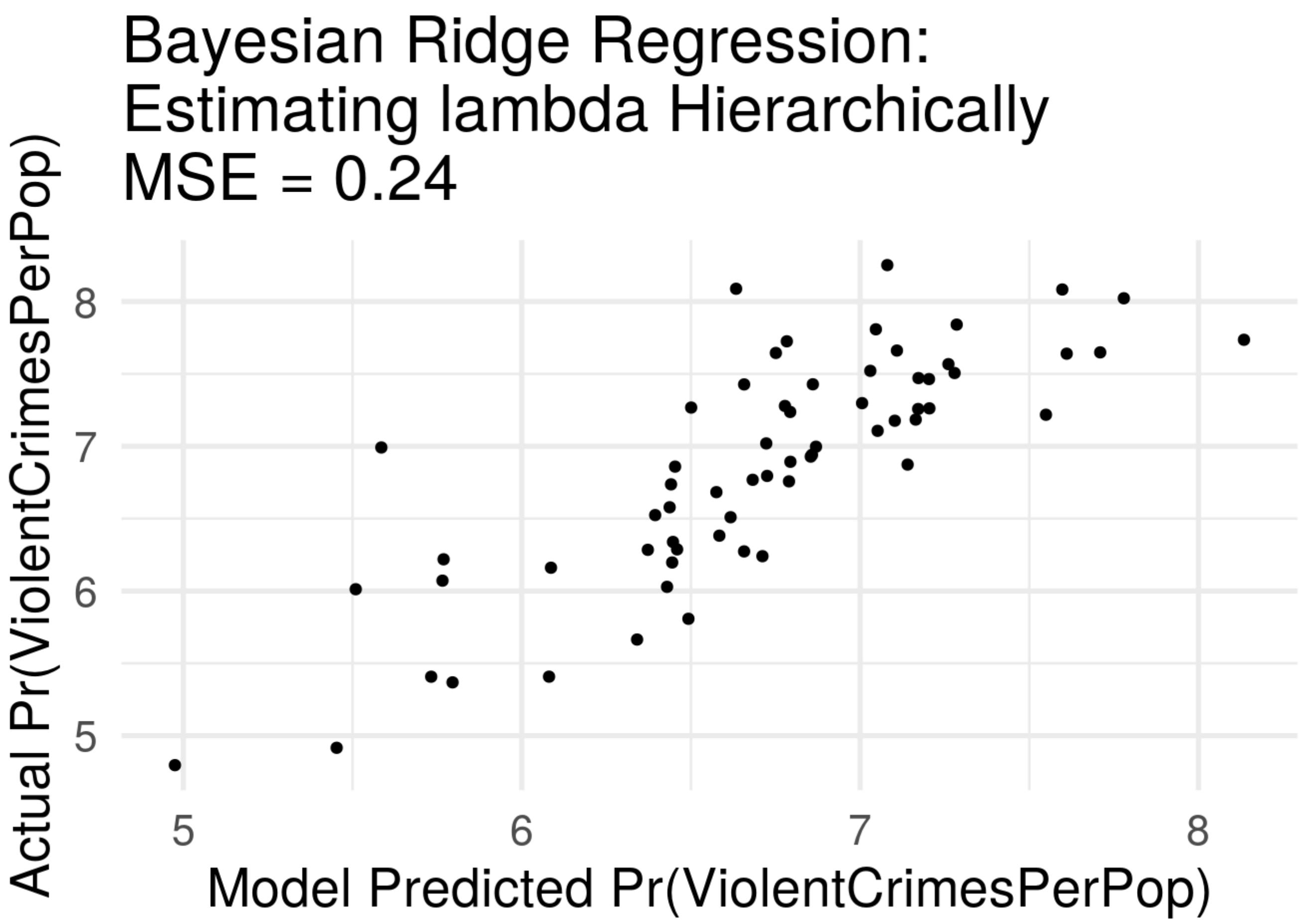
We can actually view Bayesian ridge regression as a simple hierarchical Bayesian model by jointly estimating σ_β along with individual-level β weights, where σ_β is interpreted as a group-level scaling parameter that is estimated from pooled information across individual β weights.

```
writeLines(readLines("ridge_hierarchical.stan"))
```

```
## data{
##     int N_train;             // "# training observations"
##     int N_test;              // "# test observations"
##     int N_pred;              // "# predictor variables"
##     vector[N_train] y_train; // "training outcomes"
##     matrix[N_train, N_pred] X_train; // "training data"
##     matrix[N_test, N_pred] X_test;   // "testing data"
## }
## parameters{
##  real beta_0; // intercept
##  real<lower=0> sigma; //error variance
##  vector[N_pred] beta; // regression parameters
##  real<lower=0> sigma_B; // SD across betas (hyperparameters prior)
## }
## model{
##   beta_0 ~ normal(0, 1); // prior on intercept
##   sigma ~ normal(0, 1); // model error SD
##  beta ~ normal(0, sigma_B); // prior regression coefficients: ridge
##  sigma_B ~ cauchy(0, 10000); // hierarchical prior on hyperparameter
##  y_train ~ normal(beta_0 + X_train * beta, sigma); //likelihood
## }
## generated quantities{
##  real y_test[N_test]; //predicted outputs
##  vector[N_train] log_lik;
##  for(i in 1:N_train){
##     log_lik[i] = normal_lpdf(y_train[i] | beta_0 + X_train[i,] * beta, sigma);
##   }
##  for(i in 1:N_test){
##      y_test[i] = normal_rng(beta_0 + X_test[i,] * beta, sigma);
##  }
## }
```

Now, we fit the hierarchical ridge regression model and and check the summary of the model:

```
bayes_ridge <- stan_model('ridge_hierarchical.stan')

# Fit the model using Stan's NUTS HMC sampler
fit_bayes_ridge <- sampling(bayes_ridge, data_crime, iter = 2000,
                            warmup = 500, chains = 4, cores = 4)

# Extract posterior distribution (parameters and predictions)
post_ridge <- rstan::extract(fit_bayes_ridge)

# Compute mean of the posterior predictive distribution over test set predictors,
# which integrates out uncertainty in parameter estimates
y_pred_bayes_ridge <- apply(post_ridge$y_test, 2, mean)

# Plot correlation between posterior predicted mean and actual Pr(ViolentCrimesPerPo
p)
qplot(x = y_pred_bayes_ridge, y = y_test,
      main = paste0("Bayesian Ridge Regression:\nEstimating ", expression(lambda),
                    " Hierarchically\nMSE = ", round(mean((y_test - y_pred_bayes_ridg
e)^2),2))) + # round(cor(y_test, y_pred_bayes),2)
  xlab("Model Predicted Pr(ViolentCrimesPerPop)") +
  ylab("Actual Pr(ViolentCrimesPerPop)") +
  theme_minimal(base_size = 20)
```



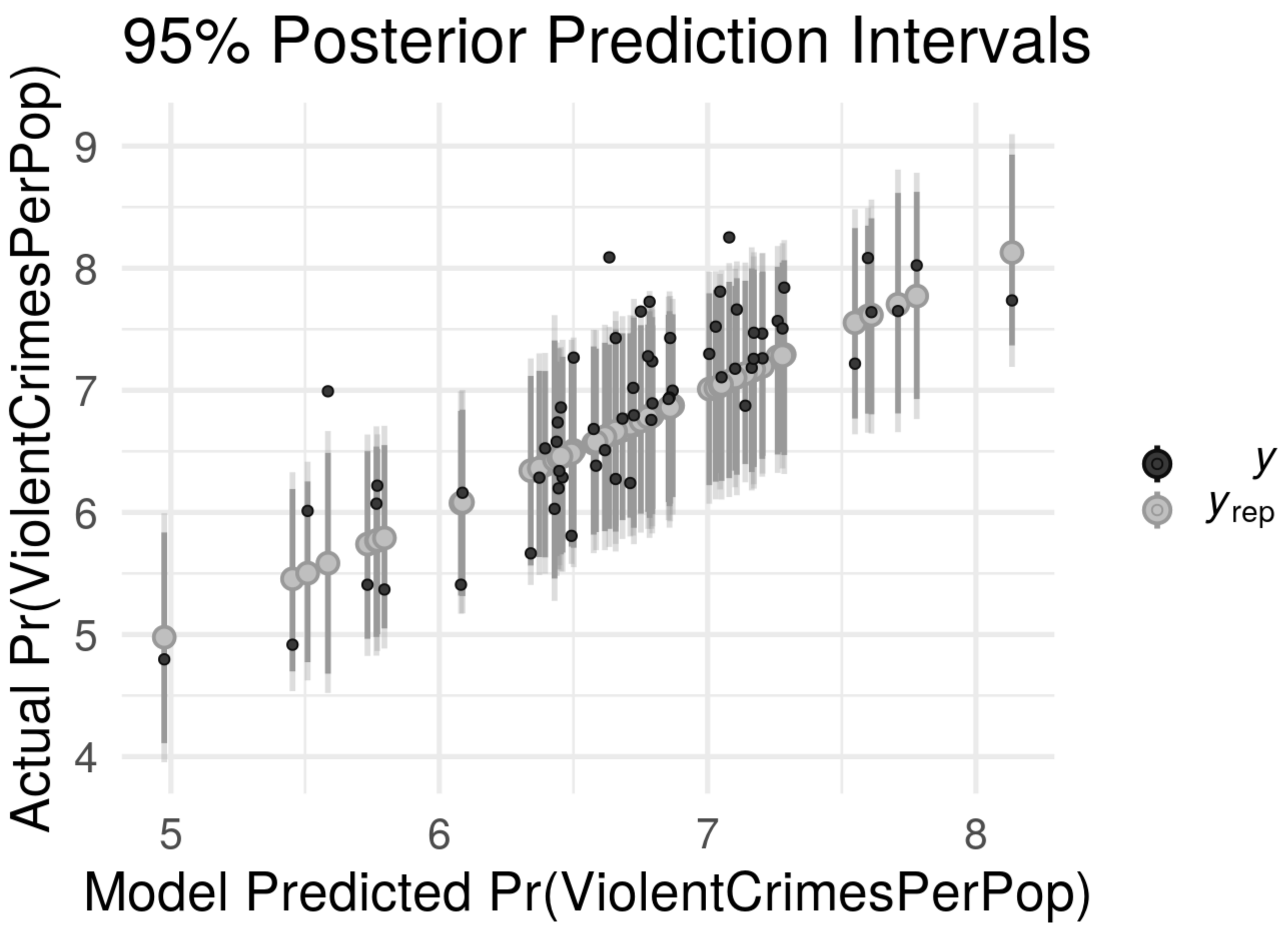
```
#print(fit_bayes_ridge)

# Convergence diagnostics (Rhat, divergences, neff)
util$check_all_diagnostics(fit_bayes_ridge)
```

```
## [1] "n_eff / iter looks reasonable for all parameters"
## [1] "Rhat looks reasonable for all parameters"
## [1] "0 of 6000 iterations ended with a divergence (0%)"
## [1] "0 of 6000 iterations saturated the maximum tree depth of 10 (0%)"
## [1] "E-BFMI indicated no pathological behavior"
```

We ensured that the split $\hat{R}$ for each parameter is close to 1. Here all of the parameters look good (not any warnings).

95% Bayesian credibility interval can simply be interpreted as the interval in which the true value lies with 95% probability (e.g., Berger, 2006). The following figure illustrates the scatterplot the prediction uncertainty that includes prediction intervals around the predicted mean estimates for each observation in the test set:

```
# `bayesplot` has many convenience functions for working with posteriors
color_scheme_set(scheme = "darkgray")
ppc_intervals(x = colMeans(post_ridge$y_test), y = y_test,
              yrep = post_ridge$y_test, prob = 0.95) +
  ggtitle("95% Posterior Prediction Intervals") +
  xlab("Model Predicted Pr(ViolentCrimesPerPop)") +
  ylab("Actual Pr(ViolentCrimesPerPop)") +
  theme_minimal(base_size = 20)
```



LASSO regression

LASSO regression only involves a minor revision to the loss function, as opposed to penalizing the model based on the sum of squared β weights in ridge regresion, we will penalize the model by the sum of the absolute value of β weights. Unlike for the ridge penalty, there are sharp corners in the geometry of LASSO penalty, which correspond to when some of β's equal 0.

The LASSO is often more useful in situations where many of the predictors are noisy and we expect the solution to be more sparse and LASSO is effective by setting β weights on noisy predictors to exactly 0.

Setting a Laplace (i.e. double-exponential) prior on the β weights is mathematically equivalent in expectation to the frequentist LASSO penalty (Tibshirani 1996):

$$\beta \sim double - expoential(0, \tau_\beta)$$

where in τ_β is a scale parameter that controls how peaked the prior distribution is around the center. For large amount of τ_β the prior reduces to a uniform prior and therefore noregularization (traditional regression). For small amount of τ_β the model assigns infinite weight on those β's that are 0, therefore there is no learning from data.

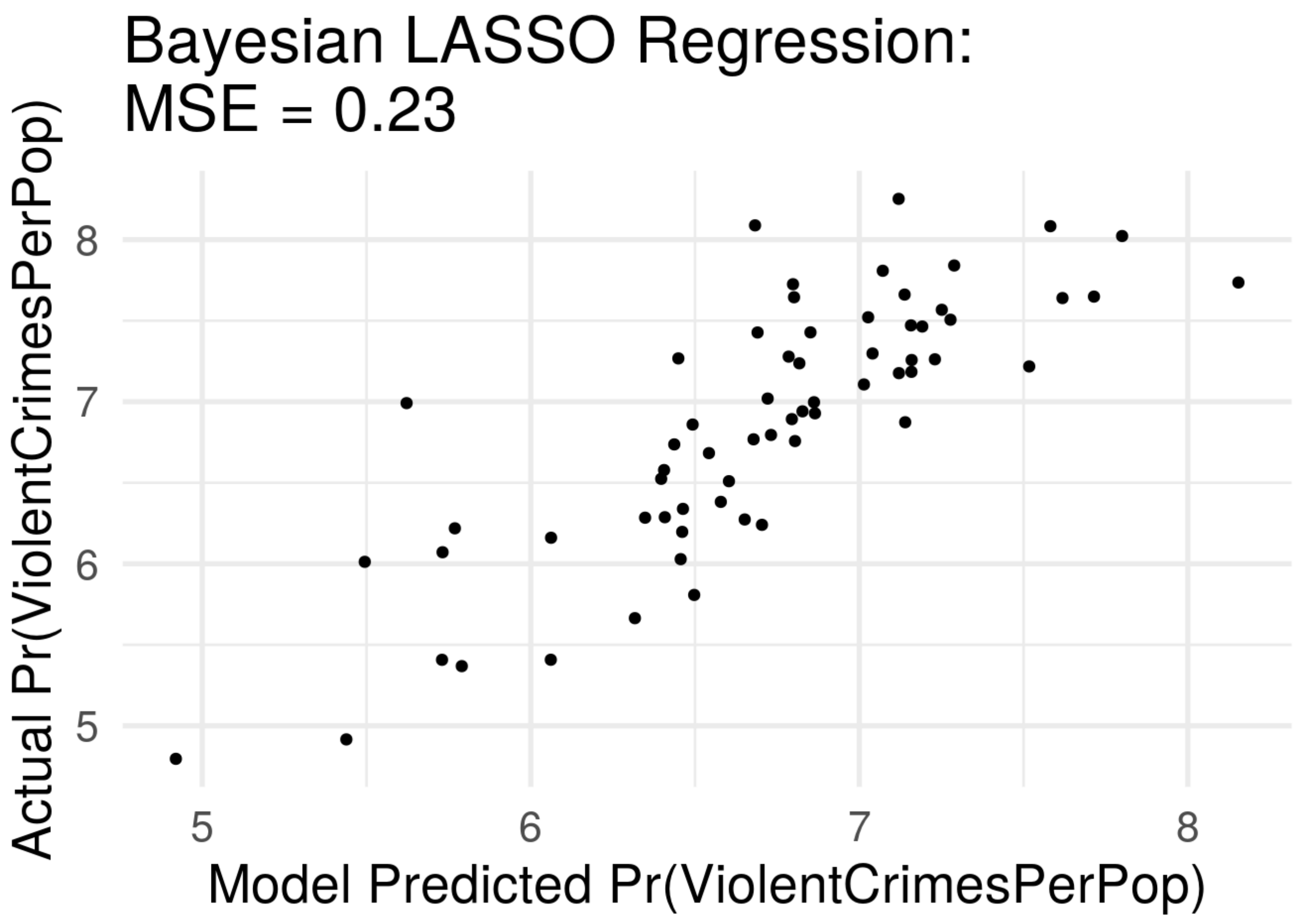
The stan code that specifies the Bayesian LASSO regression is shown in the below:

```
writeLines(readLines("lasso_hierarchical.stan"))
```

```
## data{
##     int N_train;             // # training observations
##     int N_test;              // # test observations
##     int N_pred;              // # predictor variables
##     vector[N_train] y_train; // training outcomes
##     matrix[N_train, N_pred] X_train; // training data
##     matrix[N_test, N_pred] X_test;   // testing data
## }
## parameters{
##  real beta_0; // intercept
##  real<lower=0> sigma; // error variance
##  real<lower=0> tau_B; // SD across betas (hyperparameters prior)
##  vector[N_pred] beta; // regression parameters
## }
## model{
##   beta_0 ~ normal(0, 1); // prior on intercept
##   sigma ~ normal(0, 1); // model error SD
##  beta ~ double_exponential(0, tau_B); // prior regression coefficients: lasso
##  tau_B ~ cauchy(0, 10000); // hierarchical prior on hyperparameter
##  y_train ~ normal(beta_0 + X_train * beta, sigma); //likelihood
## }
## generated quantities{
##  real y_test[N_test]; //predicted outputs
##  vector[N_train] log_lik;
##  for(i in 1:N_train){
##     log_lik[i] = normal_lpdf(y_train[i] | beta_0 + X_train[i,] * beta, sigma);
##   }
##  for(i in 1:N_test){
##      y_test[i] = normal_rng(beta_0 + X_test[i,] * beta, sigma);
##  }
## }
```

Now let’s fit the model and visualize the results:

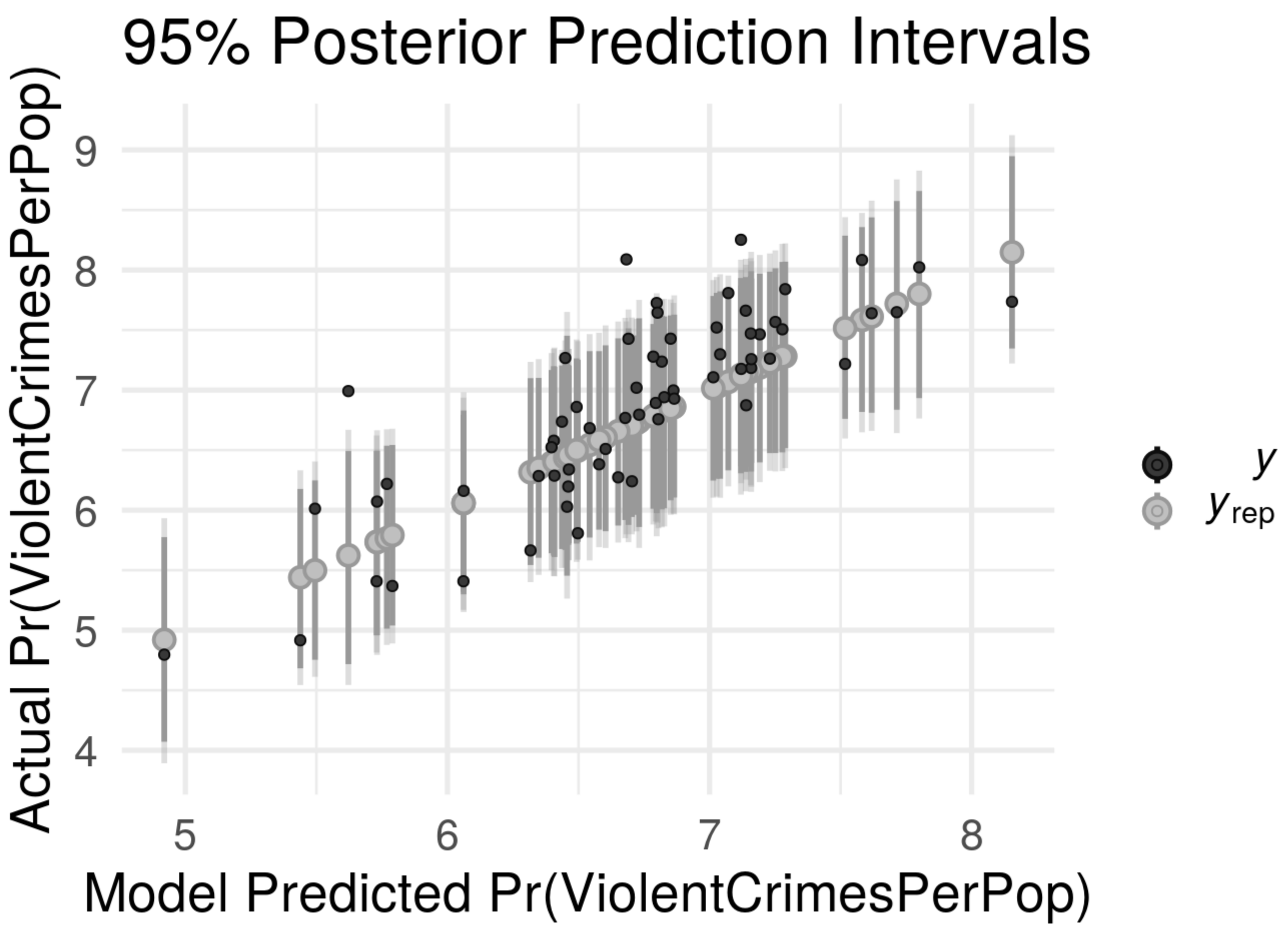
```
#fit_bayes_lasso <- stan(file="lasso.stan", data = data_crime, seed = SEED)
bayes_lasso <- stan_model('lasso_hierarchical.stan')

# Fit the model using Stan's NUTS HMC sampler
fit_bayes_lasso <- sampling(bayes_lasso, data_crime, iter = 2000,
                            warmup = 500, chains = 4, cores = 4)



# Extract posterior distribution (parameters and predictions)
post_lasso <- rstan::extract(fit_bayes_lasso)

# Compute mean of the posterior predictive distribution over test set predictors,
# which integrates out uncertainty in parameter estimates
y_pred_bayes_lasso <- apply(post_lasso$y_test, 2, mean)

# Plot correlation between posterior predicted mean and actual Pr(ViolentCrimesPerPo
p)
qplot(x = y_pred_bayes_lasso, y = y_test,
      main = paste0("Bayesian LASSO Regression:\nMSE = ", round(mean((y_test - y_pred
_bayes_lasso)^2),2))) +
    xlab("Model Predicted Pr(ViolentCrimesPerPop)") +
    ylab("Actual Pr(ViolentCrimesPerPop)") +
    theme_minimal(base_size = 20)
```



```
#print(fit_bayes_lasso)

# Convergence diagnostics (Rhat, divergences, neff)
util$check_all_diagnostics(fit_bayes_lasso)
```

```
## [1] "n_eff / iter looks reasonable for all parameters"
## [1] "Rhat looks reasonable for all parameters"
## [1] "0 of 6000 iterations ended with a divergence (0%)"
## [1] "0 of 6000 iterations saturated the maximum tree depth of 10 (0%)"
## [1] "E-BFMI indicated no pathological behavior"
```

```
# `bayesplot` has many convenience functions for working with posteriors
color_scheme_set(scheme = "darkgray")
ppc_intervals(x = colMeans(post_lasso$y_test), y = y_test,
              yrep = post_lasso$y_test, prob = 0.95) +
  ggtitle("95% Posterior Prediction Intervals") +
  xlab("Model Predicted Pr(ViolentCrimesPerPop)") +
  ylab("Actual Pr(ViolentCrimesPerPop)") +
  theme_minimal(base_size = 20)
```



Horseshoe regression

```
#fit_bayes_lasso <- stan(file="horseshoe_hierarchical.stan", data = data_crime, seed
 = SEED)
bayes_horseshoe <- stan_model('horseshoe_hierarchical.stan')
```

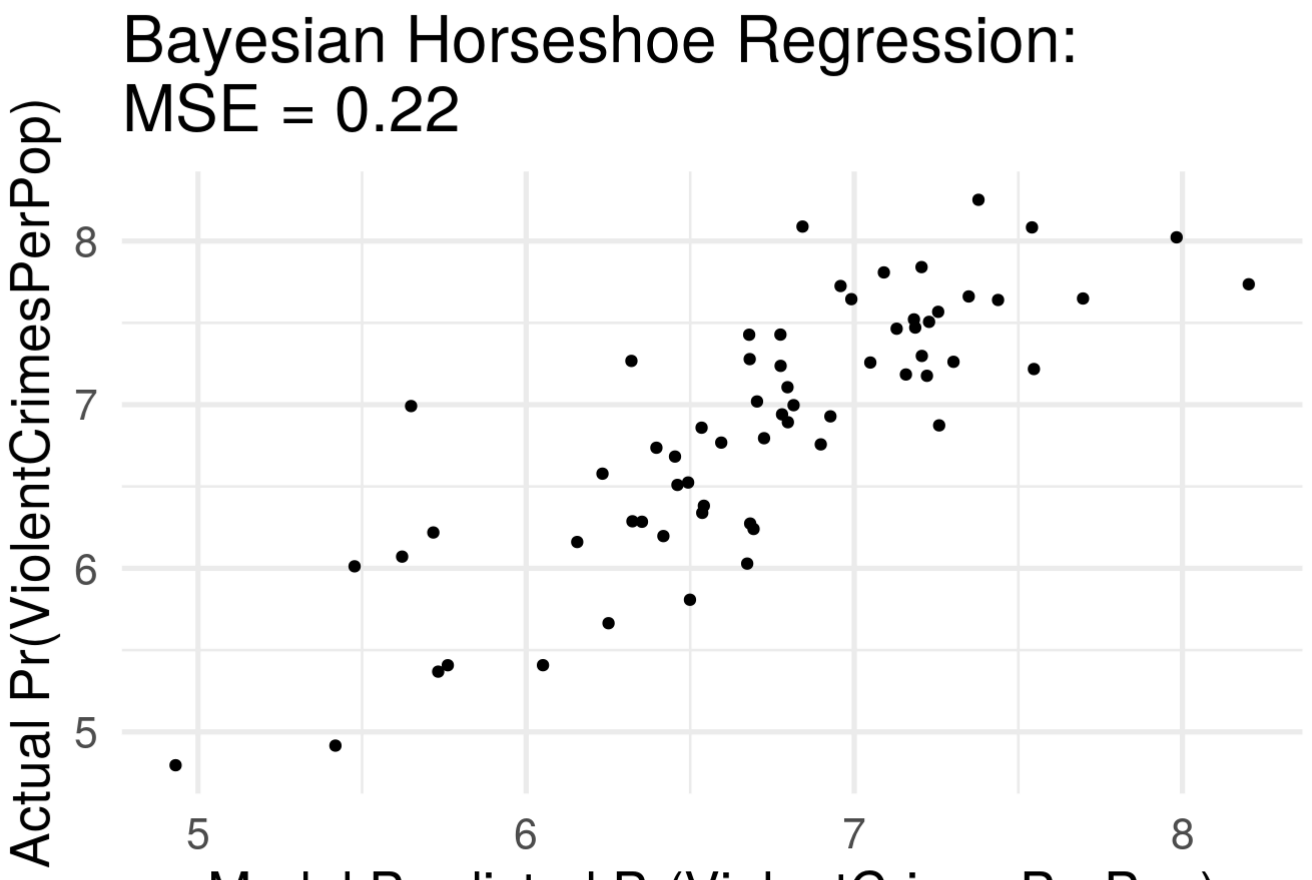
```
## Warning in readLines(file, warn = TRUE): incomplete final line found on
## '/u/12/rezaeiz1/unix/Dropbox (Aalto)/PhD/Courses/BDA/2019/bda_project/
## horseshoe_hierarchical.stan'
```

```
# Fit the model using Stan's NUTS HMC sampler
fit_bayes_horseshoe <- sampling(bayes_horseshoe, data_crime, iter = 2000,
                                warmup = 500, chains = 4, cores = 4)
```

```
## Warning: There were 232 divergent transitions after warmup. Increasing adapt_delta
above 0.8 may help. See
## http://mc-stan.org/misc/warnings.html#divergent-transitions-after-warmup
```

```
## Warning: Examine the pairs() plot to diagnose sampling problems
```

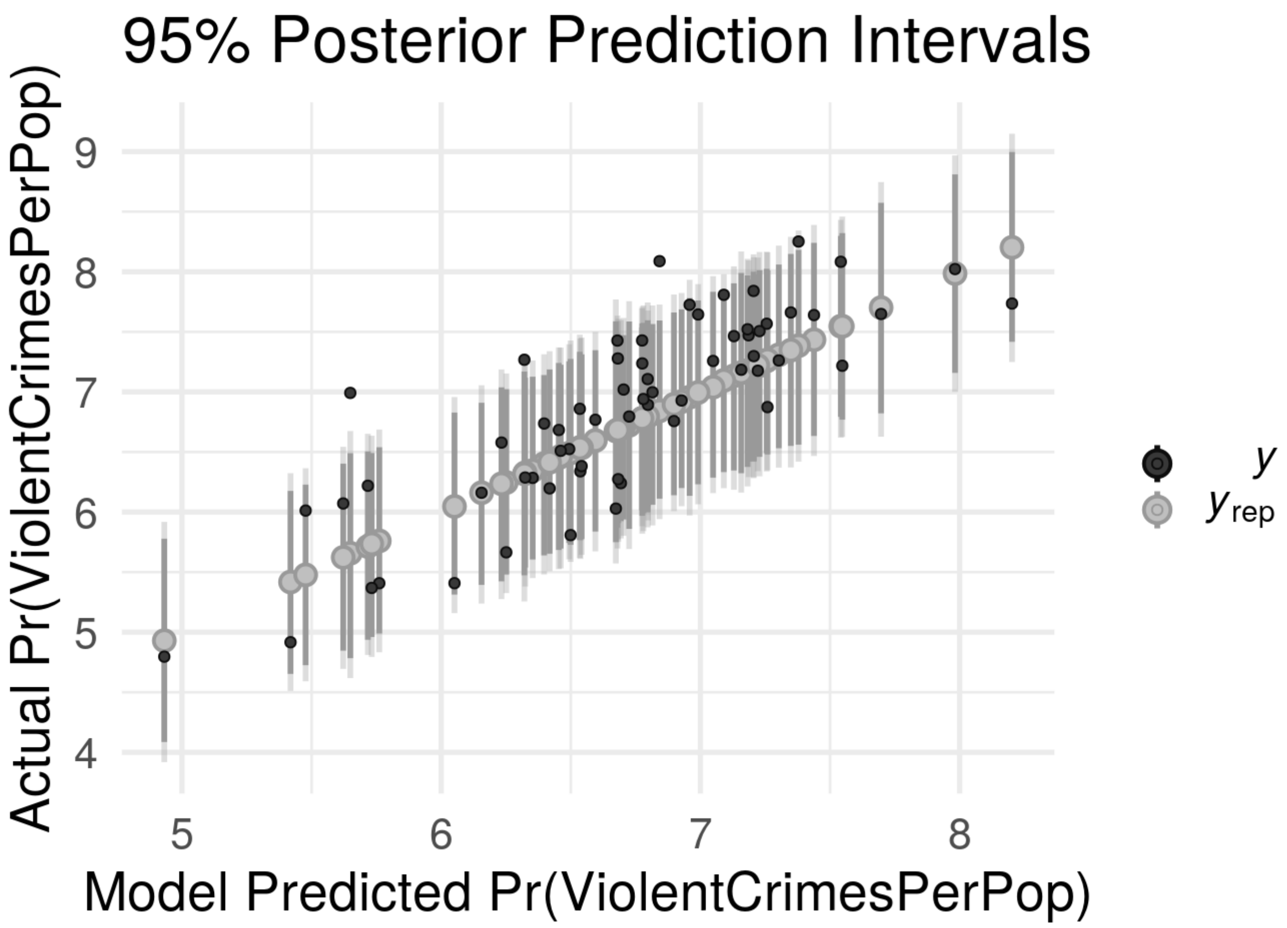
```
# Extract posterior distribution (parameters and predictions)
post_horseshoe <- rstan::extract(fit_bayes_horseshoe)

# Compute mean of the posterior predictive distribution over test set predictors,
# which integrates out uncertainty in parameter estimates
y_pred_bayes_horseshoe <- apply(post_horseshoe$y_test, 2, mean)

# Plot correlation between posterior predicted mean and actual Pr(ViolentCrimesPerPo
p)
qplot(x = y_pred_bayes_horseshoe, y = y_test,
      main = paste0("Bayesian Horseshoe Regression:\nMSE = ", round(mean((y_test - y_
pred_bayes_horseshoe)^2),2))) +
    xlab("Model Predicted Pr(ViolentCrimesPerPop)") +
    ylab("Actual Pr(ViolentCrimesPerPop)") +
    theme_minimal(base_size = 20)
```



```
#print(fit_bayes_horseshoe)

util$check_all_diagnostics(fit_bayes_horseshoe)
```

```
## [1] "n_eff / iter looks reasonable for all parameters"
## [1] "Rhat looks reasonable for all parameters"
## [1] "232 of 6000 iterations ended with a divergence (3.86666666666667%)"
## [1] "  Try running with larger adapt_delta to remove the divergences"
## [1] "0 of 6000 iterations saturated the maximum tree depth of 10 (0%)"
## [1] "E-BFMI indicated no pathological behavior"
```

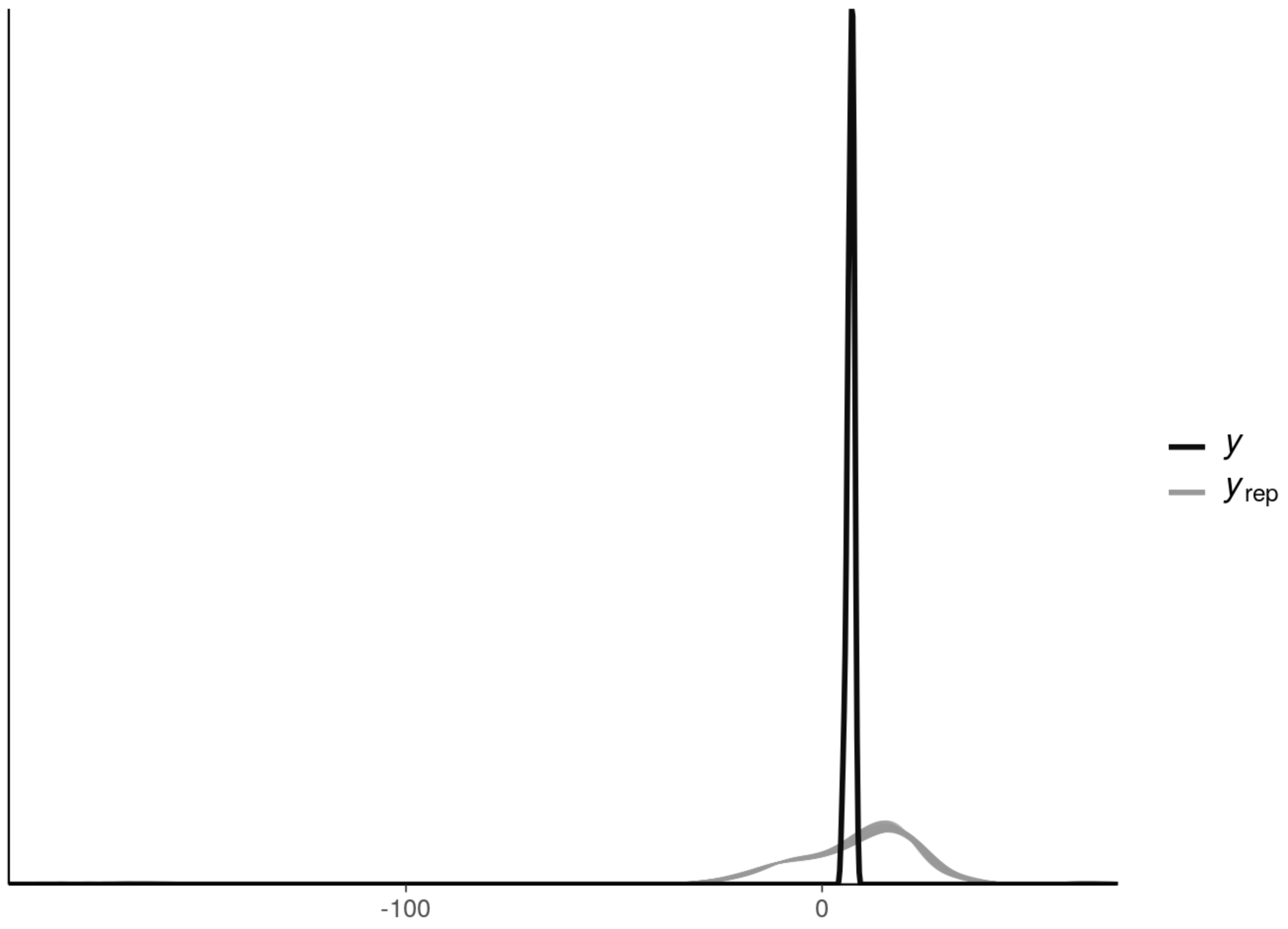
```
# `bayesplot` has many convenience functions for working with posteriors
color_scheme_set(scheme = "darkgray")
ppc_intervals(x = colMeans(post_horseshoe$y_test), y = y_test,
              yrep = post_horseshoe$y_test, prob = 0.95) +
  ggtitle("95% Posterior Prediction Intervals") +
  xlab("Model Predicted Pr(ViolentCrimesPerPop)") +
  ylab("Actual Pr(ViolentCrimesPerPop)") +
  theme_minimal(base_size = 20)
```



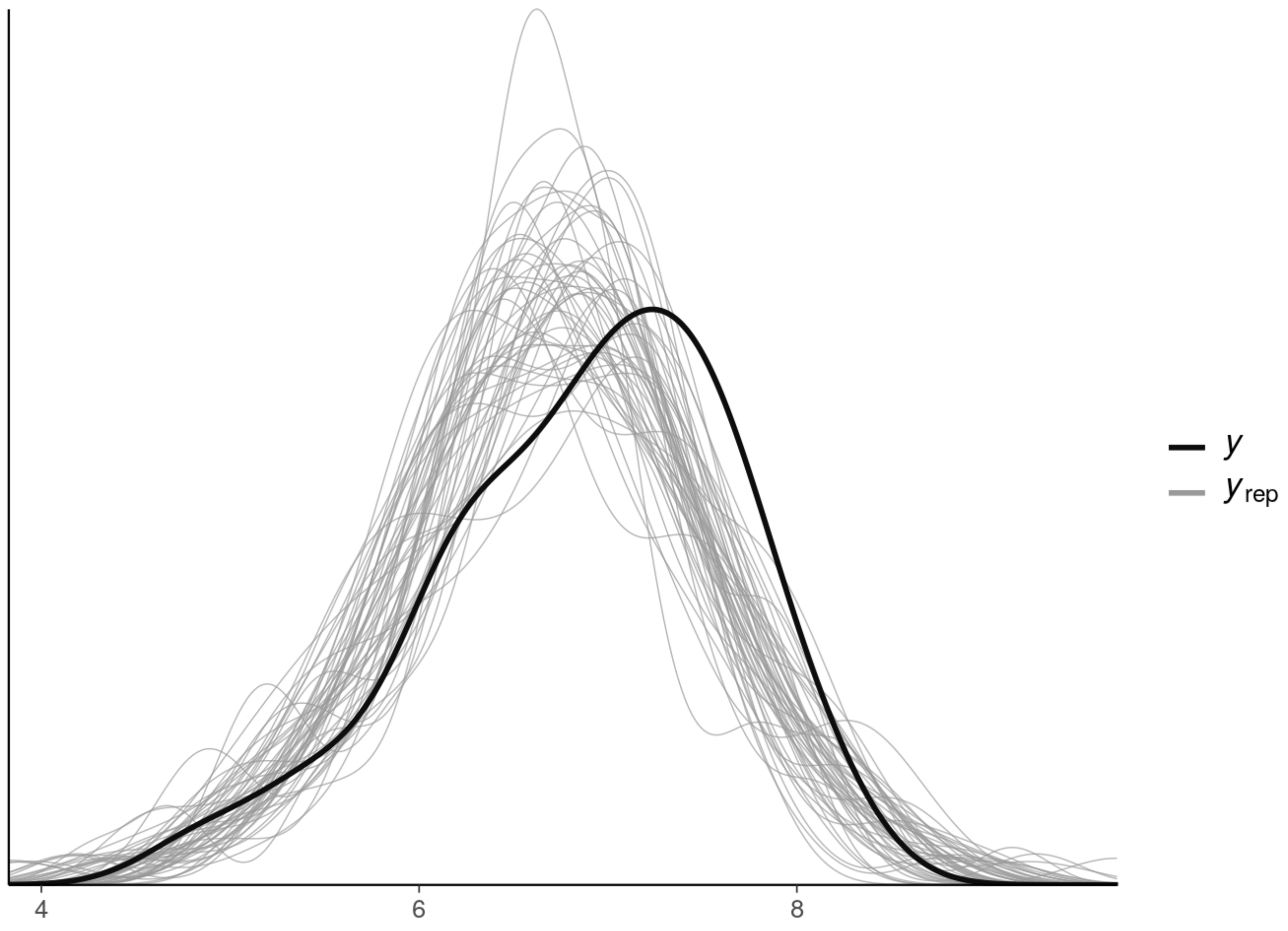
Compare density estimate of y to density estimates of a bunch of y_reps

The density estimate of the data y_test to the distributions of replicated data yrep from the posterior predictive distribution, is plotted for four intorduced models. There is a huge difference between the nonregularized model and the other 3 regularized models.

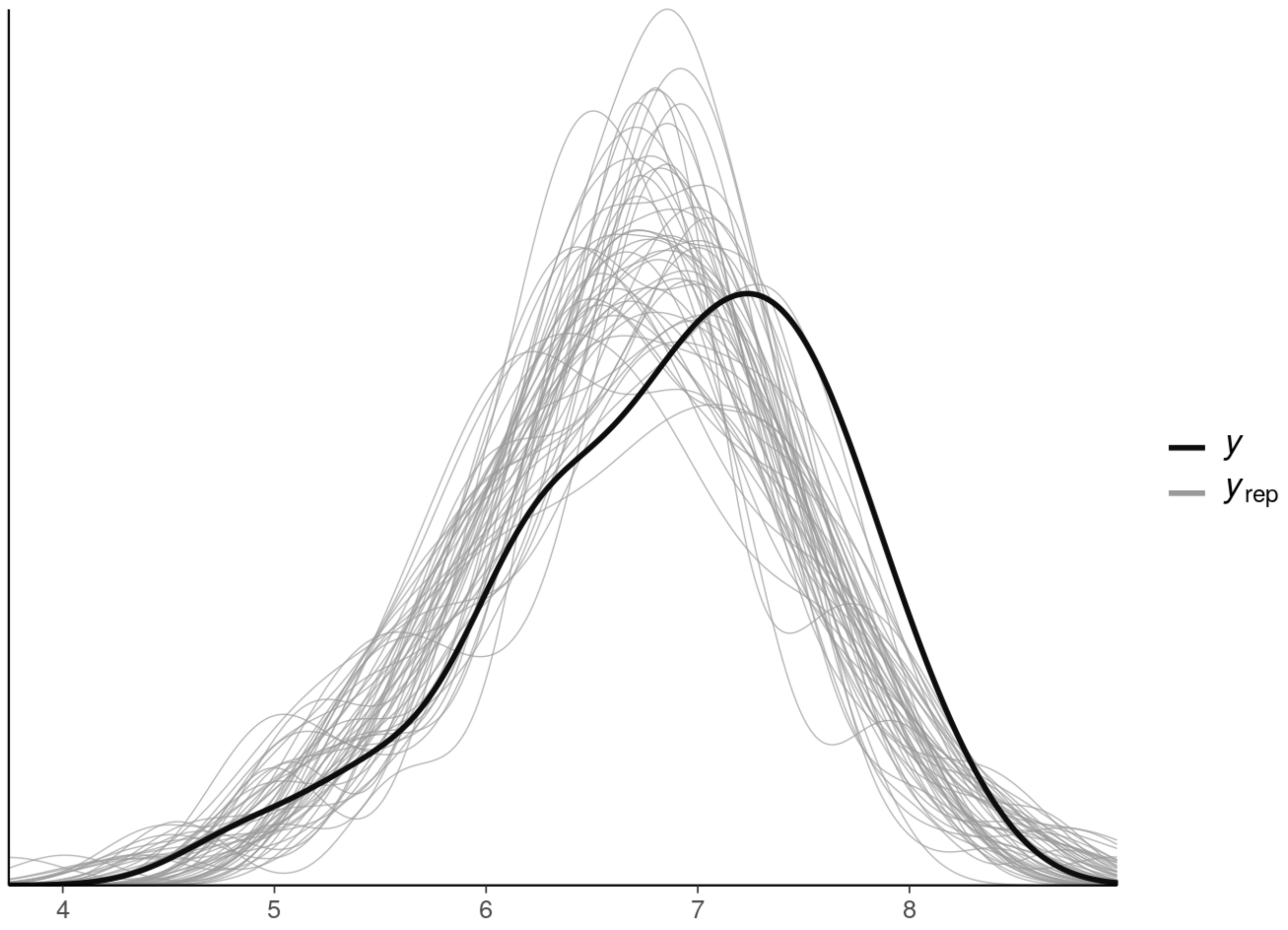
```r
y_rep_uniform <- as.matrix(fit_bayes_uniform, pars = "y_test")
y_rep_ridge <- as.matrix(fit_bayes_ridge, pars = "y_test")
y_rep_lasso <- as.matrix(fit_bayes_lasso, pars = "y_test")
y_rep_horseshoe <- as.matrix(fit_bayes_horseshoe, pars = "y_test")

par(mfrow=c(1,4))
ppc_dens_overlay(y_test, y_rep_uniform[1:50, ])
```



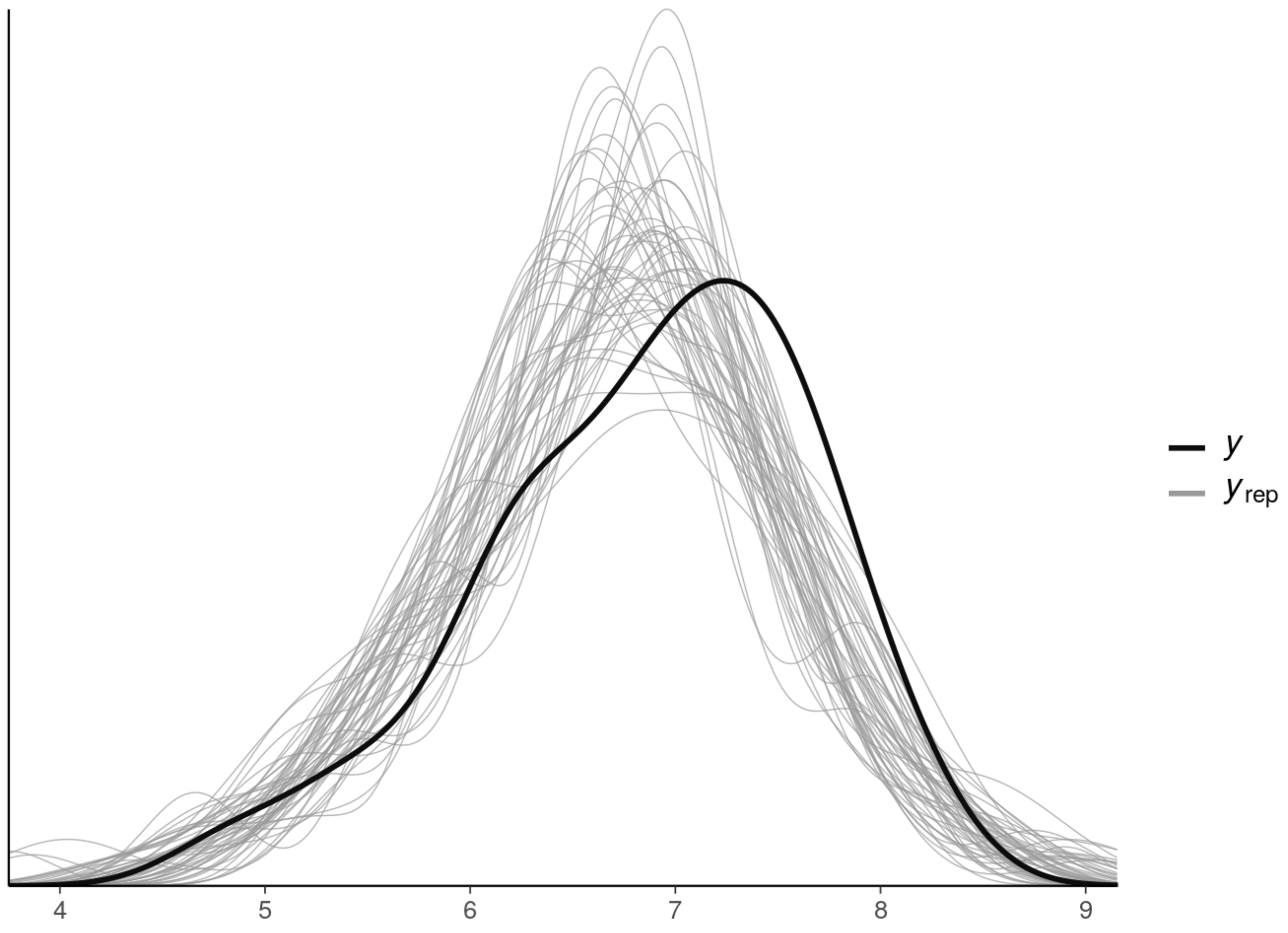
```r
ppc_dens_overlay(y_test, y_rep_ridge[1:50, ])
```



```
ppc_dens_overlay(y_test, y_rep_lasso[1:50, ])
```



```
ppc_dens_overlay(y_test, y_rep_horseshoe[1:50, ])
```



```
#p <- subplot(p_ridge, p_lasso, p_horseshoe, nrows = 1)
# chart_link <- api_create(p, filename = "subplot-basic")
# chart_link
#
# ppc_stat(y_test, y_rep_ridge, stat = "prop_zero")
# ppc_stat(y_test, y_rep_lasso, stat = "prop_zero")
# ppc_stat(y_test, y_rep_horseshoe, stat = "prop_zero")
#
# ppc_error_hist(y_test, y_rep_ridge[1:4, ], binwidth = 1) + xlim(-15, 15)
# ppc_error_hist(y_test, y_rep_lasso[1:4, ], binwidth = 1) + xlim(-15, 15)
# ppc_error_hist(y_test, y_rep_horseshoe[1:4, ], binwidth = 1) + xlim(-15, 15)
```

Computing approximate leave-one-out cross-validation using PSIS-LOO

Both visualized and tabulated are provided for $\hat{k}$ values above. If all the $\hat{k}$ values are less than 0.7, the PSIS-LOO estimate can be considered to be reliable.

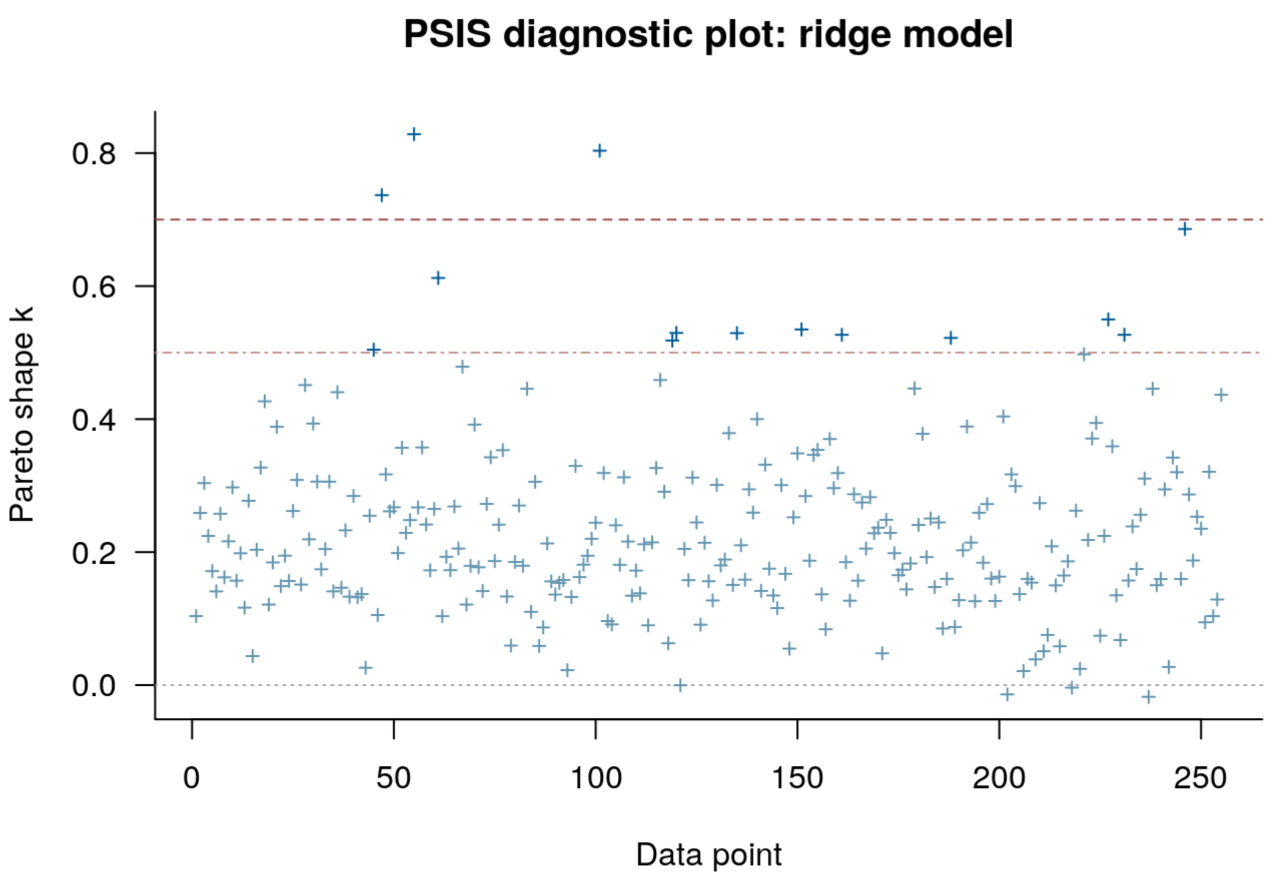
```
log_lik_ridge <- extract_log_lik(fit_bayes_ridge, merge_chains = FALSE)
r_eff_ridge <- relative_eff(exp(log_lik_ridge))
loo_ridge <- loo(log_lik_ridge, r_eff = r_eff_ridge)
```

```
## Warning: Some Pareto k diagnostic values are too high. See help('pareto-k-diagnost
ic') for details.
```

```
print(loo_ridge)
```

```
##
## Computed from 6000 by 255 log-likelihood matrix
##
##          Estimate   SE
## elpd_loo   -174.4 12.2
## p_loo        38.0  4.0
## looic       348.8 24.4
## ------
## Monte Carlo SE of elpd_loo is NA.
##
## Pareto k diagnostic values:
##                          Count Pct.    Min. n_eff
## (-Inf, 0.5]   (good)     241   94.5%   1181
##  (0.5, 0.7]   (ok)        11    4.3%   285
##    (0.7, 1]   (bad)        3    1.2%   221
##    (1, Inf)   (very bad)   0    0.0%   <NA>
## See help('pareto-k-diagnostic') for details.
```

```
plot(loo_ridge, diagnostic = c("k", "n_eff"),
  label_points = FALSE, main = "PSIS diagnostic plot: ridge model")
```



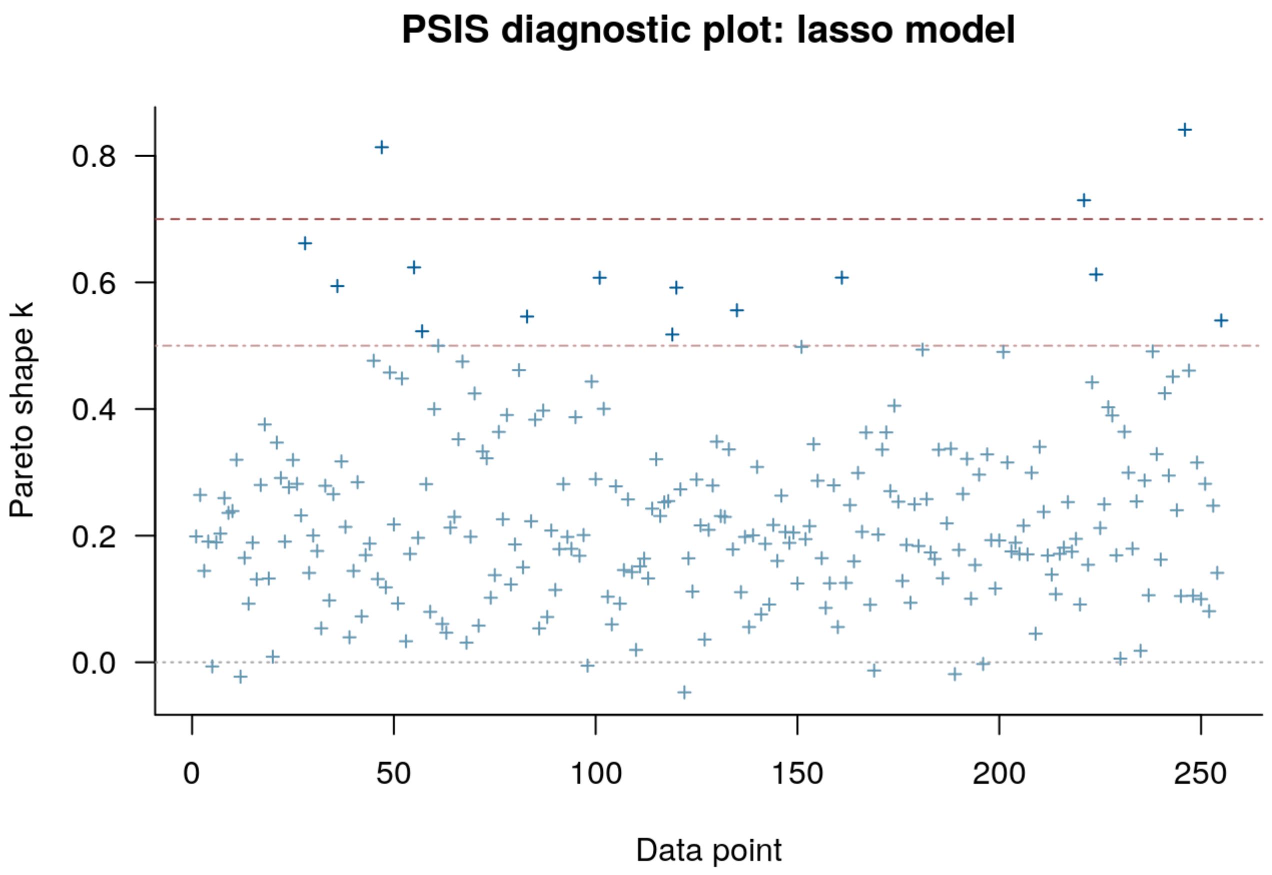
```
log_lik_lasso <- extract_log_lik(fit_bayes_lasso, merge_chains = FALSE)
r_eff_lasso <- relative_eff(exp(log_lik_lasso))
loo_lasso <- loo(log_lik_lasso, r_eff = r_eff_lasso)
```

```
## Warning: Some Pareto k diagnostic values are too high. See help('pareto-k-diagnost
ic') for details.
```

```
print(loo_lasso)
```

```
##
## Computed from 6000 by 255 log-likelihood matrix
##
##          Estimate   SE
## elpd_loo   -172.8 12.2
## p_loo        37.5  3.9
## looic       345.6 24.4
## ------
## Monte Carlo SE of elpd_loo is NA.
##
## Pareto k diagnostic values:
##                          Count Pct.    Min. n_eff
## (-Inf, 0.5]   (good)     240   94.1%   611
##  (0.5, 0.7]   (ok)        12    4.7%   243
##    (0.7, 1]   (bad)        3    1.2%   140
##    (1, Inf)   (very bad)   0    0.0%   <NA>
## See help('pareto-k-diagnostic') for details.
```

```
plot(loo_lasso, diagnostic = c("k", "n_eff"),
  label_points = FALSE, main = "PSIS diagnostic plot: lasso model")
```



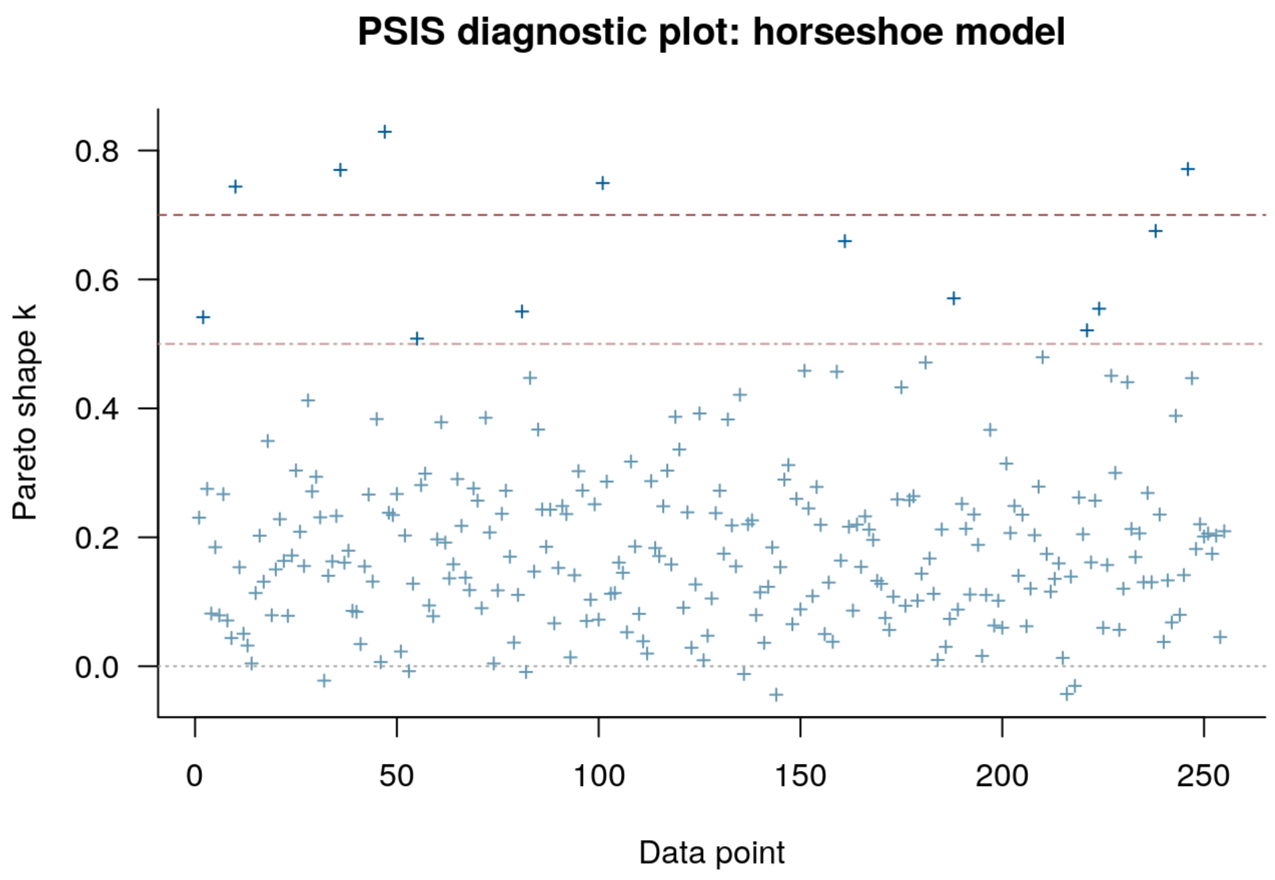
```
log_lik_horseshoe <- extract_log_lik(fit_bayes_horseshoe, merge_chains = FALSE)
r_eff_horseshoe <- relative_eff(exp(log_lik_horseshoe))
loo_horseshoe <- loo(log_lik_horseshoe, r_eff = r_eff_horseshoe)
```

```
## Warning: Some Pareto k diagnostic values are too high. See help('pareto-k-diagnost
ic') for details.
```

```
print(loo_horseshoe)
```

```
##
## Computed from 6000 by 255 log-likelihood matrix
##
##          Estimate   SE
## elpd_loo   -172.5 12.7
## p_loo        28.6  3.3
## looic       345.0 25.5
## ------
## Monte Carlo SE of elpd_loo is NA.
##
## Pareto k diagnostic values:
##                          Count Pct.    Min. n_eff
## (-Inf, 0.5]   (good)     242   94.9%   778
##  (0.5, 0.7]   (ok)         8    3.1%   388
##    (0.7, 1]   (bad)        5    2.0%   115
##    (1, Inf)   (very bad)   0    0.0%   <NA>
## See help('pareto-k-diagnostic') for details.
```

```
plot(loo_horseshoe, diagnostic = c("k", "n_eff"),
  label_points = FALSE, main = "PSIS diagnostic plot: horseshoe model")
```



As it can be seen from the plots, in the above models there are some observations with pareto $\hat{k}$ values more than 0.7, so there is a concern that these modles may be biased and they can not be considered as reliable.

Comparison

We can now compare the models on LOO using the compare function:

```
compare_ridge_lasso <- compare(loo_ridge, loo_lasso)
print(compare_ridge_lasso)
```

```
## elpd_diff        se
##       1.6       0.8
```

```
compare_ridge_horseshoe <- compare(loo_ridge, loo_horseshoe)
print(compare_ridge_horseshoe)
```

```
## elpd_diff        se
##       1.9       3.1
```

```
compare_lasso_horseshoe <- compare(loo_lasso, loo_horseshoe)
print(compare_lasso_horseshoe)
```

```
## elpd_diff        se
##       0.3       2.6
```

Based on the above results the horseshoe model is a winner in the predictive performance.

Conclusion

In this project, data from the USA Communities and Crime Data Set, sourced from the UCI Dataset Repository, was used with different shrinkage priors to predict the level of Violent Crime in USA Communities.

References

Betancourt, Michael. 2017. “Robust Statistical Workflow with RStan.” https://mc-stan.org/users/documentation/case-studies/rstan_workflow.html (https://mc-stan.org/users/documentation/case-studies/rstan_workflow.html).

Carvalho, Carlos M, Nicholas G Polson, and James G Scott. 2010. “The Horseshoe Estimator for Sparse Signals.” *Biometrika* 97 (2). Oxford University Press: 465–80.

Figueiredo, Mário. 2002. “Adaptive Sparseness Using Jeffreys Prior.” In *Advances in Neural Information Processing Systems*, 697–704.

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. 2001. *The Elements of Statistical Learning*. Vol. 1. 10. Springer series in statistics New York.

Redmond, Michael, and Alok Baveja. 2002. “A Data-Driven Software Tool for Enabling Cooperative Information Sharing Among Police Departments.” *European Journal of Operational Research* 141 (3). Elsevier: 660–78.

Tibshirani, Robert. 1996. “Regression Shrinkage and Selection via the Lasso.” *Journal of the Royal Statistical Society: Series B (Methodological)* 58 (1). Wiley Online Library: 267–88.

Van Erp, Sara, Daniel L Oberski, and Joris Mulder. 2019. “Shrinkage Priors for Bayesian Penalized Regression.” *Journal of Mathematical Psychology* 89. Elsevier: 31–50.