1-2 Introduction to Bayesian Modeling

Regression Walkthroughs

Modeling in the Real World

Bayesian Priors

Before we walk through regression templates and exercises, let's stop and review why we're here.

We're here to build competency in the application of statistical modeling to business analysis and planning. To this point, the models we've applied are based on structured data. But, navigating the real world business environment involves modeling unstructured data too *(government policies, industry strategy, capital market risk, brand awareness, customer perceptions)*. So how do we include all of these unstructured *(but very significant)* factors into analyses? For the most part, the answer has been to rely on descriptive visualization, and of course, *"very good brains"*:

"The dip in sales seems to coincide with the decision to eliminate the sales staff."

Figure 1: Visualization

But, we can integrate experience and knowledge into models using Bayesian analysis, and we do this through priors. In the last section, we looked at how priors affect posterior projections. As we continue to integrate priors, we'll find that their influence on posteriors is related to our **confidence** in the priors, which is represented by σs. The best description of the relationship between prior σ and posterior credibility is from the great philosopher, Zelda:

Priors also give us a way to generalize models at a very granular level *(parameter by parameter)*. We will see how this works shortly.

Resetting Regression Assumptions

Recall that the regression model $\hat{y} = \hat{\beta}X$ relies on assumptions about linearity, independence, heteroskedasticity and *normality* - the model can be restated as:

Figure 2: Zelda

$\hat{y} = N(\hat{\beta}X, \sigma)$

This doesn't mean that y is normally distributed *(it almost never is)*, it means that y is normally distributed **after** the model is applied *(i.e, the residuals are normally distributed)*. If we apply a simple linear model to a non-linear relationship, the residuals won't be normally distributed. So, we change the model *(polynomials, glm, splines, local regression, etc.)*.

We can also change the distribution of $\hat{y}$. You may have explored this process with log transforms and kernel transforms in machine learning. More directly, we can model to a distribution, for example the model above becomes:

$\hat{y} = SN(\hat{\beta}X, \omega, \alpha)$

If we were to build a model to predict failure of integrated circuits using the assumption that failure rate $\hat{y}$ residuals are normally distributed, it would most likely be unreliable. The same goes for transactions as they tend to cluster, usually on the lower end.

In Bayesian modeling, we need to manage the distributions of all model components - variables and parameters. For the most part, parameter estimates will follow normal distributions because of the mechanisms that model algorithms use to minimize error or maximize likelihood. And we can tweak these a bit *(e.g., using Chauchy instead of a Gaussian)*. But in most cases, that's not a difficult fit.

For Bayesian modeling, we generally restrict our ourselves to parametric *(there are 30-40 distributions in appendix A of the Stan User Guide)*. Keep in mind that this is probabilistic modeling, and the distributions are how we determine probabilities. So, we're usually better off generalizing ragged data into a distribution and producing probabilities with wide credible intervals, than going with *"very good brains"*.

Model Templates and Walkthroughs

In this section, we're going to walk through a series of Bayesian regression models, with increasing complexity, to build understanding of how these models are built. Load the following libraries and functions:

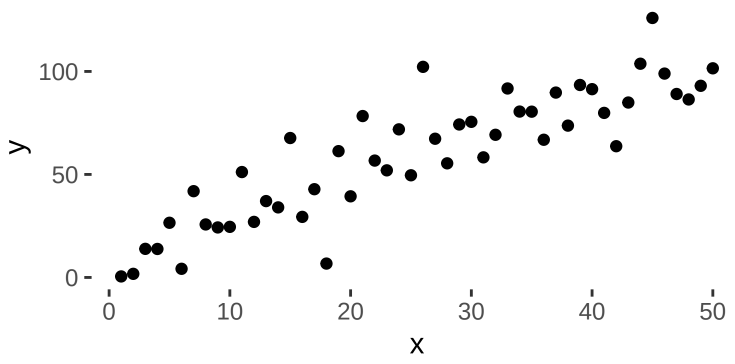
```r
library(tidyverse)
library(rstan)
library(sn)
library(lubridate)
library(stringr)

rmse <- function(error)
{
  sqrt(mean(error^2))
}

skew <- function(x,e,w,a){
  t <- (x-e)/w
  2/w * dnorm(t) * pnorm(a*t)
}

set.seed(9)
```

Basic Model

We'll start with basic regression - no priors - just bare bones template.

First generate some data:

```
# beginning assumption
n <- 50
x <- seq(from=1, to=n, by=1)
dAlpha <- 10
dBeta <- 2
noise <- rnorm(length(x), mean=0, sd=15)
y <- dAlpha + dBeta*x + noise
# put it in a dataframe
kData <- data.frame(x, y)
# build lm for baseline
# take a look
p <- ggplot(kData, aes(x, y))+geom_point() +
  theme(panel.background = element_rect(fill = "white"))
p
```

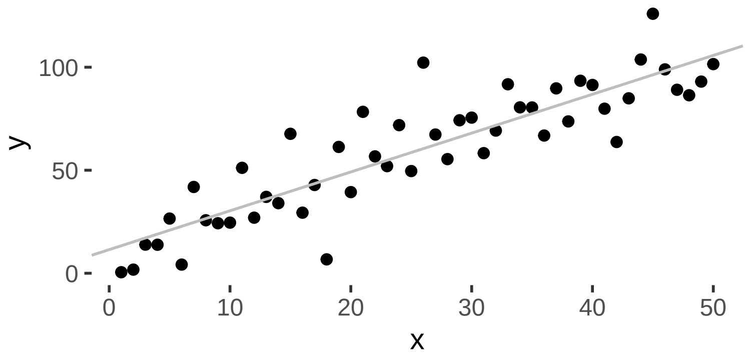


The Stan model is straight forward. We set up the parameters like we did with distributions, but this time, the mean of the model is $(alpha + beta * x)$ i.e., the linear model. So, we need to add *alpha* and *beta* to the parameters section - where we define the values we want to model to track:

```
stanMod2 <- '
data {
int<lower=0> N;
  vector[N] y;
  vector[N] x;
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
  target += normal_lpdf(y | (alpha + beta*x), sigma);
}
'
```

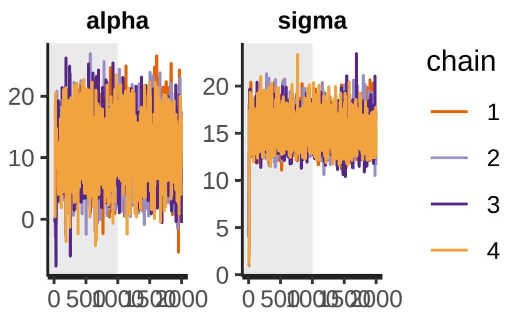
We then pass data into the model and run *stan*. The parameters are stored in the fit object *(btw, you can use any name you like)*. Then, we pull the parameter values out as shown:

```
fit <- stan(model_code = stanMod2, data = list(
  y= y,
  N=n,
  x=x),
  refresh = 0)

sumFit <- summary(fit)

p <- p + geom_abline(intercept = sumFit$summary[1,1], slope = sumFit$summary[2,1],  color = 'gray')
p
```



The sampling chains can be shown using stan_trace:

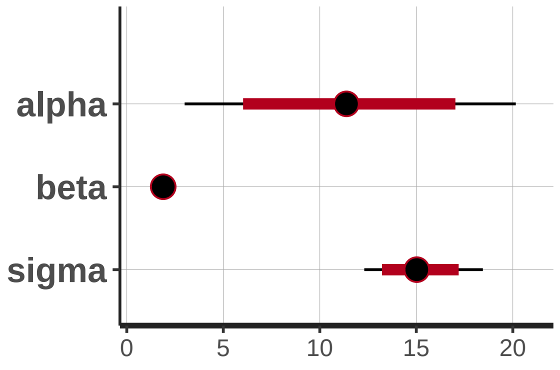
```
stan_trace(fit, inc_warmup = TRUE, pars =c("alpha","sigma"))
```



Note how the alpha trace shows a wide sample variance. We'll come back to this.

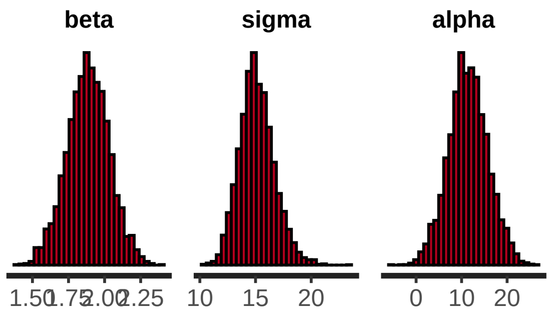
checking parameter estimates and scale *(again, notice the alpha)*:

```r
plot(fit)
```



and parameter distributions:

```r
# plot of posterior historgram for parameters
stan_hist(fit, pars =c("beta","sigma", "alpha"))
```



.. and the fit details:

```r
# print confidence intervals
print(fit, probs=c(0.025, 0.5, 0.975))
```

```
## Inference for Stan model: 0c30c8fccf3e45c755364c0384be2c92.
## 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%     50%   97.5% n_eff Rhat
## alpha   11.45    0.11 4.30    2.97   11.38   20.15  1400    1
## beta     1.89    0.00 0.15    1.59    1.89    2.18  1413    1
## sigma   15.14    0.03 1.57   12.31   15.02   18.45  2012    1
## lp__  -203.52    0.03 1.21 -206.67 -203.18 -202.14  1564    1
##
## Samples were drawn using NUTS(diag_e) at Thu Apr 09 16:27:35 2020.
## 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).
```

So, the model is a little off. This gets back to the alpha variance. Let's see if we can help it out a bit.

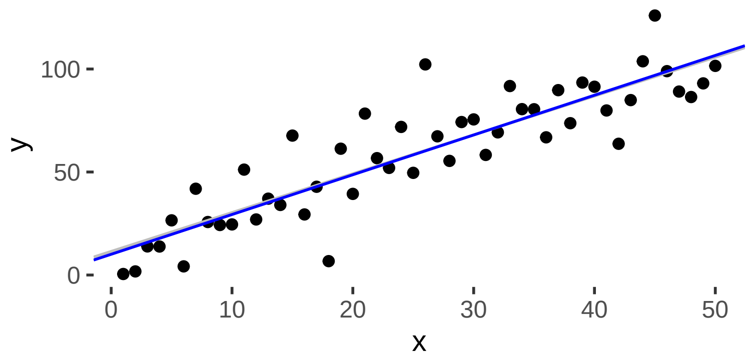
Adding Priors

Now, we'll add priors to the model. Note below that we've added an alphaMu and alphaSigma to the data section, and we set up an alpha distribution in the model section. Also in the model section, we set up priors for beta and sigma *(as you can see below, we can sometimes manually entering a value into the model, but this is NOT a good practice)*:

```
stanMod2 <- '
data {
  int<lower=0> N;
  vector[N] y;
  vector[N] x;
  real p_alpha;
  real p_alphaSigma;
  real p_beta;
  real p_betaSigma;
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
  target += normal_lpdf(y | (alpha + beta*x), sigma);
  target += cauchy_lpdf(sigma | 0, 10);
  target += normal_lpdf(alpha | p_alpha, p_alphaSigma);
  target += normal_lpdf(beta | p_beta, p_betaSigma);
}
'
```

We call *stan*, passing in the new priors *(we use the lm Alpha from earlier. We often run other models to help us find priors - then we judgmentally set the sigma).* Then plot out the new line:

```
# now fit the stan model with prior for lm

fit <- stan(model_code = stanMod2, data = list(
  y= y,
  N=n,
  x=x,
  p_alpha = 10,
  p_alphaSigma = 1,
  p_beta = 2,
  p_betaSigma = 1
  ),
  refresh = 0)

sumFit = summary(fit)
p = p + geom_abline(intercept = sumFit$summary[1,1], slope = sumFit$summary[2,1], color = 'blue')
p
```



Check parameters:

```
FitSummary = summary(fit, pars = c('alpha', 'beta'))$summary[,1]
FitSummary
```

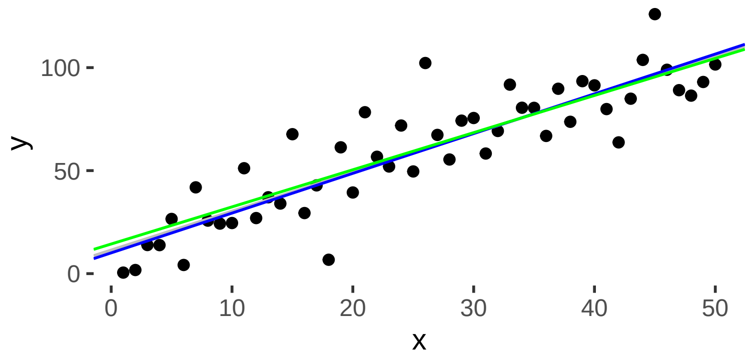
```
##     alpha      beta 
## 10.074737  1.930227
```

OK, pretty close.

Changing Priors

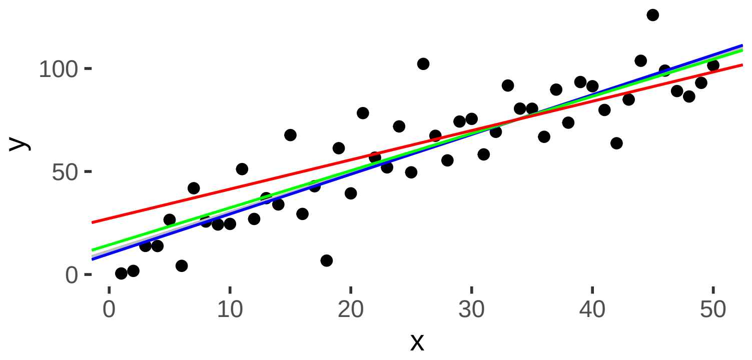
Now, let's say we know that the relationship is not realistic for the forecast period. Let's change the prior to 30. And we'll *loosen up* the sigma*, increasing to 10. Let's see how that affects the model:

```
fit <- stan(model_code = stanMod2, data = list(
  y= y,
  N=n,
  x=x,
  p_alpha = 30,
  p_alphaSigma = 10,
  p_beta = 2,
  p_betaSigma = 2
  )
  , refresh = 0)
sumFit <- summary(fit)
alpha <- sumFit$summary[1,1]
beta <- sumFit$summary[2,1]
p <- p + geom_abline(intercept = sumFit$summary[1,1], slope = sumFit$summary[2,1],  color = 'green')
p
```



OK, it pulled the alpha up a little bit, but not much because we told the model we didn't have much confidence in the prior. So, maybe we're pretty sure about this - let's tighten up the prior a bit, setting the sigma it to 2:

```r
fit <- stan(model_code = stanMod2, data = list(
  y= y,
  N=n,
  x=x,
  p_alpha = 30,
  p_alphaSigma = 2,
  p_beta = 2,
  p_betaSigma = 2
  )
  , refresh = 0)
sumFit <- summary(fit)
alpha <- sumFit$summary[1,1]
beta <- sumFit$summary[2,1]
p <- p + geom_abline(slope = beta, intercept = alpha, color = 'red')
p
```



Now if we check the fit, you can see how it pulled the alpha towards 30:

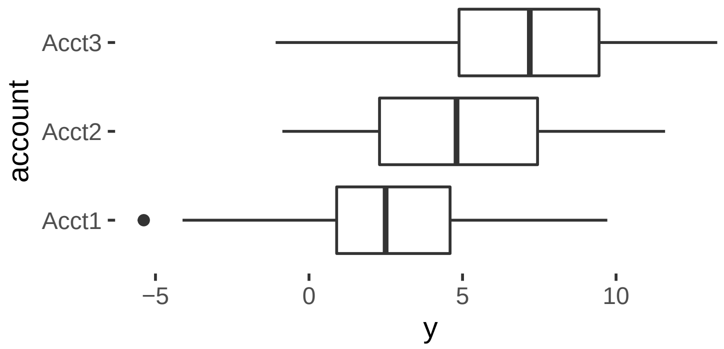
```r
# print confidence intervals
print(fit, probs=c(0.025, 0.5, 0.975))
```

```
## Inference for Stan model: bda2f4568b2da59e876fe09f17c8ba4f.
## 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%     50%   97.5% n_eff Rhat
## alpha   27.24    0.04 1.91   23.54   27.24   30.96  2519    1
## beta     1.42    0.00 0.10    1.22    1.42    1.61  2422    1
## sigma   16.87    0.04 1.79   13.80   16.75   20.87  2304    1
## lp__  -218.75    0.03 1.23 -221.95 -218.45 -217.31  1782    1
##
## Samples were drawn using NUTS(diag_e) at Thu Apr 09 16:28:23 2020.
## 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).
```

Categorical Variables

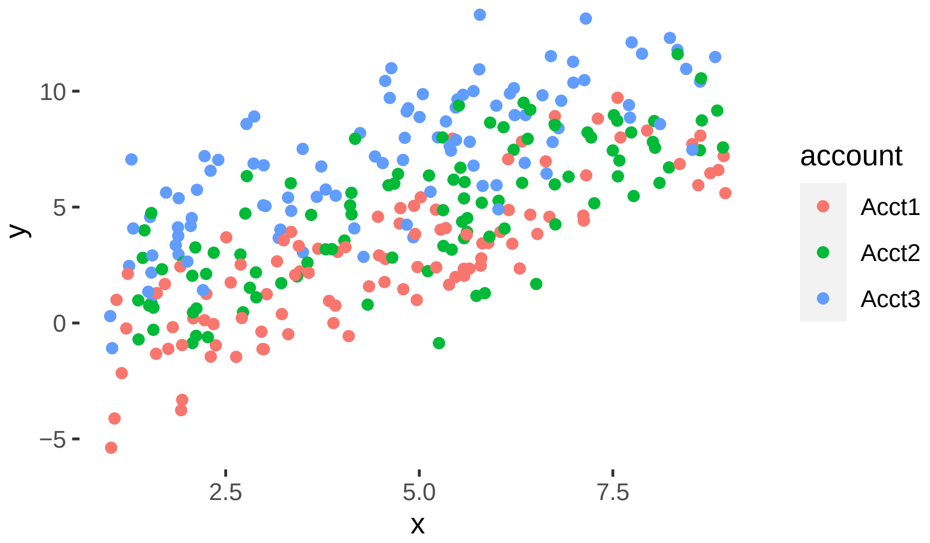
Now, let's look at categorical variables in stan. Generating data - we'll create 3 accounts with different intercepts:

```
N = 100
vAcct = c("Acct1", "Acct2", "Acct3")

N  = N*length(vAcct)
Intercept = 0
Slope = 1

likeData = data.frame(account = vAcct,
                PIntercept = c(Intercept-2, Intercept, Intercept+2),
                x = runif(N, min = 1, max = 9)) %>%
                mutate(y = PIntercept + (Slope * x) + rnorm(N, 0, sd = 2))

likeData$account <- as.factor(likeData$account)

ggplot(likeData, aes(x=account, y=y))+
  geom_boxplot() +
  theme(panel.background = element_rect(fill = "white")) +
  coord_flip()
```



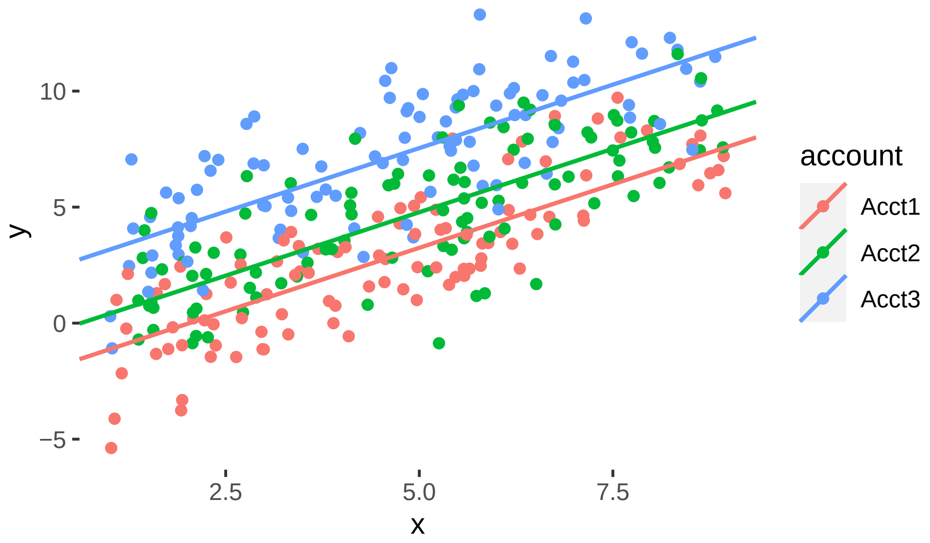
Another visualization:

```
p <- ggplot(likeData, aes(x, y, color = account))+geom_point()+
  theme(panel.background = element_rect(fill = "white"))
p
```



Baselining with lm:

```
lmMod <- lm(y ~ x + account, data=likeData)

lmCoef = data.frame(account = unique(likeData$account),
                    Intercept = c(lmMod$coefficients[1],
                                  lmMod$coefficients[1]+lmMod$coefficients[3],
                                  lmMod$coefficients[1]+lmMod$coefficients[4]),
                    Slope = c(lmMod$coefficients[2], lmMod$coefficients[2], lmMod$coefficients[2])
)

p = p +   geom_abline(data = lmCoef, aes(intercept = Intercept, slope = Slope, color = account),
                        size = .75)
p
```



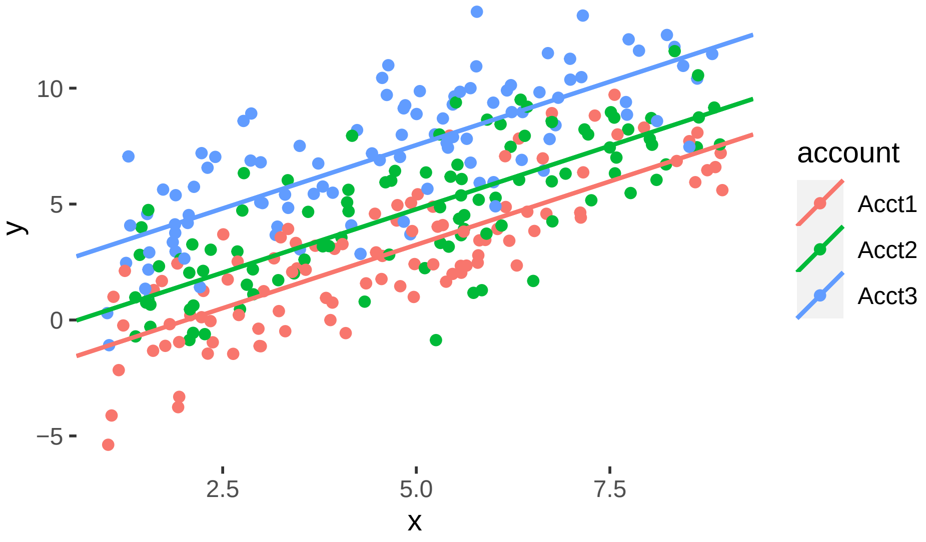
OK, let's start with the previous template *(increasing in X and beta dimensions)*:

```
stanModel <- '
data {
int<lower=0> N;
int<lower=0> K;
matrix[N, K] x; // matrix now
vector[N] y;
}
parameters {
real alpha;
vector[K] beta; // vector now
real<lower = 0> sigma;
}
model{
target += normal_lpdf(y | x * beta + alpha, sigma);
}
'
```

```
Xmatrix <- model.matrix(y ~ x + account, likeData)[,-1]
# we drop the intercept from the xMatrix [,-1]
# because we're seperating alpha here

stanData <- list(
  N=nrow(likeData),
  K=ncol(Xmatrix),
  y=likeData$y,
  x=Xmatrix
)

fit <- stan(model_code = stanModel, data = stanData, refresh = 0)
```

And pull the parameters:

```
sumFit <- summary(fit)

stanCoef = data.frame(account = unique(likeData$account),
                    Intercept = c(sumFit$summary[1,1],
                                  sumFit$summary[1,1]+ sumFit$summary[3,1],
                                  sumFit$summary[1,1]+ sumFit$summary[4,1]),
                    Slope = c(sumFit$summary[2,1], sumFit$summary[2,1], sumFit$summary[2,1])
)

p = p +   geom_abline(data = stanCoef, aes(intercept = Intercept, slope = Slope, color = account),
                        size = .75)
p
```



Let's take a look at those parameters:

```r
print(fit, probs=c(0.025, 0.5, 0.975))
```

```
## Inference for Stan model: 8596002261f6666af9a5ff0d54d90e0c.
## 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%     50%   97.5% n_eff Rhat
## alpha     -2.22    0.01 0.30   -2.82   -2.23   -1.65  2385    1
## beta[1]    1.09    0.00 0.05    1.00    1.09    1.19  2849    1
## beta[2]    1.54    0.01 0.27    0.99    1.53    2.08  2878    1
## beta[3]    4.31    0.01 0.28    3.77    4.31    4.86  2945    1
## sigma      1.93    0.00 0.08    1.78    1.93    2.10  3763    1
## lp__    -621.49    0.04 1.61 -625.46 -621.17 -619.41  1687    1
##
## Samples were drawn using NUTS(diag_e) at Thu Apr 09 16:29:10 2020.
## 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).
```

We created a parameter with the base intercept *(alpha)* and the beta[1] is the slope of x. Then beta[2] and Then beta[3] are the effects, or adjustments to the base intercept. You're probably used to this approach from lm, except it's a little out of order.

Normal Equation

Let's back up and run this data through normal equations:

```r
vY <- as.numeric(likeData$y) # set up y values in matrix
mX = model.matrix(y ~ account + x, likeData)
vBeta <- solve(t(mX)%*%mX, t(mX)%*%vY) # solve using normal equations
vBeta
```

```
##                   [,1]
## (Intercept)  -2.223548
## accountAcct2  1.534534
## accountAcct3  4.297882
## x              1.093697
```

```
vBeta <- as.numeric(vBeta)
y_hat <- t(vBeta%*%t(mX))
```

The following model uses the normal equation template, storing all the parameters in a beta vector. I find this more efficient and easier to work with if the model is simple *(no levels)*. The model:

```
stanModel1 <- '
data {
  int<lower=0> N;
  int<lower=0> K;
  matrix[N, K] x;
  vector[N] y;
  vector[K] betaPriors;
}
parameters {
  vector[K] beta;
}
model{
  target += normal_lpdf(beta | betaPriors, 1);
  target += normal_lpdf(y | x*beta, 1);
}
'
```

```
Xmatrix <- model.matrix(y ~ x + account, likeData)

stanData1 <- list(
  N=nrow(Xmatrix),
  K=ncol(Xmatrix),
  x=Xmatrix,
  y=vY,
  betaPriors  = vBeta
)

fit1 <- stan(model_code = stanModel1, data = stanData1, refresh = 0)
FitSummary1 = data.frame(summary(fit1, pars = 'beta')$summary[,1])
FitSummary1
```

```
##         summary.fit1..pars....beta...summary...1.
## beta[1]                                 -2.216704
## beta[2]                                  1.092746
## beta[3]                                  1.561572
## beta[4]                                  4.263285
```

(when you change the model, pay attention to the order of the parameters!)

When you set up a single beta vector, it's usually easier to pass priors in:

```
stanModel2 <- '
data {
  int<lower=0> N;
  int<lower=0> K;
  matrix[N, K] x;
  vector[N] y;
  vector[K] betaPriors;
}
```

```
parameters {
  vector[K] beta;
}
transformed parameters{
  vector[N] linPred;
  linPred = x*beta;
}
model{
  target += normal_lpdf(beta | betaPriors, 1);
  target += normal_lpdf(y | linPred, 1);
}
'
```

```
stanData2 <- list(
  N=nrow(Xmatrix),
  K=ncol(Xmatrix),
  x=Xmatrix,
  y=vY,
  betaPriors  = vBeta
)

fit2 <- stan(model_code = stanModel2, data = stanData2, refresh = 0)
FitSummary2 = summary(fit2, pars = 'beta')$summary[,1]
FitSummary2
```

```
##   beta[1]   beta[2]   beta[3]   beta[4]
## -2.219500  1.093591  1.558288  4.260545
```

More Complex Models

Alas, simple models don't get you very far, and we're going to need more flexible templates. The template below will accommodate a wider application range *(although we'll start with a simple one)*. The transformed parameters section provides a way to consolidate and perform operations on parameters prior to sampling *(which improves efficiency)*. Also notice the subsetting of alpha[account[i]]. This is a precursor to multilevel and grouping - it tells the model to fit alpha parameters by group.

```
stanModel3 <- '
data {

  int<lower=0> N;
  vector[N] y;

  vector[N] x;
  int account[N];

  int<lower=0> J; // Account Grps
  real alphaPriors[J];
  real betaPriors;
}

parameters {

  real<lower = 0> sigma;
  vector[J] alpha;
  real beta;
}

transformed parameters {

  vector[N] y_hat;
  for (i in 1:N)
  y_hat[i] = alpha[account[i]] + beta * x[i];

}

model {
  target += normal_lpdf(y | y_hat,  sigma);
  target += normal_lpdf(alpha | alphaPriors, 1);
  target += normal_lpdf(beta | betaPriors, 1);
}
'
```

Note we're grouping the intercept priors in the alpha vector *(so 3 priors)*. This is an easier approach - but be careful on datatypes - stan won't accept anything but numeric. Also note that we're passing many parameters in directly from the dataframe *(careful with datatypes)*

```
stanData3 <- list(
  N=nrow(likeData),
  J=length(unique(likeData$account)),
  y=likeData$y,
  x=likeData$x,
  account=as.integer(likeData$account),
  alphaPriors = vBeta[1:3],
  betaPriors = vBeta[4]
)

fit3 <- stan(model_code = stanModel3, data = stanData3, refresh = 0)
FitSummary3 = summary(fit3, pars = c('alpha', 'beta'))$summary[,1]
FitSummary3
```

```
##   alpha[1]   alpha[2]   alpha[3]       beta
## -2.0177933 -0.3874054  2.3583642  1.0473910
```

QR Decomposition

When we're sampling large populations of transactions, the QR decomposition will make a huge difference in computing times:

```
stanModel4 <- '
data {
  int<lower=0> N;
  int<lower=0> K;
  matrix[N, K] x;
  vector[N] y;
  vector[K] betaPriors;
}
// set up qr decomp

transformed data {
  matrix[N, K] Q_ast;
  matrix[K, K] R_ast;
  matrix[K, K] R_ast_inverse;

  // thin and scale the QR decomposition

  Q_ast = qr_Q(x)[, 1:K] * sqrt(N - 1);
  R_ast = qr_R(x)[1:K, ] / sqrt(N - 1);
  R_ast_inverse = inverse(R_ast);

}
parameters {
  vector[K] beta;      // coefficients on Q_ast
}
model {
  target += normal_lpdf(beta | betaPriors, .1);
  target += normal_lpdf(y | Q_ast * beta, .1);
}
'
```

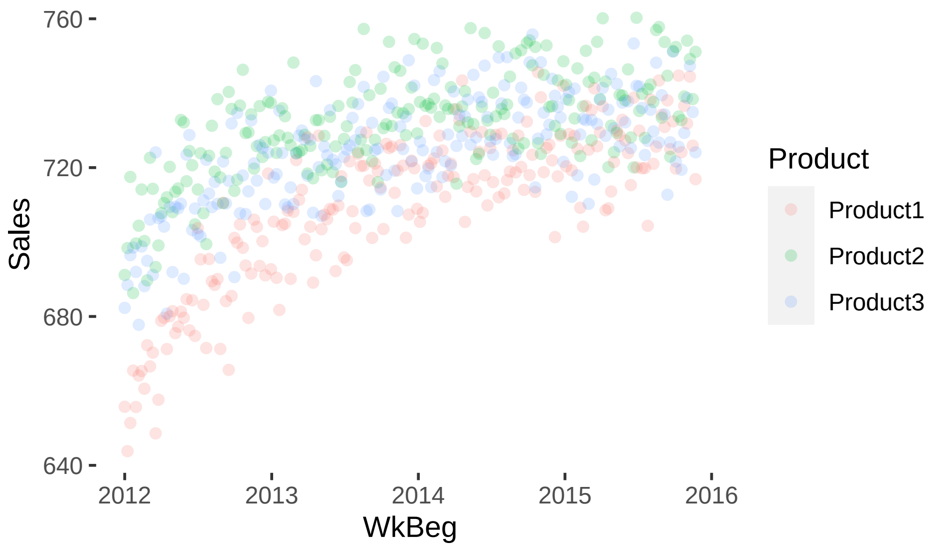
```
stanData4 <- list(
  N=nrow(Xmatrix),
  K=ncol(Xmatrix),
  x=Xmatrix,
  y=vY,
  betaPriors  = c(4, 2, -2, 1)
)

fit4 <- stan(model_code = stanModel4, data = stanData4, refresh = 0)
FitSummary4 = summary(fit4, pars = c('beta'))$summary[,1]
FitSummary4
```

```
##    beta[1]    beta[2]    beta[3]    beta[4]
##  4.8803042  2.3721688 -0.2960808  1.7550609
```

Polynomials and Simple Generalization

OK, let's do an exercise you're all familiar with. Product production curves. First, the data:

```
ProductSales = read_csv("C:/Users/ellen/Documents/UH/Spring 2020/DA2/Section 1/MIdTerm/ProductSalesv2.c

ProductSales$WkBeg = mdy(ProductSales$WkBeg)

ProductSales = pivot_longer(ProductSales, 3:5, names_to = "Product", values_to = "Sales")

p <- ggplot(data = ProductSales, aes(WkBeg, Sales, color = Product)) + geom_point(alpha = .2) +
  theme(panel.background = element_rect(fill = "white"))
p
```



Now, a solution using normal equations *(polynomial)* for baseline rmse:

```
Train = ProductSales %>% filter(WkBeg < "2015-01-01")
Test  =  ProductSales %>%  filter(WkBeg >= "2015-01-01")

mXPoly = model.matrix(Sales ~ Product + Wk + I(Wk^2), data = Train)
vY = as.numeric(Train$Sales)
vBetaPoly <- solve(t(mXPoly)%*%mXPoly, t(mXPoly)%*%vY) # solve using normal equations
yPoly = t(as.numeric(vBetaPoly)%*%t(mXPoly))
mXPolyTest = model.matrix(Sales ~ Product + Wk + I(Wk^2), data = Test)
rmse4 = rmse( Test$Sales - (t(as.numeric(vBetaPoly)%*%t(mXPolyTest))))
rmse4
```

```
## [1] 13.88266
```

Now, let's build a polynomial model in stan:

```
stanModel6 <- '
data {
  int<lower=0> N;
  int<lower=0> K;
  matrix[N, K] x;
  vector[N] y;
  vector[K] betaPriors;
  vector[K] sigmaPriors;
  real<lower = 0> mSigma;
}
parameters {
  vector[K] beta;
}
model{
  target += normal_lpdf(beta | betaPriors, sigmaPriors);
  target += normal_lpdf(y | x*beta, mSigma);
}
'
```

Bayesian generalization is not like other machine learning approaches *(which generalizes like a blanket - the greater the magnitude of a parameter, the more generalization effect)*. In Bayesian modeling, you use

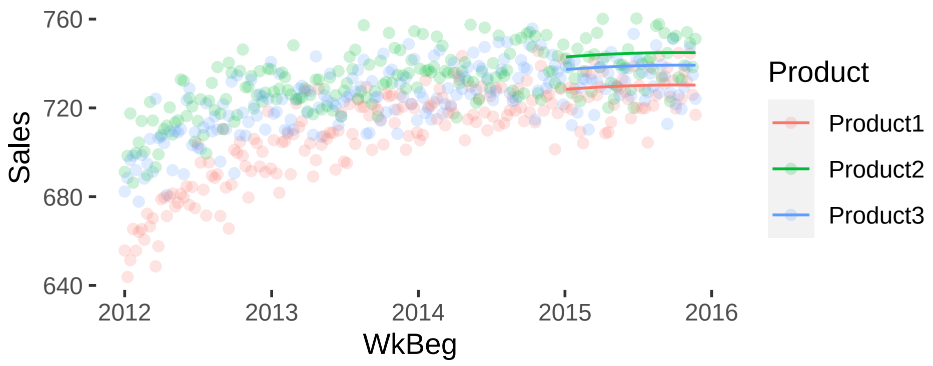
Priors to control generalization on a parameter by parameter basis. This is REALLY powerful in multilevel modeling, but it also can tune single level models like the one here. I start with the parameters from our normal equation, and then tweak the priors to tune the model. In this case, I can see the data do not follow a polynomial function, rather, it follows a typical production curve where it curves in the early stages and then levels off:

```r
vBetaPolyPriors = c(
  round(vBetaPoly[1]+10,0),
  round(vBetaPoly[2]-12,0),
  round(vBetaPoly[3]-10,),
# here, I'm "squeezing" the intercepts
# so the curve straightens out
  round(vBetaPoly[4],0),
  round(vBetaPoly[5]+.001,3))
# and here, I'm reducing the effect of the exponentiation
# the parameter was -0.002... so I added back a little
  vSigmaPriors = c(1,  1,   1,   1,  0.0001)
# then I "clamp down" the sigmas
# while leaving the model sigma some room to move
  modelSigma = 10

stanData6 <- list(
  N=nrow(mXPoly),
  K=ncol(mXPoly),
  x=mXPoly,
  y=vY,
  betaPriors  = vBetaPolyPriors,
  sigmaPriors = vSigmaPriors,
  mSigma = modelSigma
)

fit6 <- stan(model_code = stanModel6, data = stanData6, refresh = 0)
FitSummary6 = as.numeric(summary(fit6, pars = 'beta')$summary[,1])
#FitSummary6

Test$Pred =  as.numeric(t(as.numeric(FitSummary6)%*%t(mXPolyTest)))
rmse(Test$Pred - Test$Sales)
```

```
## [1] 11.26525
```

And visualizing *(you can see how it reduces the polynomial effect)*:

```r
p6 = p +  geom_line(data = Test, aes(WkBeg, Pred, color = Product))
p6
```



Nonlinear Equations

Production functions are often follow some variation of an inverse function: $\frac{(a*x)}{(b+x)}$. If you're interested you can go to https://www.desmos.com/ and create function plots *(it's really pretty cool!)*

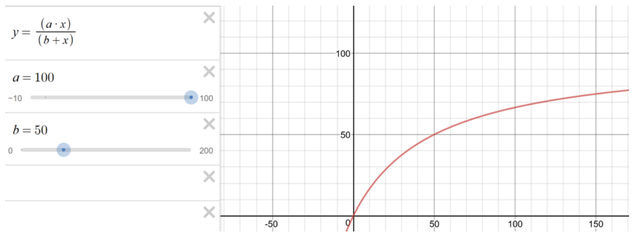


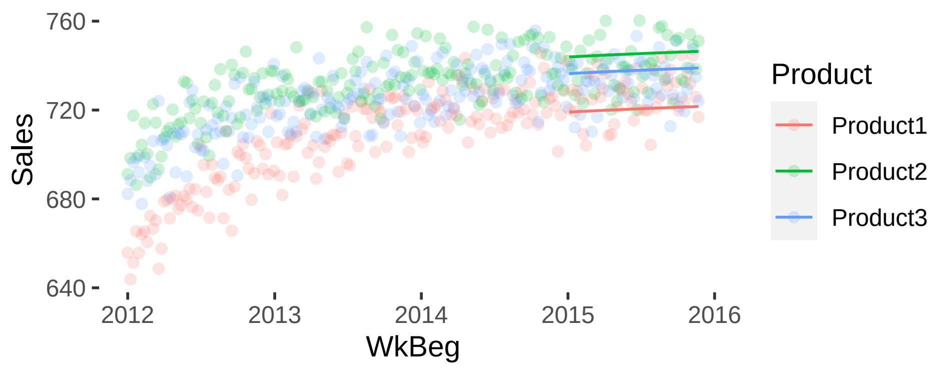
Figure 3: SalesForecast

Stan is really unlimited with how you can build your models. Below, I use a production function *(which is non-linear, so we've left linear regression now)*:

```
stanModel7 <- '
data {

  int<lower=0> N;
  vector[N] y;
  vector[N] x;
  int product[N];
  int<lower=0> J; // Product Grps
  real alphaPriors[J];
  real p_a;
  real p_b;
}

parameters {
  real a;
  real b;
  real<lower = 0> sigma;
  vector[J] alpha;
}

transformed parameters {

  vector[N] y_hat;
  for (i in 1:N)
  y_hat[i] = alpha[product[i]] + ((a*x[i])/(b+(x[i])));

}

model {
  target += normal_lpdf(y | y_hat,  sigma);
  target += normal_lpdf(alpha | alphaPriors, 50);
  target += normal_lpdf(a | p_a, 2);
  target += normal_lpdf(b | p_b, 2);
}
'
```

Running the model:

```
ModelData = Train
ModelData$Product = as.integer(factor(ModelData$Product))

stanData7 <- list(
  N=nrow(ModelData),
  J=length(unique(ModelData$Product)),
  y=ModelData$Sales,
  x=ModelData$Wk,
  product=as.integer(ModelData$Product),
  alphaPriors = c(650,660,670),
  p_a = 80,
  p_b = 30
)

fit7 <- stan(model_code = stanModel7, data = stanData7, refresh = 0)
sumFit7 = summary(fit7, pars = c('alpha', 'a', 'b'))$summary[,1]

Prod1 = filter(Test, Product == "Product1")
Prod2 = filter(Test, Product == "Product2")
Prod3 = filter(Test, Product == "Product3")
Prod1$Pred = sumFit7[1] + (sumFit7[4]*Prod1$Wk)/(sumFit7[5]+Prod1$Wk)
Prod2$Pred = sumFit7[2] + (sumFit7[4]*Prod1$Wk)/(sumFit7[5]+Prod1$Wk)
Prod3$Pred = sumFit7[3] + (sumFit7[4]*Prod1$Wk)/(sumFit7[5]+Prod1$Wk)

NewPred = bind_rows(Prod1, Prod2)
NewPred = bind_rows(NewPred, Prod3)

p7 = p + geom_line(data = NewPred, aes(WkBeg, Pred, color= Product))
p7
```



```r
rmse(NewPred$Pred - NewPred$Sales)
```

```
## [1] 11.57516
```

And we could tweak this with priors too! So, very, very powerful.

Non-Normal Data Distributions

The normal distributions dominates in statistic classes, and the skew normal dominates in the transaction world. Disconnect?

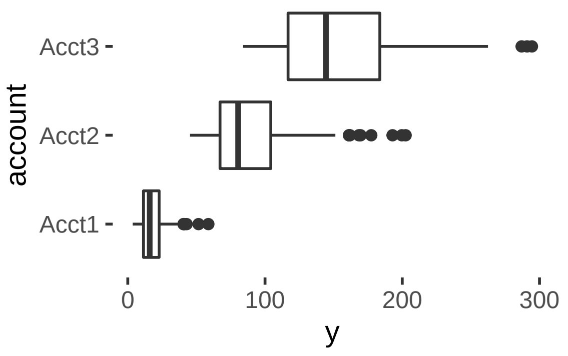
How do we model the SN in a regression model? Stan provides a range of approaches. The model below is **NOT** the preferred method *(modeling indirect effects is the way to go - for another day)*, but it should give you an idea of Stan extensibility.

Generate some SN data:

```r
N = 100
vAcct = c("Acct1", "Acct2", "Acct3")

N  = N*length(vAcct)
Intercept = 0
Slope = 2

likeDataSN = data.frame(account = vAcct,
                      PIntercept = c(Intercept, Intercept+40, Intercept+80),
                      x = runif(N, min = 1, max = 9)) %>%
              mutate(xi = PIntercept + (Slope * x),
                     omega = (PIntercept + (Slope * x)),
                     alpha = (PIntercept + (Slope * x))) %>%
              mutate(y = rsn(N, xi = xi, omega = omega, alpha = alpha))

likeDataSN$account <- as.factor(likeDataSN$account)
```

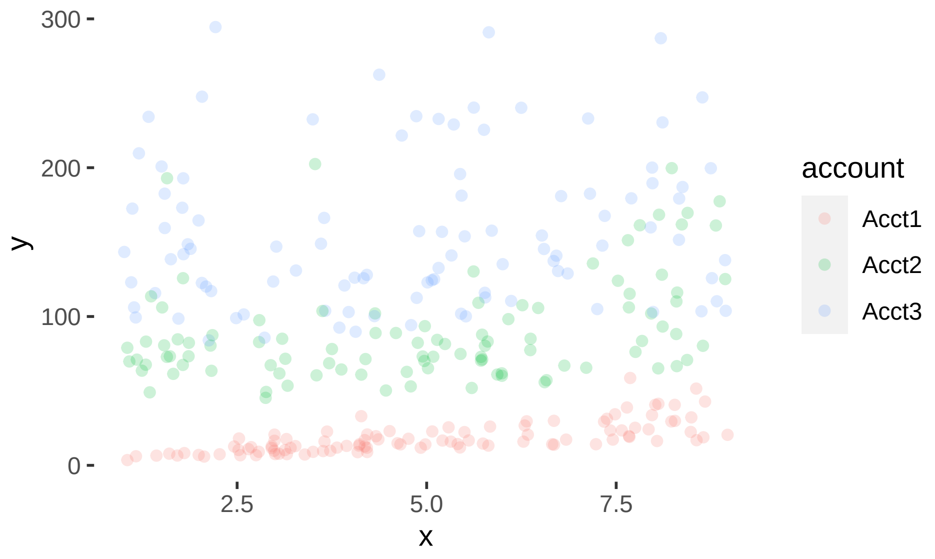
Take a look - boxplot:

```r
ggplot(likeDataSN, aes(x=account, y=y))+
  geom_boxplot() +
  theme(panel.background = element_rect(fill = "white")) +
  coord_flip()
```



and scatter:

```r
p8 = ggplot(likeDataSN, aes(x=x, y=y, color = account))+
  geom_point(alpha = .2) +
  theme(panel.background = element_rect(fill = "white"))
p8
```

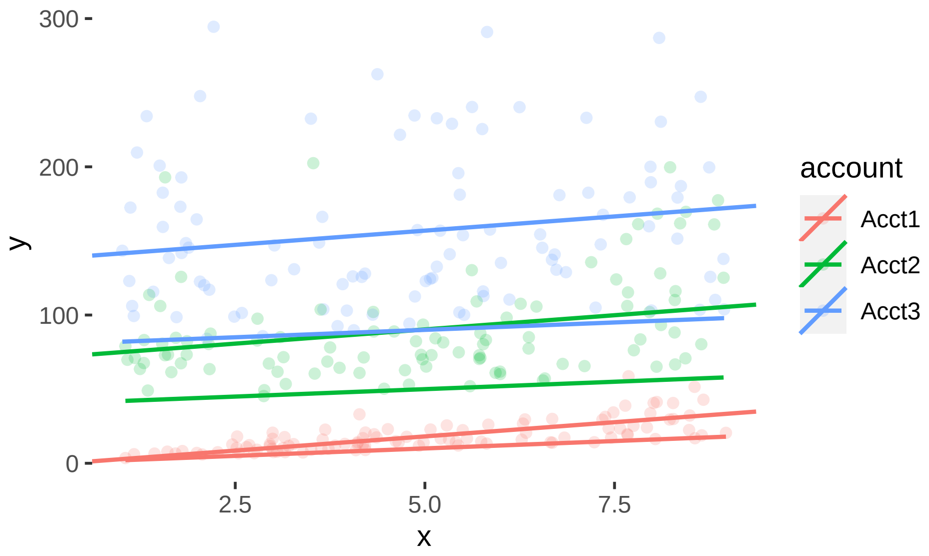


Notice how each account has a cluster of transaction amounts at the bottom and then flames up and disperses. Typical of a skew normal, and typical of transactions in business.

Let's baseline this in lm and visualize the lm prediction vs the true location values:

```r
mod8 = lm(y ~ account + x, likeDataSN)
lmCoef = data.frame(account = unique(likeDataSN$account),
  Intercept = c(mod8$coefficients["(Intercept)"],
                mod8$coefficients["(Intercept)"]+mod8$coefficients["accountAcct2"],
                  mod8$coefficients["(Intercept)"]+mod8$coefficients["accountAcct3"]),
  Slope = c(mod8$coefficients["x"],mod8$coefficients["x"],mod8$coefficients["x"])
  )


p8 = p8 +   geom_abline(data = lmCoef, aes(intercept = Intercept, slope = Slope, color = account),
                      size = .75)
p8 = p8 + geom_line(data = likeDataSN, aes(x, xi, color = account), size = .75)
```

```
p8
```



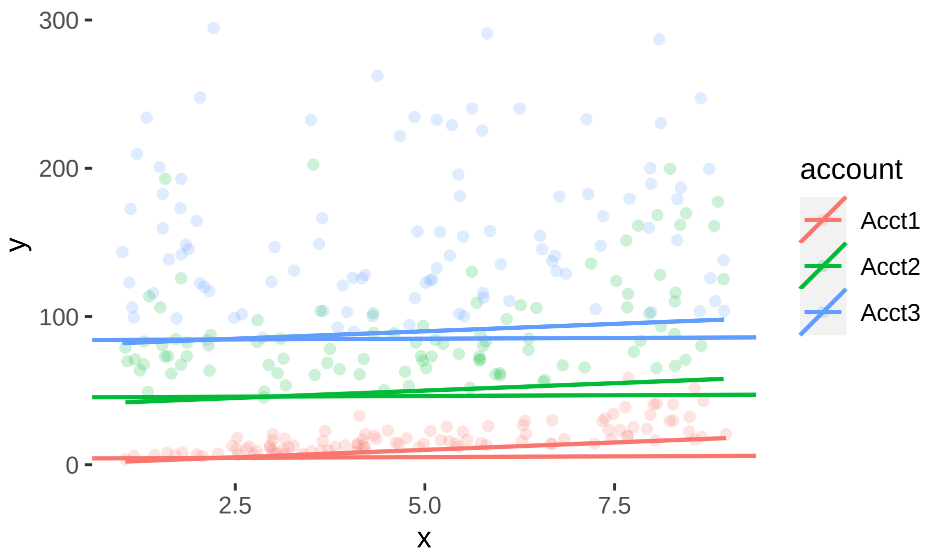
Notice how far off the lm $\hat{y}$ is from the xi location value *(especially Acct3, where scale and skew are greatest)*

Now, we'll pull the priors from the gen parameters *(yes, this is kinda cheating, but this is a theory demo)*. And set up the model as follows *(note that it's really not all that different - we've just changed the distribution where the model fits the parameters to the likelihood data - and we had to add* ω and α parameters and priors:

```
priors = likeDataSN %>% group_by(account) %>% summarise(xi = mean(xi), omega = mean(omega), alpha  = mea

xiPriors = as.numeric(priors$xi)
omegaPriors = as.numeric(priors$omega)
alphaPriors = as.numeric(priors$alpha)


stanModel8 <- '
data {

  int<lower=0> N;
  vector[N] y;

  vector[N] x;
  int account[N];

  int<lower=0> J; // Account Grps
  real alphaPriors[J];
  real betaPriors;

  real xiPriors[J];
  real omegaPriors[J];
  real snAlphaPriors[J];

}

parameters {
  vector[J] alpha;
  real beta;
  real<lower=0, upper = 100> omega[J];
  real<lower=0, upper = 100> snAlpha[J];
}

transformed parameters {

  vector[N] y_hat;
  for (i in 1:N)
  y_hat[i] = alpha[account[i]] + beta * x[i];

}

model {
  target += skew_normal_lpdf(y | y_hat[account],  omega[account], snAlpha[account]);
  target += normal_lpdf(alpha | alphaPriors, 10);
  target += normal_lpdf(beta | betaPriors, 1);

  target += normal_lpdf(omega | omegaPriors, 10);
  target += normal_lpdf(snAlpha | snAlphaPriors, 10);

}
'

stanData8 <- list(
  N=nrow(likeDataSN),
  J=length(unique(likeDataSN$account)),
  y=likeDataSN$y,
  x=likeDataSN$x,
  account=as.integer(likeDataSN$account),
  alphaPriors = c(10, 70, 140),
  betaPriors = mod8$coefficients[4],
  xiPriors = xiPriors,
  omegaPriors = omegaPriors,
  snAlphaPriors = alphaPriors
)

fit8 <- stan(model_code = stanModel8, data = stanData8, refresh = 0)
```

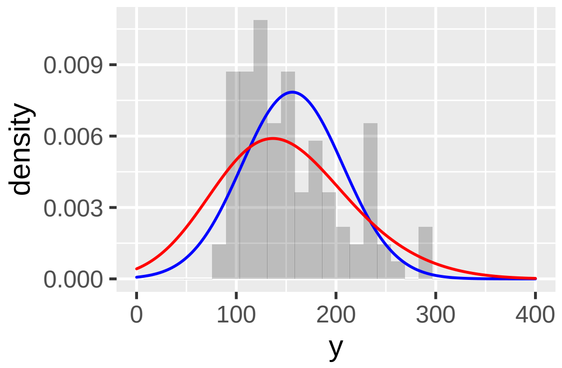
Let’s take a look:

```
FitSummary8 = as.numeric(summary(fit8, pars = c('alpha', 'beta', 'omega', 'snAlpha'))$summary[,1])

p9 = ggplot(likeDataSN, aes(x=x, y=y, color = account))+
  geom_point(alpha = .2) +
  theme(panel.background = element_rect(fill = "white"))

p9 = p9 + geom_line(data = likeDataSN, aes(x, xi, color = account), size = .75)

snCoef = data.frame(account = unique(likeDataSN$account),
                    Intercept = c(FitSummary8[1],
                                  FitSummary8[2],
                                  FitSummary8[3]),
                    Slope = c(FitSummary8[4], FitSummary8[4], FitSummary8[4])
)

p9 = p9 +   geom_abline(data = snCoef, aes(intercept = Intercept, slope = Slope, color = account),
                        size = .75)
p9
```



And comparing models. The important point here is what's missed. Let's say we're modeling transactions for audit reasonableness and we used the frequentist approach. There's a large group of transactions that would be classified as exceptions *(which means we go do unnecessary boring substantive testing, we generate excess fees, and generally piss off the client)*. Know your data, know your distributions, and know how models work.

```
Acct3Data = filter(likeDataSN, account == "Acct3")

snPred = rsn(nrow(Acct3Data), mean(FitSummary8[3] + Acct3Data$x*FitSummary8[4]),FitSummary8[7], FitSumma


p8 = ggplot(Acct3Data, aes(x = y, y = ..density..)) +
 geom_histogram(bw = 10, fill = "black", alpha = .2) +
 geom_density(aes(predict(mod8, Acct3Data)), bw = 50, color = "blue") +
 geom_density(aes(snPred), bw = 50, color = "red") +
 xlim(0, 400)
p8
```



Homework.

You have a pretty decent menu of model templates from which to work now. Go out an find some data and apply 2 different Bayesian models. Summarize findings.