BDA 2021 Project. Which hour is the post to post on HackerNews?

Contents

Introduction

HackerNews (http://news.ycombinator.com) is a popular technical news board where people share links on various resources. Each post has a score that accumulates accorging to user upvotes and downvotes. Recent posts with high scores get a high ranking on the news board. Then, post rank gradually degrades with time. The actual scoring algorithm can be explored here.

Goals of this project are:

1. Try to identify which is the **best hour to post** in HN using Bayesian methods
2. Explore different methods that allow to fit and evaluate Stan models on large datasets

```r
library(fs)
library(tidyverse)
library(jsonlite)
library(lubridate)
library(rstan)
library(cmdstanr)
library(bayesplot)
library(loo)
library(posterior)
library(glue)

SEED = 42
set.seed(SEED)
```

Data loading

Let's load the historical HN post data from this repository. The data is stored in JSONL format, where each line in a file contains JSON object that represents a single post.

First, we load all JSONL file contents into strings:

```r
json_data <- dir_ls('./hackernews-post-datasets', glob = "*.json") %>%
  map_dfr(~ paste(readLines(.), collapse = ""))
```

Then, we parse the JSONL strings using the `jsonlite` package and combine all records into a single tibble:

```r
hn_posts <- as_tibble(
  json_data %>% map_dfr(fromJSON, flatten=TRUE)
  ) %>%
  drop_na(score) %>%
  mutate(
    time = as_datetime(time),
    hour = hour(time) + 1,
    weekday = as.factor(weekdays(time)),
    year = year(time)
  )
```

Exploratory Data Analysis

Now, we will explore the dataset. The primary things we are interested for this task are score distributions for each hour and weekday.

The dataset contains only posts and no comments, so we don't need to filter it by type.

```r
hn_posts %>% select(type) %>% unique
```

```
## # A tibble: 1 x 1
##   type
##   <chr>
## 1 story
```

Let's see the total number of data points:

```r
hn_posts %>% count
```

```
## # A tibble: 1 x 1
##        n
##    <int>
## 1 239417
```

This number is significant and we will likely have problems using HMC sampler on this scale.

Score summary statistics:

```
summary(hn_posts$score)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00   10.00   43.00   92.87  116.00 6015.00
```

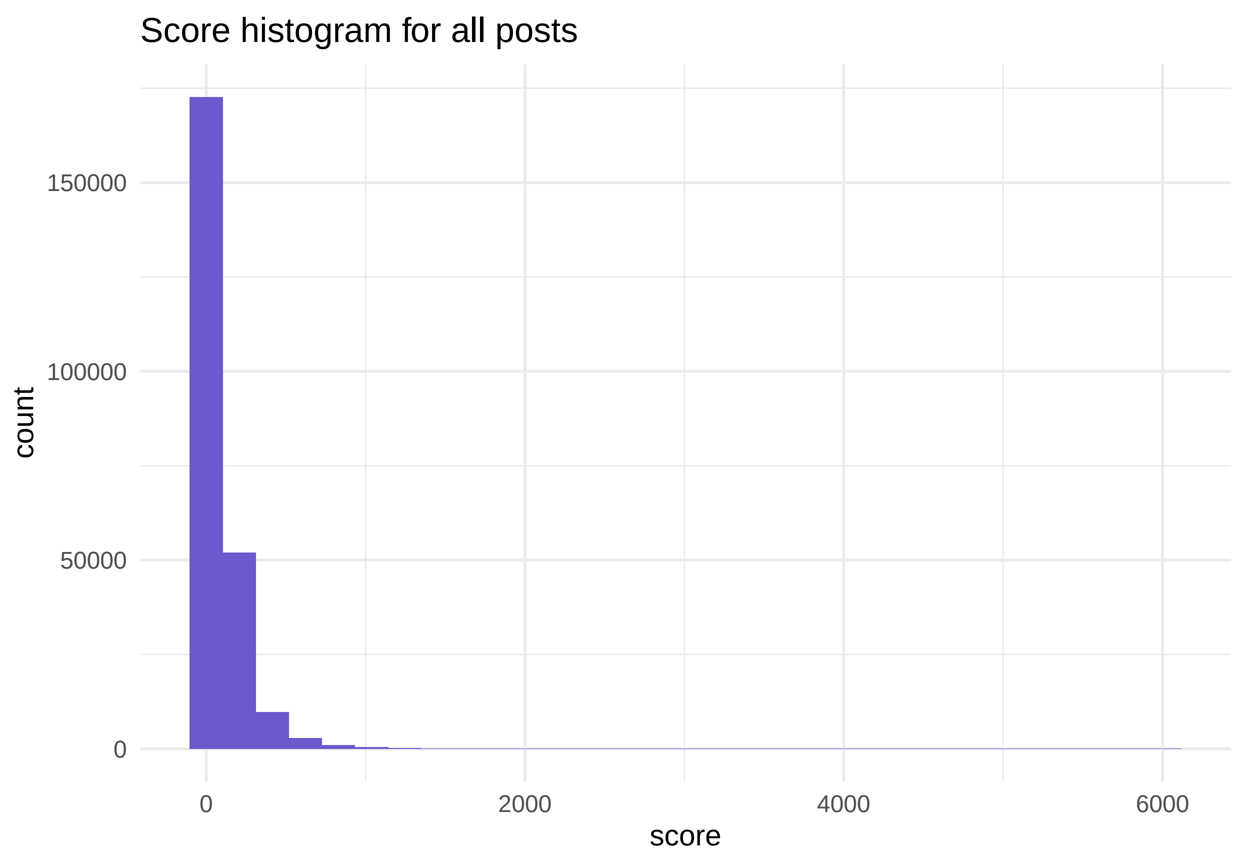
We have several years worth of data:

```
hn_posts %>%
  mutate(year = year(time)) %>%
  group_by(year) %>%
  count()
```

```
## # A tibble: 6 x 2
## # Groups:   year [6]
##    year     n
##   <dbl> <int>
## 1  2015 18878
## 2  2016 44531
## 3  2017 35925
## 4  2018 44016
## 5  2019 48722
## 6  2020 47345
```

```
ggplot(hn_posts) +
  geom_histogram(aes(score), fill='slateblue') +
  ggtitle("Score histogram for all posts") +
  theme_minimal()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



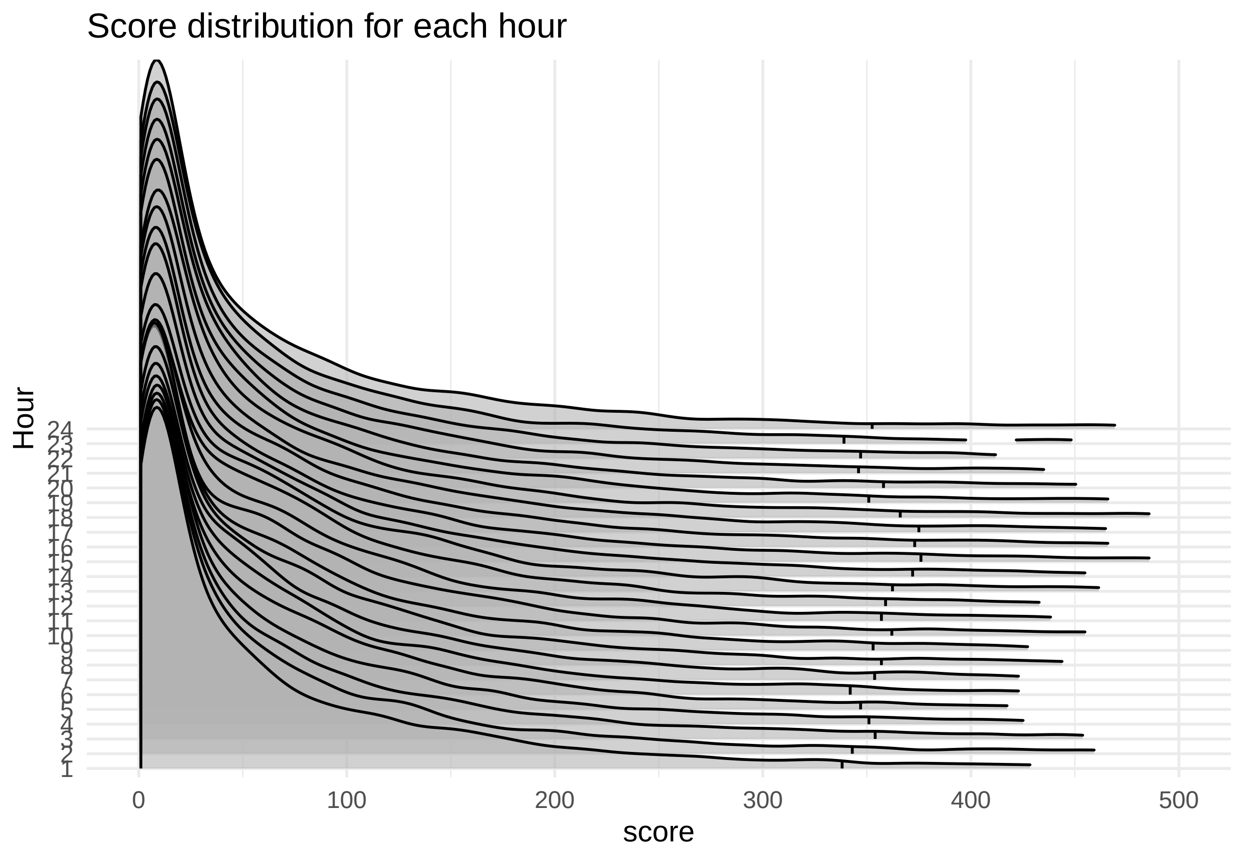
Now, let's look at how score is distributed hourly

```r
library(ggridges)
score_rigdeplot <- function (df, col, title="Score distribution for each hour", ylab="Hour") {
  qcol <- enquo(col)
  ggplot(df, aes(x = score, y = as.factor(!!qcol))) +
    stat_density_ridges(alpha = 0.6,
                        rel_min_height = 0.01,
                        scale = 25,
                        calc_ecdf = TRUE,
                        quantiles = c(0.025, 0.975),
                        quantile_lines = TRUE) +
    xlim(0, 500) +
    ylab(ylab) +
    ggtitle(title) +
    theme_minimal()
}

hn_posts %>%
  score_rigdeplot(., hour)
```

```
## Picking joint bandwidth of 10.7
```

```
## Warning: Removed 5302 rows containing non-finite values (stat_density_ridges).
```

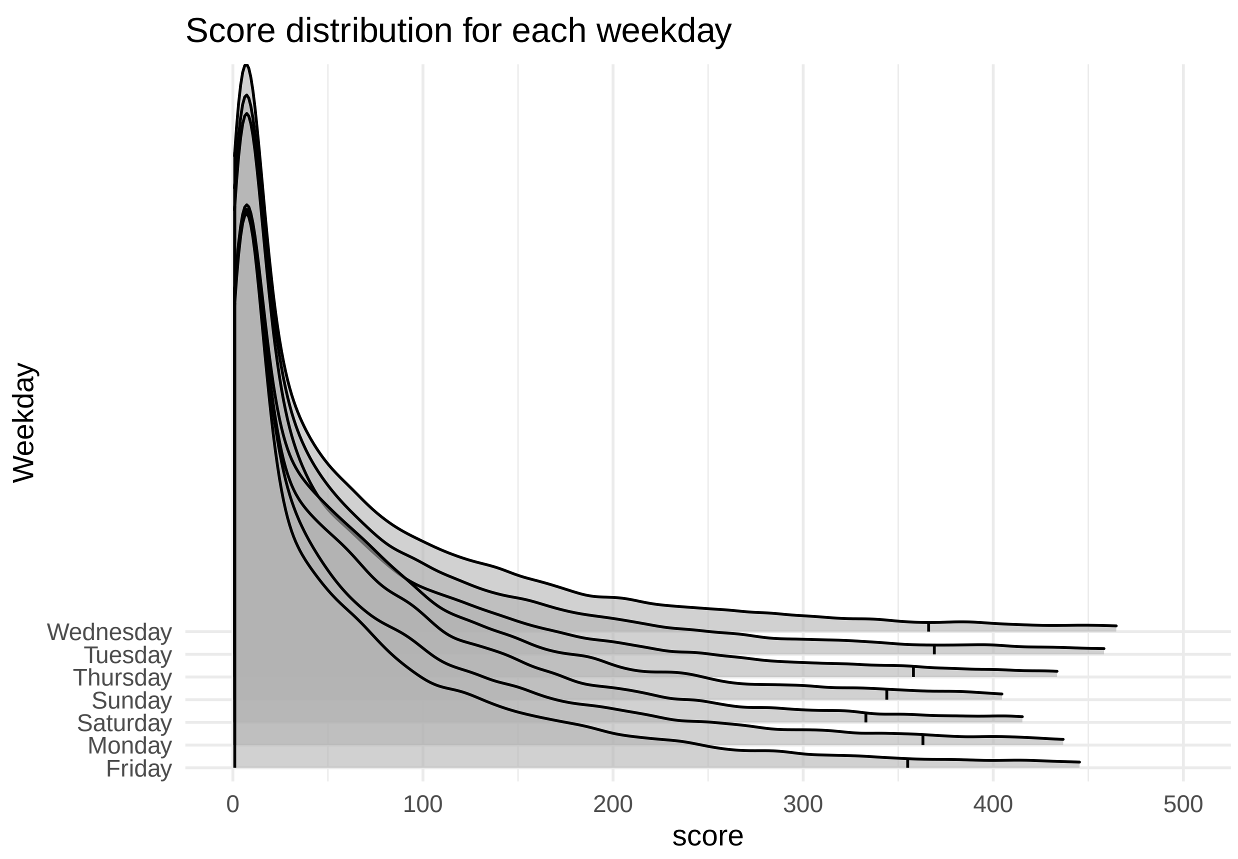


Now, let's do the same plot for weekdays:

```r
hn_posts %>%
  score_rigdeplot(., weekday, title="Score distribution for each weekday", ylab="Weekday")
```

```
## Picking joint bandwidth of 8.28
```

```
## Warning: Removed 5302 rows containing non-finite values (stat_density_ridges).
```



We see that distribution changes significantly from day to day.

The score distribution looks like Possion. In Possion distribution, sample mean $\hat{\mu}$ is equal to sample std $\hat{\sigma}$. Let's check that:

```r
hn_posts %>%
  drop_na(score) %>%
  group_by(hour) %>%
  summarise(mean = mean(score), sd = sd(score)) %>%
  mutate(diff = mean - sd) %>%
  arrange(desc(diff))
```

```
## # A tibble: 24 x 4
##     hour  mean    sd  diff
##    <dbl> <dbl> <dbl> <dbl>
##  1     5  86.7  125. -38.2
##  2     6  86.1  126. -39.6
##  3    12  97.0  137. -40.4
##  4    11  93.8  139. -45.5
##  5     3  88.0  134. -46.1
##  6    13 101.   147. -46.2
##  7    10  93.3  141. -47.4
##  8     8  90.6  140. -49.4
##  9     7  91.3  143. -51.2
## 10     2  84.1  137. -53.2
## # ... with 14 more rows
```

Seems like all hourly distributions are overdispersed. This suggests that we should use Negative-Binomial distribution instead of Poisson, as it has heavier tails.

Let's confirm the same for the weekdays:

```r
hn_posts %>%
  drop_na(score) %>%
  group_by(weekday) %>%
  summarise(mean = mean(score), sd = sd(score)) %>%
  mutate(diff = mean - sd) %>%
  arrange(desc(diff))
```

```
## # A tibble: 7 x 4
##   weekday    mean    sd  diff
##   <fct>     <dbl> <dbl> <dbl>
## 1 Saturday   86.3  129. -42.6
## 2 Sunday     89.8  134. -44.5
## 3 Friday     90.2  141. -50.6
## 4 Monday     95.1  152. -57.4
## 5 Thursday   93.7  155. -60.9
## 6 Tuesday    96.0  160. -64.1
## 7 Wednesday  96.3  165. -69.1
```

Modelling

To fit models on large dataset this project relies on various optimization techniques that we will explore further.

1. Parallel computation using FORK processes on Unix systems
2. Automatic Differentiation Variational Inference to calculate approximations for posterior distributions to allow faster inference on large datasets
3. LOO subsampling technique that allows to calculate PSIS LOO on a fraction of data samples instead of performing computation on the full dataset

```r
options(mc.cores = parallel::detectCores())
```

The current release of stan seems to be affected by a memory leak in the OpenCL code, so this project uses latest Stan and cmdstan versions that were built directly from the github repos. To install the dev version of `cmdstanr` use the following command: `remotes::install_github("stan-dev/cmdstanr")`. Also, should compile `cmdstan` manually to include all latest bug fixes. See this topic if you want to run the model, otherwise you will run out of memory.

As noted in https://discourse.mc-stan.org/t/stan-is-not-working-on-gpu-in-linux/21331/6 make sure to add `CXXFLAGS += -fpermissive` line to the makefile to avoid errors while compiling Stan models.

```r
set_cmdstan_path('/home/kdubovikov/lib/cmdstan')
```

```
## CmdStan path set to: /home/kdubovikov/lib/cmdstan
```

```
check_cmdstan_toolchain()
```

```
## The C++ toolchain required for CmdStan is setup properly!
```

Simple model

First, we will use simple model that fits Negative Binomial distribution to the data. The model uses complete pooling, meaning that all observations share the same parameters. As a digression, here are a few discussions featuring Andrew Gelman about priors for Negative Binomial models:

- https://groups.google.com/g/stan-users/c/__xxNlLGn2BI
- https://groups.google.com/g/stan-users/c/8kTIm5aPpwo

To fit the model for all data points we will use OpenCL. It is a framework that allows to perform likelihood computations for Stan models on GPUs so that the inference becomes highly parallelized.

To compile our model with OpenCL support we use `stan_opencl` flag.

```
simple_model <- cmdstan_model('./simple-model.stan', quiet = FALSE, cpp_options = list(stan_opencl = TRU
```

```
##
## --- Translating Stan model to C++ code ---
## bin/stanc --use-opencl --name='simple-model_model' --o=/tmp/RtmpHBXgZS/model-1ccd5093cec3.hpp /tmp/R
##
## --- Compiling, linking C++ code ---
## g++ -fpermissive -std=c++1y -pthread -D_REENTRANT -Wno-sign-compare -Wno-ignored-attributes   -I star
## g++ -fpermissive -std=c++1y -pthread -D_REENTRANT -Wno-sign-compare -Wno-ignored-attributes   -I star
## rm -f /tmp/RtmpHBXgZS/model-1ccd5093cec3.o
```

The source code of the simple model:

```
simple_model
```

```
## Warning in readLines(self$stan_file()): incomplete final line found on '/home/
## kdubovikov/Projects/bda-2021/project/./simple-model.stan'
```

```
## data {
##   int<lower=0> N;        // number of observations
##   int<lower=0> score[N]; // post scores
## }
##
## parameters {
##   real<lower=0> mu;
##   real<lower=0> phi;
## }
##
## model {
##   mu ~ cauchy(0, 10);
##   phi ~ cauchy(0, 10);
##   score ~ neg_binomial_2_log(mu, phi);
## }
```

Now, let's fit the simple model using NUTS sampler. OpenCL optimizations will be applied automatically for the likelihood computation.

```
data <- list(score = hn_posts$score - 1, N = length(hn_posts$score))

simple_fit <- simple_model$sample(data = data, chains = 4, seed = SEED)
```

```
## Running MCMC with 4 chains, at most 32 in parallel...
##
## Chain 1 Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 3 Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 2 Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 4 Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 3 Iteration:  100 / 2000 [  5%]  (Warmup)
## Chain 2 Iteration:  100 / 2000 [  5%]  (Warmup)
## Chain 4 Iteration:  100 / 2000 [  5%]  (Warmup)
## Chain 1 Iteration:  100 / 2000 [  5%]  (Warmup)
## Chain 3 Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 2 Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 4 Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 1 Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 3 Iteration:  300 / 2000 [ 15%]  (Warmup)
## Chain 2 Iteration:  300 / 2000 [ 15%]  (Warmup)
## Chain 4 Iteration:  300 / 2000 [ 15%]  (Warmup)
## Chain 1 Iteration:  300 / 2000 [ 15%]  (Warmup)
## Chain 3 Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 2 Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 1 Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 4 Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 3 Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 2 Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 4 Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 1 Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 3 Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 2 Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 4 Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 1 Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 3 Iteration:  700 / 2000 [ 35%]  (Warmup)
## Chain 2 Iteration:  700 / 2000 [ 35%]  (Warmup)
## Chain 4 Iteration:  700 / 2000 [ 35%]  (Warmup)
## Chain 1 Iteration:  700 / 2000 [ 35%]  (Warmup)
## Chain 3 Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 2 Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 4 Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 1 Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 3 Iteration:  900 / 2000 [ 45%]  (Warmup)
## Chain 2 Iteration:  900 / 2000 [ 45%]  (Warmup)
## Chain 4 Iteration:  900 / 2000 [ 45%]  (Warmup)
## Chain 1 Iteration:  900 / 2000 [ 45%]  (Warmup)
## Chain 3 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 3 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 2 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 2 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 1 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 1 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 4 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 4 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 3 Iteration: 1100 / 2000 [ 55%]  (Sampling)
## Chain 2 Iteration: 1100 / 2000 [ 55%]  (Sampling)
## Chain 4 Iteration: 1100 / 2000 [ 55%]  (Sampling)
## Chain 1 Iteration: 1100 / 2000 [ 55%]  (Sampling)
## Chain 3 Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 2 Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 4 Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 1 Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 3 Iteration: 1300 / 2000 [ 65%]  (Sampling)
## Chain 2 Iteration: 1300 / 2000 [ 65%]  (Sampling)
## Chain 4 Iteration: 1300 / 2000 [ 65%]  (Sampling)
## Chain 1 Iteration: 1300 / 2000 [ 65%]  (Sampling)
## Chain 3 Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 2 Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 4 Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 1 Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 3 Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 2 Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 4 Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 1 Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 3 Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 2 Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 4 Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 1 Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 3 Iteration: 1700 / 2000 [ 85%]  (Sampling)
## Chain 2 Iteration: 1700 / 2000 [ 85%]  (Sampling)
## Chain 4 Iteration: 1700 / 2000 [ 85%]  (Sampling)
## Chain 1 Iteration: 1700 / 2000 [ 85%]  (Sampling)
## Chain 2 Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 3 Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 1 Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 4 Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 2 Iteration: 1900 / 2000 [ 95%]  (Sampling)
## Chain 3 Iteration: 1900 / 2000 [ 95%]  (Sampling)
## Chain 4 Iteration: 1900 / 2000 [ 95%]  (Sampling)
## Chain 1 Iteration: 1900 / 2000 [ 95%]  (Sampling)
## Chain 2 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 2 finished in 65.1 seconds.
## Chain 3 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 3 finished in 65.3 seconds.
## Chain 1 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 4 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 4 finished in 66.7 seconds.
## Chain 1 finished in 66.9 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 66.0 seconds.
## Total execution time: 67.6 seconds.
```

Let's run model diagnostics:

```
simple_fit$cmdstan_diagnose()
```

```
## Processing csv files: /tmp/RtmpHBXgZS/simple-model-202105181124-1-48975f.csv, /tmp/RtmpHBXgZS/simple-
##
## Checking sampler transitions treedepth.
## Treedepth satisfactory for all transitions.
##
## Checking sampler transitions for divergences.
## No divergent transitions found.
##
## Checking E-BFMI - sampler transitions HMC potential energy.
## E-BFMI satisfactory for all transitions.
##
## Effective sample size satisfactory.
##
## Split R-hat values satisfactory all parameters.
##
## Processing complete, no problems detected.
```

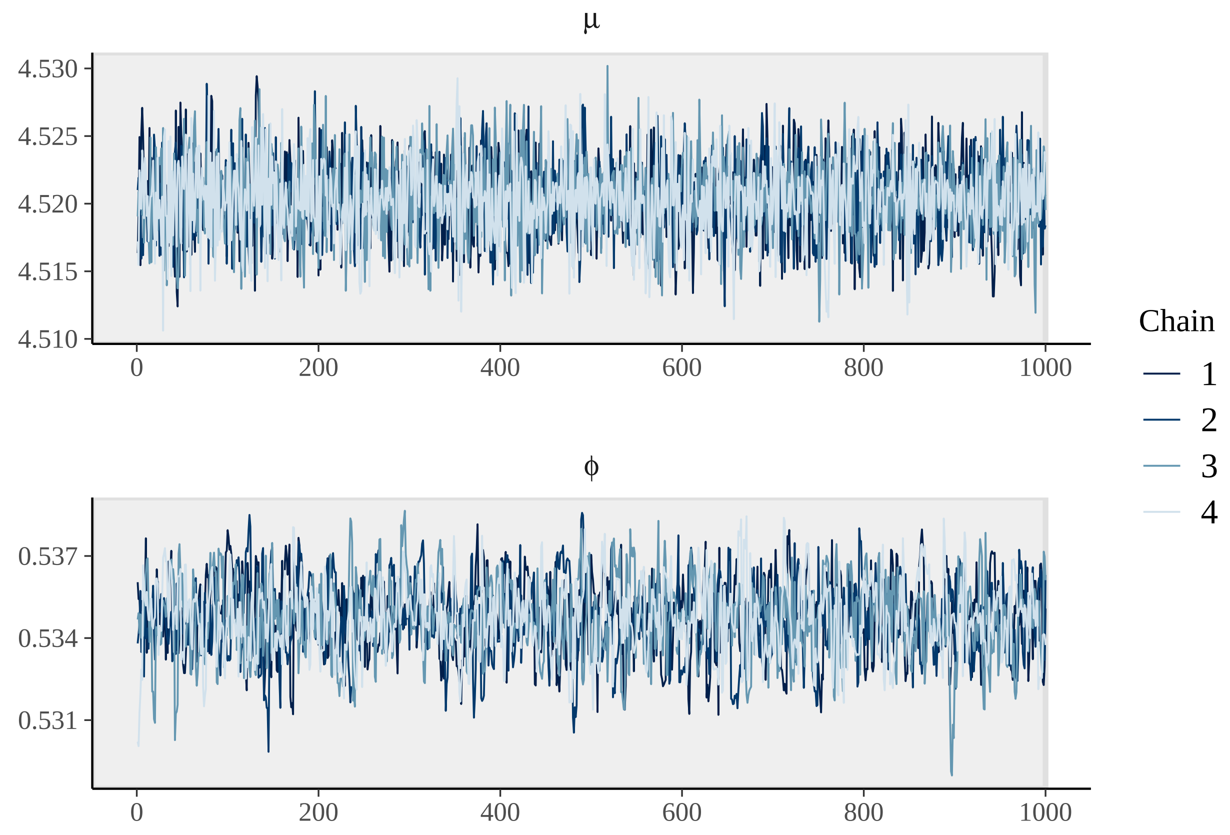
```
simple_fit$summary()
```

```
## # A tibble: 3 x 10
##   variable      mean    median      sd     mad        q5      q95  rhat ess_bulk
##   <chr>        <dbl>     <dbl>   <dbl>   <dbl>     <dbl>    <dbl> <dbl>    <dbl>
## 1 lp__      -1.29e+6  -1.29e+6 0.499   0        -1.29e+6 -1.29e+6  1.00    2873.
## 2 mu         4.52e+0   4.52e+0 0.00283 0.00285   4.52e+0  4.52e+0  1.00    4335.
## 3 phi        5.35e-1   5.35e-1 0.00135 0.00132   5.32e-1  5.37e-1  1.00    1316.
## # ... with 1 more variable: ess_tail <dbl>
```

Now, let's extract parameter draws and explore plots using `bayesplot` package:

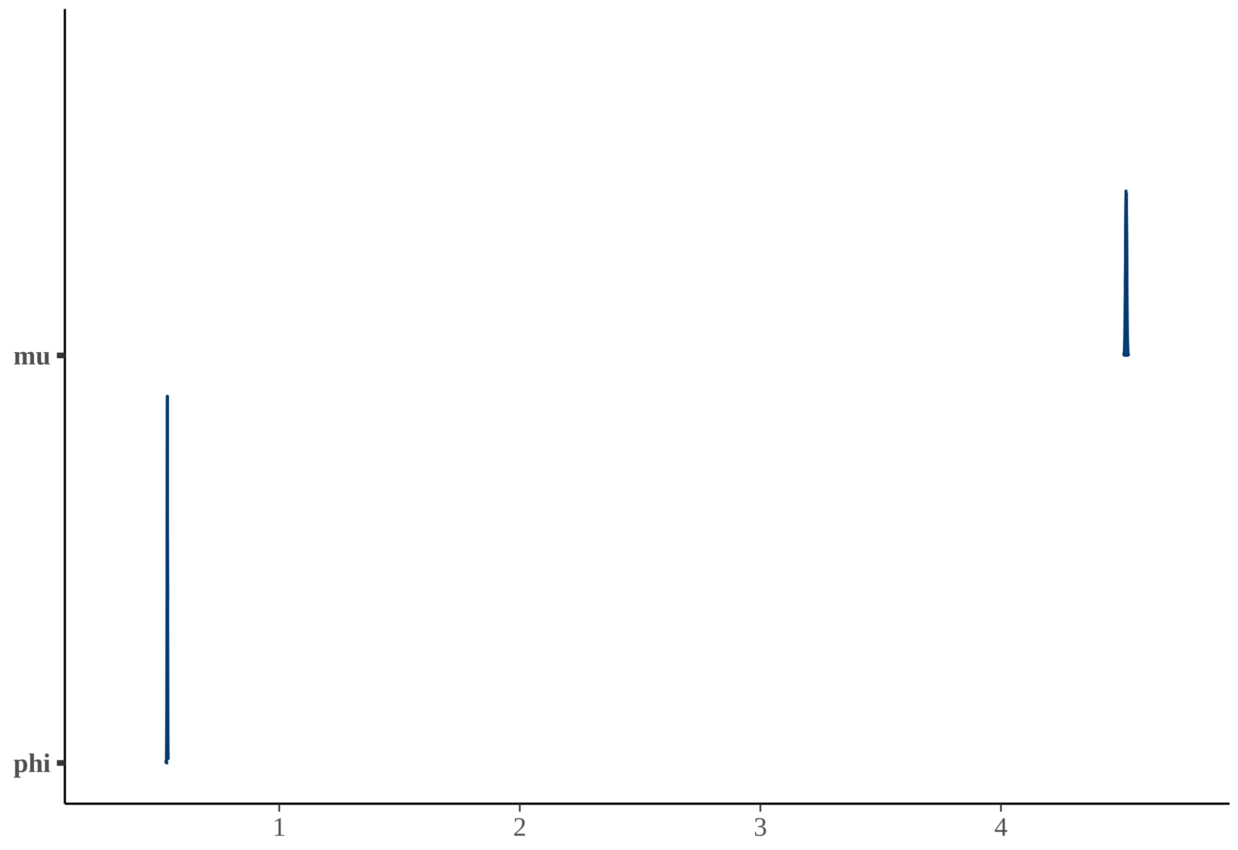
```
simple_draws <- simple_fit$draws(variables = c('mu', 'phi'))
```

```
mcmc_trace(simple_draws,  pars = c("mu", "phi"), n_warmup = 1000,
                facet_args = list(nrow = 2, labeller = label_parsed))
```



Chain plots for ϕ show slight autocorrelation. Otherwise, everything looks good.

```
mcmc_areas(as_draws_matrix(simple_draws),
           prob = 0.8,
           pars = c('mu', 'phi'))
```



We have very sharp parameter estimates because of the large dataset.

PSIS LOO

Now let's compute PSIS LOO estimates. As our dataset is big, computing log likelihood inside `generated quantities` block will consume much memory. We will use `loo_subsample` function which allows to compute PSIS LOO estimate using a subsample of the dataset. The downside of this is that we will need to define log likelihood function in R. For this model, the implementation is in the `llfun_neg_binomial` function below:

```r
llfun_neg_binomial <- function(data_i, draws, log = TRUE) {
  dnbinom(data_i$score, mu = draws[, 'mu'], size = draws[, 'phi'], log = log)
}

simple_params <- as_draws_matrix(simple_draws)

r_eff <- relative_eff(llfun_neg_binomial,
                      log = FALSE, # relative_eff wants likelihood not log-likelihood values
                      chain_id = rep(1:4, each = 1000),
                      data = as.data.frame(data),
                      draws = simple_params,
                      cores = 25)

loo_simple <- loo_subsample(llfun_neg_binomial,
              data = as.data.frame(data),
              draws = simple_params,
              r_eff = r_eff,
              observations = 5000,
              cores = 25)
```

```
## [1] 239417      2
## [1] 1 2
## [1] 25
```

```
loo_simple
```

```
##
## Computed from 4000 by 5000 subsampled log-likelihood
## values from 239417 total observations.
##
##            Estimate      SE subsampling SE
## elpd_loo -3246855.2  8482.4           17.9
## p_loo         475.1     5.3           36.3
## looic     6493710.4 16964.8           35.8
## ------
## Monte Carlo SE of elpd_loo is 0.1.
##
## All Pareto k estimates are good (k < 0.5).
## See help('pareto-k-diagnostic') for details.
```

We get all $\hat{k}$ values lower than 0.5, which indicates reliable ELPD estimate. We see, that the current model has very low ELPD, which indicates that it is too simple for this dataset.

Hierarchical model

Now, let's move closer to answering our question: which hours are the best to post on HN? To answer it, we will build a hierarchical model using additional information about time to bring new structure to the model. In particular, we will look how different years, weekdays and hours affect post scores.

First, let's fit a hierarchical model on a subset of data to see how well it will perform:

```
hierarchical_model <- cmdstan_model('./hierarchical-model.stan', quiet = FALSE)
```

```
##
## --- Translating Stan model to C++ code ---
## bin/stanc --name='hierarchical-model_model' --o=/tmp/RtmpHBXgZS/model-1ccd5a36589.hpp /tmp/RtmpHBXgZS
##
## --- Compiling, linking C++ code ---
## g++ -fpermissive -std=c++1y -pthread -D_REENTRANT -Wno-sign-compare -Wno-ignored-attributes      -I
## g++ -fpermissive -std=c++1y -pthread -D_REENTRANT -Wno-sign-compare -Wno-ignored-attributes      -I
## rm -f /tmp/RtmpHBXgZS/model-1ccd5a36589.o
```

Model code:

```
print(hierarchical_model)
```

```
## Warning in readLines(self$stan_file()): incomplete final line found on '/home/
## kdubovikov/Projects/bda-2021/project/./hierarchical-model.stan'
## data {
##   int<lower=0> N;       // number of observations
##   int<lower=0> year[N]; // year hierarchy level
##   int<lower=0> day[N];  // day hierarchy level
##   int<lower=0> hour[N]; // hour hierarchy level
##
##   int<lower=0> num_years;
##   int<lower=0> num_days;
##   int<lower=0> num_hours;
##
##   int<lower=0> score[N]; // post scores
## }
##
## parameters {
##   vector<lower=0>[num_years] year_mu;
##   vector<lower=0>[num_years] year_phi;
##   vector<lower=0>[num_days] days_mu;
##   vector<lower=0>[num_days] days_phi;
##   vector<lower=0>[num_hours] hours_mu;
##   vector<lower=0>[num_hours] hours_phi;
## }
##
## model {
##   year_mu ~ normal(0, 100);
##   year_phi ~ normal(0, 10);
##   days_mu ~ normal(0, 50);
##   days_phi ~ normal(0, 10);
##   hours_mu ~ normal(0, 10);
##   hours_phi ~ normal(0, 10);
##
##   score ~ neg_binomial_2_log(
##     year_mu[year] + days_mu[day] + hours_mu[hour],
##     year_phi[year] + days_phi[day] + hours_phi[hour]
##   );
## }
```

Here, we sample random 5000 posts from 2019 and 2020 to fit the model using NUTS sampler. On this scale, working on CPUs is faster than relying on OpenCL which gives certain overhead when moving data back and forth between GPU and RAM.

```
hn_posts_2020 <- hn_posts %>%
  filter(year %in% c(2020, 2019)) %>%
  mutate(score = score - 1) %>%
  sample_n(5000)

prepare_dataset <- function(hn_posts) {
  data <- list(score = hn_posts$score,
               N = length(hn_posts$score),
               year = as.integer(as.factor(hn_posts$year)),
               day = as.integer(hn_posts$weekday),
               hour = hn_posts$hour,
               num_years = n_distinct(hn_posts$year),
               num_days = n_distinct(hn_posts$weekday),
               num_hours = n_distinct(hn_posts$hour))
```

```r
  data
}

hierarchical_data <- prepare_dataset(hn_posts_2020)
hierarchical_fit_mcmc <- hierarchical_model$sample(data = hierarchical_data, seed = SEED)
```

```
## Running MCMC with 4 chains, at most 32 in parallel...
##
## Chain 1 Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 2 Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 3 Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 4 Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 4 Iteration:  100 / 2000 [  5%]  (Warmup)
## Chain 2 Iteration:  100 / 2000 [  5%]  (Warmup)
## Chain 3 Iteration:  100 / 2000 [  5%]  (Warmup)
## Chain 1 Iteration:  100 / 2000 [  5%]  (Warmup)
## Chain 3 Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 4 Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 3 Iteration:  300 / 2000 [ 15%]  (Warmup)
## Chain 2 Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 3 Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 1 Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 4 Iteration:  300 / 2000 [ 15%]  (Warmup)
## Chain 3 Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 2 Iteration:  300 / 2000 [ 15%]  (Warmup)
## Chain 4 Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 3 Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 1 Iteration:  300 / 2000 [ 15%]  (Warmup)
## Chain 2 Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 4 Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 3 Iteration:  700 / 2000 [ 35%]  (Warmup)
## Chain 1 Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 3 Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 2 Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 1 Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 3 Iteration:  900 / 2000 [ 45%]  (Warmup)
## Chain 4 Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 2 Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 3 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 3 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 1 Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 4 Iteration:  700 / 2000 [ 35%]  (Warmup)
## Chain 2 Iteration:  700 / 2000 [ 35%]  (Warmup)
## Chain 3 Iteration: 1100 / 2000 [ 55%]  (Sampling)
## Chain 1 Iteration:  700 / 2000 [ 35%]  (Warmup)
## Chain 4 Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 2 Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 3 Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 1 Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 4 Iteration:  900 / 2000 [ 45%]  (Warmup)
## Chain 3 Iteration: 1300 / 2000 [ 65%]  (Sampling)
## Chain 2 Iteration:  900 / 2000 [ 45%]  (Warmup)
## Chain 1 Iteration:  900 / 2000 [ 45%]  (Warmup)
## Chain 4 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 4 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 3 Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 2 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 2 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 3 Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 4 Iteration: 1100 / 2000 [ 55%]  (Sampling)
## Chain 1 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 1 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 3 Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 2 Iteration: 1100 / 2000 [ 55%]  (Sampling)
## Chain 4 Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 3 Iteration: 1700 / 2000 [ 85%]  (Sampling)
## Chain 1 Iteration: 1100 / 2000 [ 55%]  (Sampling)
## Chain 2 Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 3 Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 4 Iteration: 1300 / 2000 [ 65%]  (Sampling)
## Chain 1 Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 3 Iteration: 1900 / 2000 [ 95%]  (Sampling)
## Chain 2 Iteration: 1300 / 2000 [ 65%]  (Sampling)
## Chain 4 Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 3 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 3 finished in 602.0 seconds.
## Chain 1 Iteration: 1300 / 2000 [ 65%]  (Sampling)
## Chain 4 Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 2 Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 4 Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 1 Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 2 Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 4 Iteration: 1700 / 2000 [ 85%]  (Sampling)
## Chain 1 Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 2 Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 4 Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 1 Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 2 Iteration: 1700 / 2000 [ 85%]  (Sampling)
## Chain 4 Iteration: 1900 / 2000 [ 95%]  (Sampling)
## Chain 1 Iteration: 1700 / 2000 [ 85%]  (Sampling)
## Chain 2 Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 4 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 4 finished in 804.1 seconds.
## Chain 1 Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 2 Iteration: 1900 / 2000 [ 95%]  (Sampling)
## Chain 2 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 2 finished in 880.8 seconds.
## Chain 1 Iteration: 1900 / 2000 [ 95%]  (Sampling)
## Chain 1 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 1 finished in 921.2 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 802.0 seconds.
## Total execution time: 921.3 seconds.
```

```r
hierarchical_summary <- hierarchical_fit_mcmc$summary()
hierarchical_summary
```

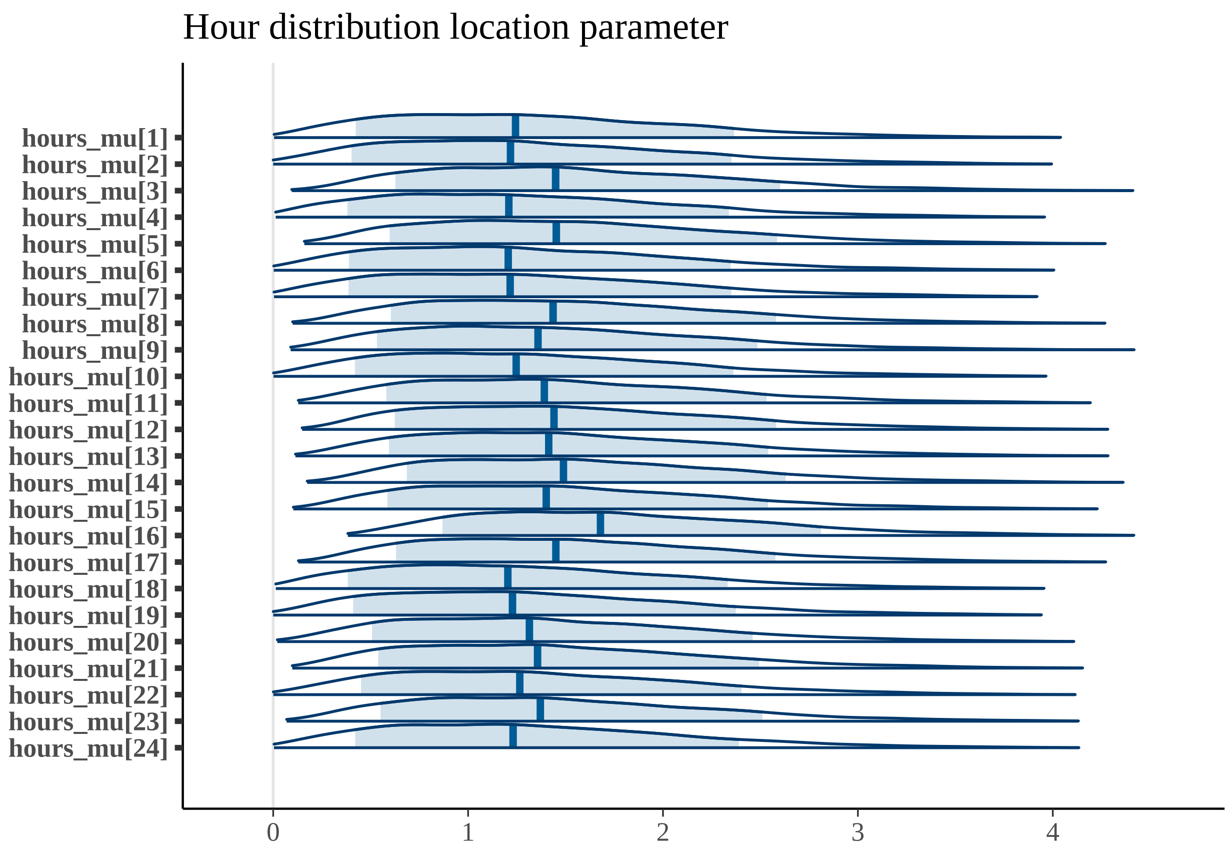
```
## # A tibble: 67 x 10
##    variable       mean   median      sd     mad       q5      q95  rhat ess_bulk
##    <chr>         <dbl>    <dbl>   <dbl>   <dbl>    <dbl>    <dbl> <dbl>    <dbl>
##  1 lp__       -2.70e+4 -2.70e+4 16.2    16.2    -2.70e+4 -2.70e+4  1.11     78.6
##  2 year_mu[~   1.42e+0  1.36e+0  0.920   1.07    1.08e-1  3.09e+0  1.08     52.6
##  3 year_mu[~   1.43e+0  1.36e+0  0.920   1.07    1.17e-1  3.09e+0  1.08     52.2
##  4 year_phi~   1.41e-1  1.26e-1  0.0853  0.0900  2.78e-2  3.03e-1  1.02    176.
##  5 year_phi~   1.14e-1  9.69e-2  0.0841  0.0883  7.76e-3  2.77e-1  1.02    177.
##  6 days_mu[~   1.61e+0  1.56e+0  0.979   1.18    2.11e-1  3.30e+0  1.02    134.
##  7 days_mu[~   1.60e+0  1.54e+0  0.978   1.17    2.09e-1  3.28e+0  1.02    135.
##  8 days_mu[~   1.53e+0  1.48e+0  0.976   1.16    1.39e-1  3.23e+0  1.02    133.
##  9 days_mu[~   1.73e+0  1.67e+0  0.978   1.17    3.25e-1  3.41e+0  1.02    135.
## 10 days_mu[~   1.64e+0  1.58e+0  0.979   1.16    2.45e-1  3.31e+0  1.03    134.
## # ... with 57 more rows, and 1 more variable: ess_tail <dbl>
```

Let's dig into the parameter visualizations

```
hierarchical_draws <- hierarchical_fit_mcmc$draws()
hierarchical_draws <- as_draws_df(hierarchical_draws)
```

```
mcmc_areas(hierarchical_draws %>%
             select(contains("hours_mu")) %>%
             as.matrix,
           prob = 0.8) +
  ggtitle("Hour distribution location parameter")
```

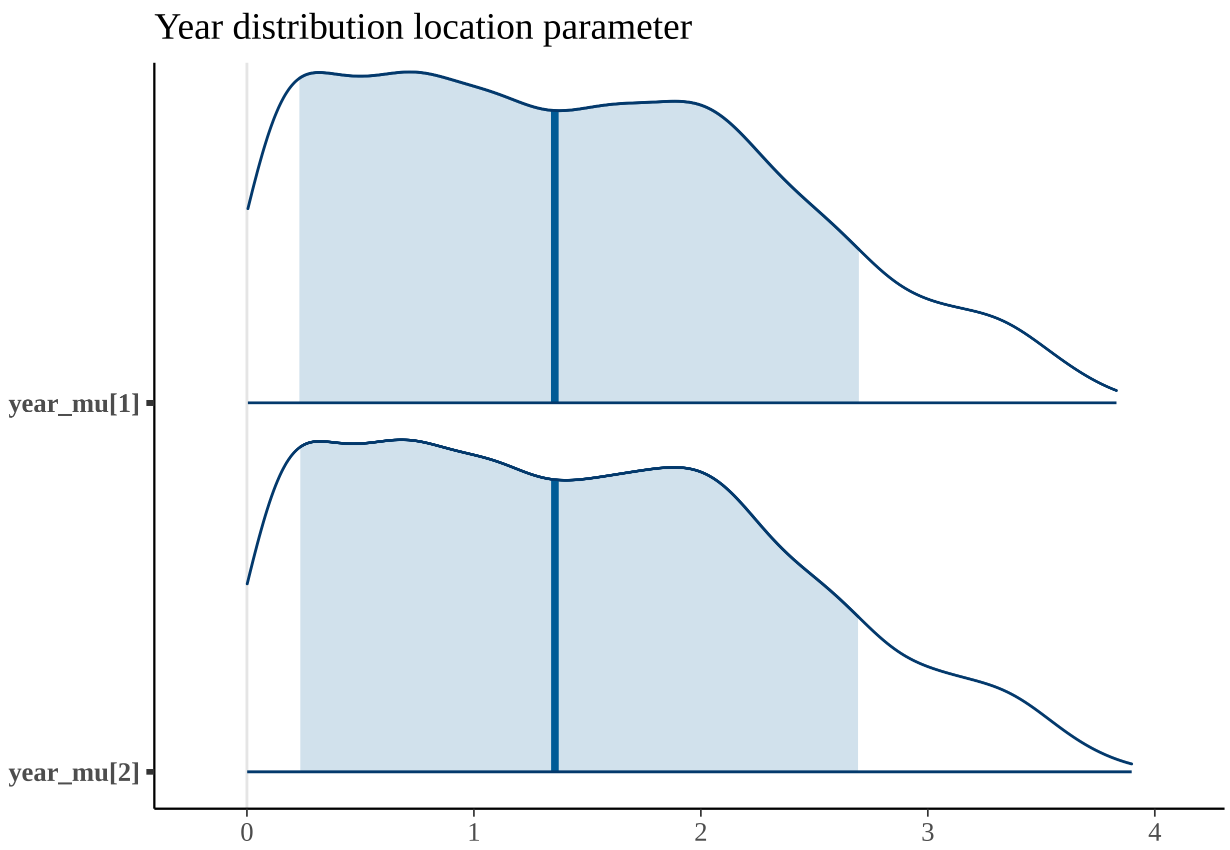
```
## Warning: Dropping 'draws_df' class as required metadata was removed.
```



For the hours, there seems to be no significant difference. The distributions are very wide and noisy, suggesting that we need to use more data to get better estimates.

```r
mcmc_areas(hierarchical_draws %>%
             select(contains("year_mu")) %>%
             as.matrix,
           prob = 0.8) +
  ggtitle("Year distribution location parameter")
```

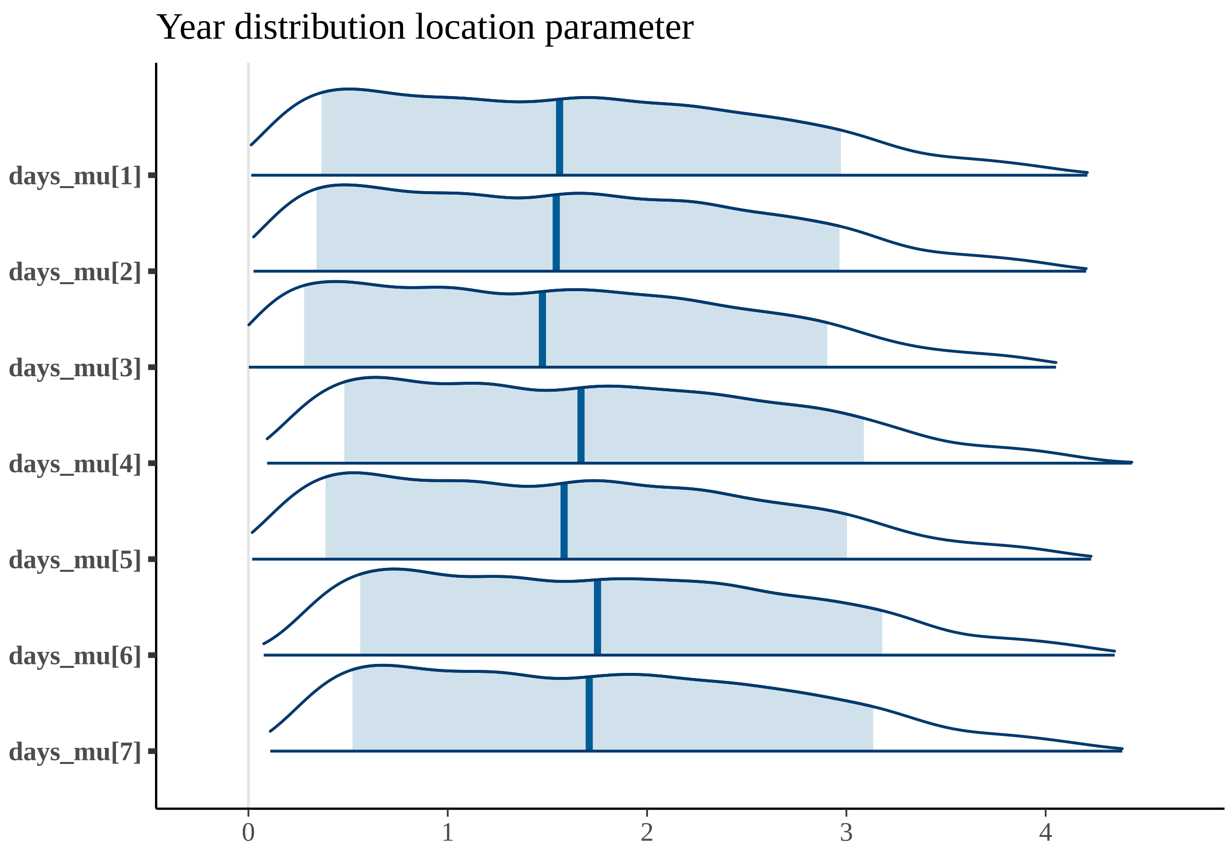
```
## Warning: Dropping 'draws_df' class as required metadata was removed.
```



There seems to be no large difference between 2019 and 2020 in terms of the scores.

```
mcmc_areas(hierarchical_draws %>%
             select(contains("days_mu")) %>%
             as.matrix,
           prob = 0.8) +
  ggtitle("Year distribution location parameter")
```

```
## Warning: Dropping 'draws_df' class as required metadata was removed.
```



This plot suggests that Mondays, Fridays and Sundays could be a bit better, but not by a large margin.

Now, let's compute the PSIS LOO estimates for this model

```
llfun_hierarchical <- function(data_i, draws, log = TRUE) {
  year_mu <- glue("year_mu[{data_i$year}]")
  year_phi <- glue("year_phi[{data_i$year}]")
  days_mu <- glue("days_mu[{data_i$day}]")
  days_phi <- glue("days_phi[{data_i$day}]")
  hours_mu <- glue("hours_mu[{data_i$hour}]")
  hours_phi <- glue("hours_phi[{data_i$hour}]")

  #print(dim(draws))
  #print(draws[, c(year_mu, days_mu, hours_mu)])
  mu_sum <- rowSums(
    as.matrix(draws[, c(year_mu, days_mu, hours_mu)])
  )

  phi_sum <- rowSums(
    as.matrix(draws[, c(year_phi, days_phi, hours_phi)])
  )

  dnbinom(data_i$score, mu = mu_sum, size = phi_sum, log = log)
}

hierarchical_params <- as_draws_matrix(hierarchical_draws)
```

```r
r_eff <- relative_eff(llfun_hierarchical,
                      log = FALSE, # relative_eff wants likelihood not log-likelihood values
                      chain_id = rep(1:4, each = 1000),
                      data = as.data.frame(hierarchical_data),
                      draws = hierarchical_params,
                      cores = 25)

loo_hierarchical <- loo_subsample(llfun_hierarchical,
                          data = as.data.frame(hierarchical_data),
                          draws = hierarchical_params,
                          r_eff = r_eff,
                          observations = 500,
                          cores = 25)
```

```
## [1] 5000    8
## [1]  1 67
## [1] 25
```

```r
loo_hierarchical
```

```
##
## Computed from 4000 by 500 subsampled log-likelihood
## values from 5000 total observations.
##
##          Estimate     SE subsampling SE
## elpd_loo -75730.0 1498.7         1771.0
## p_loo     10790.1  505.3         1515.8
## looic    151460.0 2997.5         3541.9
## ------
## Monte Carlo SE of elpd_loo is NA.
##
## Pareto k diagnostic values:
##                          Count Pct.    Min. n_eff
## (-Inf, 0.5]   (good)     400   80.0%   116
##  (0.5, 0.7]   (ok)        29    5.8%   58
##    (0.7, 1]   (bad)       33    6.6%   9
##    (1, Inf)   (very bad)  38    7.6%   1
## See help('pareto-k-diagnostic') for details.
```

We got much lower ELPD this time, but we need to consider two poits:

1. Some of our $\hat{k}$ values are much higher this time, indicating unreliable ELPD estimate
2. We use a small subsample of the data instead of the full dataset which makes our estimate overly optimistic

Variational inference

To run the hierarchical model on full dataset we will need to use variational inference. Stan implements an anglorithm called AVDI that allows much faster model fits. However, this price comes with the accuracy penalty as we will be using posterior approximations instead of direct posterior samples compared to the MCMC NUTS sampler.

```r
# re-compile the model to use OpenCL for faster computation on full data
hierarchical_model <- cmdstan_model('./hierarchical-model.stan',
                                   quiet = FALSE,
                                   force_recompile = TRUE,
                                   cpp_options = list(stan_opencl = TRUE))
```

```
##
## --- Translating Stan model to C++ code ---
## bin/stanc --use-opencl --name='hierarchical-model_model' --o=/tmp/RtmpHBXgZS/model-1ccd76d5f454.hpp /
##
## --- Compiling, linking C++ code ---
## g++ -fpermissive -std=c++1y -pthread -D_REENTRANT -Wno-sign-compare -Wno-ignored-attributes   -I stan
## g++ -fpermissive -std=c++1y -pthread -D_REENTRANT -Wno-sign-compare -Wno-ignored-attributes   -I stan
## rm -f /tmp/RtmpHBXgZS/model-1ccd76d5f454.o
```

```r
# remove data filtering, we will look at all years now and without random subset sampling
hn_posts_vb <- hn_posts %>% mutate(score = score - 1)

hierarchical_data_vb <- prepare_dataset(hn_posts_vb)

# run ADVI
hierarchical_fit_vb <- hierarchical_model$variational(data = hierarchical_data_vb,
                                                      output_samples = 10000,
                                                      algorithm = "fullrank",
                                                      seed = SEED)
```

```
## ------------------------------------------------------------
## EXPERIMENTAL ALGORITHM:
##   This procedure has not been thoroughly tested and may be unstable
##   or buggy. The interface is subject to change.
## ------------------------------------------------------------
## Gradient evaluation took 0.092634 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 926.34 seconds.
## Adjust your expectations accordingly!
## Begin eta adaptation.
## Iteration:   1 / 250 [  0%]  (Adaptation)
## Iteration:  50 / 250 [ 20%]  (Adaptation)
## Iteration: 100 / 250 [ 40%]  (Adaptation)
## Iteration: 150 / 250 [ 60%]  (Adaptation)
## Iteration: 200 / 250 [ 80%]  (Adaptation)
## Iteration: 250 / 250 [100%]  (Adaptation)
## Success! Found best value [eta = 0.1].
## Begin stochastic gradient ascent.
##   iter             ELBO   delta_ELBO_mean   delta_ELBO_med   notes
##    100     -4350249.768             1.000            1.000
##    200     -2304001.978             0.944            1.000
##    300     -1573641.188             0.784            0.888
##    400     -1427771.974             0.614            0.888
##    500     -1344625.883             0.503            0.464
##    600     -1319183.912             0.423            0.464
##    700     -1304500.634             0.364            0.102
##    800     -1297964.703             0.319            0.102
##    900     -1292661.660             0.284            0.062
##   1000     -1290059.621             0.256            0.062
##   1100     -1289702.146             0.156            0.019
##   1200     -1288388.377             0.067            0.011
##   1300     -1287284.749             0.021            0.005   MEDIAN ELBO CONVERGED
## Drawing a sample of size 10000 from the approximate posterior...
## COMPLETED.
## Finished in  267.5 seconds.
```

Now, let's run PSIS LOO on the variational approximation. `loo_subsample` uses special approach to compute LOO for posterior approximations. It uses `log_p` (log probability) and `log_g` (approximated log probability) parameters and a different algorithm for computing PSIS LOO over a subsample under the hood.

When I was testing this function I have noticed that it executes really slow. Surprisingly, this lead to which I consider to be **the main result of this project**: a pull request to the `loo` package that removes computational bottleneck inside `loo_subsample` to allow much faster parallel computation of the log likelihood. You can see the changes that I have suggested here: https://github.com/stan-dev/loo/pull/171.

```
hierarchical_draws_vb <- hierarchical_fit_vb$draws()
hierarchical_draws_vb <- as_draws_df(hierarchical_draws_vb)
```

```
log_p <- hierarchical_fit_vb$lp()
log_g <- hierarchical_fit_vb$lp_approx()

hierarchical_params_vb <- as_draws_matrix(hierarchical_draws_vb)

loo_hierarchical_vb <- loo_subsample(llfun_hierarchical,
                          data = as.data.frame(hierarchical_data_vb),
                          draws = hierarchical_params_vb,
                          log_p = log_p,
                          log_g = log_g,
                          observations = 3000 ,
                          cores = 25)
```

```
## [1] 239417      8
## [1]  1 76
## [1] 25
```

```
loo_hierarchical_vb
```

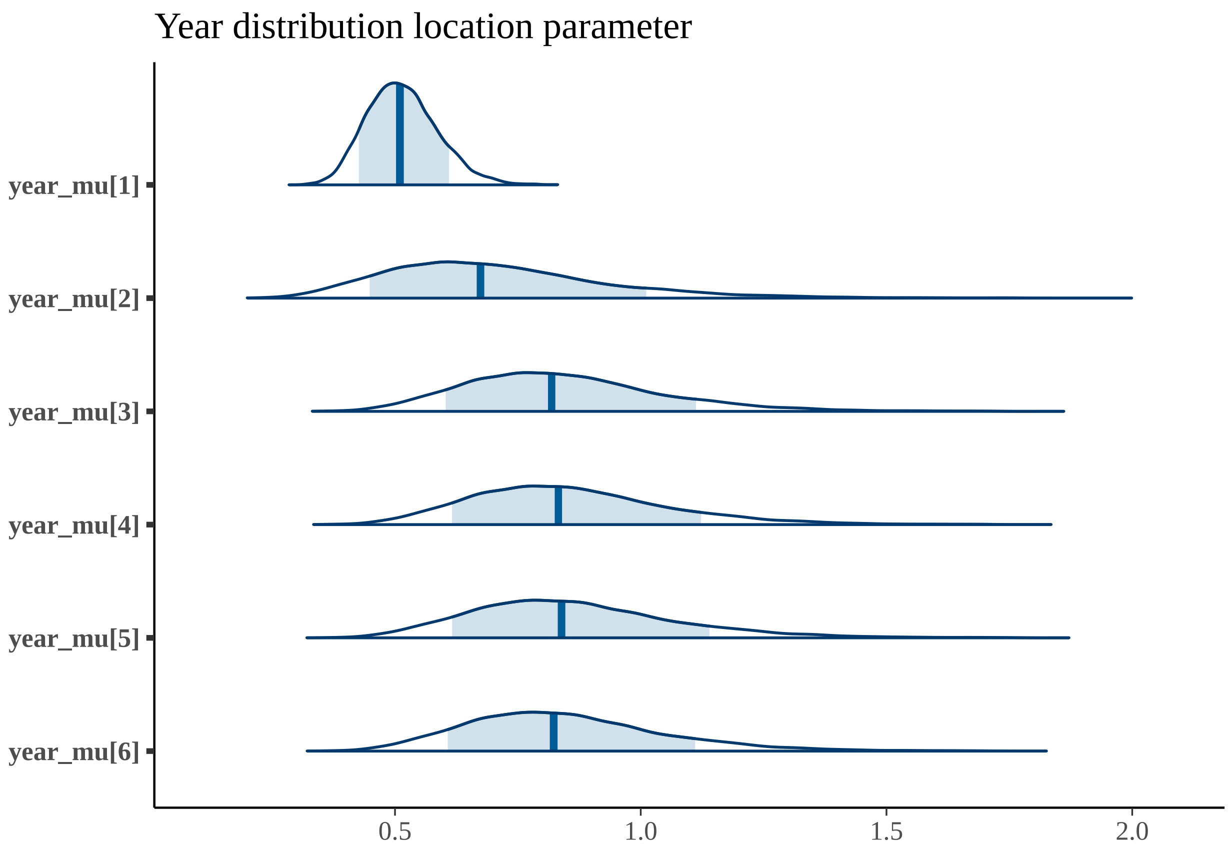
```
##
## Computed from 10000 by 3000 subsampled log-likelihood
## values from 239417 total observations.
##
##            Estimate      SE subsampling SE
## elpd_loo -3214669.8  8025.8        22401.2
## p_loo      113470.8   980.3         8702.4
## looic     6429339.5 16051.6        44802.3
## ------
## Posterior approximation correction used.
## Monte Carlo SE of elpd_loo is NA.
##
## Pareto k diagnostic values:
```

```
##                           Count Pct.    Min. n_eff
## (-Inf, 0.5]   (good)         0    0.0%  <NA>
##  (0.5, 0.7]   (ok)           0    0.0%  <NA>
##    (0.7, 1]   (bad)          0    0.0%  <NA>
##    (1, Inf)   (very bad) 3000  100.0%  4
## See help('pareto-k-diagnostic') for details.
```

The ELPD estimate we got is roughy equivalent to the ELPD of the simple model. Unfortunately, all of the $\hat{k}$ values are larger than 1, which indicates that the PSIS LOO estimate is unreliable. Variational approximations tend to be fast, but inaccurate. Let's explore parameter plots to see how they've changed when model is using the full dataset.

```
mcmc_areas(hierarchical_draws_vb %>%
             select(contains("year_mu")) %>%
             as.matrix,
           prob = 0.8) +
  ggtitle("Year distribution location parameter")
```

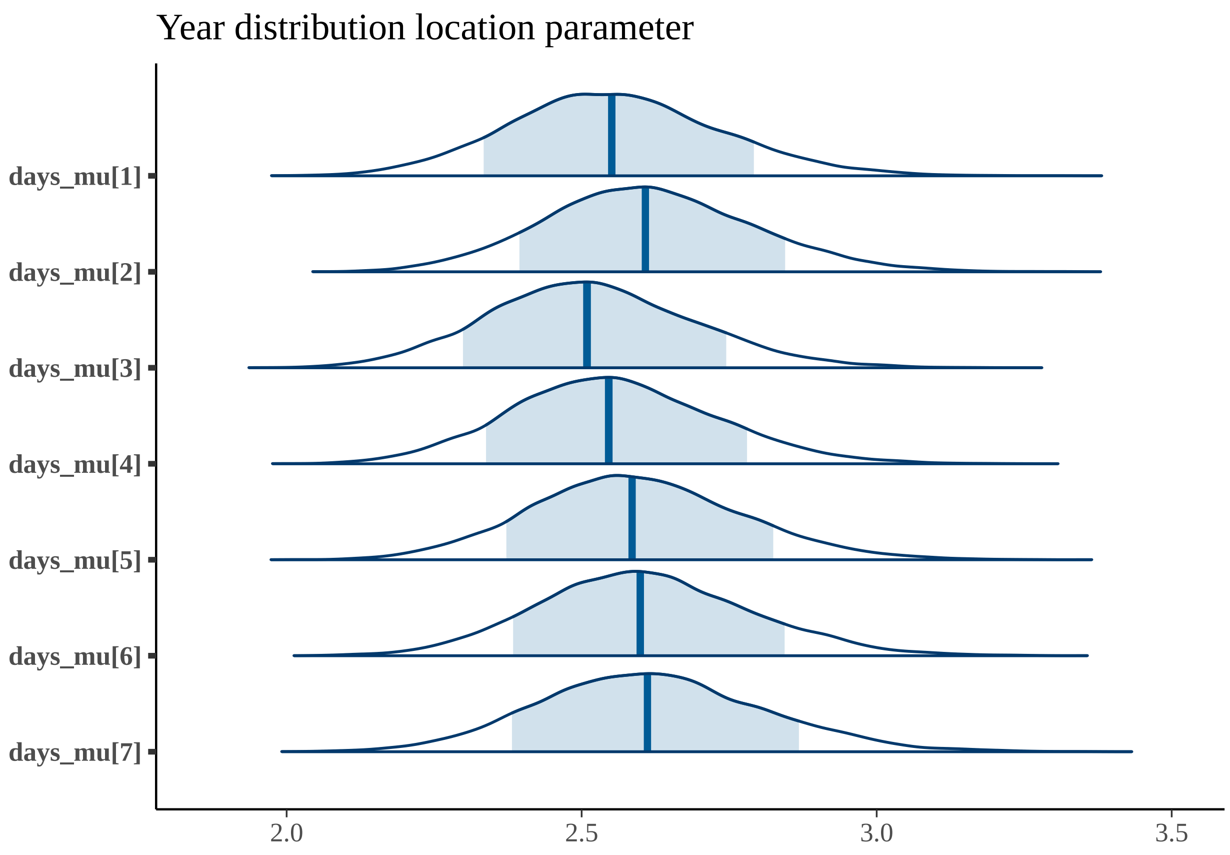
```
## Warning: Dropping 'draws_df' class as required metadata was removed.
```



From this plot we see, that if the model is correct, years do not affect score distribution significantly, apart from one year, which has a lower estimate.

```
mcmc_areas(hierarchical_draws_vb %>%
             select(contains("days_mu")) %>%
             as.matrix,
           prob = 0.8) +
  ggtitle("Year distribution location parameter")
```

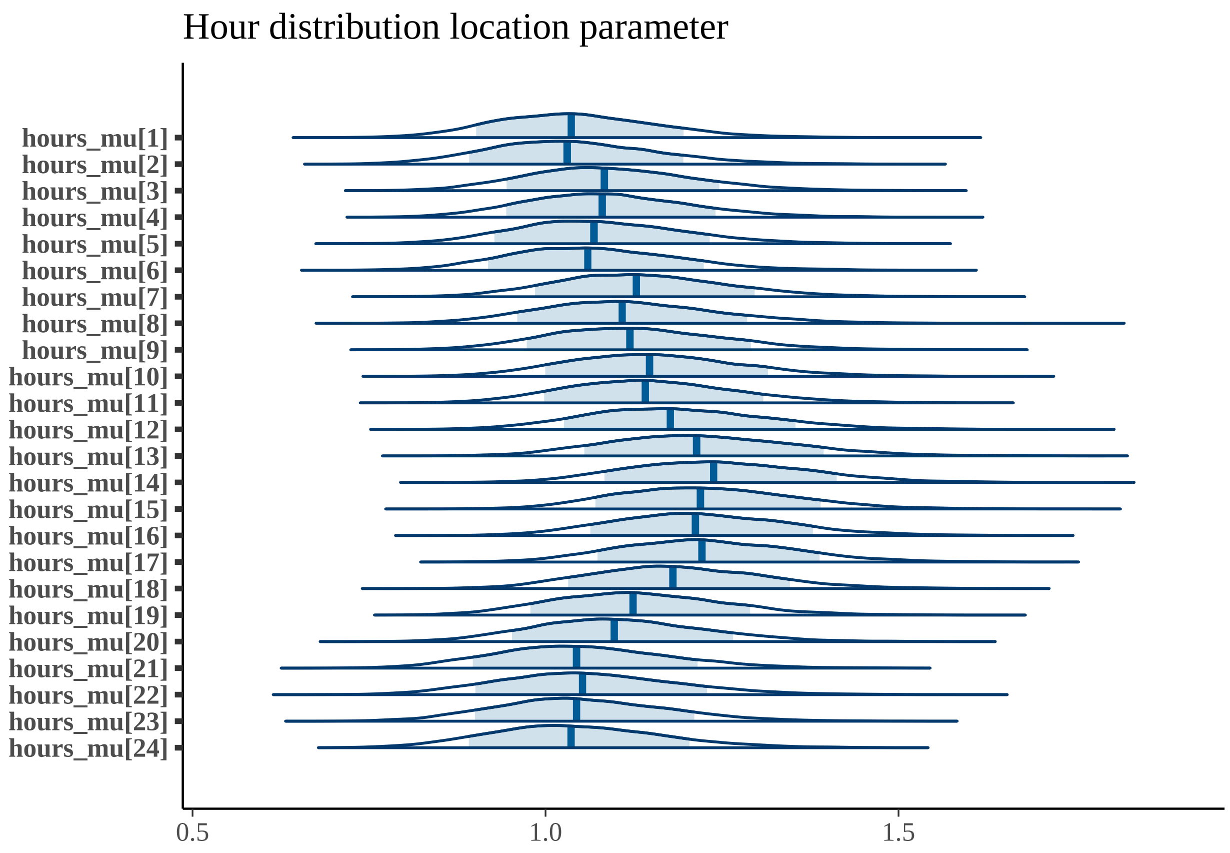
```
## Warning: Dropping 'draws_df' class as required metadata was removed.
```



Here, we see that midweek distributions have slightly lower estimates. Tuesday seems to be a good day to post.

```
mcmc_areas(hierarchical_draws_vb %>%
             select(contains("hours_mu")) %>%
             as.matrix,
           prob = 0.8) +
  ggtitle("Hour distribution location parameter")
```

```
## Warning: Dropping 'draws_df' class as required metadata was removed.
```



As in the previous models, posting in daytime indicates better response than posting at night or early mornings.

Conclusion

Technical part

In this project, we have looked at how to run Stan models on large datasets. Tools that the Stan ecosystem provides for this are:

1. OpenCL support which allows to run likelihood computations on GPUs
2. LOO subsampling that allows to scale PSIS LOO on large datasets using parallel computation over likelihood functions defined in R
3. Variational inference support that allows for much faster but less accurate posterior approximations

As a result of this project I suggested an improvement for the `loo` package that you can see here: https://github.com/stan-dev/loo/pull/171.

Modelling part

We have looked over 3 models:

1. Simple Negative binomial model that was fit on the full dataset using NUTS

2. Hierarchical model that included time-related information that was fit on the small subsample of the dataset using NUTS
3. Hierarchical model that included time-related information that was fit on the full dataset using ADVI

Models suggest that the best time to post on HackerNews is during daytime on Mondays, Fridays and Saturdays, but all models clearly need an improvement and additional data to give more reliable parameter estimates.

Further directions

This project is fairly technically focused because I spent a lot of time working with the `loo` package source code instead of modelling, so there is a lot left on the modelling side:

1. Explore mixture models to improve model quality
2. Add additional features to the model. I have a code that does topic modeling over this dataset which can be used to generate additional features that should improve resulting metrics
3. Prior analysis. This should not be as significant since we have a lot of data and low prior sensitivity as a result