

Foundations for Bayesian Analysis

This document provides an introduction (*or review, depending on prereqs*) to the concepts and skills foundational for Bayesian Analysis. There are two major sections: **Conditional Probability in Regression Analysis** and **Foundations for Optimization**:

Understanding Conditional Probability

Before getting into Conditional Probability in Regression Analysis, let's preface with a caveat about regression and distributions: Regression variables can be any distribution, but the residuals ($y - \hat{y}$) should be normally distributed (*remember, \hat{y} is the result of model application (e.g., $b_0 + b_1x$ in a simple linear case).* **Data variables in transaction environments are seldom normally distributed**, and neither application of the model, nor sampling will transform these distributions to normal distributions (*recall that the central limit theorem applies to sampling means - not population parameters, and you can't use a sample mean to project a population distribution when the population is not normally distributed (and they seldom are).*

We will get into skewed distributions shortly, but for this exercise, we'll force normal distributions. With this in mind, let's look at how the distribution of y changes when we set conditions on x .

Load the following libraries:

```
library(tidyverse)
library(stringr)
library(lubridate)
library(kableExtra)
library(cowplot)
library(ggExtra)
library(sfsmisc)
library(janitor)
```

.. and generate some data (*using discrete variables here to add a little clarity*), and create a plot where we can see both the regression and the distributions:

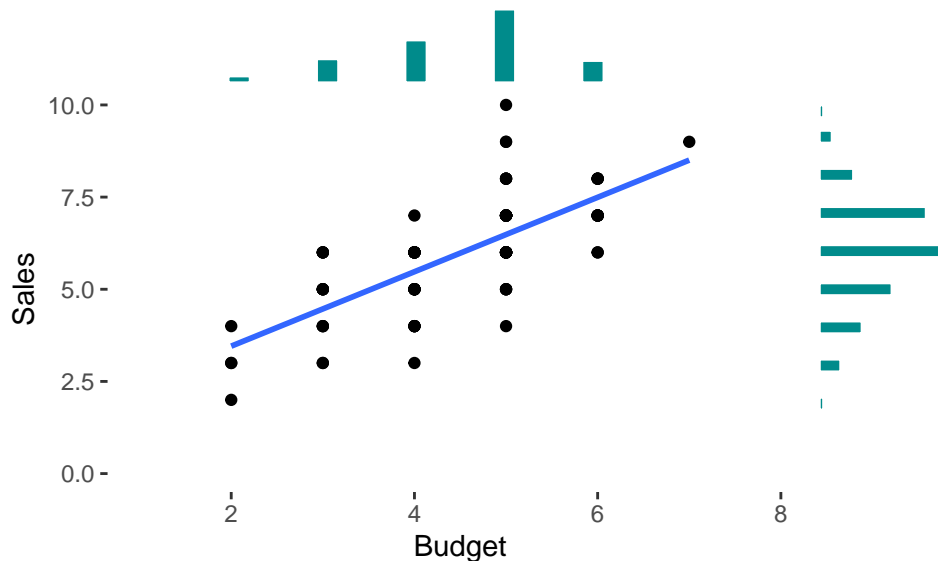
```
set.seed(222)

intercept = 2
slope = 1
N = 100

# generate discrete data and apply distribution AFTER the regression equation
ExampleData1 = data.frame(x = as.integer((rnorm(N, 5, sd = 1)))) %>%
  mutate(y = as.integer((intercept + (slope * x)) + rnorm(N, 0, sd = 1)))

# plot using ggextra format
plot_center = ggplot(ExampleData1, aes(x=x,y=y)) +
  geom_point(width = .05) +
  geom_smooth(method="lm", se = F) +
  theme(panel.background = element_rect(fill = "white")) +
  xlim(1, 8) + ylim(0, 10) +
  ylab("Sales") + xlab("Budget")
```

```
p1 = ggMarginal(plot_center, type="histogram", fill = "cyan4", color = "white")
p1
```



..and gathering some metrics we'll use later:

```
mod = lm(y~x, data = ExampleData1)
muX = mean(ExampleData1$x)
muY = round(mean(ExampleData1$y),1)
sigmaX = round(sd(ExampleData1$x),1)
sigmaY = round(sd(ExampleData1$y),1)
```

OK, in the plot above, we can see y (lets call it Sales in M's to keep it real) and x (let's call it Advertising in K's), with a linear relationship determining the **mean** of y , defined by $b_0 + b_1x$ for each value of x . So, the model is **parameterized** with two coefficients (b_0, b_1). The overall distributions of x and y are shown on the margins. (keep in mind that values can repeat, so one point could include many observations - therefore, the points and histograms may not appear to match up - but they do).

Knowing the **mean** for a specific value of x is a good start, and we can get the variance of the distribution of y from the standard error. But what if we want to know the **probability** for a range of y , given a specific value for x ? For example: What's the probability that sales = 5M, given the advertising budget = 3k? (do you think a business manager might ask this type of question?). This is a conditional probability, and can be written, generally, as a joint occurrence of random variables, or **joint probability density function**, which (in continuous case) can be defined as:

$$P(y_1 \leq y \leq y_2, x_1 \leq x \leq x_2) = \int_{y_1}^{y_2} \int_{x_1}^{x_2} p(x, y) dx dy$$

So in our example (simplified temporarily by the discrete variables), the pmf could be stated as:

$$P(y = 5|x = 3) = \Pi(y = 5|x = 3)$$

And we can get a **conditional mean and variance** of y by **averaging** the conditional **mean** over the *marginal* distribution of x where the inner expectation averages over y , conditional on x , and the outer expectation averages over x (BDA formula 1.8)

$$E(y) = E(E(y|x))$$

This sounds fancy, but really, it's just a mean weighted by the joint probability of the variables, which, in R, can be estimated using a weighted.mean function:

$$EY = \text{weighted.mean}(y, \text{conditional.density})$$

And similarly, the conditional variance can be determined by the mean of the conditional variance and the variance of the conditional mean, which can be estimated with another weighted mean with the densities:

$$CV = \text{weighted.mean}((y - EY), \text{conditional.density})$$

(these are shortcuts to the precise methods outlined in BDA, which is painfully tedious. We really don't have to do this my hand, so just work trough this a few times, and we'll introduce other methods)

We'll create a conditional probability table here (review C1_Classification_LogReg.pptx if needed)

```
AnalysisSummary1 = ExampleData1 %>% group_by(x,y) %>%
  summarise(Cnt = n(), Prob = round(Cnt/N,2))

MarAnalysis = AnalysisSummary1 %>%
  select(y, x, Prob) %>%
  pivot_wider(names_from = x, values_from = Prob) %>%
  adorn_totals("row") %>%
  adorn_totals("col")

knitr::kable(MarAnalysis) %>%
  kable_styling(full_width = F, bootstrap_options = "striped", font_size = 9)
```

y	2	3	4	5	6	7	Total
2	0.01	NA	NA	NA	NA	NA	0.01
3	0.02	0.02	0.01	NA	NA	NA	0.05
4	0.01	0.03	0.05	0.01	NA	NA	0.10
5	NA	0.05	0.08	0.04	NA	NA	0.17
6	NA	0.04	0.10	0.14	0.02	NA	0.30
7	NA	NA	0.01	0.17	0.07	NA	0.25
8	NA	NA	NA	0.04	0.04	NA	0.08
9	NA	NA	NA	0.02	NA	0.01	0.03
10	NA	NA	NA	0.01	NA	NA	0.01
Total	0.04	0.14	0.25	0.43	0.13	0.01	1.00

To get a visual, we can then use these parameters to generate the conditional distribution of y , given $x = 3k$, as illustrated below:

```
# filter for advertising budget of 3k
AnalysisSummary2 = AnalysisSummary1 %>%
  filter(x == 3)

# estimated expected value and variance (probs from column x=3 above)
EY = round(weighted.mean(AnalysisSummary2$y, AnalysisSummary2$Prob),2)
CV = weighted.mean((AnalysisSummary2$y-EY)^2, AnalysisSummary2$Prob)

# summarize for unique values of x (eliminate recurring values which have the same densities)
AnalysisSummary3 = AnalysisSummary2 %>%
  group_by(x) %>% summarise(my = mean(y))

# plot using ggextra format
plot_center = ggplot(AnalysisSummary2, aes(x=x,y=y)) +
  geom_point() +
  theme(panel.background = element_rect(fill = "white")) +
  xlim(-2, 9) + ylim(1, 10) +
```

```

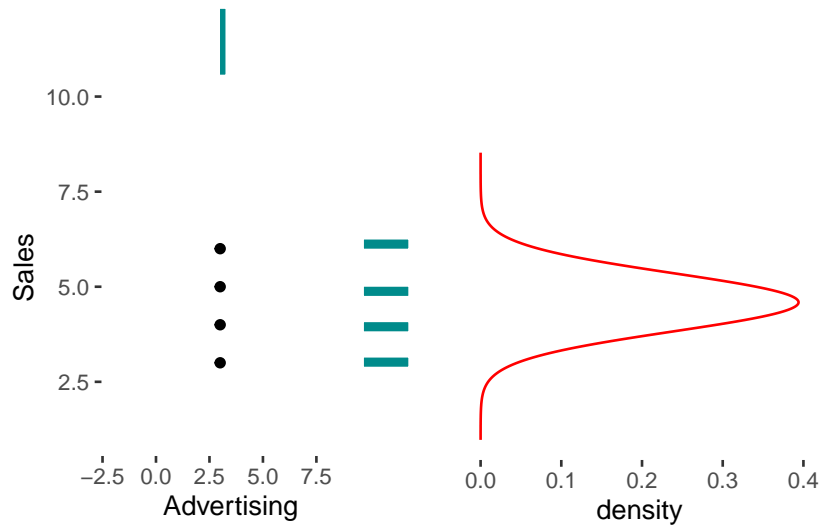
  ylab("Sales") + xlab("Advertising")
p2 = ggMarginal(plot_center, type="histogram",
               fill = "cyan4", color = "white")
base <- data.frame(x = seq(0, 10, by = .01))

# forcing a normal dist here for visual understanding

PF1 = ggplot(base, aes(xy)) +
  geom_line(aes(x,y= dnorm(x, mean = EY, sd = sqrt(CV))), color = "red") +
  theme(panel.background = element_rect(fill = "white"),
        axis.text.y = element_blank(),
        axis.ticks.y = element_blank()) +
  xlim(0, 14) +
  xlab("") +
  ylab("density") +
  coord_flip()

# bring all the plots together using cowplot
library(cowplot)
plot_grid(p2, PF1, nrow = 1, ncols = 2)

```



The conditional mean $E(E(y|x))$ is not the mean of y where $x = 3$, it is the conditional **mean** of $x = 3$ over the *marginal* distribution - not the same. In this case, the conditional mean should be a little higher than the distribution of y where $x = 3$. It will get “pulled” towards the center of the marginal y .

Let’s ignore conditional parameters, and just estimate the mean of y just from data from $x = 3$:

```

mod2 = lm(y~x, AnalysisSummary2)
y2 = predict(mod2, AnalysisSummary2)
EY2 = mean(y2)
CV2 = summary(mod2)$coefficient[, 'Std. Error']

PF = ggplot(base, aes(x)) +
  geom_line(aes(x,y= dnorm(x, mean = EY, sd = CV)), color = "red") +
  geom_line(aes(x,y= dnorm(x, mean = EY2, sd = CV2)), color = "blue") +
  theme(panel.background = element_rect(fill = "white"),

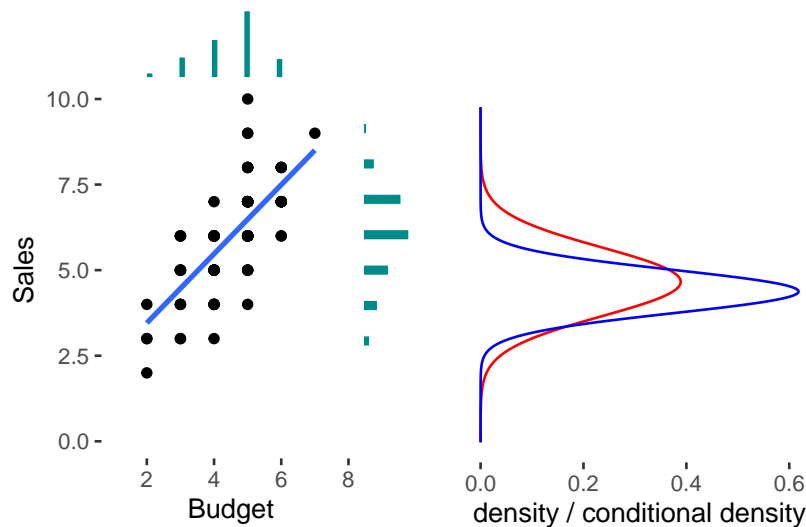
```

```

axis.text.y = element_blank(),
axis.ticks.y = element_blank()) +
xlim(0, 12) +
xlab("") + ylab("density / conditional density") +
coord_flip()

library(cowplot)
plot_grid(p1, PF, nrow = 1, ncols = 2)

```



Notice that the conditional mean is closer to the overall mean of y . It's *weighted* by the marginal population. Let's talk about how this works.

The second estimate (*from a model based on $x=3$ data only*) ignores that there is a bigger market for y , with a tendency to “pull” y up, towards the mean. Which one do you think is the better estimate? Now, let's look at all 3 Distributions:

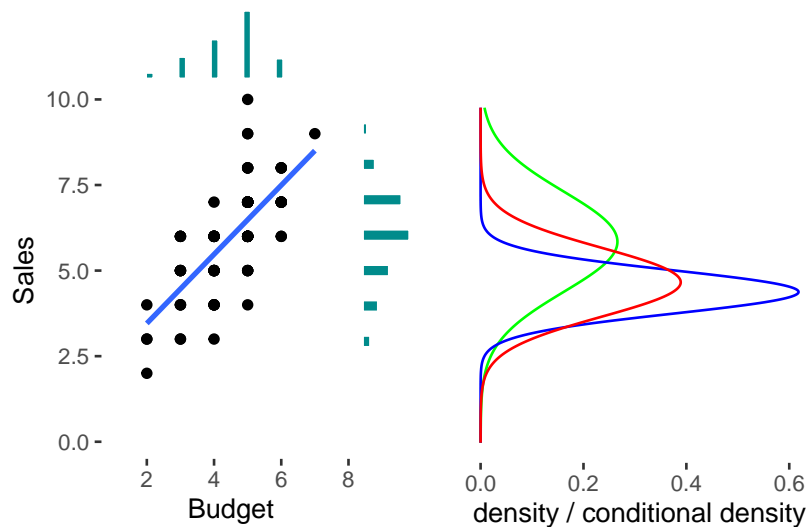
1. Distribution based on weighted mean, conditioned on $x = 3$ (*red*)
2. Distribution based on data limited to $x = 3$ (*blue*)
3. Distribution of y with no condition (*green*)

```

base <- data.frame(x = seq(0, 10, by = .01))
PF = ggplot(base, aes(x)) +
  geom_line(aes(x,y= dnorm(x, mean = muY, sd = sigmaY)), color = "green") +
  geom_line(aes(x,y= dnorm(x, mean = EY2, sd = CV2)), color = "blue") +
  geom_line(aes(x,y= dnorm(x, mean = EY, sd = CV)), color = "red") +
  theme(panel.background = element_rect(fill = "white"),
        axis.text.y = element_blank(),
        axis.ticks.y = element_blank()) +
  xlim(0, 12) +
  xlab("") + ylab("density / conditional density") +
  coord_flip()

library(cowplot)
plot_grid(p1, PF, nrow = 1, ncols = 2)

```



Note that location and scale shifts - the weighted distribution is a **compromise** between the data conditioned on $x=3$ and the unconditioned distribution of y (*the marginal*). Which one is most realistic for $x = 3$? Which one would you bet your job on?

Now, let's go back to the original question: The probability that sales = 5M, given the advertising budget = 3k? In this case, we have a discrete question. Repeating the conditional table for convenience:

```
knitr::kable(MarAnalysis) %>%
  kable_styling(full_width = F, bootstrap_options = "striped", font_size = 9)
```

y	2	3	4	5	6	7	Total
2	0.01	NA	NA	NA	NA	NA	0.01
3	0.02	0.02	0.01	NA	NA	NA	0.05
4	0.01	0.03	0.05	0.01	NA	NA	0.10
5	NA	0.05	0.08	0.04	NA	NA	0.17
6	NA	0.04	0.10	0.14	0.02	NA	0.30
7	NA	NA	0.01	0.17	0.07	NA	0.25
8	NA	NA	NA	0.04	0.04	NA	0.08
9	NA	NA	NA	0.02	NA	0.01	0.03
10	NA	NA	NA	0.01	NA	NA	0.01
Total	0.04	0.14	0.25	0.43	0.13	0.01	1.00

A Bayes Theorem solution:

$$P(\text{Sales} = 5 | \text{Budget} = 3) = \frac{P(\text{Budget}=3 | \text{Sales}=5) * P(\text{Sales}=5)}{P(\text{Budget}=3)}$$

$$P(\text{Sales} = 5 | \text{Budget} = 3) = \frac{(.05 / .17) * .17}{.14} = .357$$

How does this compare with the $P(\text{Sales} = 5 | \text{Budget} = 3)$ if we just use data partitioned for $\text{Budget} = 3$ (in which case, probability = .29)

Another way of solving this is with a Bayesian Update Table (*again, review C1_Classification_LogReg.pptx if needed*). In the example below, we back into the likelihood (*still not realistic - for pedagogical purposes*):

```
BUT = data.frame(
  y = MarAnalysis$y,
  Prior = MarAnalysis$Total,
  Like = (MarAnalysis$`3` / MarAnalysis$Total))
BUT = BUT %>%
```

```

  filter(y != 'Total') %>%
  mutate(
    BayesNum = Prior * Like,
  )
Den = sum(BUT$BayesNum, na.rm = T)
BUT$Posterior = round(BUT$BayesNum/Den,3)
BUT$Like = round(BUT$Like,3)

knitr::kable(BUT) %>%
  kable_styling(full_width = F, bootstrap_options = "striped", font_size = 9)

```

y	Prior	Like	BayesNum	Posterior
2	0.01	NA	NA	NA
3	0.05	0.400	0.02	0.143
4	0.10	0.300	0.03	0.214
5	0.17	0.294	0.05	0.357
6	0.30	0.133	0.04	0.286
7	0.25	NA	NA	NA
8	0.08	NA	NA	NA
9	0.03	NA	NA	NA
10	0.01	NA	NA	NA

Notice above that we just multiplied the prior * likelihood and then normalized the Bayes numerator to the the posterior (*which agrees with the formula answer above*).

This is important because we seldom have conditional probabilities like in the conditional table above, and we usually don't know the marginal probabilities either. So reality is a long way away from this academic exercise. What do we do? For starters, we can just sample the data to get the Likelihood, and we can estimate priors.

To finish this discrete case exercise, there are few other distributions that will be important in our analysis - those of the parameters. We'll just pull those for awareness now:

```

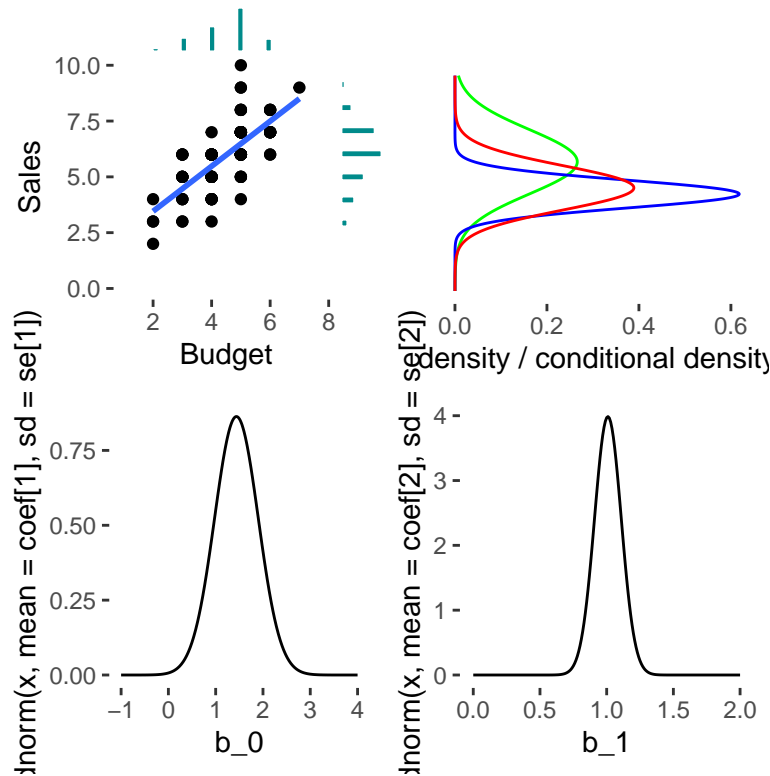
coef = summary(mod)$coeff[, "Estimate"]
se = summary(mod)$coeff[, "Std. Error"]

dfData <- data.frame(x = seq(-1, 4, by = .01))
pp1 <- ggplot(dfData) +
  geom_line(aes(x,y= dnorm(x, mean = coef[1], sd = se[1]))) +
  theme(panel.background = element_rect(fill = "white")) +
  xlab("b_0")

dfData <- data.frame(x = seq(0, 2, by = .01))
pp2 <- ggplot(dfData) +
  geom_line(aes(x,y= dnorm(x, mean = coef[2], sd = se[2]))) +
  theme(panel.background = element_rect(fill = "white")) +
  xlab("b_1")

plot_grid(p1, PF, pp1, pp2, rows = 2, cols = 2)

```



The distributions of our parameter (β_0 and β_1 in this case), gives us a metric about the credibility of our analysis. A **LOT** more on this later. So, this is a fairly complete picture of a conditional analysis, and a good start on understanding Bayesian data analysis.

Continuous Case

Let's change the discrete case and say that y is a continuous variable, say management wants to know the probability that sales could be 5 or greater, given a budget of 3 (*a more realistic case*). So:

$$P(\text{Sales} \geq 5 | \text{Budget} = 3) = \frac{P(\text{Budget}=3 | \text{Sales} \geq 5) * P(\text{Sales}=5)}{P(\text{Budget}=3)}$$

Recreating the data without forcing a discrete y :

```

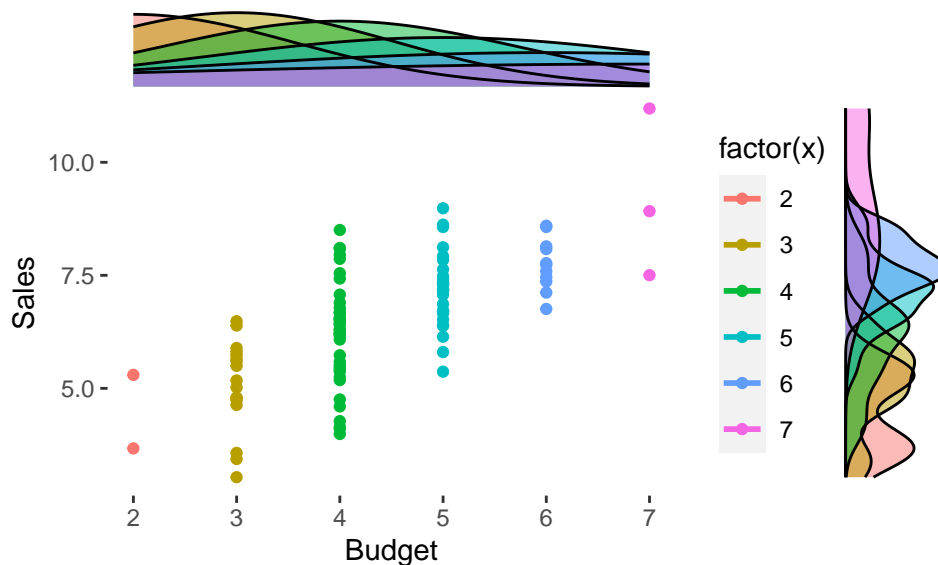
intercept = 2
slope = 1
N = 100

# generate discrete data
ExampleData1c = data.frame(x = as.integer((rnorm(N, 5, sd = 1)))) %>%
  mutate(y = (intercept + (slope * x)) + rnorm(N, 0, sd = 1))

# plot using ggextra format (you can use density with discrete variables too)

plot_center = ggplot(ExampleData1c, aes(x=x, y=y, colour = factor(x))) +
  geom_point(width = .05) +
  geom_smooth(method="lm", se = F) +
  theme(panel.background = element_rect(fill = "white")) +
  ylab("Sales") + xlab("Budget")
pC = ggMarginal(plot_center, type="density", groupColour = FALSE, groupFill = TRUE)
pC

```

ggmarginal won't let us show a discrete on one margin and continuous on another, so just focus on the Sales margin for now, keeping in mind that budget is discrete.

Since we're now trying to find $\text{Sales} > 5$, we'll need to define a continuous distribution with a condition of $\text{budget} = 3$. And like the discrete cases above, this distribution does NOT project the densities of the conditional case, it defines the data before it is projected to a posterior. That must consider the denominator (like we did with the BAT above). But this is a continuous case, so we need another way. There are a couple of ways to do this (and we'll study these both in later classes)

1. Directly determine the parameters (must be conjugate, and limited to a few distributions) Conjugate distributions are those of the same type (i.e. normal, binomial, etc.) . In a normal case, we can project $P(\text{Sales} >= 5 | \text{Budget} = 3)$ using formulas:

$$\sigma = \sqrt{\frac{1}{\sigma_1^{-2} + \sigma_2^{-2}}}$$

$$\mu = \text{posterior} \sigma^2 * \frac{\frac{\mu_1}{\sigma_1^2} + \frac{\mu_2}{\sigma_2^2}}{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}}$$

This is all for building understanding, but it's not very practical.

2. Multiplying the Distributions and then figuring out the parameters. This is the more practical approach, based on a restatement of Bayes Rule to assess the **probability of the parameters**. $P(\theta | \text{Data}) = \frac{P(\theta) * P(\text{Data} | \theta)}{P(\text{Data})}$. This works well for inverting probabilities in a closed system - such as this case, where the denominator $P(\text{Data})$ is (the marginal probability of the population)* is known and can serve as the normalizing constant (bayes denominator).

Important attribute here - it is a **constant**. You may recall how, in maximum likelihood, we dropped the constant portion of the binomial density function $\frac{n!}{n-h!}$ while varying p for likelihood of a parameter (which is what we're doing here). So, we can rewrite the equation as:

$$P(\theta | \text{Data}) \propto P(\theta) * P(\text{Data} | \theta), \text{ or}$$

$\text{Posterior}(\theta | \text{Data}) \propto \text{Prior}(\theta) * \text{Likelihood}(\text{Data} | \theta)$, where \propto means "proportionate to". We can normalize later. So we can just multiple the prior x likelihood to get the posterior.

This is a big deal. This means we really don't need the marginal distribution. And we can estimate the prior. Then all we have to do is sample the data for likelihood. This is getting easy, no?

```
plm = ggplot(ExampleData1c, aes(x = x, y= y)) +
  geom_point() +
```

```

theme(panel.background = element_rect(fill = "white"))

dfData <- data.frame(x = seq(1, 10, length.out = nrow(ExampleData1c)))
denY = data.frame(y = dfData$x, density = dnorm(dfData$x, mean = mean(ExampleData1c$y), sd = sd(ExampleData1c$y)))
denY$cDensity3 = dnorm(denY$y, mean = mean(filter(ExampleData1c, x == 3)$y) , sd = sd(filter(ExampleData1c, x == 3)$y))

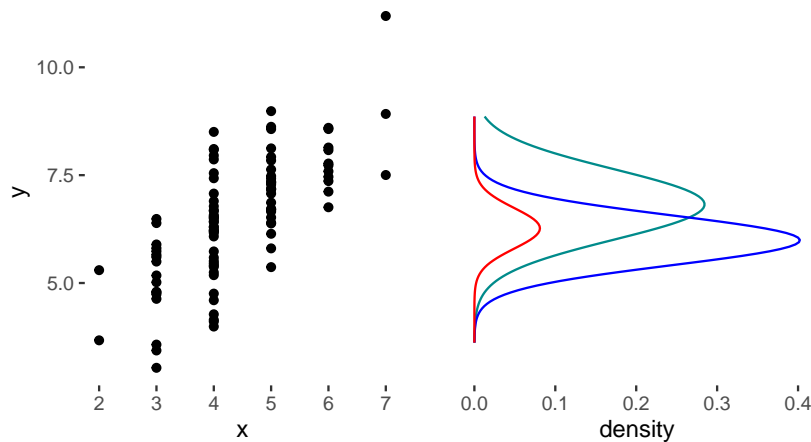
PF3 = ggplot(denY, aes(x = y, y= density)) +
  geom_line(color = "cyan4") +
  geom_line(aes(x = y, y= cDensity3), color = "blue") +
  theme(panel.background = element_rect(fill = "white"),
        axis.text.y = element_blank(),
        axis.ticks.y = element_blank()) +
  xlim(0, 14) +
  xlab("") +
  ylab("density") +
  coord_flip()

denY$pDensity3 = denY$density*denY$cDensity3

PF3 = PF3 +
  geom_line(aes(x = y, y= denY$pDensity3), color = "red")

plot_grid(plm, PF3, nrow = 1, ncol = 2)

```



You may notice that the posterior density has smaller values. That's because we computed it using $(\text{density} * \text{density})$, and (small values * small values) are really small values. But it doesn't matter, because the relative proportion (\propto) is what we're after. And we can convert density to probability whenever we want - all we have to do is divide it by the sum of itself (*i.e.*, we “normalize” it)

Why do we go through all this distribution stuff? Because people want to know the probability of a decision or opinion. **No distribution = No probability.** and **No probability = no credibility.**

Simulation

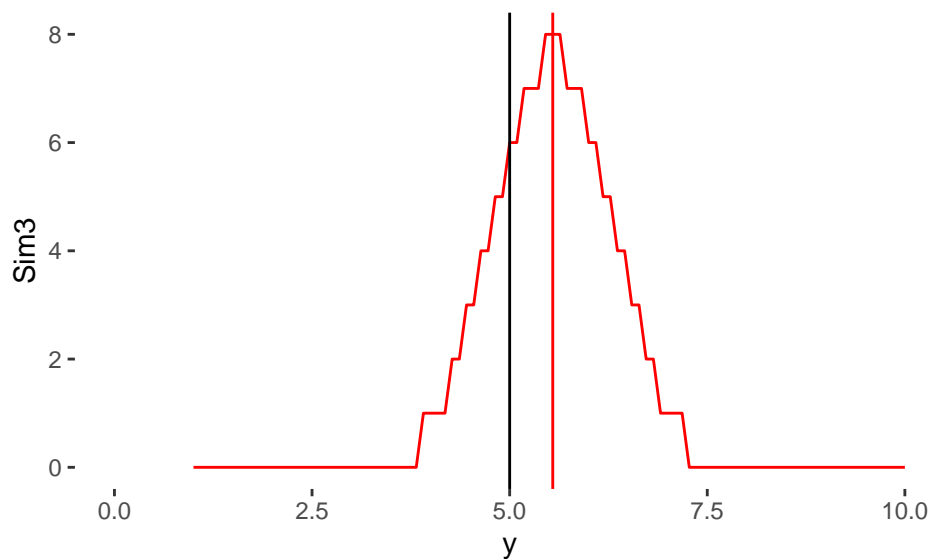
Now that we have the distributions, we can compute the probability of sales > 5 , given a budget of 3. First, let's simulate (*predict, project*) what a distribution of $P(\text{Sales} \geq 5 | \text{Budget} = 3)$ would look like (*back in a*

discrete case):

```
N2 = nrow(denY)
denY$Sim3 = as.integer(denY$pDensity3*N2)
simMean = denY %>% filter(Sim3 > 0) %>% summarise(Mean = weighted.mean(y, Sim3)) %>% as.numeric()
simSD = denY %>% filter(Sim3 > 0) %>% summarise(SD = sd(Sim3)) %>% as.numeric()

PF5 = ggplot(denY, aes(x = y, y = Sim3)) +
  geom_line(color = "red") +
  geom_vline(xintercept = 5) +
  geom_vline(xintercept = simMean, color = "red") +
  xlim(0, 10) +
  theme(panel.background = element_rect(fill = "white"))
```

PF5



So, using the pnorm functions, we can get the probabilities (of sales > 5 where Budget = 3):

```
(1 - pnorm(5, simMean, simSD))
```

```
## [1] 0.5887106
```

Foundations for Optimization

When I interviewed job applicants for analyst and data science positions, I focused on generalization and optimization - that quickly told me if they were an experienced resource.

Optimization is especially important in Bayesian modeling because the approach uses Markov Chain Monte Carlo (*MCMC*) sampling. MCMC is really the only practical approach to defining probabilities in complex, multi-distribution, multilevel spaces, but it requires a significant amount of computational resources *and time* to wander through those spaces and find parameters. This is partly because of the way the sampler works:

High performance computing (*clusters*) helps a lot, but it won't compensate for poor model design. Transformation, Factorization and Reparametrization are essential skills in model design. You may have touched on these topics in prerequisite courses, for example:

- **Transformation** includes scaling (*and centering*), log transforms, and vectorization (*using vectors instead of iterative operations*).
- **Factorization** includes QR (*the method utilized by lm*) and Cholesky matrix decomposition (*you practiced factoring quadratics in high school, hopefully - it's the same idea*).
- **Reparameterization**, as a general term, includes factorization, and adds other techniques. We'll introduce these as needed during exercises.

Many optimization approaches are intended to reduce the total data and the total variance (*keeping in mind that we have to leave breadcrumbs so we can get back to the original scale*), and we'll explore these later. But first, we need to build a foundation to understand this all better. We'll first look at some alternative ways to get the parameters of the regression distributions μ, σ, b_0, b_1 . We want to know this because

We'll break this into two sections I'll call variance and decomposition:

Variance

Let's start with the familiar. Taking our model from above (*mod*), let's summarize:

```
summary(mod)

##
## Call:
## lm(formula = y ~ x, data = ExampleData1)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.4846 -0.4846  0.0200  0.5246  3.5154
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   1.4387     0.4622   3.113  0.00243 **
## x              1.0092     0.1001  10.086 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.045 on 98 degrees of freedom
## Multiple R-squared:  0.5093, Adjusted R-squared:  0.5043
## F-statistic: 101.7 on 1 and 98 DF,  p-value: < 2.2e-16
```

You should be familiar with rmse as a measure of model fit (*1.03 here*). We often used the following function to calculate rmse for out-of-sample data (*feeding residuals to the function and we used to evaluate model fit*):

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

We now need to drill down more on what it means and how it's calculated (*we'll use a normal equation the rest of the way so we can break up the components*). Recall that the normal equation is: $\beta = (X^T X)^{-1} (X^T Y)$, $\hat{y} = \beta(X^T)$, and error is $\sum (y - \hat{y})^2 / df$ where df is degrees of freedom (*number of observations - estimated coefficients*). So, if we have std error $\sqrt{\sum (y - \hat{y})^2 / df}$, we can compute the covariance matrix $Var(\beta|X) = \sigma^2 (X^T X)^{-1}$. Let's compute these values and compare to lm.

First, come manual covariance calculations:

```
mX = model.matrix(y ~ x, ExampleData1)
vY = ExampleData1$y
```

```

vBeta2 <- solve(t(mX)%*%mX) %*% t(mX) %*% vY

# degrees of freedom (just getting technical here - n is fine in transaction environments)
df = nrow(mX) - ncol(mX)
# calculate mean error
sigmaSq = sum((vY - (t(as.vector(vBeta2)%*%t(mX))))^2)/df
# and standardize
rmse = sqrt(sigmaSq)
# so the variance-covariance is:
vCv = solve(t(mX)%*%mX) * sigmaSq
# check with lm
vCvlm = vcov(mod)
# create std error
vStdErr <- sqrt(diag(vCv))

knitr::kable(vCv) %>%
  kable_styling(full_width = F, bootstrap_options = "striped", font_size = 9)

```

	(Intercept)	x
(Intercept)	0.2136614	-0.0450551
x	-0.0450551	0.0100122

Comparing with covariance from lm:

	(Intercept)	x
(Intercept)	0.2136614	-0.0450551
x	-0.0450551	0.0100122

Manual Std. Error:

	x
(Intercept)	0.4622352
x	0.1000612

And if we don't have b (and se), we can determine those parameters just from the contrivance of (x, y) :

$$Cov(x, y) = \frac{\sum (x - \bar{x})(y - \bar{y})}{n-1}, \text{ and } b_1 = \frac{Cov(x, y)}{var(x)},$$

So, b_1 can be estimated as follows:

```

# doing this the long way (so you can see)
CovXY2 = sum(((ExampleData1$x - mean(ExampleData1$x)) * (ExampleData1$y - mean(ExampleData1$y))))/(nrow(ExampleData1)-1)

varX = var(ExampleData1$x)

b1 = CovXY2/varX
round(b1, 2)

```

```
## [1] 1.01
```

Pretty cool huh?

And then we can back into b_0 : $b_0 = \bar{y} - b\bar{x}$

We're not done yet! Another way to get the covariance is *(it's just a little different, but you may find it easier at times)*:

$$\text{Cov}(x, y) = \frac{n(\bar{xy} - \bar{x} * \bar{y})}{n-1}$$

The code below uses this approach and compares the covariances:

```
CovXY3 = nrow(ExampleData1)*(mean(ExampleData1$x*ExampleData1$y) - mean(ExampleData1$x)* mean(ExampleData1$y))

bo = data.frame(Description = c("covXY2", "covXY3", "b1"), Value = c(CovXY2, CovXY3, b1))
bo$Value = round(bo$Value, 2)

knitr::kable(bo) %>%
  kable_styling(full_width = F, bootstrap_options = "striped", font_size = 9)
```

Description	Value
covXY2	1.11
covXY3	1.11
b1	1.01

You can also use solve and treat the covariance matrices as a system of equations:

```
b0 <- cov(as.matrix(ExampleData1$x))
b1 <- cov(as.matrix(ExampleData1$x), as.matrix(ExampleData1$y))

vBeta1 <- solve(b0, b1)[, 1] # Coefficients from the covariance matrix

vBeta1

## [1] 1.009174
```

Let's take a look at the full covariance matrix from the model:

```
Out <- data.frame(vcov(mod))
knitr::kable(Out) %>%
  kable_styling(full_width = F, bootstrap_options = "striped", font_size = 9)
```

	X.Intercept.	x
(Intercept)	0.2136614	-0.0450551
x	-0.0450551	0.0100122

Just to review, the covariance matrix provides the variance of each variable on the diagonal, and the covariances outside *(which is why it's really called a variance-covariance matrix - I just use covariance for expediency)*.

Another way to get the vcov is using Cholesky decomposition *(we'll look at Cholesky soon)*

```
mX = model.matrix(y ~ x, ExampleData1)
vY = ExampleData1$y
vBeta2 <- solve(t(mX)%*%mX, t(mX)%*%vY)
y_hat = t(as.numeric(vBeta2)%*%t(mX))
# estimate of sigma-squared
dSigmaSq <- sum((vY - mX%*%vBeta2)^2)/(nrow(mX)-ncol(mX))
# variance covariance matrix
mVarCovar <- dSigmaSq*chol2inv(chol(t(mX)%*%mX))
mVarCovar
```

```
##           [,1]           [,2]
## [1,]  0.21366136 -0.04505511
## [2,] -0.04505511  0.01001225
# coeff. est. standard errors
vStdErr <- sqrt(diag(mVarCovar))
vStdErr

## [1] 0.4622352 0.1000612
```

Factorization

QR QR decomposition factors the model matrix into a Q and a R matrix (*which by definition, can be multiplied to yield the original matrix*). The R is rectangular, and the bottom row is all zeros except the last column - which will align with a beta value. The we just have to back-solve to get the beta values (*we don't actually have to back solve ourselves, we can use matrix algebra to do it*).

How come we can do this? Let's start with a normal equation and assume we can factor X into Q and R. Then, see how it can used to solve the beta values:

$$\begin{aligned}(X^T X)\beta &= X^T Y, \text{ so:} \\ (QR^T QR)\beta &= (QR)^T Y \\ (R^T Q^T Q)R\beta &= R^T Q^T Y \\ (R^T)^{-1}R^T R\beta &= (R^T)^{-1}R^T Q^T Y \\ R\beta &= Q^T Y\end{aligned}$$

And there are two ways to solve using QR - fat and thin. The thin way is faster (*I have included some thin QR functions in this document for illustration - you don't need to know them - just understand the benefits outlined below*). Being faster is what we're after!! Let's look at it:

```
# functions
# ----- Don't worry about these functions - just here for reference -----#

inner_prod = function(v1, v2) {
  stopifnot(length(v1) == length(v2))
  len = length(v1)
  res = vector("numeric", length = len)
  for (i in 1:len) {
    res[i] = v1[i] * v2[i]
  }
  return(sum(res))
}

inner = function(v1, v2) {
  stopifnot(length(v1) == length(v2))
  return(sum(v1 * v2))
}

# thin QR

thinQR = function(x) {
  n = nrow(x)
  p = ncol(x)
  q1 = matrix(0, nrow = n, ncol = p)
  r1 = matrix(0, nrow = p, ncol = p)
  u = matrix(0, nrow = n, ncol = p)
  u[, 1] = x[, 1]
  for (k in 2:ncol(x)) {
```

```

u[, k] = x[, k]
# successive orthogonalization
for (ctr in seq(1, (k - 1), 1)) {
  u[, k] = u[, k] - ((inner(u[, ctr], u[, k]) / inner(u[, ctr], u[, ctr])) * (u[, ctr]))
}
}
q1 = apply(u, 2, function(x) { x / sqrt(inner(x, x)) })
r1 = crossprod(q1, x) # t(q1) %*% x
return(list(q = q1, r = r1, u = u))
}

```

The following sets up the data and solves the equation using `lm` first (*just for a baseline*). Next, we use the `qr` function in R to get the Q matrix and the R matrix, and then solve using the equation above. Finally, we solve using thin QR (*from the functions above*).

As you can see, we get the same answers for each method (*which shouldn't be a big surprise because `lm` also uses QR*). First the `lm` beta:

```

modQR = lm(y~x, ExampleData1)
lmBeta = coef(modQR)

fatQRBeta <- solve(qr.R(qr(mX))) %*% t(qr.Q(qr(mX))) %*% vY

res = thinQR(mX)
thinQRBeta <- solve(res$r) %*% t(res$q) %*% vY

knitr::kable(lmBeta) %>%
  kable_styling(full_width = F, bootstrap_options = "striped", font_size = 9)

```

	x
(Intercept)	1.438716
x	1.009174

The Fat QR beta:

```

knitr::kable(fatQRBeta) %>%
  kable_styling(full_width = F, bootstrap_options = "striped", font_size = 9)

```

(Intercept)	1.438716
x	1.009174

and Thin QR:

```

knitr::kable(thinQRBeta) %>%
  kable_styling(full_width = F, bootstrap_options = "striped", font_size = 9)

```

(Intercept)	1.438716
x	1.009174

OK, so they get the same answer - so what's the big deal?

Keep in mind that an algorithm to solve a regression equation is going to try to minimize error, and most of the algorithms that solve the more complex regressions, do so using derivative based methods and maximum likelihood. The Bayesian world uses sampling to find parameters, but it samples derivatives of the log

probability of the joint distributions (*which can be hundreds or even thousands - especially in multilevel cases*). And the magnitude and variances of the parameters and data has a **BIG** effect on performance. With that in mind, let's just look at magnitude and variance of the data vs the thin Q and R:

```
var(mX)
```

```
##           (Intercept)           x
## (Intercept)      0 0.00000
## x                0 1.10101
```

```
var(res$q)
```

```
##      [,1]      [,2]
## [1,]    0 0.00000000
## [2,]    0 0.01010101
```

Winner! Thin QR. That's the idea anyway.

Cholesky One more tool. Cholesky decomp works the same way as QR, but it handles sparse matrices better (*sparse matrices are ones with lots of zeros and it's very common in transaction analysis - think of ERP tables, and how that translates into a model matrix with indicator variables - lots of them*). We'll just run a simple example:

```
#use Cholesky to estimate beta
```

```
vBeta3 = chol2inv(chol(t(mX)%*%mX)) %*% t(mX)%*% vY
vBeta3
```

```
##      [,1]
## [1,] 1.438716
## [2,] 1.009174
```

chol produces the Cholesky decomposition and chol2inv produces the inverse, so we're back to $\beta = (X^T X)^{-1}(X^T Y)$!! but in a process that can deal with LARGE datasets.