2.3 Practical: posterior

Benjamin Rosenbaum

October 25, 2022

In this session, we will learn how to fit a model and to interpret the output.

Specifically, we learn how to deal with the posterior distribution to make inference and predictions.

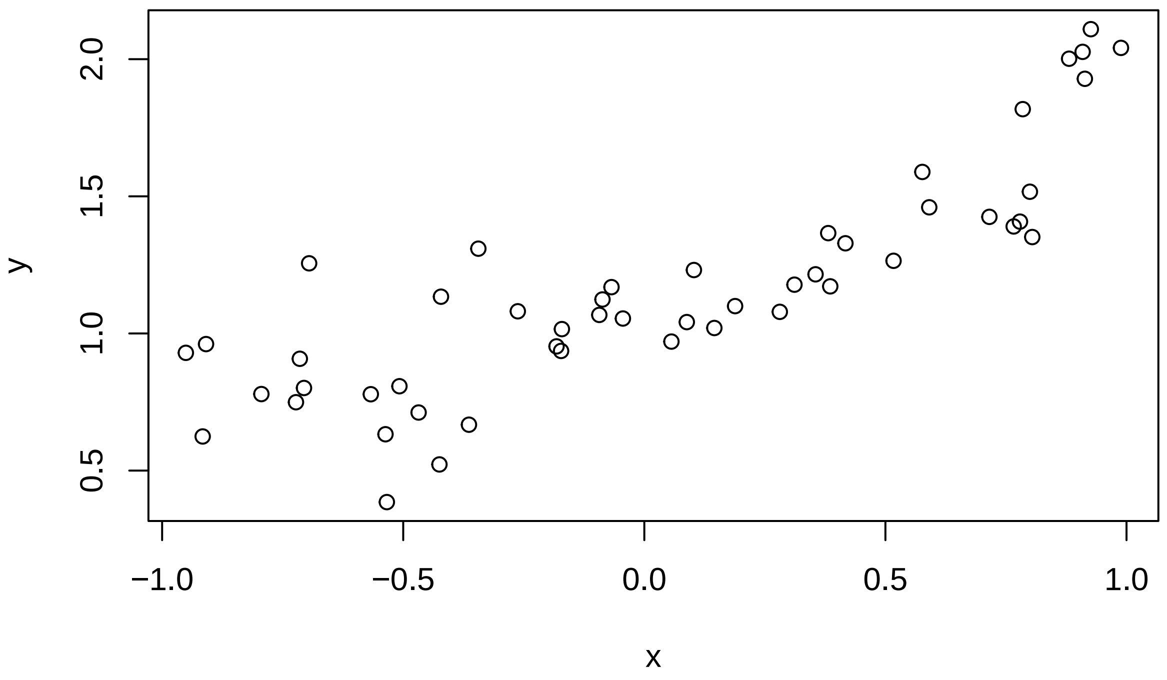
Again, we will use linear regression.

Setup

```r
rm(list=ls())
library(rstan)
library(coda)
library(BayesianTools)

rstan_options(auto_write = TRUE)
options(mc.cores = 4) # number of CPU cores
```

Generate data

```r
set.seed(123) # initiate random number generator for reproducability

n=50

a=1.0
b=0.5
c=0.4
sigma=0.2

x = runif(n=n, min=-1, max=1)
y = rnorm(n=n, mean=a+b*x+c*x^2, sd=sigma)

df = data.frame(x=x,
                y=y)

plot(df)
```



Statistical model

Statistical model for linear regression

$$y_i \sim \text{normal}(\mu_i, \sigma)$$
$$\mu_i = a + b \cdot x_i$$

```
stan_code = '
data {
  int n;
  vector[n] x;
  vector[n] y;
}
parameters {
  real a;
  real b;
  real<lower=0> sigma;
}
model {
  // priors
  a ~ normal(0, 10);
  b ~ normal(0, 10);
  sigma ~ normal(0, 10);
  // likelihood
  y ~ normal(a+b*x, sigma);
}
'
```

Data and sampler preparation, MCMC sampling

```r
data = list(n=n,
            x=df$x,
            y=df$y)


stan_model = stan_model(model_code=stan_code)
# save(file="stan_model_test.RData", list="stan_model")
# load("stan_model_test.RData")

fit  = sampling(stan_model,
                data=data)
```

Explore the posterior distribution

First, we look at the output and check `n_eff` and `Rhat`.

`Rhat<1.01` for all parameters, that's good. `n_eff` looks good, too (compare to `n_total`).

```r
print(fit, digits=3, probs=c(0.025, 0.975))
```

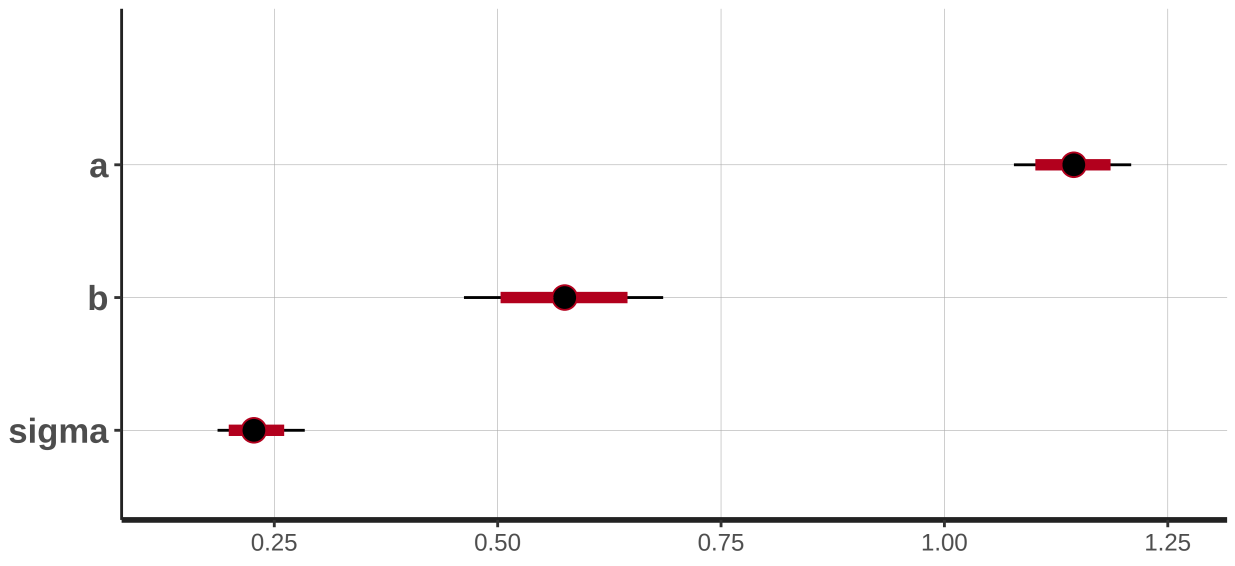
```
## Inference for Stan model: 8ad33c5d8a4bcdd6e619c1be283d2f6e.
## 4 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=4000.
##
##         mean se_mean    sd   2.5%  97.5% n_eff  Rhat
## a      1.144   0.001 0.033  1.078  1.209  3509 1.000
## b      0.575   0.001 0.056  0.462  0.685  3780 1.000
## sigma  0.229   0.000 0.025  0.187  0.284  3555 1.002
## lp__  48.043   0.032 1.292 44.603 49.508  1661 1.002
##
## Samples were drawn using NUTS(diag_e) at Tue Oct 18 13:23:42 2022.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

Plot the standard Stan output.

```r
plot(fit)
```

```
## ci_level: 0.8 (80% intervals)
```

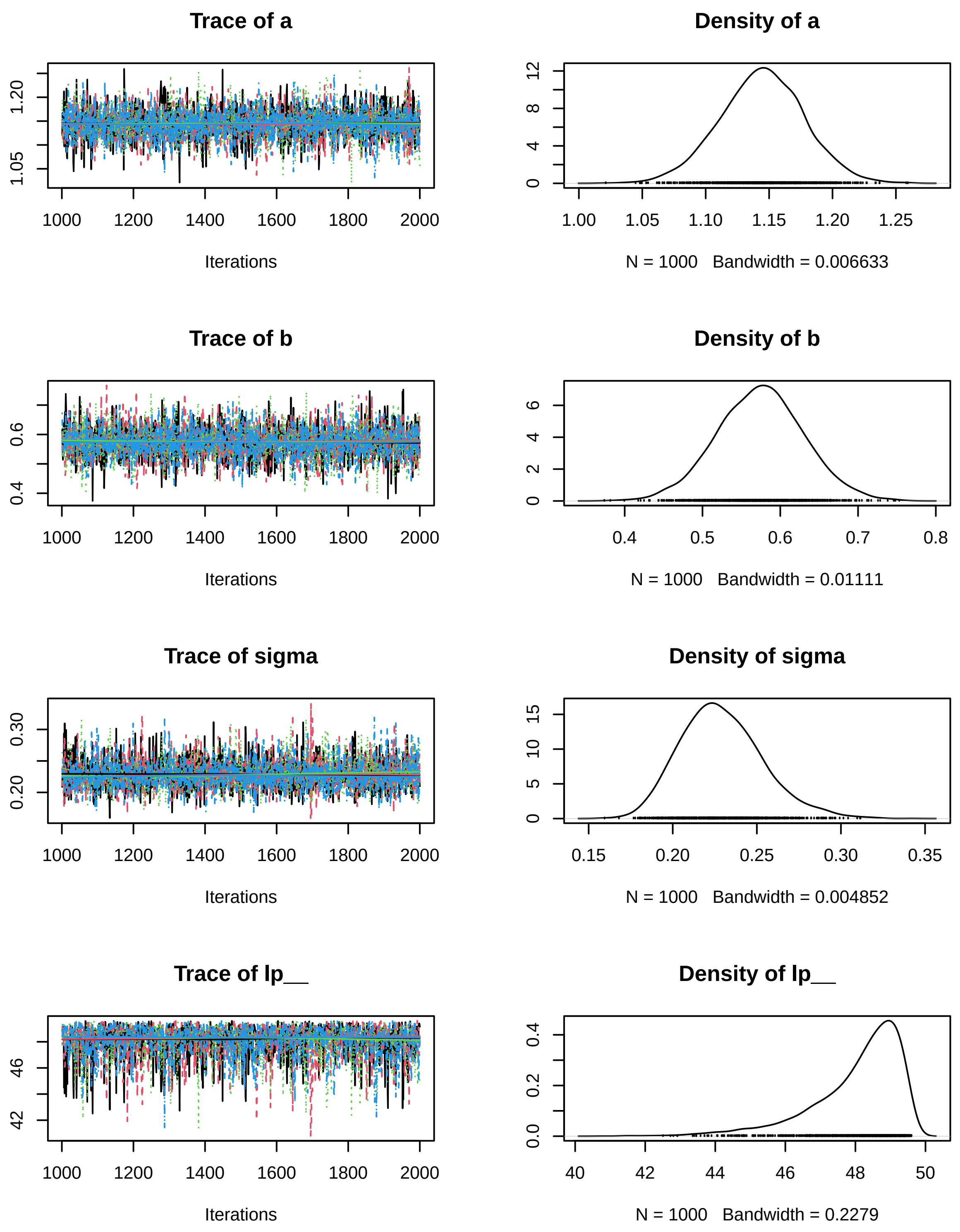
```
## outer_level: 0.95 (95% intervals)
```



Plotting from coda package (covert fit object to `mcmc.list` object) also shows traceplots of the 3 chains.

They look like a **"fat hairy caterpillar"**, so we assume the chains are a good representation of the true posterior distribution.

```
plot(As.mcmc.list(fit)) # from coda package
```



What is the posterior exactly?

Convert fit object into a **matrix**. Now all chains (here: 4) are concatenated to 1.

It has `n_total=4000` rows and 1 column per parameter (3 parameters + “lp___“, we ignore the last one)

```
posterior=as.matrix(fit)
str(posterior)
```

```
##  num [1:4000, 1:4] 1.12 1.17 1.15 1.14 1.15 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ iterations: NULL
##   ..$ parameters: chr [1:4] "a" "b" "sigma" "lp__"
```

```
head(posterior)
```

```
##           parameters
## iterations        a         b     sigma     lp__
##       [1,] 1.120521 0.6173368 0.2086787 48.84575
##       [2,] 1.167061 0.5480164 0.2382805 49.01637
##       [3,] 1.154086 0.5136227 0.2215158 48.92314
##       [4,] 1.135057 0.5351571 0.2334256 49.14544
##       [5,] 1.149297 0.6386438 0.2216895 48.90987
##       [6,] 1.212371 0.6337799 0.2938154 44.48180
```

Each **column** contains all posterior samples of 1 parameter.

We can look at this (marginal) posterior distribution. This is the same as the plotting commands above.

We can index the columns by number or by name.

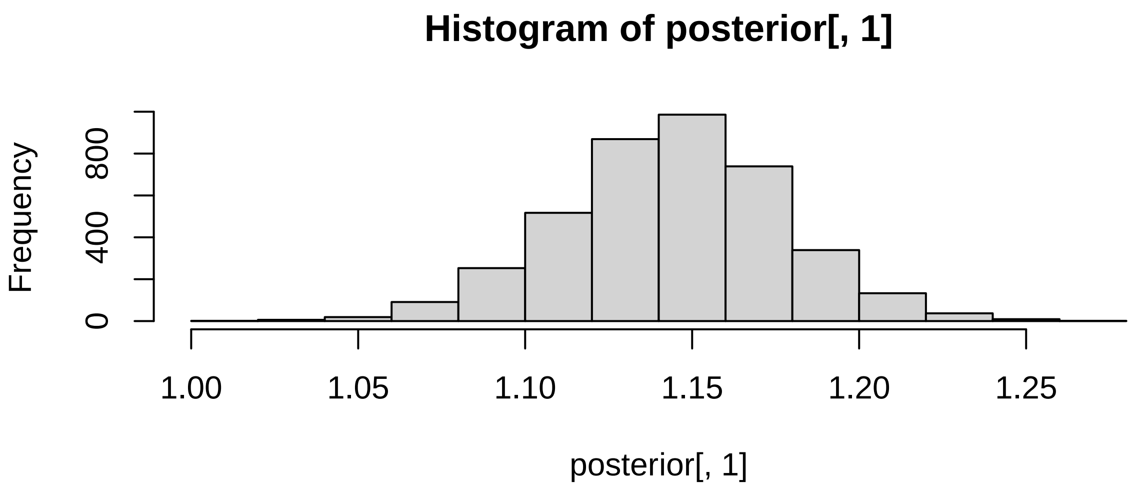
```
str(posterior[, 1])
```

```
##  num [1:4000] 1.12 1.17 1.15 1.14 1.15 ...
```

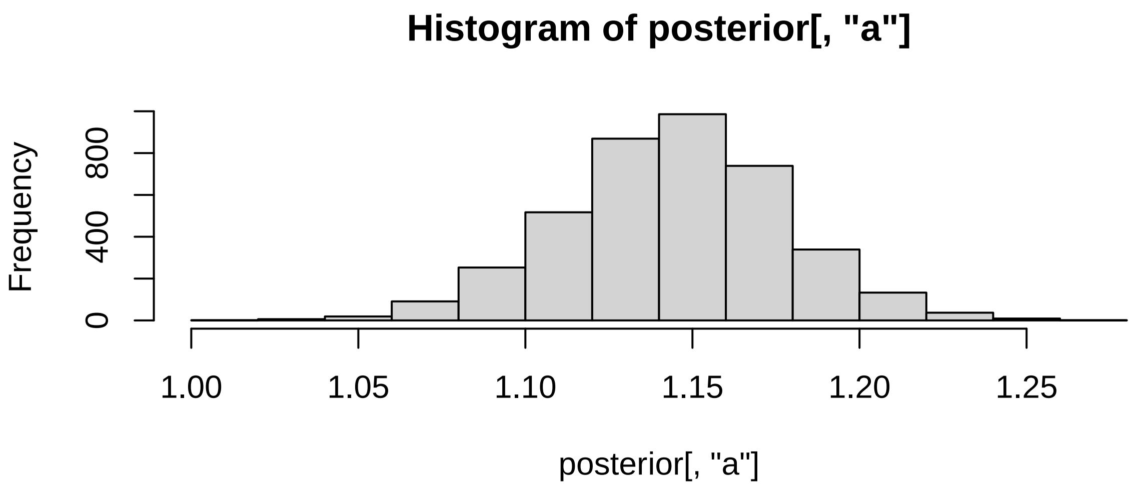
```
head(posterior[, 1])
```

```
## [1] 1.120521 1.167061 1.154086 1.135057 1.149297 1.212371
```

```
hist(posterior[, 1])
```



```
hist(posterior[, "a"])
```



Each **row** contains one sample of the multidimensional posterior.

Important: each draw / sample consists of a multidimensional vector for `a`,`b`,`sigma`.

If you change the order of one column (permutation), the whole thing is not a representation of the posterior anymore!

Each element of the matrix is linked to the other elements in that row!

```
str(posterior[1, ])
```

```
##  Named num [1:4] 1.121 0.617 0.209 48.846
##  - attr(*, "names")= chr [1:4] "a" "b" "sigma" "lp__"
```

```
posterior[1, ]
```

```
##          a          b      sigma       lp__
##  1.1205208  0.6173368  0.2086787 48.8457455
```

The reason is that the parameters can be correlated.

Typically, there is some correlation between intercept and slope in linear regression (especially if the data is not centered).

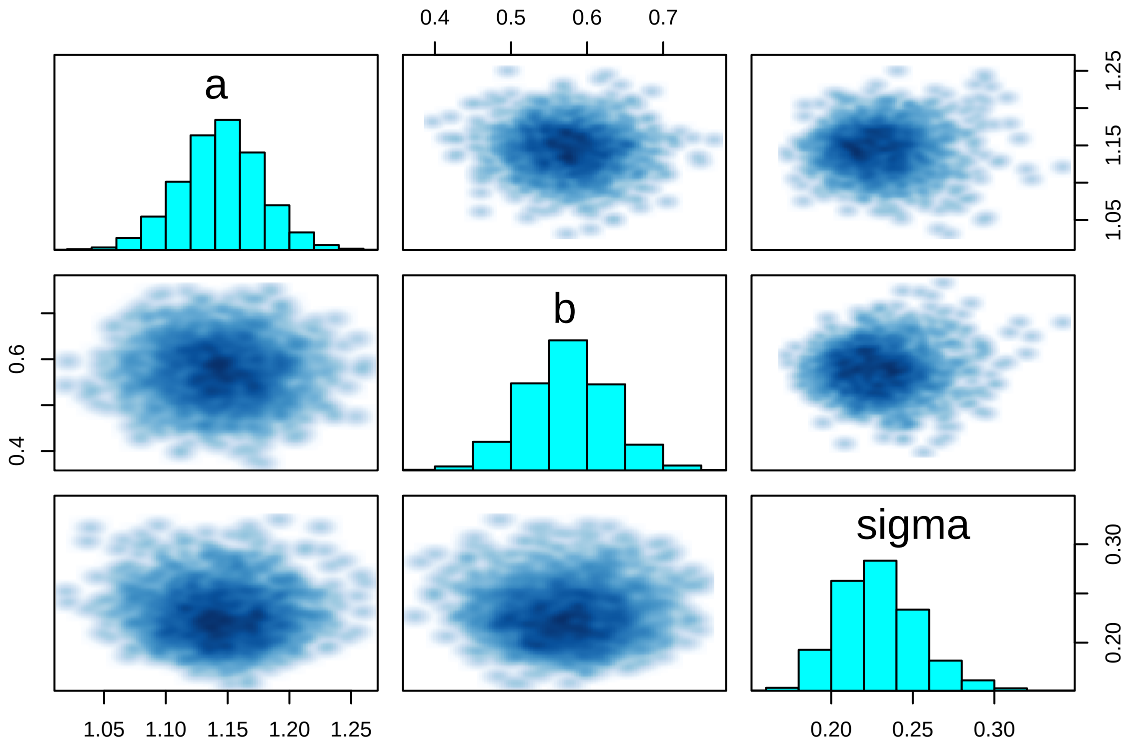
`correlationPlot()` shows pairwise plots and correlation coefficients.

Some correlation is generally not a problem in Bayesian statistics!

Perfect correlation (samples perfectly distributed along a line or a curve), however, would indicate some problem with the model (unidentifiability).

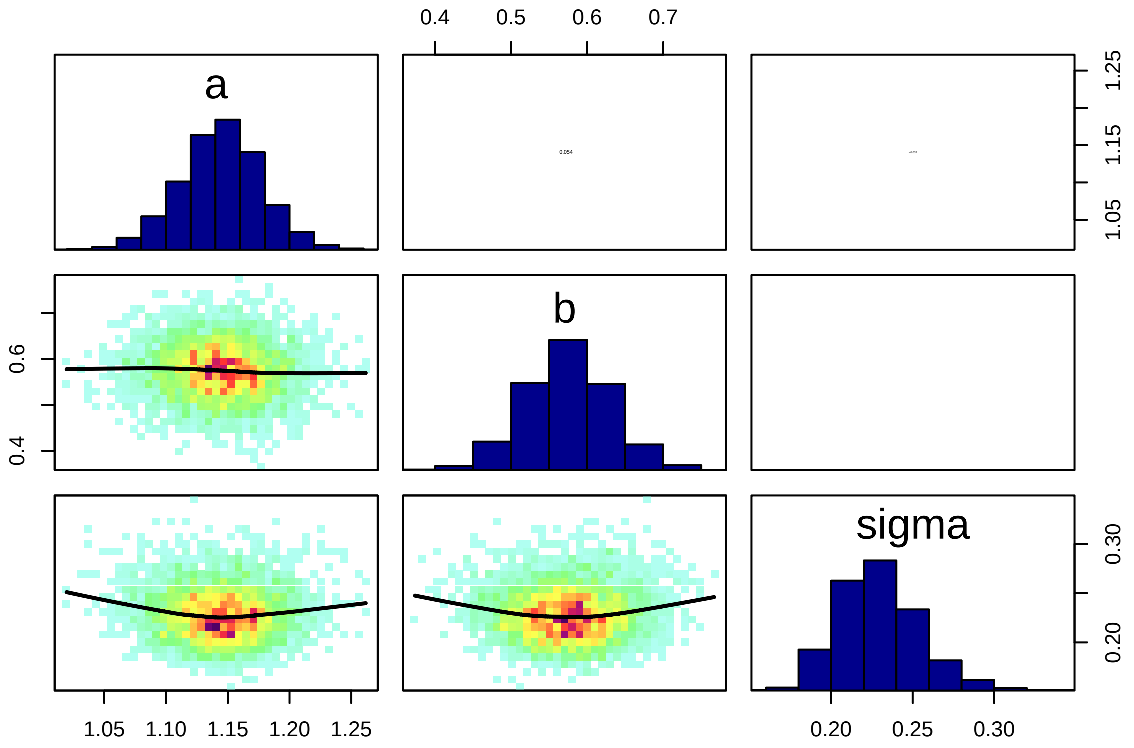
```
pairs(fit, pars=c("a","b","sigma"))
```

```
## Warning in par(usr): argument 1 does not name a graphical parameter

## Warning in par(usr): argument 1 does not name a graphical parameter

## Warning in par(usr): argument 1 does not name a graphical parameter
```



```
correlationPlot(posterior[, 1:3], thin=1) # from BayesianTools package
```

```
## Warning in par(usr): argument 1 does not name a graphical parameter

## Warning in par(usr): argument 1 does not name a graphical parameter

## Warning in par(usr): argument 1 does not name a graphical parameter

## Warning in par(usr): argument 1 does not name a graphical parameter

## Warning in par(usr): argument 1 does not name a graphical parameter

## Warning in par(usr): argument 1 does not name a graphical parameter
```



Now it's time for some **inference**!

We want to test if there is a positive effect of predictor x on response y.

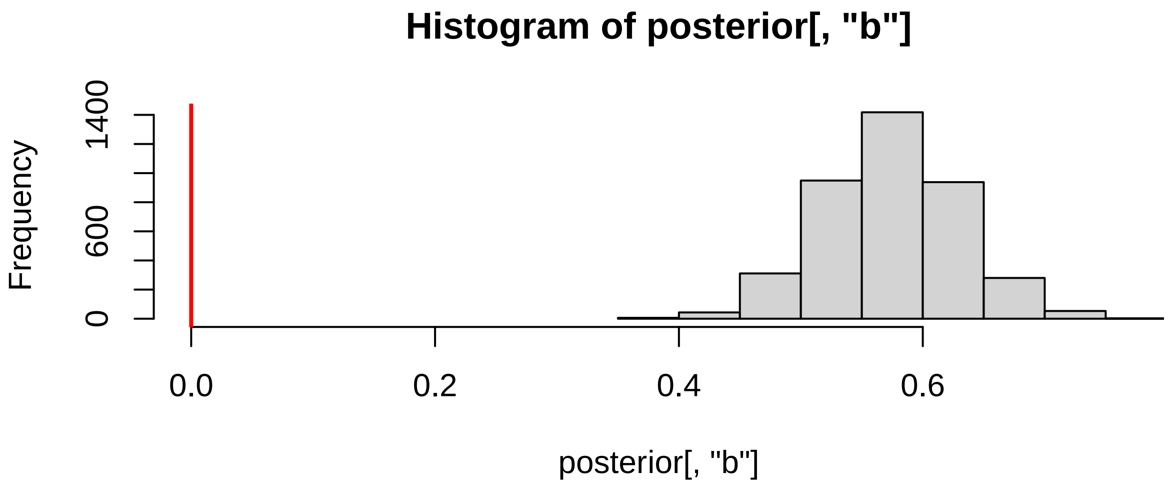
Posterior samples of slope b represent posterior distribution of b given the data $p(b|y)$.

So we can actually compute the posterior probability of a positive effect given the data $P(b > 0|y)$.

How to do that? Just count the number the event $(b > 0)$ occurs in the posterior samples, divide by total number of samples and that's the probability!

Given the data, we are 100% sure that the effect is positive.

```
hist(posterior[, "b"], xlim=c(0,max(posterior[, "b"])))
abline(v=0, col="red", lwd=2)
```



```
sum(posterior[, "b"]>0)/nrow(posterior)
```

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

Similarly, we can compute the probability of the effect being larger than 0.7 or effect being in the interval [0.4, 0.6].

```
sum(posterior[, "b"]>0.7)/nrow(posterior)
```

```
## [1] 0.01375
```

```
sum( (posterior[, "b"]>0.4) & (posterior[, "b"]<0.6) )/nrow(posterior)
```

```
## [1] 0.68025
```

Posterior predictions

Usually, it is **not** sufficient just to check if the MCMC sampler converged.

That doesn't tell us anything about if our statistical model (deterministic part and stochastic part) describes the data adequately!

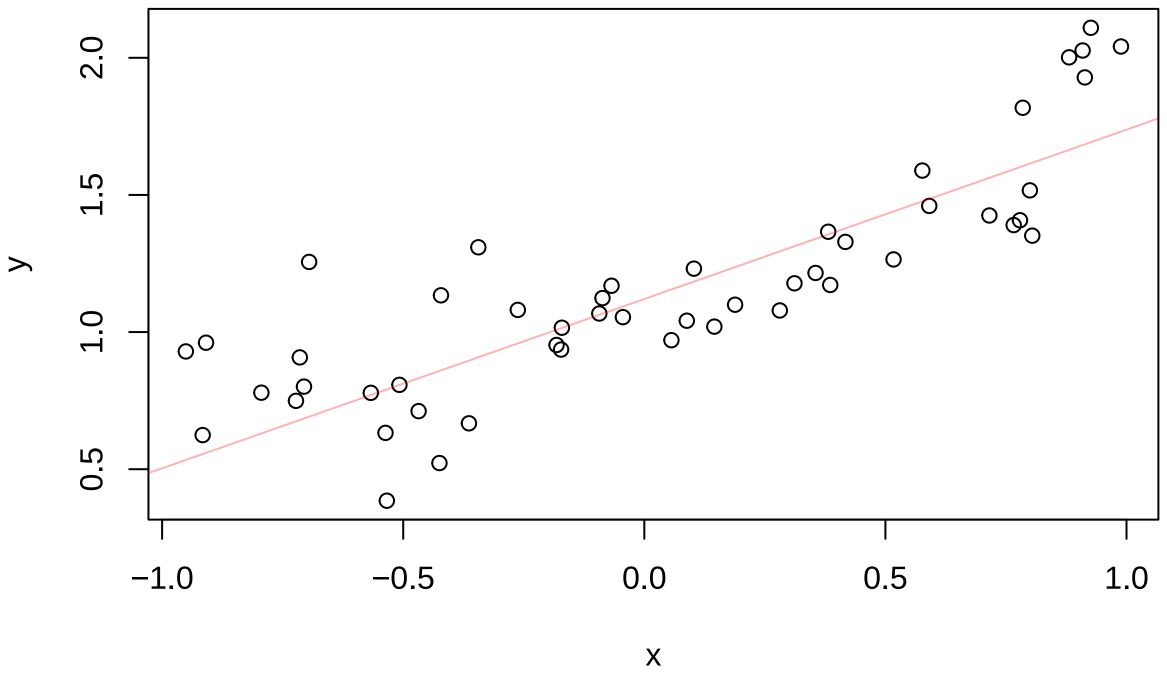
For that, we have to compare observed and predicted values.

Each row of the posterior matrix contains a sample of the posterior, i.e. intercept a and slope b.

We can evaluate or plot the deterministic model (regression line $a + b \cdot x$) using these parameters.

E.g. plot the deterministic model for the first sample using the `abline()` command.

```
plot(df)
abline(posterior[1,"a"], posterior[1,"b"], col=adjustcolor("red", alpha.f=0.3))
```



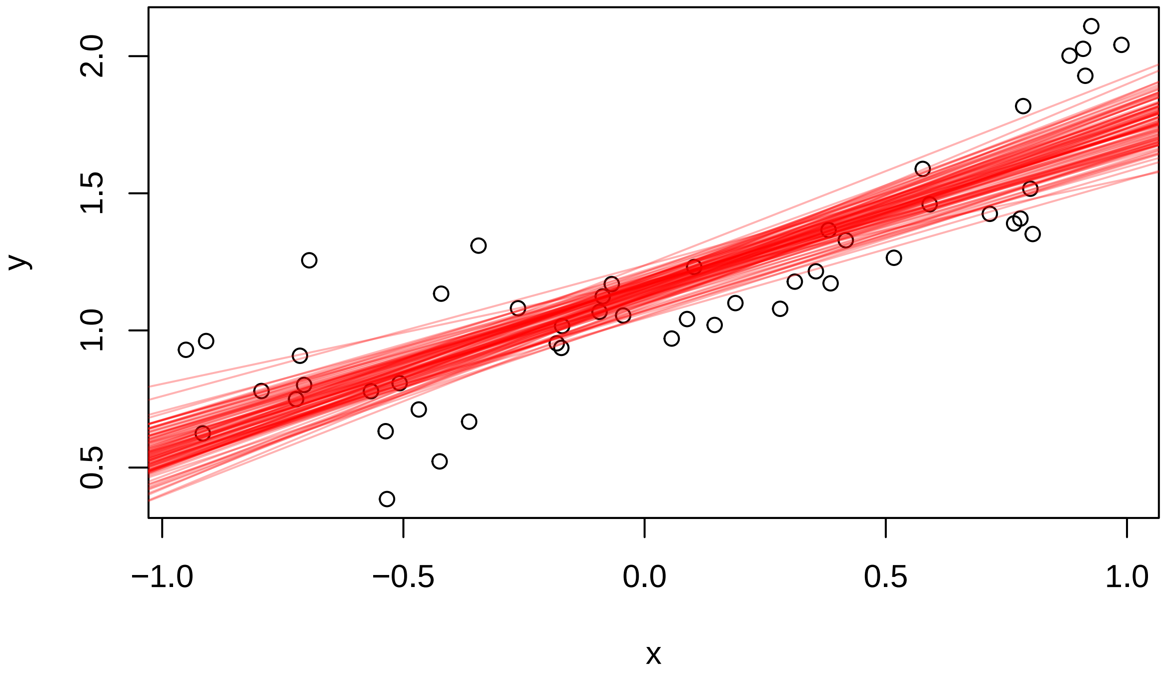
Or plot the deterministic model for the first 100 samples.

We can see the uncertainty associated with the predictions.

Remember that each row is a sample, i.e. we have to use intercept `a_i` and slope `b_i`.

Never mix up the order, `a_i` with slope `b_j`!

```r
plot(df)
for(i in 1:100){
  abline(posterior[i,"a"], posterior[i,"b"], col=adjustcolor("red", alpha.f=0.3))
}
```



`abline()` is a fancy command for plotting lines, but if we want a more generalized approach (more complex models later), we have to code the deterministic model ourself.

`x.pred` is a vector of predictor values for which we want to make predictions.

`y.cred` is a matrix that will contain all predictions.

We call it "cred" because we will use it for computing **"credible intervals"** / "confidence intervals" of the **deterministic model part**.

See below for "prediction intervals"

In Bayesian statistics, everything is a distribution. So also the predictions will be a distribution.

There are 3000 samples in the posterior, i.e. 3000 parameter combinations of intercept and slope.

This means we can make 3000 predictions and these will be samples from a posterior predictive distribution.

```r
x.pred = seq(from=-1, to=1, by=0.1)
y.cred = matrix(0, nrow=nrow(posterior), ncol=length(x.pred))

for(i in 1:nrow(posterior)){
  y.cred[i, ] = posterior[i,"a"] + posterior[i,"b"]*x.pred
}
```

The element `y.cred[i,j]` (ith row, jth column) is the prediction for MCMC sample `i` (using parameters `a_i`,`b_i`) for predictor value `x.pred[j]`.

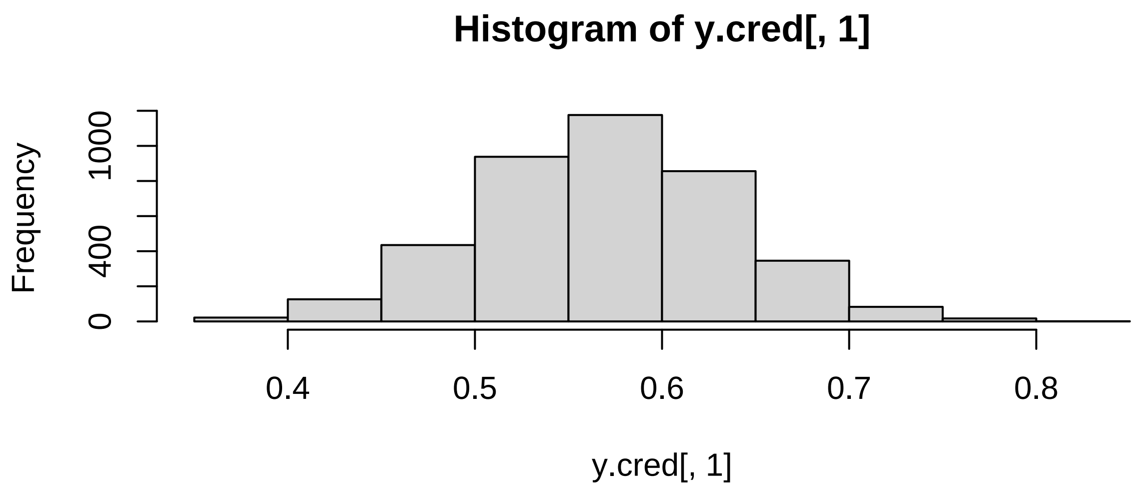
Each row `i` contains predictions for all `x.pred` using single parameter set `a_i`,`b_i`.

Each column `j` contains 3000 predictions (for all posterior samples) for one predictor value `x.pred[j]`.

```r
head(y.cred)
```

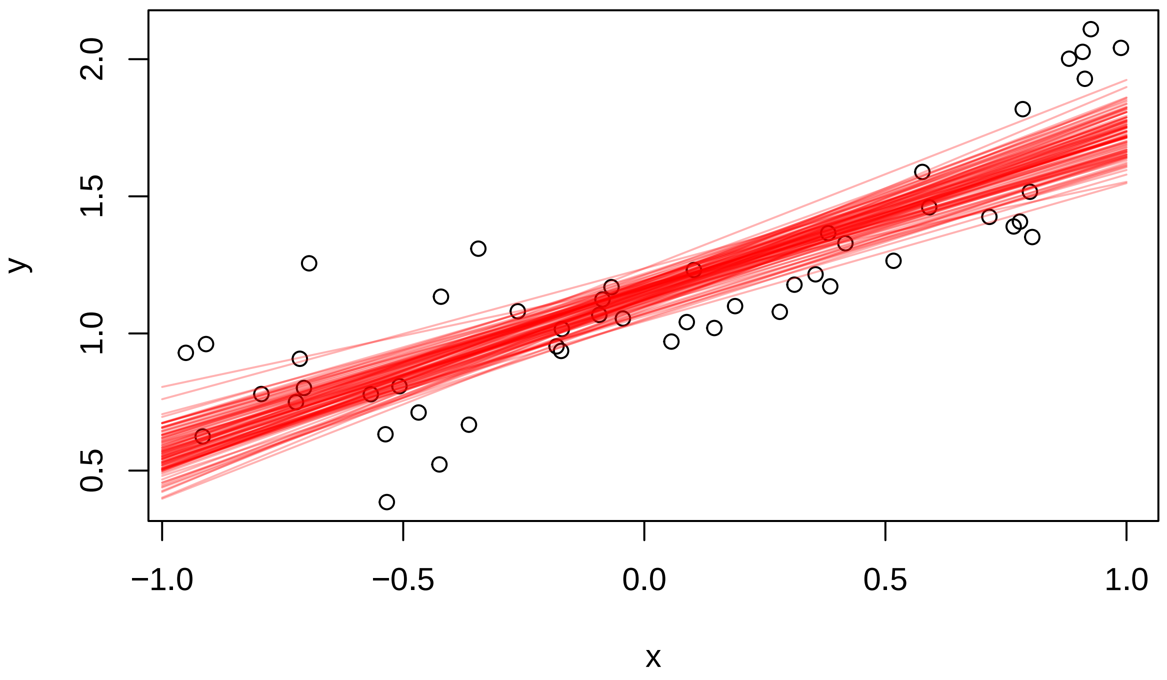
```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 0.5031840 0.5649177 0.6266514 0.6883850 0.7501187 0.8118524 0.8735861
## [2,] 0.6190443 0.6738459 0.7286476 0.7834492 0.8382509 0.8930525 0.9478541
## [3,] 0.6404637 0.6918259 0.7431882 0.7945505 0.8459127 0.8972750 0.9486373
## [4,] 0.5998994 0.6534152 0.7069309 0.7604466 0.8139623 0.8674780 0.9209937
## [5,] 0.5106531 0.5745175 0.6383819 0.7022463 0.7661106 0.8299750 0.8938394
## [6,] 0.5785912 0.6419692 0.7053472 0.7687252 0.8321032 0.8954812 0.9588592
##           [,8]      [,9]    [,10]    [,11]    [,12]    [,13]    [,14]    [,15]
## [1,] 0.9353198 0.9970534 1.058787 1.120521 1.182254 1.243988 1.305722 1.367456
## [2,] 1.0026558 1.0574574 1.112259 1.167061 1.221862 1.276664 1.331466 1.386267
## [3,] 0.9999995 1.0513618 1.102724 1.154086 1.205449 1.256811 1.308173 1.359535
## [4,] 0.9745094 1.0280251 1.081541 1.135057 1.188572 1.242088 1.295604 1.349119
## [5,] 0.9577038 1.0215682 1.085433 1.149297 1.213161 1.277026 1.340890 1.404754
## [6,] 1.0222371 1.0856151 1.148993 1.212371 1.275749 1.339127 1.402505 1.465883
##         [,16]    [,17]    [,18]    [,19]    [,20]    [,21]
## [1,] 1.429189 1.490923 1.552657 1.614390 1.676124 1.737858
## [2,] 1.441069 1.495871 1.550672 1.605474 1.660275 1.715077
## [3,] 1.410898 1.462260 1.513622 1.564985 1.616347 1.667709
## [4,] 1.402635 1.456151 1.509667 1.563182 1.616698 1.670214
## [5,] 1.468619 1.532483 1.596348 1.660212 1.724076 1.787941
## [6,] 1.529261 1.592639 1.656017 1.719395 1.782773 1.846151
```

```r
hist(y.cred[,1])
```



As above, we can plot the first 100 predictions.

```r
plot(df)
for(i in 1:100){
  lines(x.pred, y.cred[i, ], col=adjustcolor("red", alpha.f=0.3))
}
```



Since each column contains samples of a posterior distribution, we can make statistics, e.g. mean or confidence intervals.

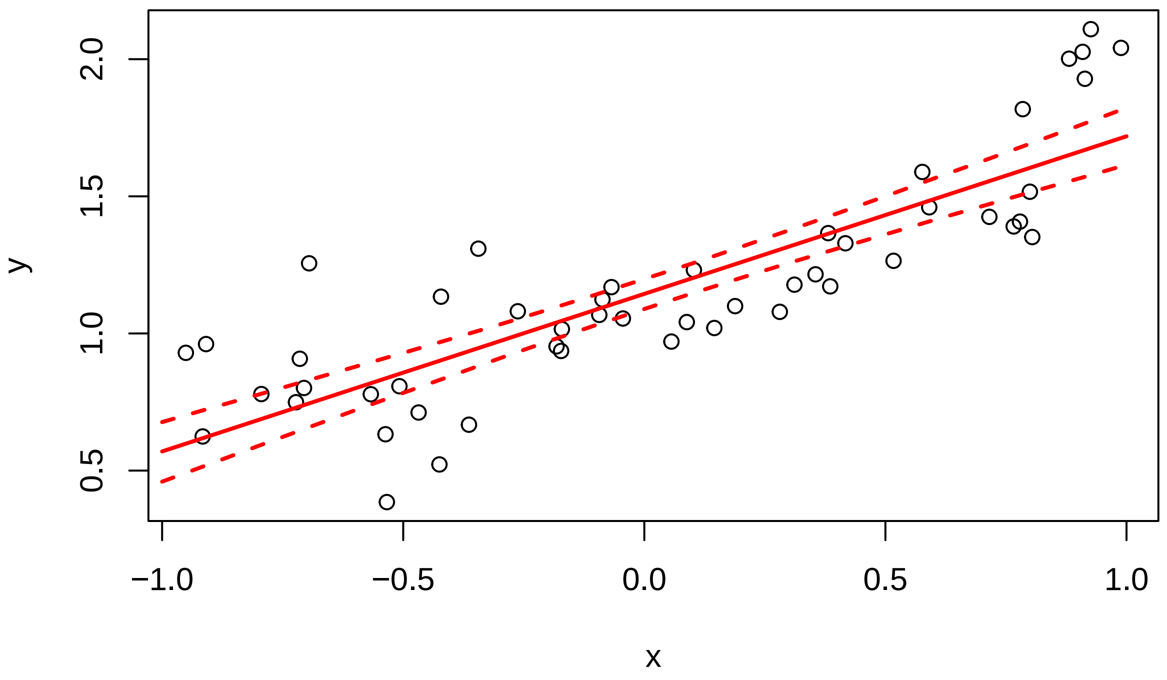
In Bayesian stats, these confidence intervals are often called **"credible intervals"**.

We will now plot the mean and the 90% credible intervals (using 5% and 95% quantiles).

Why 90% and not 95%? 95% is an arbitrary number. Choose your own credible interval!

We use the `apply()` function to use the `mean()` and `quantile()` commands on each column of the matrix.

```
plot(df)

y.cred.mean = apply(y.cred, 2, function(x) mean(x))
lines(x.pred, y.cred.mean, col="red", lwd=2)

y.cred.q05 = apply(y.cred, 2, function(x) quantile(x, probs=0.05))
lines(x.pred, y.cred.q05, col="red", lwd=2, lty=2)

y.cred.q95 = apply(y.cred, 2, function(x) quantile(x, probs=0.95))
lines(x.pred, y.cred.q95, col="red", lwd=2, lty=2)
```



A statistical model contains a **deterministic and stochastic part**.

The credible / confidence intervals are computed using distribution of the deterministic part only!

They are confidence intervals for the regression line, not for the data!

Now we will compute **true prediction intervals** also using the **stochastic model part**. (data are normally distributed around regression line with standard deviation `sigma`).

`y.pred` is structured as y.cred above:

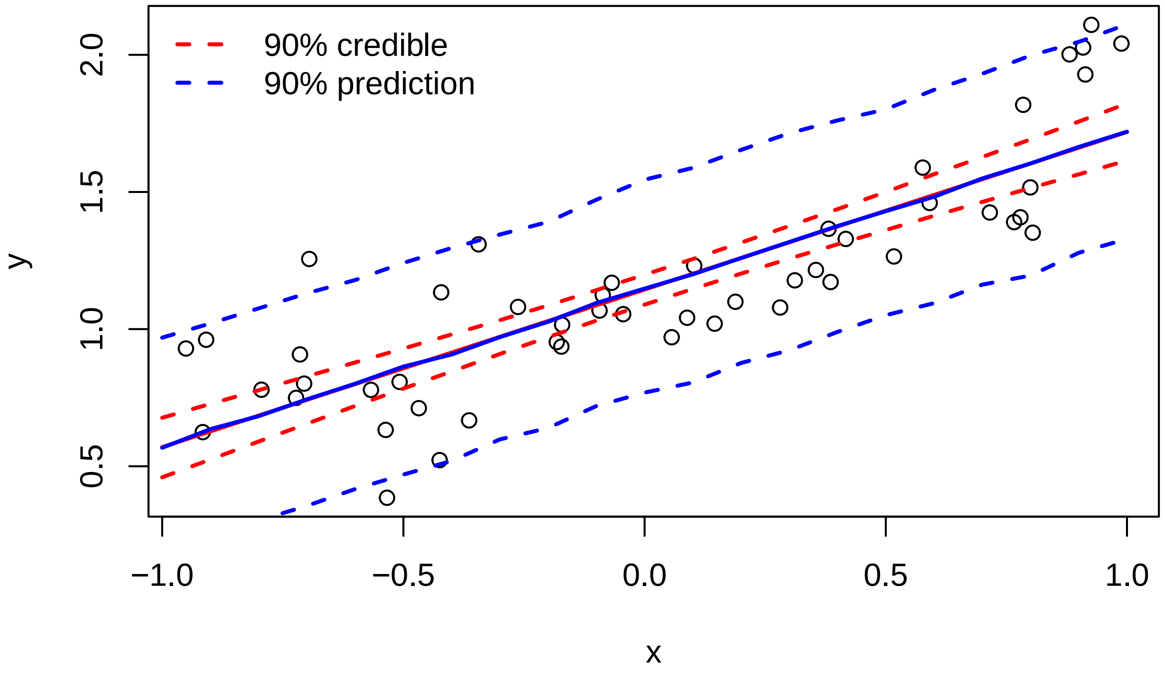
Each row `i` contains predictions for all `x.pred` using single parameter set `a_i`,`b_i` (and `sigma_i`).

Each column `j` contains 3000 predictions (for all posterior samples) for one predictor value `x.pred[j]`.

But now, each prediction is a random draw from `normal(a_i+b_i*x,sigma_i)` (deterministic part `a_i+b_i*x` was already computed in `y.cred`).

```r
y.pred = matrix(0, nrow=nrow(posterior), ncol=length(x.pred))

for(i in 1:nrow(posterior)){
  y.pred[i, ] = rnorm(n=length(x.pred), mean=y.cred[i, ], sd=rep(posterior[i, "sigma"],length(x.pred))
}

plot(df)

lines(x.pred, y.cred.mean, col="red", lwd=2)
lines(x.pred, y.cred.q05, col="red", lwd=2, lty=2)
lines(x.pred, y.cred.q95, col="red", lwd=2, lty=2)

y.pred.mean = apply(y.pred, 2, function(x) mean(x))
lines(x.pred, y.pred.mean, col="blue", lwd=2)

y.pred.q05 = apply(y.pred, 2, function(x) quantile(x, probs=0.05))
lines(x.pred, y.pred.q05, col="blue", lwd=2, lty=2)
```

```
y.pred.q95 = apply(y.pred, 2, function(x) quantile(x, probs=0.95))
lines(x.pred, y.pred.q95, col="blue", lwd=2, lty=2)

legend("topleft", legend=c("90% credible","90% prediction"), lwd=c(2,2), col=c("red","blue"), bty="n",
```



The 90% **credible interval** (red) tells us that we are 90% sure that the **regression line** is in that interval.

The 90% **prediction interval** (blue) tells us that we are 90% sure that the **data** are in that interval.

4 out of 50 datapoints are outside the prediction interval.

46 out of 50 datapoints are inside the prediction interval, that's 92% $\approx$ 90%.

Note: you can also make predictions while fitting using the `generated_quantities{}` block.

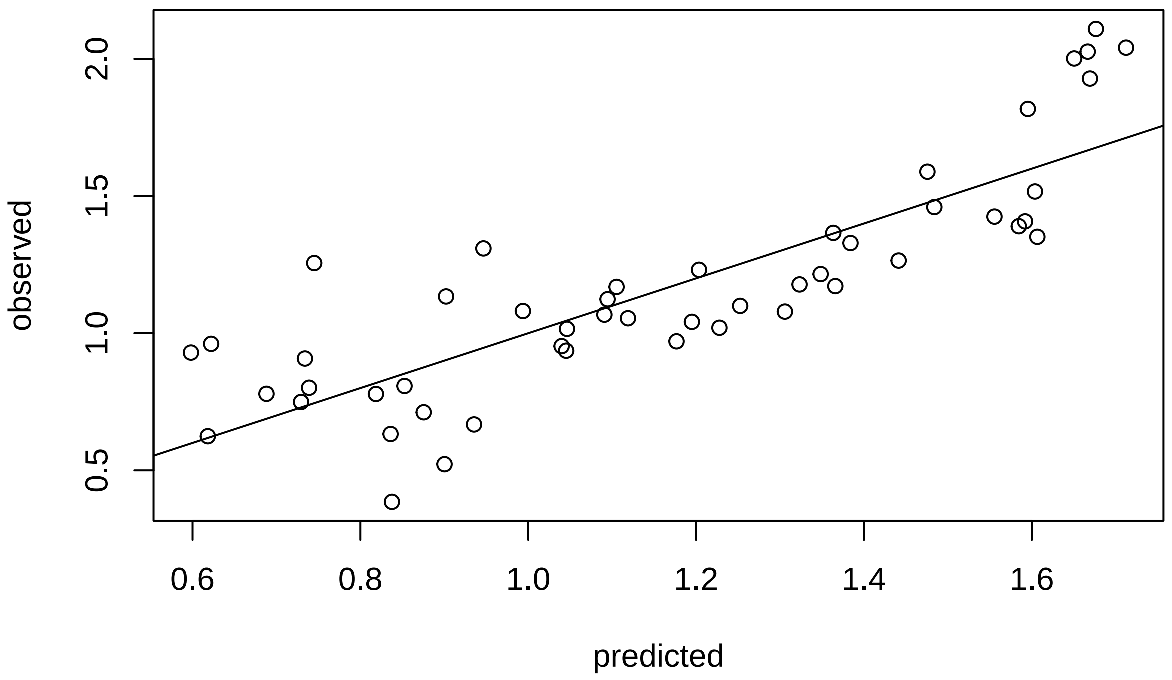
Observed vs. predicted

In the previous section we used a sequence of predictor values `x.pred` for making nice plots.

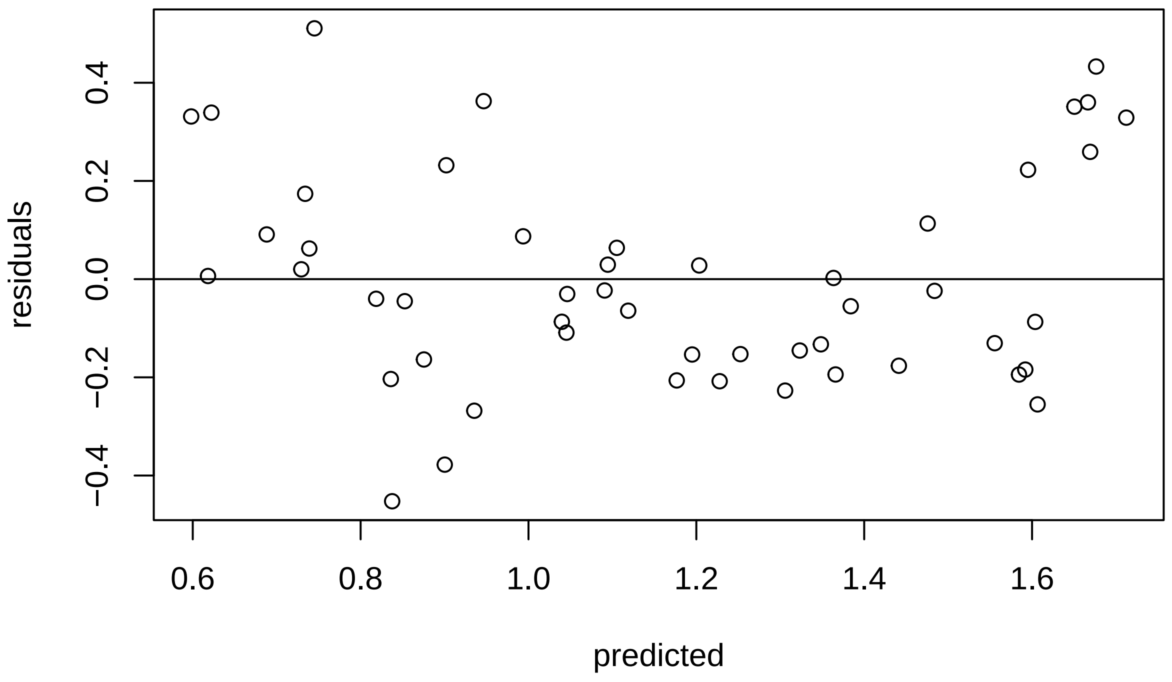
For model validation, we should make predictions for the actual data.

Then we can compare observed and predicted values (even if we have many predictors / groups and nice plots aren't possible).

```
x.pred = df$x # this are the actual predictor values from the data
y.cred = matrix(0, nrow=nrow(posterior), ncol=length(x.pred))

for(i in 1:nrow(posterior)){
  y.cred[i, ] = posterior[i,"a"] + posterior[i,"b"]*x.pred
}

y.cred.mean = apply(y.cred, 2, function(x) mean(x))

plot(y.cred.mean, df$y, ylab="observed", xlab="predicted")
abline(0,1)
```



```
plot(y.cred.mean, df$y-y.cred.mean, ylab="residuals", xlab="predicted")
abline(0,0)
```



There seems to be a little systematic bias for high and low values. (the dataset contains a small quadratic effect, see above). It also shows in the residual plot.

Note: you can also make predictions while fitting using the `generated_quantities{}` block.

Model comparison, AIC etc

For Stan models we will not cover it here. For brms models, see afternoon session!

Use the “loo” package for an information criterion that you can use similar to AIC.

You have to calculate the pointwise log-likelihood values in your model in the “generated quantities{}” block.

see https://cran.r-project.org/web/packages/loo/vignettes/loo2-with-rstan.html

Pitfalls of predictions

That’s a lot of code above just for predictions. Can’t we just use the mean fitted parameters to make predictions?

Please don’t do that! In Bayesian stats, everything is a distribution, also the predictions.

For simple linear models, the **mean of predictions** can be equal to the **predictions** using the **mean parameters**.

This is not the case for more complex models!

This phenomenon is called Jensen’s inequality.

For a nonlinear function $f(\theta)$ and some samples $\theta = \theta_1, ..., \theta_K$: $\text{mean}\{f(\theta_1), ..., f(\theta_K)\} \neq f(\text{mean}\{\theta_1, ..., \theta_K\})$

```
print(fit)
```

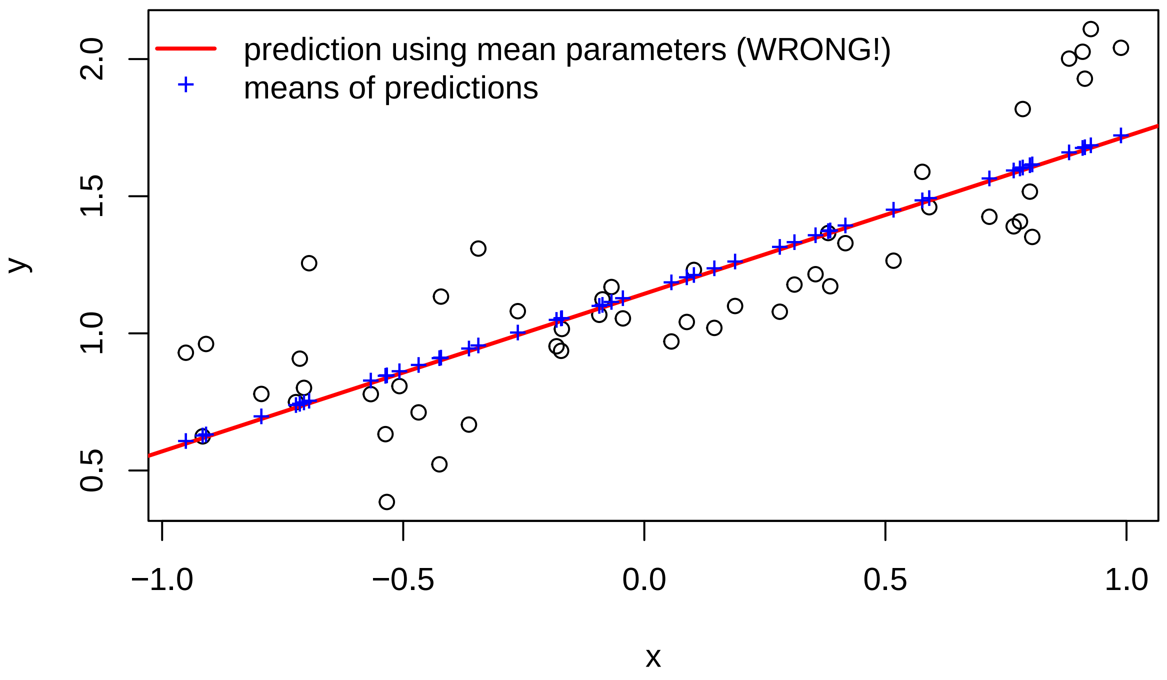
```
## Inference for Stan model: 8ad33c5d8a4bcdd6e619c1be283d2f6e.
## 4 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=4000.
##
##        mean se_mean   sd  2.5%   25%   50%   75% 97.5% n_eff Rhat
## a      1.14    0.00 0.03  1.08  1.12  1.14  1.17  1.21  3509    1
## b      0.57    0.00 0.06  0.46  0.54  0.58  0.61  0.69  3780    1
## sigma  0.23    0.00 0.02  0.19  0.21  0.23  0.24  0.28  3555    1
## lp__  48.04    0.03 1.29 44.60 47.47 48.41 48.98 49.51  1661    1
##
## Samples were drawn using NUTS(diag_e) at Tue Oct 18 13:23:42 2022.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

```
summary.fit = summary(fit)$summary

summary.fit
```

```
##             mean      se_mean         sd       2.5%        25%        50%
## a      1.1443377 0.0005591462 0.03312253  1.0777533  1.1227234  1.1447992
## b      0.5745159 0.0009051840 0.05565063  0.4622920  0.5371809  0.5750225
## sigma  0.2290889 0.0004110622 0.02450837  0.1865109  0.2118357  0.2271689
## lp__  48.0426144 0.0316930597 1.29180470 44.6025426 47.4651756 48.4076287
##              75%      97.5%    n_eff      Rhat
## a      1.1667733  1.2090224 3509.103 0.9998404
## b      0.6109410  0.6852184 3779.779 0.9996261
## sigma  0.2440545  0.2840872 3554.788 1.0017691
## lp__  48.9788635 49.5082695 1661.366 1.0015343
```

```
plot(df)
abline(summary.fit["a","mean"], summary.fit["b","mean"], col="red", lwd=2)
points(x.pred,y.cred.mean, col="blue", pch="+")
```

```
legend("topleft", legend=c("prediction using mean parameters (WRONG!)","means of predictions"), bty="n"
       lty=c(1,NA), pch=c(NA,"+"), col=c("red","blue"), lwd=c(2,2))
```



(However, in this linear regression, both are the same)