Foundations for Regression Analysis

PreReq Skills for BZAN X6 / ACCT 7374 Bayesian Transaction Analysis

Introduction

Hopefully, you're familiar with a linear equation $\hat{y} = \alpha + \hat{\beta}X + \epsilon$. Generally, we have the data X and Y, and we want to find the best values for the $\hat{\beta}$ parameters that predict estimated values $\hat{y}$ where the model error $(y - \hat{y})$ is minimized, given the random error ϵ.

This can be solved by creating a cost function of model errors $S_{min} \sum (y - \hat{y})^2$, finding the point where error is minimized *(using a derivative-based solution)*. We call this Ordinary Least Squares *(OLS)*. The nice thing about a simple OLS solution is that there exists a stable, unique solution. As the problem gets more complicated, this approach starts running into barriers. Fortunately, for simple problems, we can use a normal equation, which is a closed form solution to estimate OLS. This is derived from the linear equation above:

$Y = \alpha + \beta X + \epsilon$
(note that β is a vector and X *is a matrix, and dropping hats for readability)*. So:

$X^T X \beta = X^T Y$
Multiplying both sides by X^T

$(X^T X)^{-1} X^T X \beta = (X^T X)^{-1} X^T Y$
Multiplying both sides by the inverse matrix *(note: an inverse multiplied by the original matrix = 1)*, and that gets us to an normal equation we can use to solve for β:

$\beta = (X^T X)^{-1} (X^T Y)$ Normal Equation

Another way to estimate the OLS solution is using matrix decomposition. We won't get into decomp at this level, but fortunately, the lm function in R does a good job of wrapping decomp into an easy working format:

Regression Modeling Process

Model 1

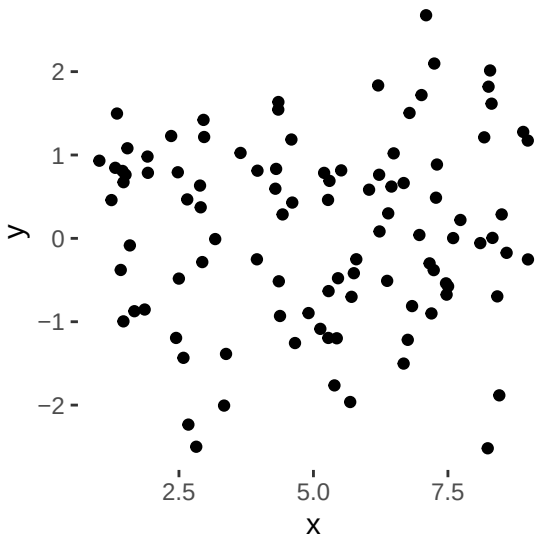
First, let's build a regression model on data that is completely random *(no relationship)*. This may seem silly, but it's a good way to start *(and btw, most of the world is random - finding relationships is a rare treat)*.

First, let's generate data with no relationship *(pay attention to how data is generated, as you will need to do this)*:

Let's generate some data with no relationship and take a look at it:

```
N  = 100
AnalysisData1 = data.frame(x = runif(N, min = 1, max = 9),
                           y = rnorm(N, 0, sd = 1))

p1 = ggplot(AnalysisData1, aes(x, y)) +
  geom_point()+
  theme(panel.background = element_rect(fill = "white"))
p1
```



I created two sets of unrelated, variables: one a random uniform distribution for X, and one a random normal distribution for y.

The linear model relies on a number of assumptions, a few of which are:

1. There is a linear relationship between independent x and dependent y *(obviously not the case here - purely random)*.
2. The **residuals** are normally distributed. This confuses students at first, but it's really just a way of saying that Y is normally distributed **after the model equation** $a + bx$ **has been applied**. So, the residuals $(y - \hat{y})$ will also be normally distributed **IF** the model structure *(not all models are linear)* and the estimated coefficients $\hat{\beta}$ **fit** the data. If not, the residuals will skew off.
3. There is also an assumption about homoscedasticity *(or homogeneity of variance - i.e., it looks elliptical in 2 dimensions)*, which also relates to the distribution of y.

Now, let's run lm and see what we get:

```
mod1 = lm(y~x, AnalysisData1)
summary(mod1)
```

```
##
## Call:
## lm(formula = y ~ x, data = AnalysisData1)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.6635 -0.8083  0.1103  0.7579  2.5486
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.01901    0.26674   0.071    0.943
## x            0.01534    0.04759   0.322    0.748
##
## Residual standard error: 1.11 on 98 degrees of freedom
## Multiple R-squared:  0.00106,    Adjusted R-squared:  -0.009134
## F-statistic: 0.1039 on 1 and 98 DF,  p-value: 0.7478
```

The format of lm is: dependent variable ~ *(is a function of)* dependent variable *(or list of dependent variables*

added + together).

The summary() function gives us a read-out of the modeling results.

Let's walk through the model output:

1. The formula we used.
2. Quartiles of the residuals ($\hat{y} - y$). You're looking to see if these appear to be evenly distributed *(back to the normality of the residuals. So, if 1Q is 10x the size of 3Q, we have a problem).*

Now, skipping down to the model level measures *(we'll come back to the Coefficient level later)*:

1. The Residual standard error *(also called the residual mean standard error - rmse)* standardizes the squared residuals by dividing by the degrees of freedom *(n - number of estimated coefficients)* and taking the square root of the squared errors *(so the error is converted to a magnitude - otherwise it would sum to 0)* $\sqrt{\sum(\hat{y} - y)^2}$.
2. R-Squared is a measure of the portion of variance that is explained by the model *(we'll see this computed soon)*
3. The p-value *(in line with traditional hypothesis testing)* estimated the probability of **falsely rejecting** the **null hypothesis**. The null hypothesis is skeptical, and the NULL here is: "*nothing is going on*". So the p-value means that there's a 75% chance that we will FALSELY REJECT the null hypothesis and say "*hey, there's something going on here*". There's not - but we knew that because we created the data :). The p-values of the coefficients follow the same process *(so in this case, the null hypothesis regarding the intercept and X coefficients should not be rejected - there's no effect here)*

We'll cover the rest of these measures later, as they relate to multivariate models.

Now that we have the model *(mod1)*, we can use it to generate predictions as follows:

```
AnalysisData1$y_hat = predict(mod1, AnalysisData1)
```

Easy enough eh?

Calculating Model Metrics

The model metrics above can all be calculated manually, which helps understand origin and purpose. Let's work through a few of these:

```
# ------------------------------------------------------------#

# sum of squares Regression (total variance related to the model)
# the model prediction less the prediction if there is no relationship (mean of y)
# so in this case, it will be very small - there's no difference
AnalysisData1$y_bar = mean(AnalysisData1$y)
SSR = sum((AnalysisData1$y_hat - AnalysisData1$y_bar)^2)

# sum of squares Error - this is the SS, not the rmse (square root)
SSE = sum((AnalysisData1$y_hat - AnalysisData1$y)^2)

# sum of squares Total
SST = SSR+SSE

# now R2 can be computed as portion of variance explained by model
R2 = SSR/SST

# Determine the number of estimated coefficients
```

```r
# Subtract one to ignore intercept which
k=length(mod1$coefficients)-1
# Number of Observations
n=length(mod1$residuals)
# now compute rmse
rmseMan = sqrt(SSE/(n-(1+k))) #Residual Standard Error/ degrees of freedom (df)

Out = data.frame(Description = c("R2 = ", "rmse = "), Vales = c(round(R2,4), round(rmseMan,4)))

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

Description	Vales
R2 =	0.0011
rmse =	1.1098

Covariance is important in building models that explain variance *(duh)*. From a high level, we're concerned with two sources of variance *(the data and the model)*, and two kinds of variance: within and between *(or covariance)*:

```r
# variance of X
varX = sum(AnalysisData1$x - mean(AnalysisData1$x))

# covariance of XY
covarXY  = sum((AnalysisData1$x - mean(AnalysisData1$x))*(AnalysisData1$y - mean( AnalysisData1$y)))

# variance of residuals
res = sum(AnalysisData1$y_hat - AnalysisData1$y)

# and residual mean standard error (again)
rmseAgain = sqrt(sum((AnalysisData1$y_hat - AnalysisData1$y)^2)/(n-(1+k)))
rmseAgain
```

```
## [1] 1.10981
```

In the modeling process, we also estimate the coefficients, and we need to know the variance of those estimates *(this tells us how confident we can be in our models)*. For example, the model estimated that the x coefficient *(slope)* is 0.01534 *(which is really zero, which is what we'd expect)*. And the std error of that estimate can be calculated as the ratio of model standard error to the standard error of the variable:

```r
# get the standarized error of the variable being estimated:
seX  = sqrt(sum((AnalysisData1$x - mean(AnalysisData1$x))^2))
# Now, divide the overall std error of the model by that variable error
seBeta = rmseMan/seX
seBeta
```

```
## [1] 0.04759019
```

We can also get this from the model covariance *(of the coefficients)* matrix:

```r
vcov(mod1)
```

```
##             (Intercept)            x
## (Intercept)  0.07114993 -0.011543260
## x           -0.01154326  0.002264826
```

The coefficient variance-covariance matrix shows the variances on the diagonal and the covariances on the

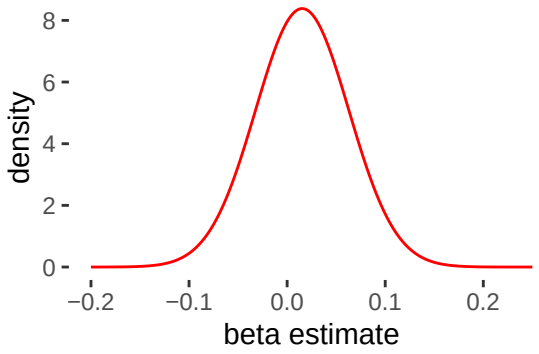
side. W were just interested in x here *(this gets a little more complex with multivariate models, where we're trying to estimate the effect of each variable, while holding the other variables constant)*. And we need to standardize, so:

```
sqrt(vcov(mod1)[2,2])
```

```
## [1] 0.04759019
```

That's two ways to get it *(there are more using matrix decomposition - so it really depends on what you have to work from - sometimes you don't even get X and Y, seriously)*. Now that we have the standard error of the coefficient estimate, we can define the distribution of our estimate of the coefficient:

```
base <- data.frame(x = seq(-.2, .25, by = .001))
ggplot(base, aes(x)) +
  geom_line(aes(x,y= dnorm(x, mean = mod1$coefficients[2], sd = seBeta)), color = "red") +
  theme(panel.background = element_rect(fill = "white")) +
  xlim(-.2, .25) +
  xlab("beta estimate") + ylab("density")
```



Think about it this way: there a 95% probability that the true value of β_1 is between -0.07 and 0.11 *(the estimated value of the coefficient, mean, +- 2SE)*:

```
LL = mod1$coefficients[2] - (2*seBeta)
UL = mod1$coefficients[2] + (2*seBeta)

Out = data.frame(Description = c("LL = ", "UP = "), Vales = c(round(LL,4), round(UL,4)))

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

Description	Vales
LL =	-0.0798
UP =	0.1105

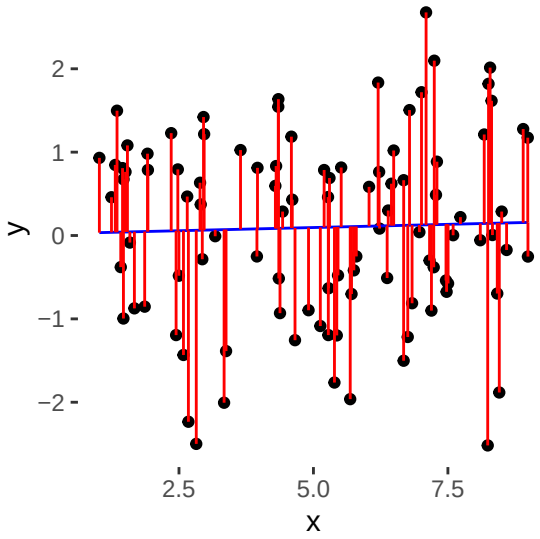
We'll cover the intercept estimate later. We can also get the model level std errors this way *(think about why this works)*:

```
se = sqrt(diag(vcov(mod1)))
se
```

```
## (Intercept)           x
##  0.26673944  0.04759019
```

One last thing before we move on, just for intuition. Let's visualize the error distribution:

```
p1 = p1 +
  geom_line(data = AnalysisData1, aes(x, y_hat), color = "blue")
AnalysisData1$error =  AnalysisData1$y - AnalysisData1$y_hat
p1 = p1 +
  geom_errorbar(data = AnalysisData1, aes(ymin = if_else((y_hat < y), y_hat, y),
                                          ymax = if_else((y_hat < y), y, y_hat)),
                color = "red")
p1
```



There's your residuals!

Model 2

This time, we're adding a real relationship *(woo-hoo!)*. Note how this is done *(you'll need to do this for homework)*. Just create a uniform distribution for x, then **create a model formula** to project the **mean** of y, and then add some error noise.

Also, add the following rmse function as we'll be using this more often later. Then, create the model and then get your coefficients, then draw the regression line using geom_abline:

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

# add model parameters

N  = 100
Intercept = 0
Slope = 1

# now adding a relationship

AnalysisData2 = data.frame(x = runif(N, min = 1, max = 9)) %>%
  mutate(y = (Intercept + (Slope * x)) + rnorm(N, 0, sd = 1))
```

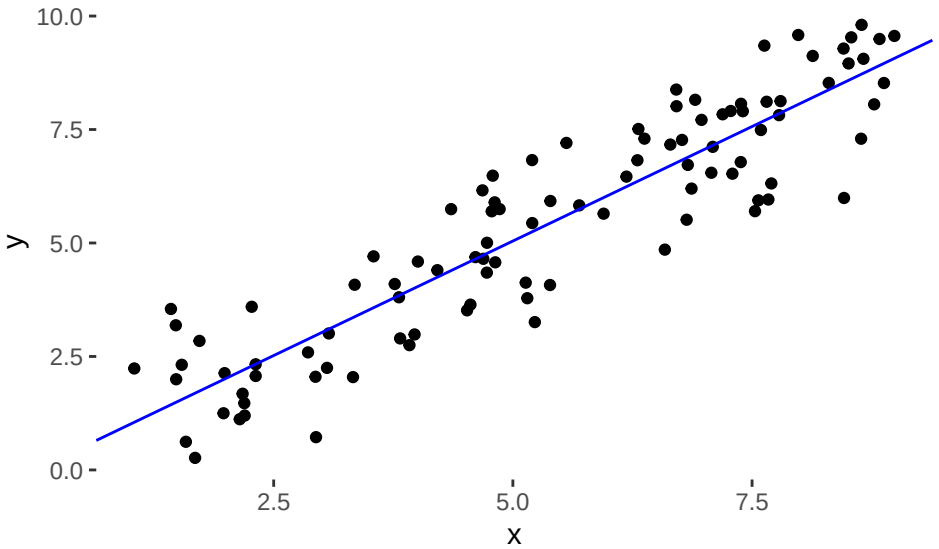
```r
mod2 = lm(y~x, AnalysisData2)
summary(mod2)
```

```
##
## Call:
## lm(formula = y ~ x, data = AnalysisData2)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.5443 -0.8329  0.1407  0.7553  2.1095
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.001867   0.268444  -0.007    0.994
## x            1.009028   0.045904  21.981   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.044 on 98 degrees of freedom
## Multiple R-squared:  0.8314, Adjusted R-squared:  0.8297
## F-statistic: 483.2 on 1 and 98 DF,  p-value: < 2.2e-16
```

```r
AnalysisData2$y_hat  = predict(mod1, AnalysisData2)
```

Visualize the model using abline *(which takes an intercept and slope argument)*. So, we need to pull the coefficient estimates from the model. Note how we pull the coefficient values out of the model *(just use a model$coefficients to see the full vector in order, then use [] to access the elements you want)*:

```r
p3 = ggplot(AnalysisData2, aes(x, y)) +
  geom_point() +
  geom_abline(intercept = mod2$coefficients[1], slope = mod2$coefficients[2], color = "blue") +
  theme(panel.background = element_rect(fill = "white"))
p3
```



And we'll use our rmse function to estimate model error:

```
AnalysisData2$y_hat = predict(mod2, AnalysisData2)
Mod2Error = rmse(mod2$residuals)
Mod2Error
```

```
## [1] 1.033456
```

The rmse function doesn't use df, just uses n. So, rmse will be lower. When you get into transaction analysis, it doesn't matter that much - transaction datasets usually have large Ns. We need a separate rmse function that we can apply to validation and out-of-sample data later.

Model 3

Now let's add categorical variables. We'll create a vector of different product groups, and then generate data across those groups.

```
vProduct = c("Product1", "Product2", "Product3")
N  = N*length(vProduct)
Intercept = 0
Slope = 1

AnalysisData3 = data.frame(Product = vProduct,
                PIntercept = c(Intercept-2, Intercept, Intercept+2),
                x = runif(300, min = 1, max = 9)) %>%
                mutate(y = PIntercept + (Slope * x) + rnorm(300, 0, sd = 2))
```

Now, build a model *(adding in our categorical variables)*

```
mod3 = lm(y~ Product + x, data = AnalysisData3)
summary(mod3)
```

```
##
## Call:
## lm(formula = y ~ Product + x, data = AnalysisData3)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.3192 -1.1815  0.1101  1.4195  5.7460
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)     -2.06705    0.32760  -6.310 1.02e-09 ***
## ProductProduct2  2.13859    0.29158   7.334 2.15e-12 ***
## ProductProduct3  3.96112    0.29212  13.560  < 2e-16 ***
## x                1.01621    0.05236  19.410  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.06 on 296 degrees of freedom
## Multiple R-squared:  0.6713, Adjusted R-squared:  0.668
## F-statistic: 201.5 on 3 and 296 DF,  p-value: < 2.2e-16
```

OK, when you have categorical variables, the model will, by default, create a varying intercepts model. This means there's a different intercept **EFFECT** for each group. Effects are added to the base estimate.

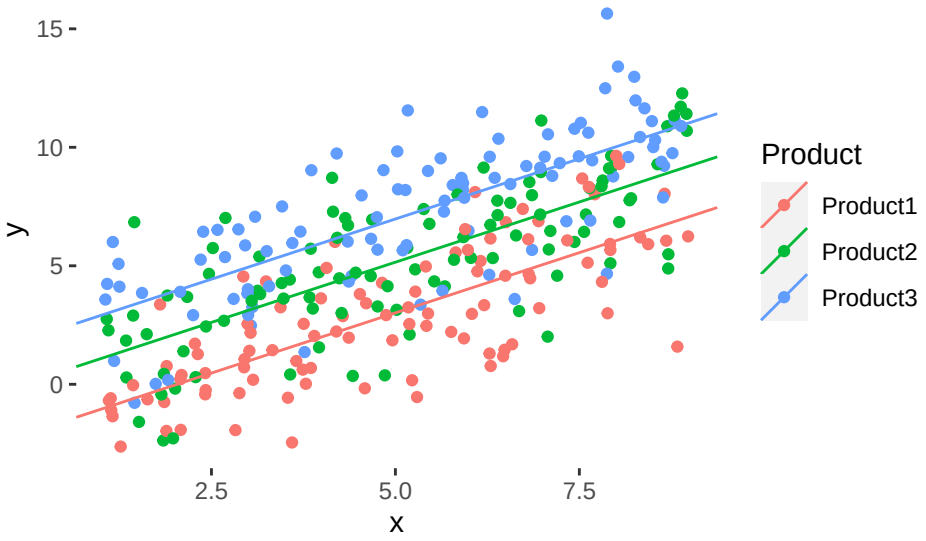
So, you'll need to find the base intercept, and adjust that for each group's effects. This is easy to figure out in lm - the first group is the base intercept, and the other groups are added to that. So let's look at how that's done.

We'll create a new dataframe with the categorical values, the intercept for each *(base + effect)*, and the slope *(which stays the same in a variable intercept model)*:

```
# create dataframe with coefficients from lm
# again, if you get confused about where this comes from
# inspect mod3$coefficients and note the order

dfCoef = data.frame(Product = c("Product1", "Product2", "Product3"),
                    Intercept = c(mod3$coefficients[1], mod3$coefficients[1] + mod3$coefficients[2], mo
                    Slope = c(mod3$coefficients[4], mod3$coefficients[4], mod3$coefficients[4]))

p5 = ggplot(AnalysisData3, aes(x, y, color = Product)) +
  geom_point()+
  geom_abline(data = dfCoef, aes(intercept = Intercept, slope = Slope, color = Product)) +
  theme(panel.background = element_rect(fill = "white"))
p5
```



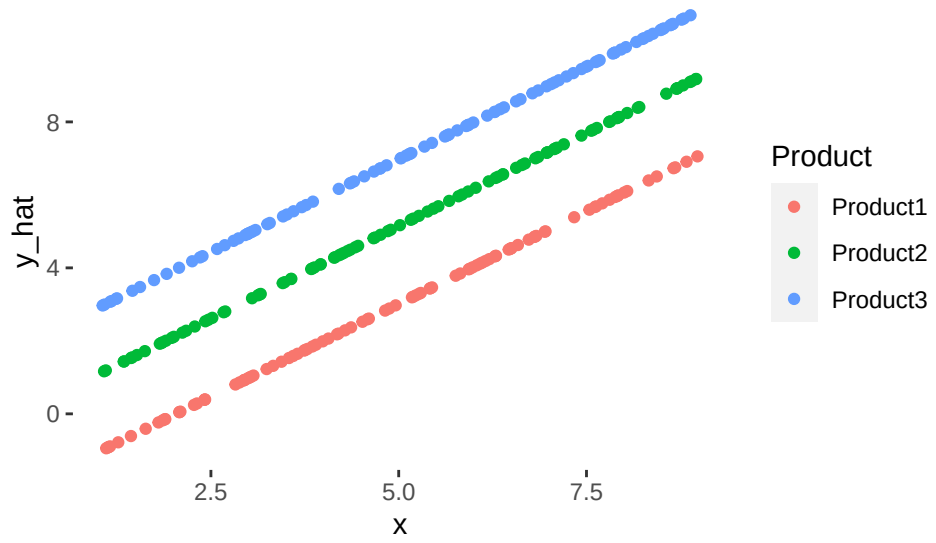
Note how each product has a separate line *(and equation, which produces different projections)*:

```
AnalysisData3$y_hat = predict(mod3, AnalysisData3)
rmse(AnalysisData3$y_hat - AnalysisData3$y)
```

```
## [1] 2.045868
```

And we can visualize our projections:

```
p6 = ggplot(AnalysisData3, aes(x, y_hat, color = Product)) + geom_point() +
  theme(panel.background = element_rect(fill = "white"))
p6
```



Do you see where the "regression line" comes from? This is all generated from the equation: $\hat{y} = Intercept_{product} + \hat{\beta}_x * X$.

So, let's use the equation to predict, and then check against model predictions:

```
AnalysisData3 = AnalysisData3 %>% inner_join(dfCoef, by = "Product") %>%
  mutate(EqY_hat  = Intercept + Slope*x)
sum(AnalysisData3$y_hat - AnalysisData3$EqY_hat)
```

```
## [1] 0
```

Look right? Yep.

*(Note: Normally, we do **NOT** predict using the same data that you use to build (or train) the model. That doesn't prove anything. You usually split your data into validation and test sets at least. I'm doing this here to just to keep it simple while demonstrating concepts. But don't get confused about that..)*

Homework

Take the example in Model 3 and generate your own data. You can use the overall approach, but you can't copy and paste - the data has to be different categorical variables, and different β values. Then show:
1. Model Summary with comments on output *(in # comments)*
2. Plot of variables in color with lines created with abline from coefficients.
3. Predictions using the equation with test to make sure the equation gets the exact same answer as lm got.