Bishop's PRML, Chapter 3

May, 2015

- Linear Basis Models (section 3.1)
  - Maximum Likelihood
  - Regularization (section 3.1.4)
  - Sequence Learning (section 3.1.3)
- Bayesian Linear Regression (section 3.3)
  - Animating the sequential inference process
  - Predictive Distribution (section 3.3.2)

This page contains source code relating to chapter 3 of Bishop's *Pattern Recognition and Machine Learning* (2009)

Linear Basis Models (section 3.1)

Linear models are defined as

$$y(x, w) = w_0, w_1 x_1 + \ldots + w_n x_N$$

where x is the datapoint and w are the parameters of the model.

A model extension is to consider linear combinations of fixed nonlinear functions ϕ over x which are called *basis functions*:

$$y(x, w) = \sum_{i=0}^{N} w_i \phi_i(x_i)$$

where, by convention, $\phi_0(x) = 1$.

An example of basis is the gaussian basis:

$$\phi(x) = exp\{-\frac{(x-\mu)^2}{2\sigma^2}\}$$

Maximum Likelihood

If we want to find the maximum likelihood, under the assumption of normal noise, the formula is given by:

$$w_{ML} = (\Phi^T \Phi)^{-1} \Phi^T y$$

where y are the training data output, and Φ is the design matrix such that $\Phi_{i,j} = \phi_{j+1}(x_i)$.

The next function computes it:

```
# compute weights for a given basis phi, using maximum likelihood
compute_w <- function(X, Y, phi) {
  Phi <- sapply(phi, function(base) base(X))  # make design matrix
  solve(t(Phi) %*% Phi) %*% t(Phi) %*% Y      # find maximum likelihood
}
```

With the value of parameters w, and a given basis $\phi_1, \phi_2, \ldots$ we can estimate results for new points.

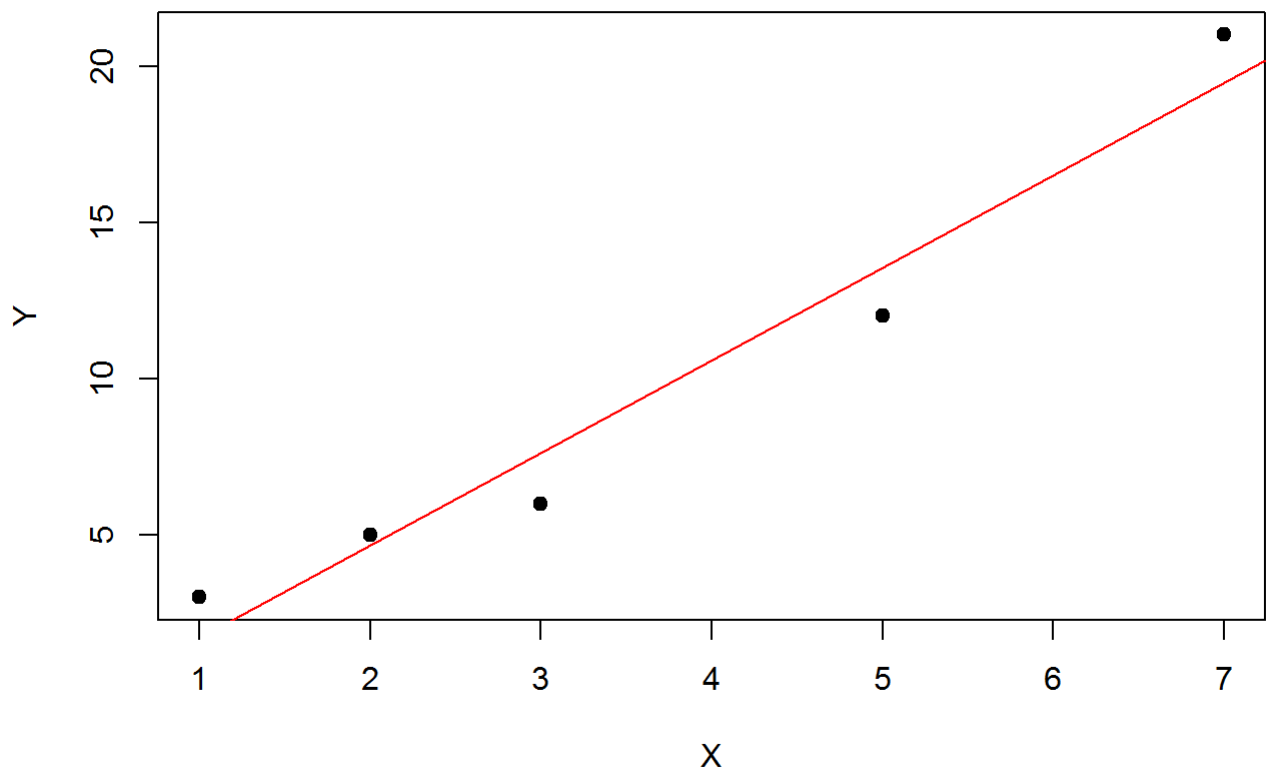
```
# compute estimates for points in x, given weigths W and respective basis
regression_curve <- function(xs, W, basis) {
  m <- sapply(phi,function(base) base(xs))  # values for each base
  apply(m, 1, function(row) sum(row*W))     # add them together, row by row
}

# function to draw regression line
draw_regression <- function(X, W, phi) {
  xs     <- seq(min(X),max(X),len=50)
  ys_hat <- regression_curve(xs, W, phi)
  points(xs, ys_hat, type="l", col="red")
}
```

Let's define some basis functions ϕ:

```
# some basis egs
one <- function(x) rep(1,length(x))
id  <- function(x) x
sq  <- function(x) x^2
x3  <- function(x) x^3
x4  <- function(x) x^4
```

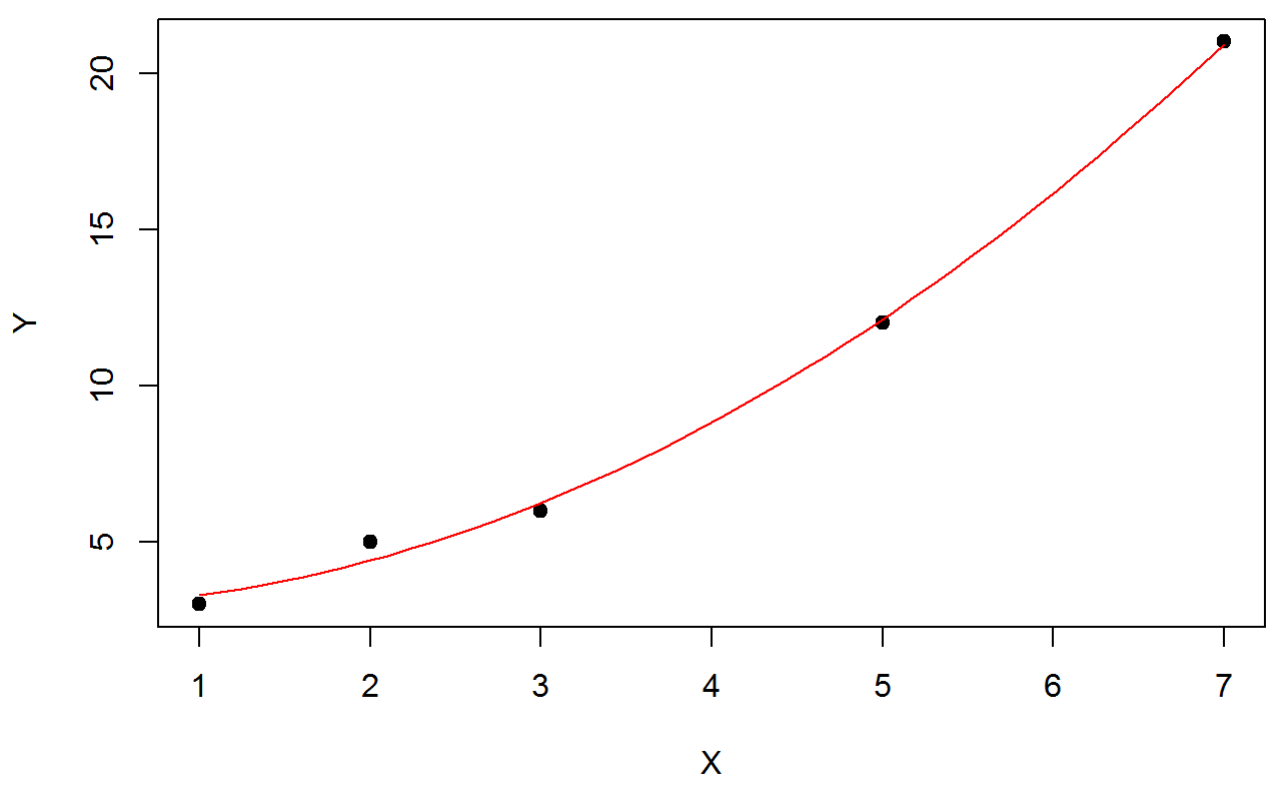
And let's apply this to some egs. First to the standard linear regression:

```
# some data
X <- c(1,2,3,5,7)
Y <- c(3,5,6,12,21)

# basis for linear regression
phi <- c(one, id)
W <- compute_w(X, Y, phi)

plot(X,Y,pch=19)
abline(W, col="red")
```



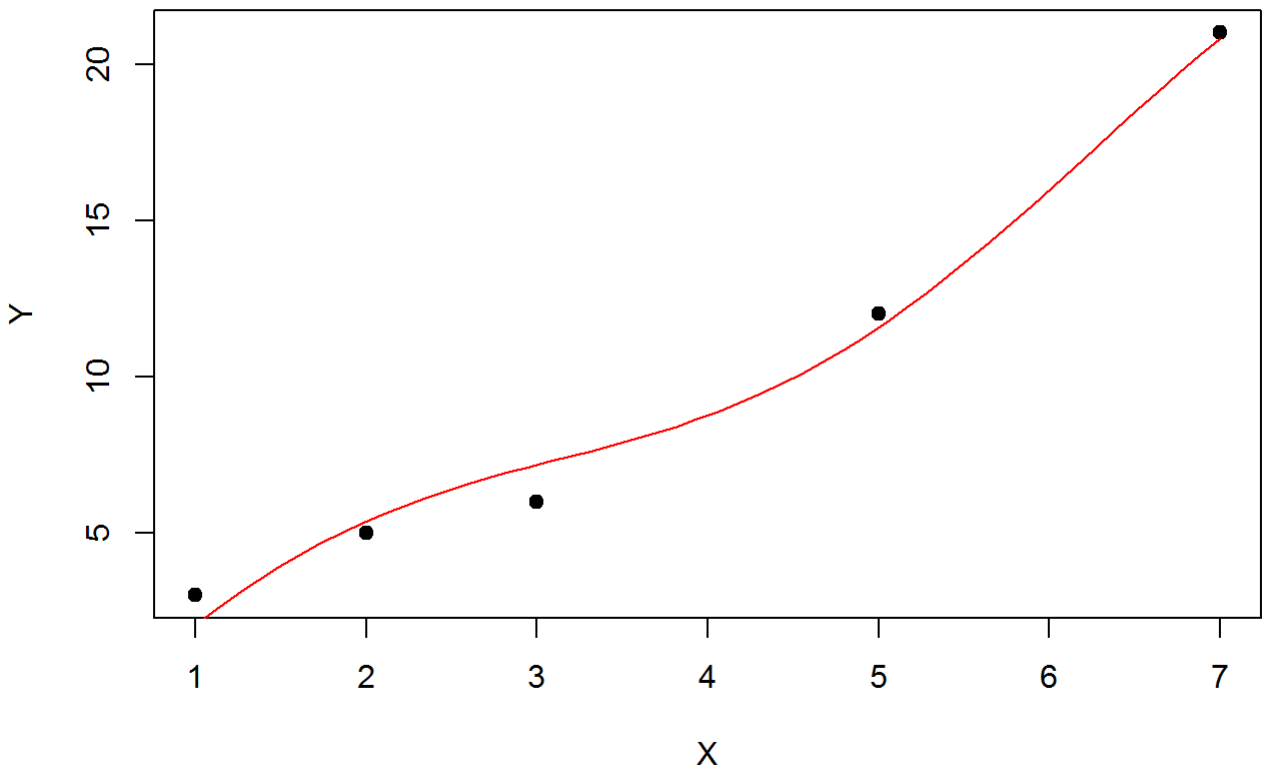
Then to quadratic regression. Notice that here, the basis are $\{\phi_0, \phi_1(x) = x, \phi_2(x) = x^2\}$:

```
# basis for quadratic regression
phi <- c(one, id, sq)
W <- compute_w(X, Y, phi)

plot(X,Y,pch=19)
draw_regression(X,W,phi)
```



And a strange one that uses a sine basis:

```
# including a sine into the linear regression basis
phi <- c(one, id, function(x) sin(x))
W <- compute_w(X, Y, phi)

plot(X,Y,pch=19)
draw_regression(X,W,phi)
```



Regularization (section 3.1.4)

The goal of regularization is to provide a way to prevent overfit in the parameter values, ie, sometimes the model tends to place very extreme values over w to fit as best as possible to the given x. Regularization defines a kind of budget that prevents to much extreme values in the parameters. This is especially relevant in complex models that have great expressivity to adjust to the dataset, which means that they could easily overfit.

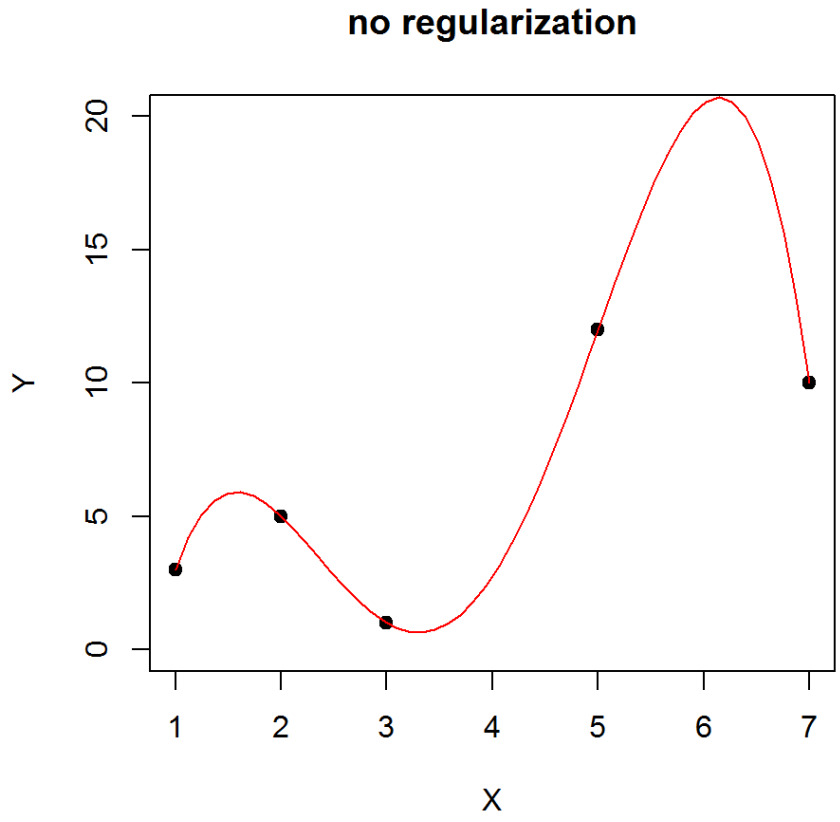
Regularization uses a parameter λ to tune the budget amount.

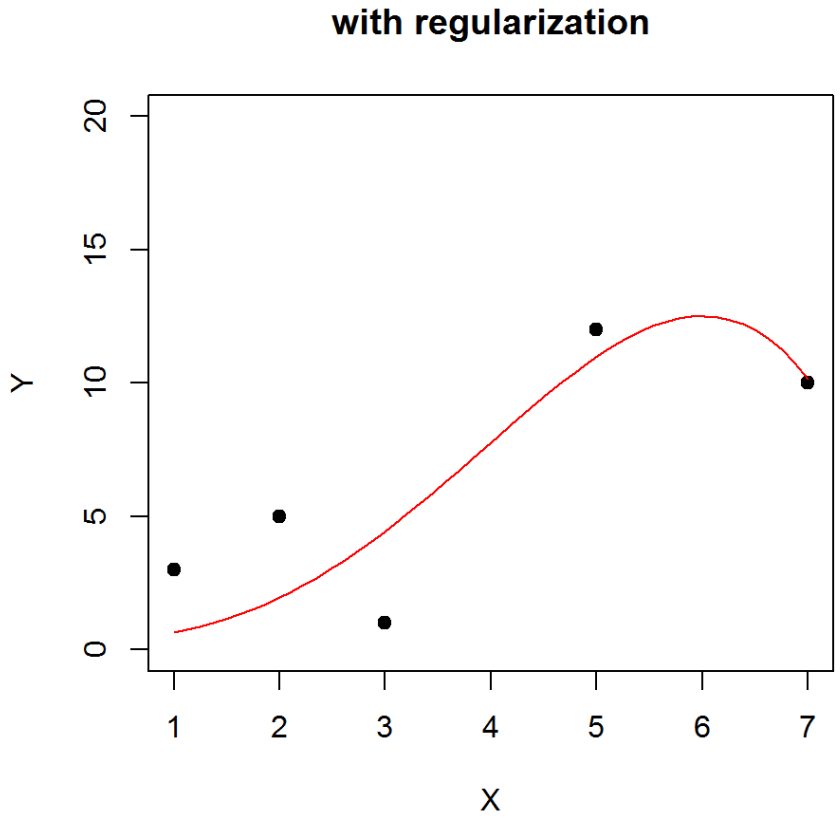
The next function computes w using a value λ given by the user:

```
compute_w_reg <- function(X, Y, phi, lambda) {
  Phi <- sapply(phi, function(base) base(X))  # make design matrix
  solve(lambda * diag(length(phi)) + t(Phi) %*% Phi) %*% t(Phi) %*% Y
}
```

An eg:

```
X <- c(1,2,3,5,7)
Y <- c(3,5,1,12,10)
phi <- c(one, id, sq, x3, x4)    # quartic regression

par(mfrow=c(1,2))
W <- compute_w(X, Y, phi)        # without regularization
plot(X,Y,pch=19,ylim=c(0,20), main="no regularization")
draw_regression(X,W,phi)

W <- compute_w_reg(X, Y, phi, lambda=11) # with regularization
plot(X,Y,pch=19,ylim=c(0,20), main="with regularization")
draw_regression(X,W,phi)
```






```
par(mfrow=c(1,1))
```

Sequence Learning (section 3.1.3)

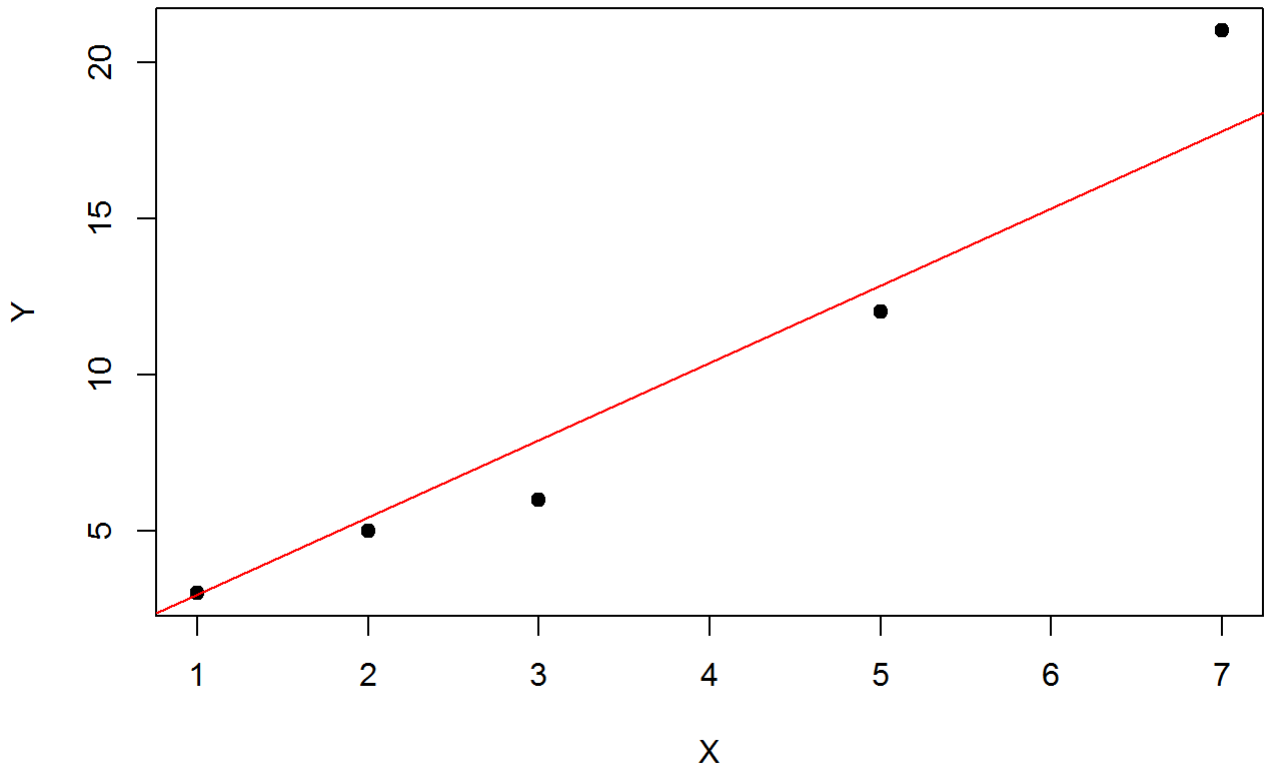
This section deals with the problem of not being able to infer all the datapoints at the same time. This could happen because making inference with all of them is very costly, or that we don't have them all at the moment, but need to update the model when more datapoints arrive.

The next functions implement a kind of gradient descent, updating parameters w for each new data point. This method is sub-optimal and might not converge. The next section uses Bayesian methods that do not suffer from this problem.

```
update_w <- function(W, x_n, y_n, phi, eta=0.01) {
  phi_n <- matrix(sapply(phi, function(base) base(x_n)), ncol=1) # make it a vector
  W + eta * phi_n %*% (y_n - t(W)%*%phi_n)
}

compute_w_batch <-  function(X, Y, phi, convt=1e-3, eta=0.01) {
  W <- rnorm(length(phi),0,0.1)             # initialization to small random values
  for(i in 1:length(X)) {                   # batch update
    W <- update_w(W, X[i], Y[i], phi, eta)
  }
  W
}
```

An eg:

```
X <- c(1,2,3,5,7)
Y <- c(3,5,6,12,21)
phi <- c(one, id) # basis for linear regression

W <- compute_w_batch(X, Y, phi,eta=0.015) # choosing value for eta is tricky...
plot(X,Y,pch=19)
abline(W, col="red")
```



Bayesian Linear Regression (section 3.3)

The next eg uses function $f(x) = -0.3 + 0.5x$ to generate a dataset, adding a gaussian noise with sd $\sigma = 0.2$. The x come from $\mathcal{U}(-1, 1)$.

```
make.X <- function(n) {
  runif(n,-1,1)
}

a0 <- -0.3 # the true values (unknown to model)
a1 <-  0.5

sigma <- 0.2
beta  <- 1/sigma^2  # precision

make.Y <- function(xs) {
  a0 + a1*xs + rnorm(length(xs),0,sigma)
}
```

Our model is also a linear regression:

$$p(y|w, x) = w_0 + w_1 x$$

$$p(w) \sim \mathcal{N}(m, S)$$

$$p(x) \sim \mathcal{U}(-1, 1)$$

Analytically, it can be shown that given a prior

$$p(w) = \mathcal{N}(w|m_0, S_0)$$

the posterior, given dataset x, y with n points, is

$$p(w|x, y) = \mathcal{N}(w|m_N, S_N)$$

where

$$m_N = S_N(S_0^{-1} m_0 + \beta \Phi^T y)$$

$$S_N^{-1} = S_0^{-1} + \beta \Phi^T \Phi$$

β is the precision (1/variance) we assumed fixed (if we also wanted to estimate the precision, then it would be easier to use MCMC methods).

The next function does this parameter update, it must receive the new data x, y and the previous parameter values (m, S):

```
# uses linear regression basis (phi) by default
compute_posterior <- function(X, Y, m_old, S_old, phi= c(one, id)) {
  Phi <- sapply(phi, function(base) base(X))  # make design matrix

  if(length(X)==1)  # type hack, with just 1 point, R makes a vector, not a matrix
    Phi <- t(as.matrix(Phi))

  S_new <- solve(solve(S_old) + beta * t(Phi) %*% Phi)
  m_new <- S_new %*% (solve(S_old) %*% m_old + beta * t(Phi) %*% Y)

  list(m=m_new, S=S_new)  # return the new updated parameters
}
```

Let's apply this to a given dataset. We will start with a prior, m_0, centered at the origin (without loss of generality, since we can always center the dataset), and with a wide enough covariance matrix S_0. So, the initial parameter values w are:

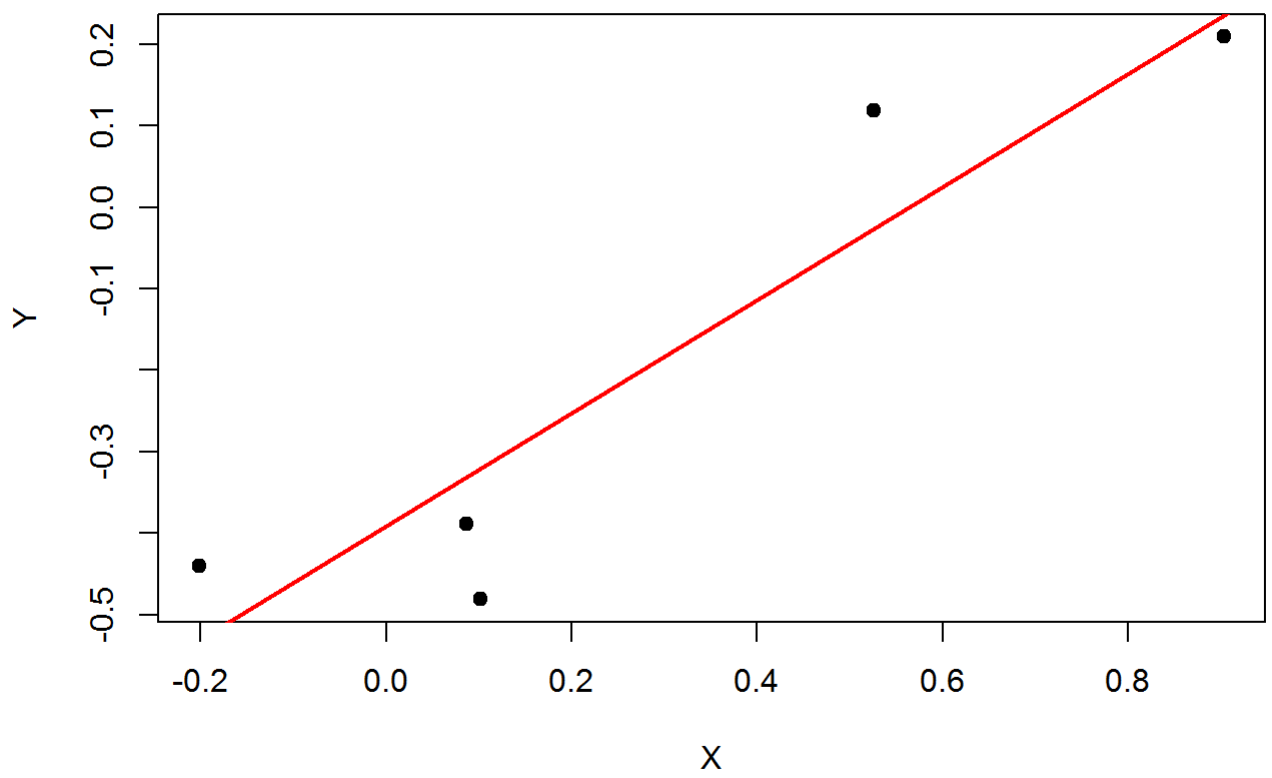
```
alpha <- 2.0
m_0 <- c(0,0)         # we know the mean is (0,0), otherwise, center first
S_0 <- alpha*diag(2)  # relatively uninformative prior
```

Now we make some points and update the model with them:

```
set.seed(121)
X <- make.X(5) # make some points
Y <- make.Y(X)

posterior_1 <- compute_posterior(X, Y, m_0, S_0)
posterior_1$m
```

```
##           [,1]
## [1,] -0.391059
## [2,]  0.693193
```

```
plot(X, Y, pch=19, col="black")
abline(posterior_1$m, col="red", lwd=2)
```

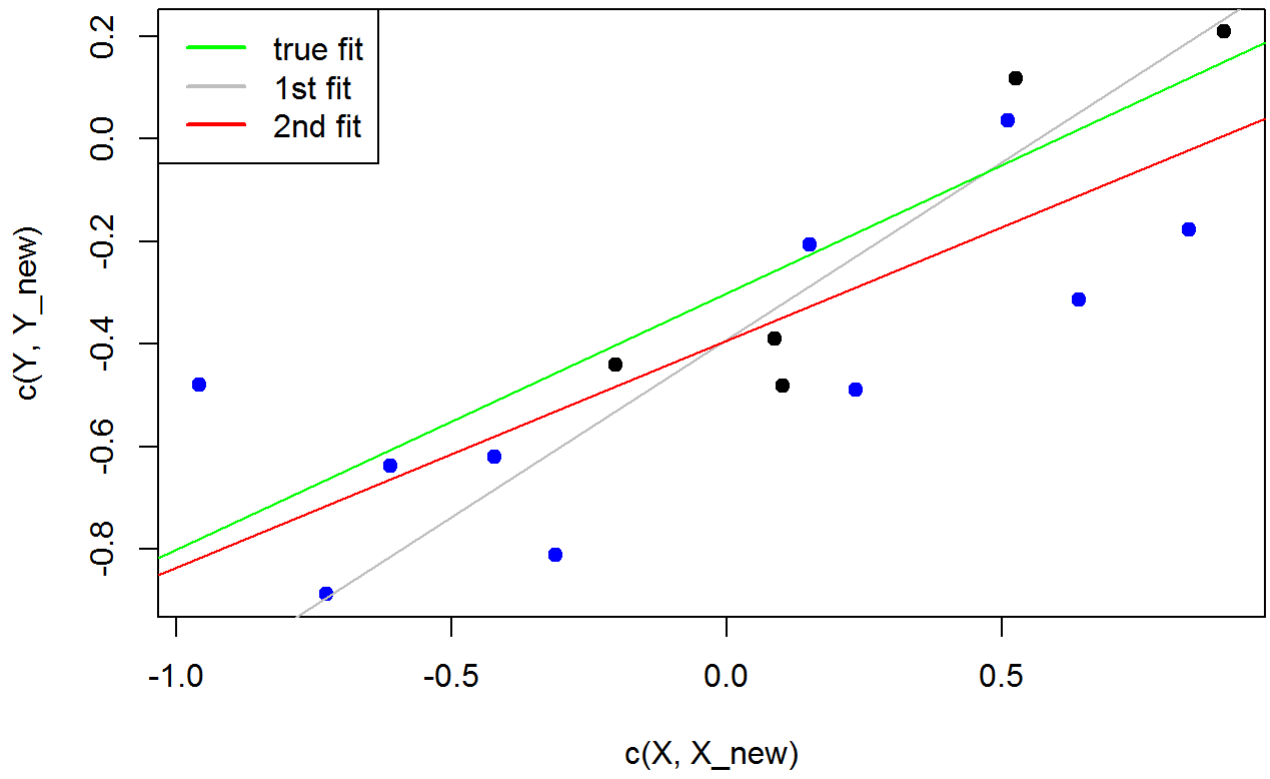


if we have more points afterwards, we can again update the model:

```
X_new <- make.X(10) # more points are available!
Y_new <- make.Y(X_new)

posterior_2 <- compute_posterior(X_new,  Y_new, posterior_1$m, posterior_1$S)
posterior_2$m
```

```
##           [,1]
## [1,] -0.3930011
## [2,]  0.4430177
```

```
plot(c(X,X_new),c(Y,Y_new),type="n")
legend("topleft",c("true fit","1st fit","2nd fit"),
       col=c("green","grey","red"), lty=1, lwd=2)
points(X    , Y    , pch=19, col="black")
points(X_new, Y_new, pch=19, col="blue")
abline(posterior_1$m, col="grey")  # old fit
abline(posterior_2$m, col="red")   # new fit
abline(c(-0.3,.5), col="green")    # target function (true parameter values)
```

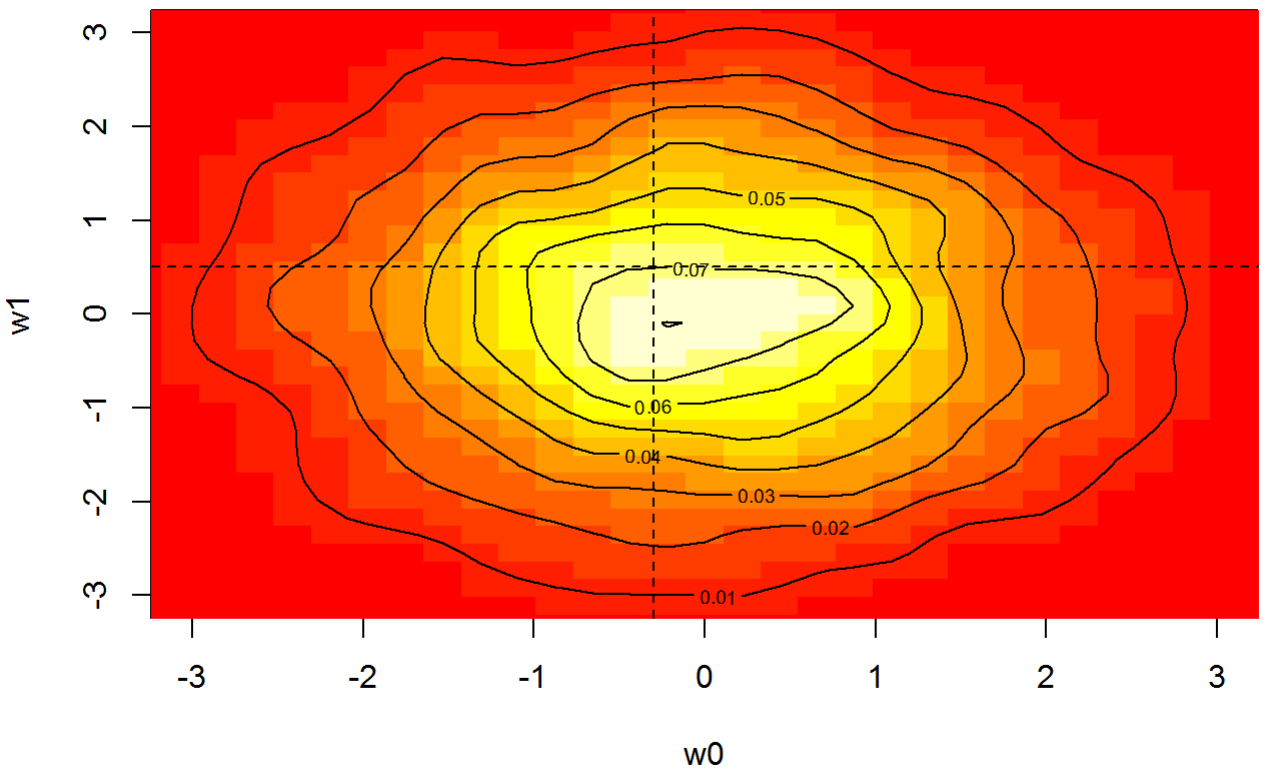


The next function draws the joint distribution $p(w_0, w_1)$:

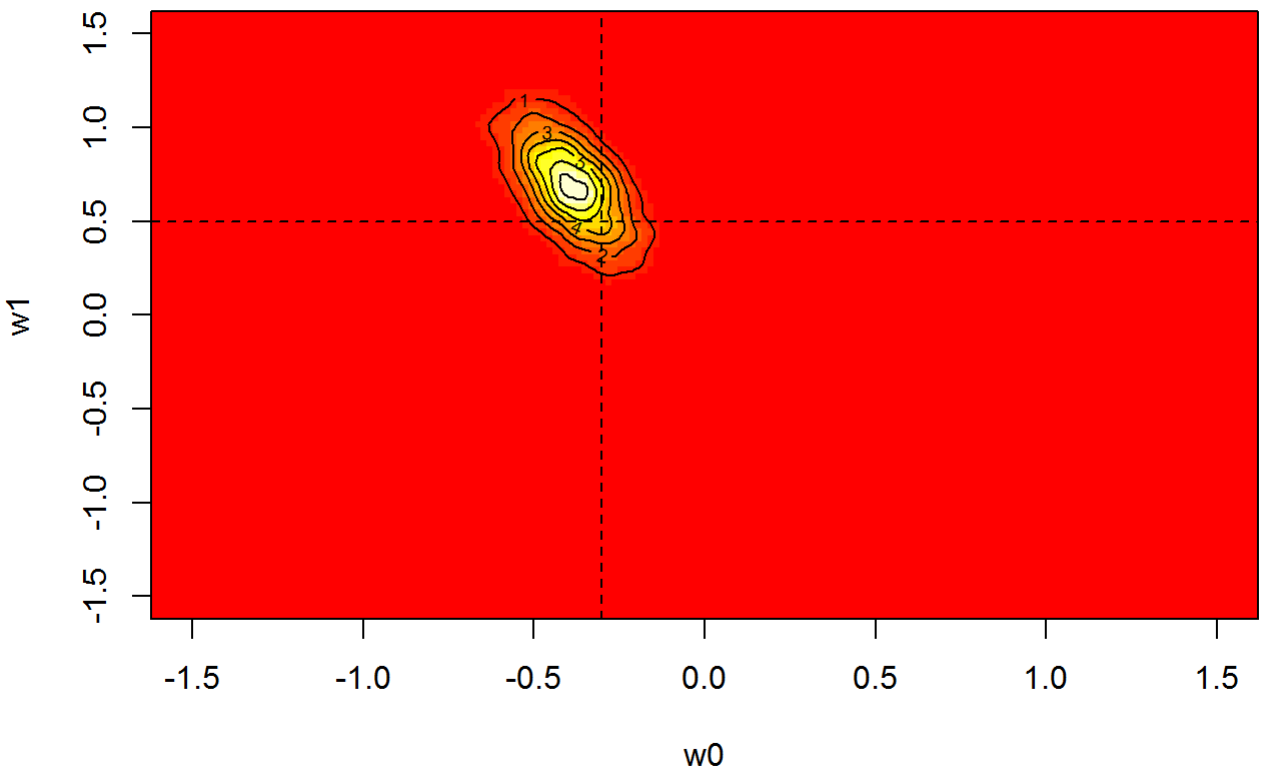
```
library(MASS)

draw_bivariate <- function(m, S, lims=c(-1.5,1.5)) {
  bivn <- mvrnorm(5e3, mu = m, Sigma = S)
  bivn.kde <- kde2d(bivn[,1], bivn[,2], n = 50)
  plot(0,type="n", xlim=lims, ylim=lims, xlab="w0", ylab="w1")
  # draw a red rectangle over the plot area
  rect(par("usr")[1], par("usr")[3], par("usr")[2], par("usr")[4], col =
"red")
  image(bivn.kde, xlim=lims, ylim=lims, add=T)
  contour(bivn.kde, add = T)
  abline(v=-0.3, h=0.5, lty=2) # mark the real values
}
```

Let's see how the joint distribution evolved during these three steps, ie, for prior $p(w_0, w_1)$, and the two posteriors:

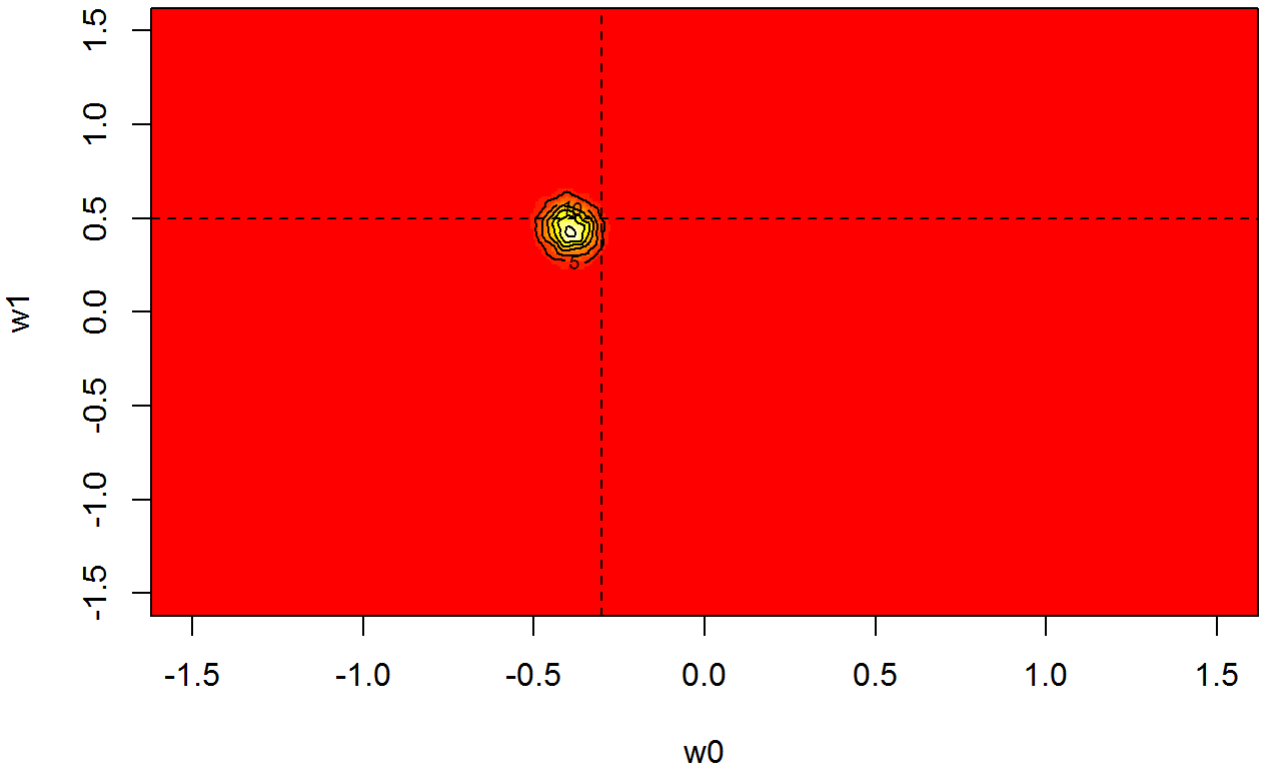
```
draw_bivariate(m_0, S_0, lims=c(-3,3))
```




```
draw_bivariate(posterior_1$m, posterior_1$S)
```

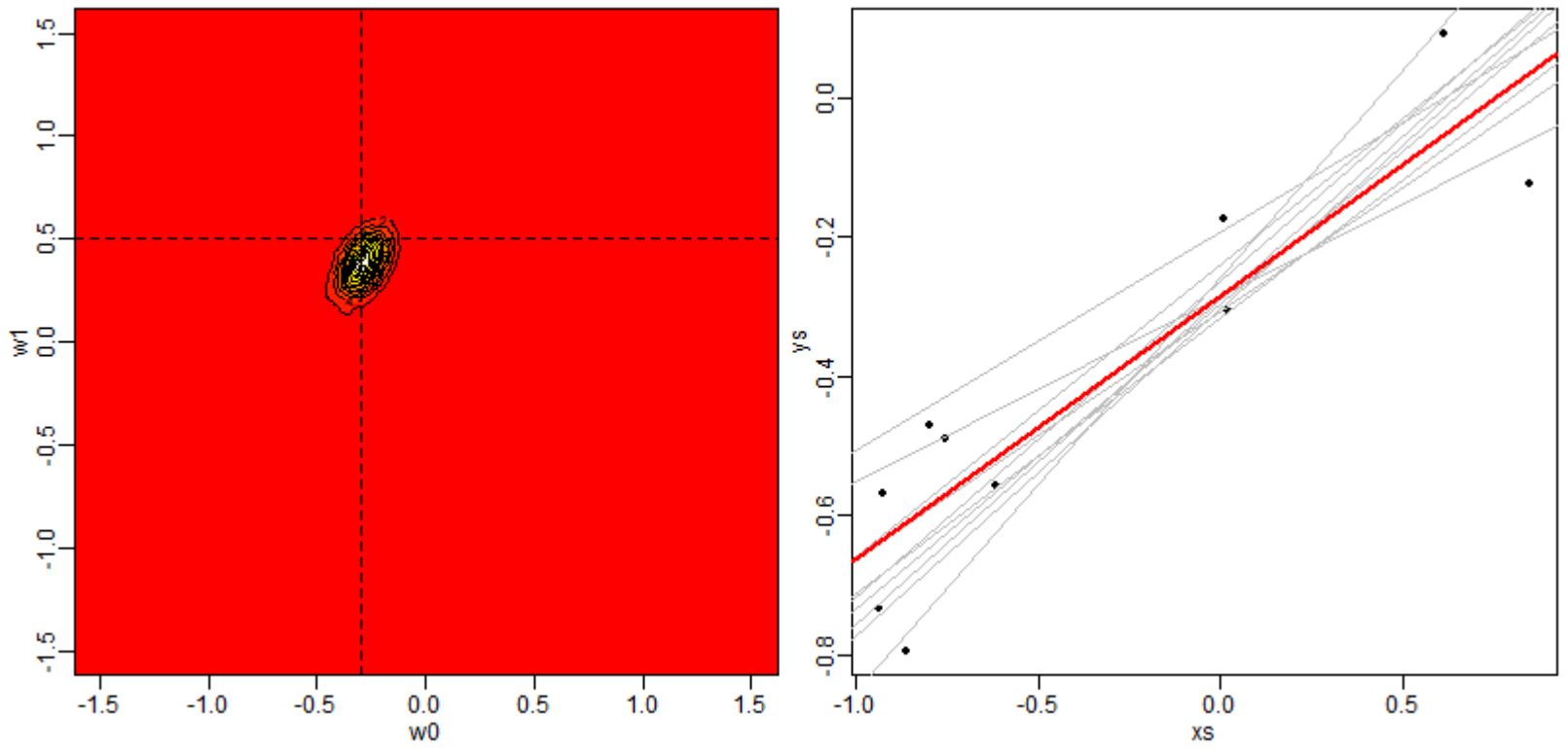



```
draw_bivariate(posterior_2$m, posterior_2$S)
```



Animating the sequential inference process

The next code, when executed, produces a stand-alone html page, which was embedded here (click the buttons to control the animation):



The grey lines are some candidates given by the current parameter values of the model. The red line is the mean of many candidates, so it's a possible estimation for the parameter values of the target function that produced the data.

```
library(animation)

bayes_reg <- function () {
  n = ani.options("nmax")

  par(mfrow=c(1,2)) # prepare two lateral plots

  # draw first left panel
  pdf <- list(m=c(0,0) , S=2.0*diag(2))
  draw_bivariate(pdf$m, pdf$S)

  # draw first right panel
  plot(0, type="n", xlim=c(-3,3), ylim=c(-3,3))
  estimates <- mvrnorm(10, mu = pdf$m, Sigma = pdf$S)
  for(i in 1:nrow(estimates))
    abline(a=estimates[i,1], b=estimates[i,2], col="grey")

  xs <- c() # the vectors that will collect all X's and Y's
  ys <- c()

  for(i in 1:n) {
    X <- make.X(1)
    Y <- make.Y(X)

    ani.pause()  # pauses and flushes the current plots into the animation

    # draw left panel
    pdf <- compute_posterior(X,Y, pdf$m, pdf$S)
    draw_bivariate(pdf$m, pdf$S)

    # draw right panel
    xs <- c(xs,X)  # add values X and Y into the vectors
    ys <- c(ys,Y)
    plot(xs,ys)
    estimates <- mvrnorm(500, mu=pdf$m, Sigma=pdf$S)   # make many regressions
    for(i in 1:10)
      abline(a=estimates[i,1], b=estimates[i,2], col="grey") # draw 10
    abline(apply(estimates,2,mean), col="red", lwd=2)        # draw the mean

  }
  dev.hold() # flushes the last plots into the animation
}

saveHTML({
     par(mar = c(3, 2.5, 1, 0.2), pch = 20, mgp = c(1.5, 0.5, 0))
     ani.options(nmax = ifelse(interactive(), 30, 30), interval = 1)
     bayes_reg()
   },
   img.name = "bayes_reg",
   htmlfile = "bayes_reg.html",
   ani.height = 400,
   ani.width  = 800,
   title = "Bayesian Linear Regression",
   description = c(
     "This is a reproduction of Bishop -- Pattern Recognition and Machines Learning's example of pages 154,155",
     "The true generator is the function f(x) = -0.3x + 0.5 with a normal noise of sd 0.2",
     "The model receives one point per iteration and updates itself using Bayes rule.",
     "The left panel shows a countour of the join distribution for the estimates of f(x), namely w0 and w1.",
     "The right panel shows the current set of points, 10 linear estimations given the model's current knowledge (in grey),
and the mean (in red).")
)
```

```
## HTML file created at: bayes_reg.html
```

Predictive Distribution (section 3.3.2)

Usually we don't care that much about the values of the parameters, instead we wish to predict y given a new datapoint x. This is given by the predictive distribution:

$$p(y|Y,X,\alpha,\beta) = \int p(y|w,\beta)p(w|Y,X,\alpha,\beta)dw$$

where X, Y are the input and output values of the previous data.

After some hairy math, we find that

$$p(y|x,X,Y,\alpha,\beta) = \mathcal{N}(y|m_N^T\phi(x), \sigma_N^2(x))$$

where $\sigma_N^2(x) = 1/\beta + \phi(x)^T S_N \phi(x)$

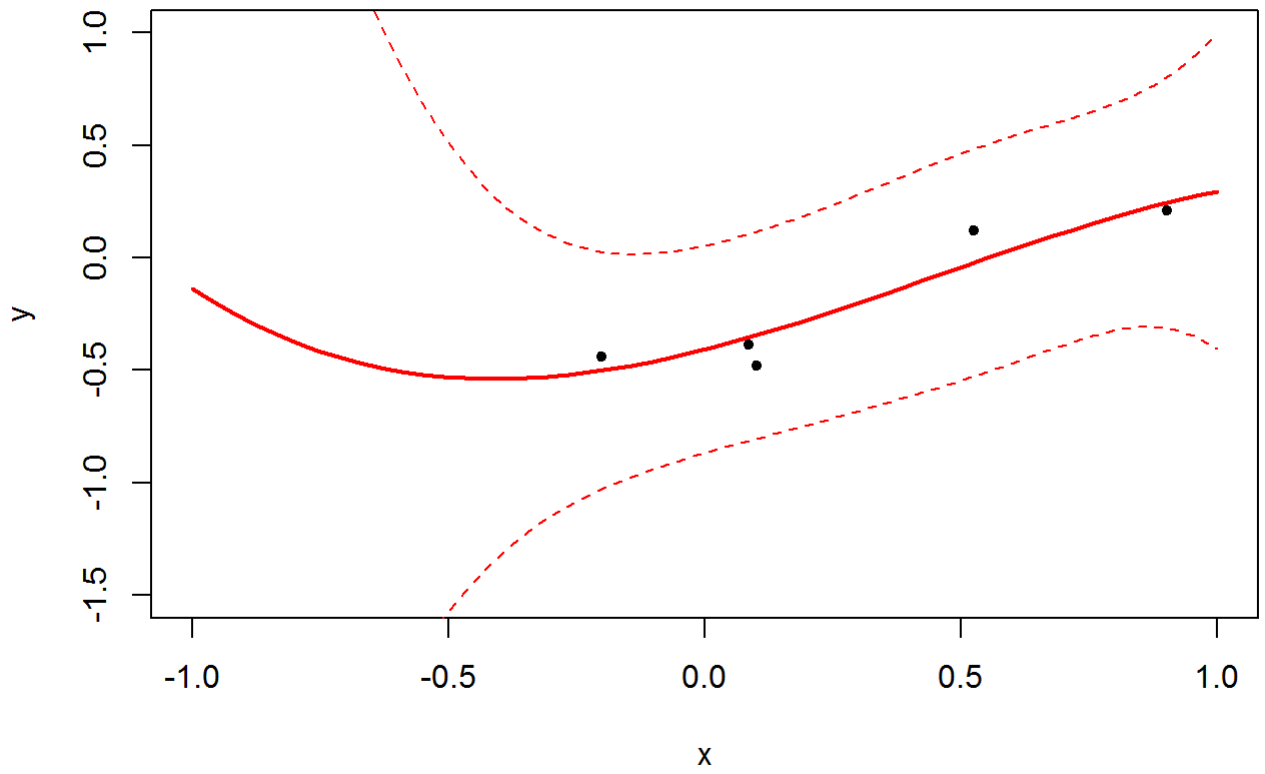
```
# return the predictive distribution's mean and 95% density interval
get_predictive_vals <- function(x, m_N, S_N, phi) {
  phix <- sapply(phi, function(base) base(x))
  mean_pred <- t(m_N) %*% phix
  sd_pred  <- sqrt(1/beta + t(phix) %*% S_N %*% phix)

  c(mean_pred, mean_pred-2*sd_pred, mean_pred+2*sd_pred)
}
```

The next function is for drawing purposes:

```
draw_predictive <- function(xs, m_N, S_N, phi) {
  vs <- rep(NA, length(xs))
  ys <- data.frame(means=vs, p2.5=vs, p97.5=vs)  # init dataframe

  for (i in 1:length(xs)) {  # compute predictive values for all xs
    ys[i,] <- get_predictive_vals(xs[i],m_N, S_N, phi)
  }

  # draw mean and 95% interval
  lines(xs, ys[,1], col="red", lwd=2)
  lines(xs, ys[,2], col="red", lty="dashed")
  lines(xs, ys[,3], col="red", lty="dashed")
}
```

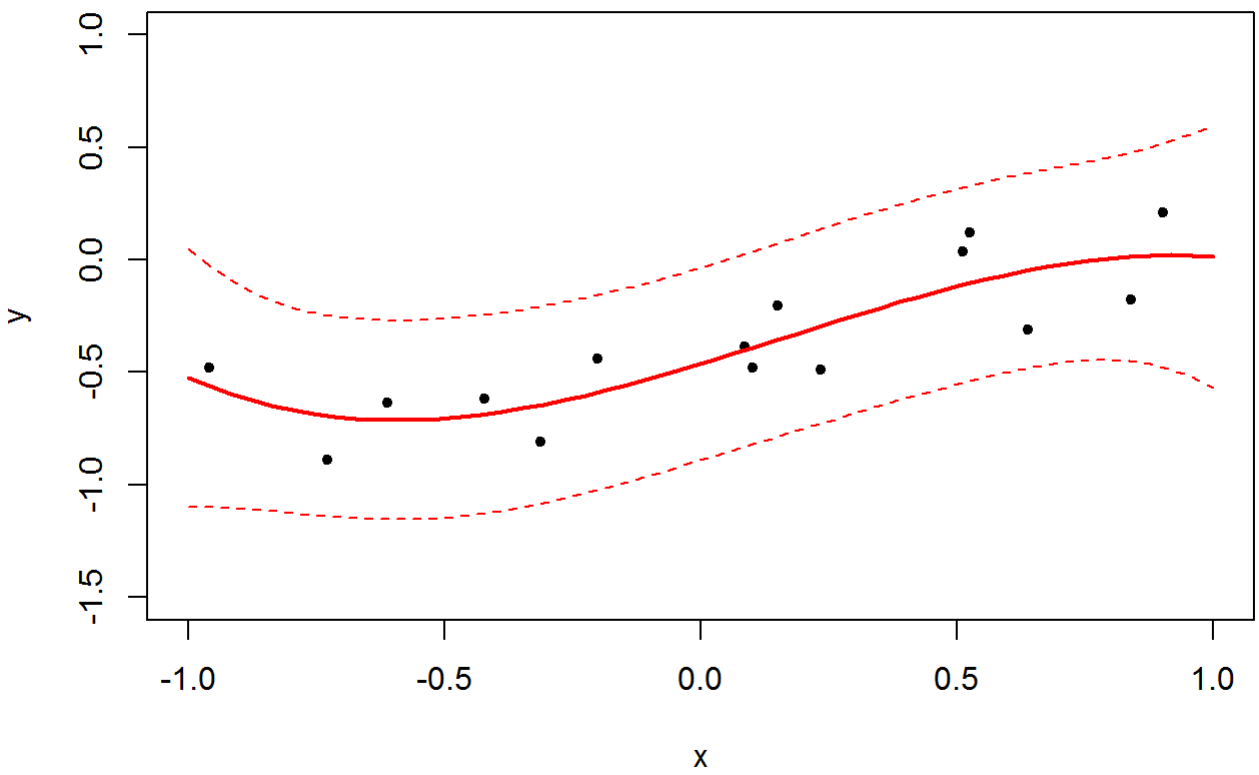
Let's see what's the predictive distribution of a cubic regression after learning some initial information:

```
set.seed(121)
X <- make.X(5) # make some points
Y <- make.Y(X)

phi <- c(one,id,sq,x3) # basis for the cubic regression
m_0 <- c(0,0,0,0)      # priors
S_0 <- alpha*diag(4)

posterior_1 <- compute_posterior(X, Y, m_0, S_0, phi=phi)
m_N <- posterior_1$m
S_N <- posterior_1$S

plot(X, Y, pch=20, ylim=c(-1.5,1), xlim=c(-1,1), ylab="y", xlab="x")
xs <- seq(-1,1,len=50)
draw_predictive(xs, m_N, S_N, phi=phi)
```



Now let's update the model with more points:

```
X_new <- make.X(10) # more points are available!
Y_new <- make.Y(X_new)

posterior_2 <- compute_posterior(X_new, Y_new, posterior_1$m, posterior_1$S, phi=phi)
m_N <- posterior_2$m
S_N <- posterior_2$S

plot(c(X,X_new), c(Y,Y_new), pch=20, ylim=c(-1.5,1), xlim=c(-1,1), ylab="y", xlab="x")
draw_predictive(xs, m_N, S_N, phi=phi)
```



Since we have more information, the predictive distribution has less uncertainty, especially in the extreme values around -1, since among the new datapoints, there are information nearby.