Regression Part 1

Homework

Prof. Bisbee

Due Date: 2023-02-22

Motivation: predicting movie revenues

“Nobody knows anything…… Not one person in the entire motion picture field knows for a certainty what’s going to work. Every time out it’s a guess and, if you’re lucky, an educated one.”

-William Goldman, screenwriter of The Princess Bride and All the President’s Men.

In this series of lectures we’re going to see if we can predict which movies will bring in more money. Predicting movie revenues is known to be a very difficult problem, as some movies vastly outperform expectations, while other (very expensive) movies flop badly. Unlike other areas of the economy, it’s not always easy to know which characteristics of movies are associated with higher gross revenues. Nevertheless, we shall persist!

It’s typical for an investor group to have a model to understand the range of performance for a given movie. Investors want to know what range of return they might expect for an investment in a given movie. We’ll try and get started on just such a model.

The Data

Load in libraries.

```
library(tidyverse)
library(plotly)
library(scales)
```

Load in data.

```
mv<-read_rds("../data/mv.Rds")%>%
  filter(!is.na(budget))

glimpse(mv)
```

```
## Rows: 3,191
## Columns: 20
## $ title         <chr> "Almos…
## $ rating        <chr> "R", "…
## $ genre         <chr> "Adven…
## $ year          <dbl> 2000, …
## $ released      <chr> "Septe…
## $ score         <dbl> 7.9, 7…
## $ votes         <dbl> 260000…
## $ director      <chr> "Camer…
## $ writer        <chr> "Camer…
## $ star          <chr> "Billy…
## $ country       <chr> "Unite…
## $ budget        <dbl> 932896…
## $ gross         <dbl> 736774…
## $ company       <chr> "Colum…
## $ runtime       <dbl> 122, 1…
## $ id            <dbl> 877, 6…
## $ imdb_id       <chr> "01818…
## $ bechdel_score <dbl> 3, 3, …
## $ boxoffice_a   <dbl> 505860…
## $ language      <chr> "Engli…
```

This data comes from the Internet Movie Data Based, with contributions from several other sources.

Name	Description
title	Film Title
rating	MPAA Rating
genre	Genre: Adventure, Action etc
year	Year
released	Date released
score	IMDB Score
votes	Votes on IMDB
director	Director
writer	Screenwriter (top billed)
star	Top billed actor
country	Country where produced
runtime	Running time in minutes
id	Alternate ID
imdb_id	IMDB unique ID

Name	Description
bechdel_score	1=two women characters, 2=they speak to each other, 3= about something other than a man, 0=none of the above
box_office_a	Box office take
language	Languages spoken in the movie
gross	Gross revenue

Can we make money in the movie business?

There are a couple of ways we can answer this question. First of all, let's look at gross minus budget (I'm avoiding the word profit because movie finance is deeply weird (https://en.wikipedia.org/wiki/Hollywood_accounting)).

```
mv%>%
  mutate(gross_less_budget=gross-budget)%>%
  summarize(dollar(mean(gross_less_budget,na.rm=TRUE)))
```

```
## # A tibble: 1 × 1
##   `dollar(mean(gross_less_bu…`
##   <chr>
## 1 $113,277,345
```

Looks good! But wait a second, some movies must lose money right? Let's see how many that is:

```
mv%>%
  mutate(gross_less_budget=gross-budget)%>%
  mutate(made_money=ifelse(gross_less_budget>0,1,0))%>%
  group_by(made_money)%>%
  drop_na()%>%
  count()%>%
  ungroup()%>%
  mutate(prop=n/sum(n))
```

```
## # A tibble: 2 × 3
##   made_money     n  prop
##        <dbl> <int> <dbl>
## 1          0   336 0.170
## 2          1  1645 0.830
```

Oof, so something like 18 percent of movies in this dataset lost money. How much did they lose?

```
mv%>%
  mutate(gross_less_budget=gross-budget)%>%
  mutate(made_money=ifelse(gross_less_budget>0,1,0))%>%
  group_by(made_money)%>%
  summarize(dollar(mean(gross_less_budget)))%>%
  drop_na()
```

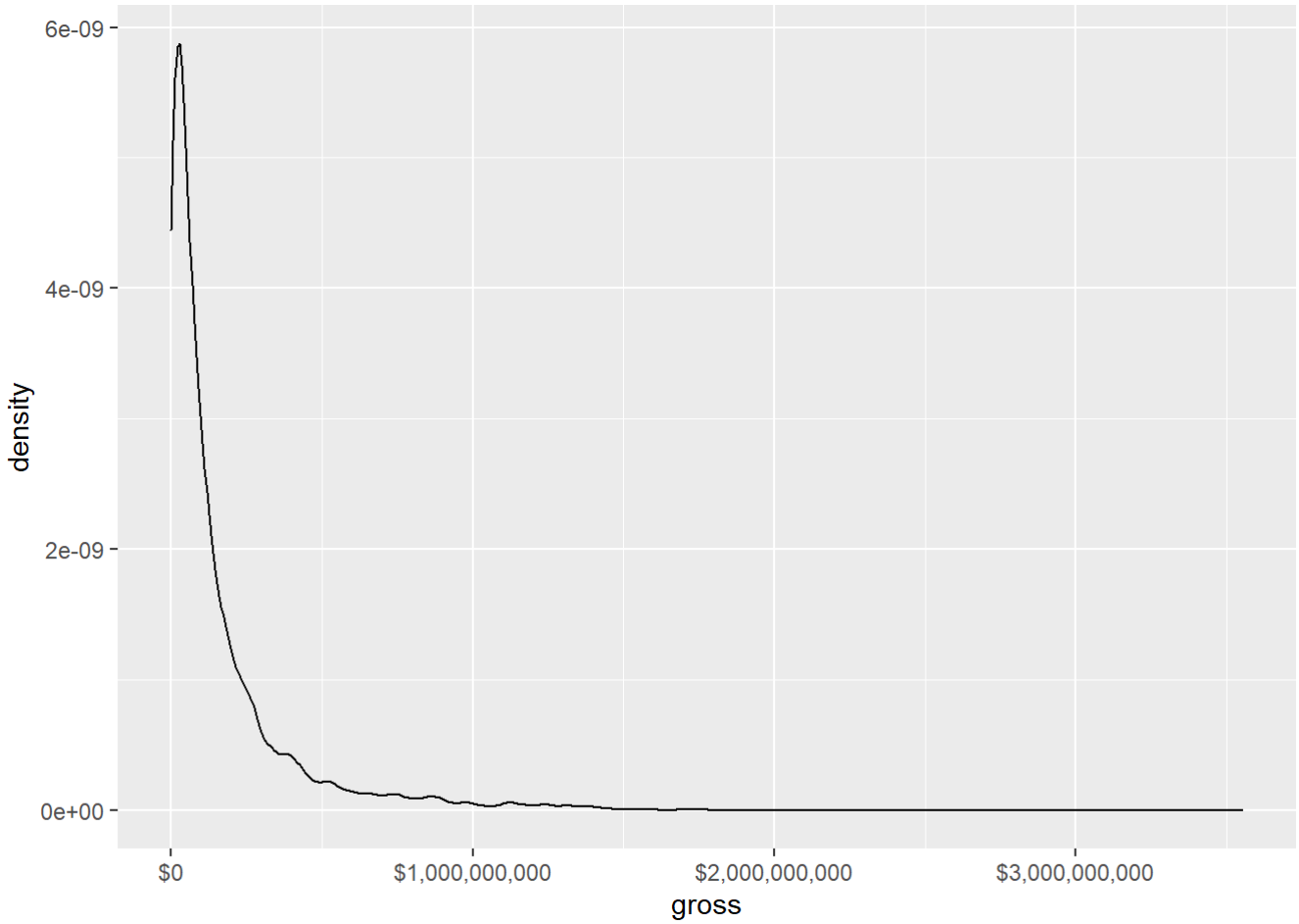
```
## # A tibble: 2 × 2
##   made_money `dollar(mean(gr…`
##        <dbl> <chr>
## 1          0 -$16,705,383
## 2          1 $155,755,006
```

16 million on average. We better get this right! Okay, Let's get started on our model.

Dependent Variable

Our dependent variable will be gross from the movie which is a measure of revenue from all sources: box office, streaming, dvds etc.

```
mv%>%
  ggplot(aes(gross))+
  geom_density()+
  scale_x_continuous(labels=dollar_format())
```

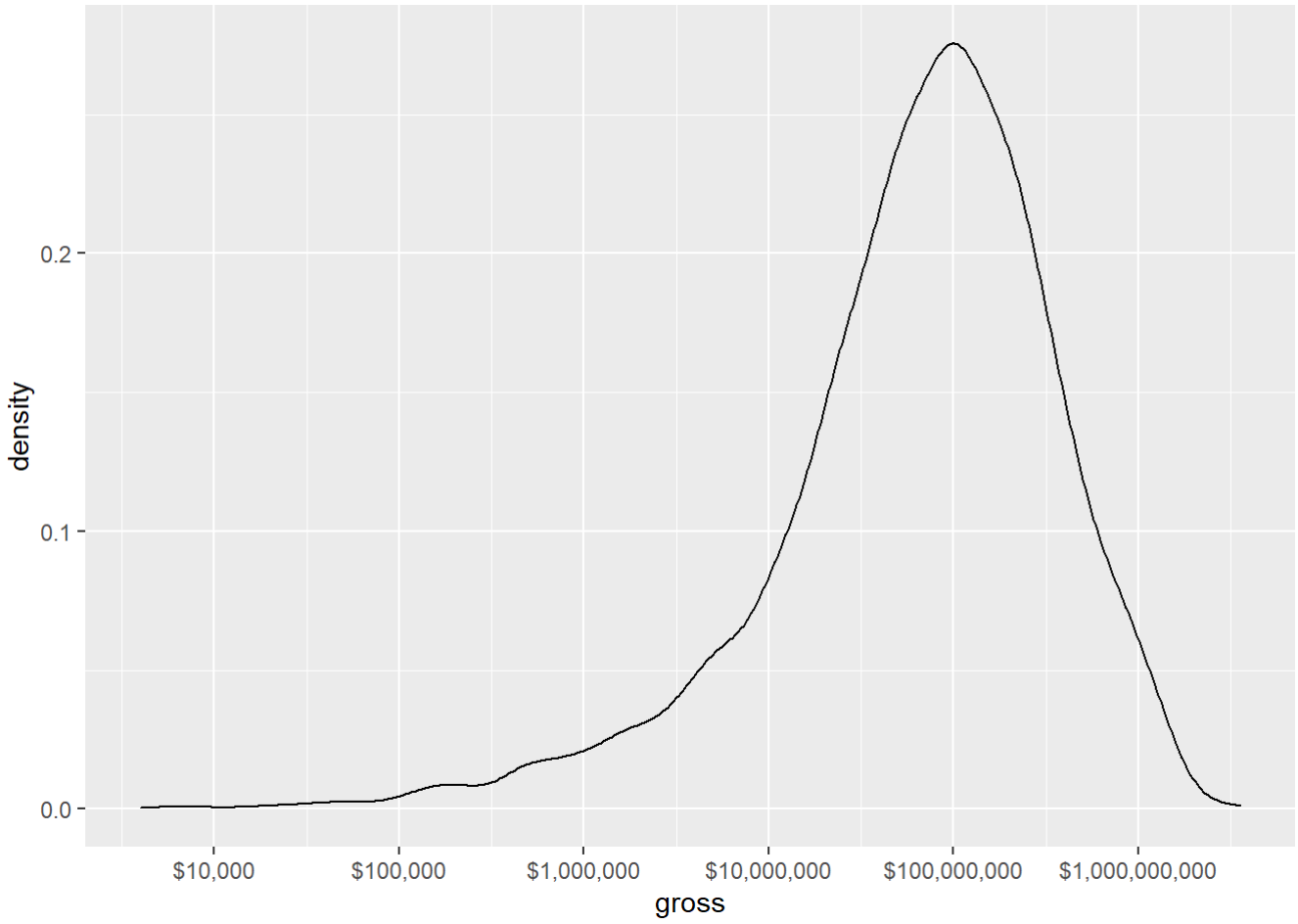


Well, that looks weird. Some movies make A LOT. Most, not so much. We need to work with this on an appropriate scale. When working with almost any data that has to do with money we're going to see things on an exponential scale. Our solution will be to transform this to be on a "log scale". Really what we're doing is taking the natural log of a number, which is the amount that Euler's constant e would need to be raised to in order to equal the nominal amount.

$$log_e(y) = x, \equiv e^x = y$$

Let's transform gross to be on a log scale. We can do this within ggplot to allow for some nicer labeling.

```
mv%>%
  ggplot(aes(x=gross))+
  geom_density()+
  scale_x_continuous(trans="log",
                     labels=dollar_format(),
                     breaks=breaks_log(n=6))
```



That looks somewhat better. Notice how the steps on the log scale imply very different amounts. Effectively what we're doing is changing our thinking to be about percent changes. A 10% increase in gross above 10,000 is 1000. A 10% increase in gross above 100,000 is 10,000. On a log scale, these are (sort of) equivalent steps. Transforming this on the front end will have some implications on the back end, but we'll deal with that in good time.

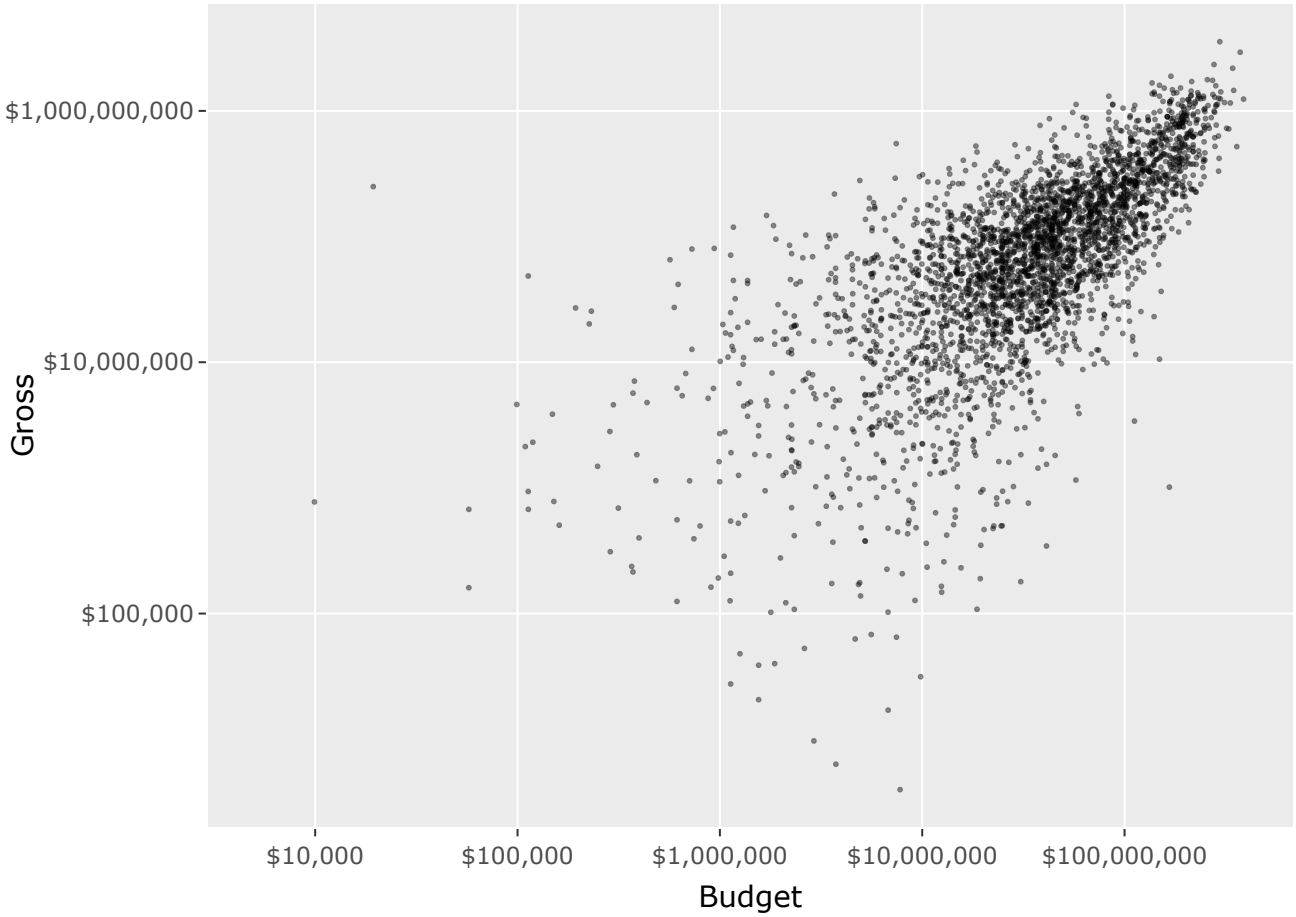
Gross as a Function of Budget

It's quite likely that gross revenues will be related to the budget, but how likely? Let's plot the relationship and find out.

```
gg<-mv%>%
  ggplot(aes(x=budget,y=gross,text=paste(title,"<br>",
                                         "Budget:", dollar(budget), "<br>",
                                         "Gross:" ,dollar(gross))))+
    geom_point(size=.25,alpha=.5)+

  scale_x_continuous(trans="log",
                     labels=label_dollar(),
                     breaks=log_breaks())+
   scale_y_continuous(trans="log",
                     labels=label_dollar(),
                     breaks=log_breaks())+
  xlab("Budget")+
  ylab("Gross")

ggplotly(gg,tooltip = "text")
```



You can navigate this plot and find out what the titles are. This is made possible by the ggplotly command, which uses the plotly library.

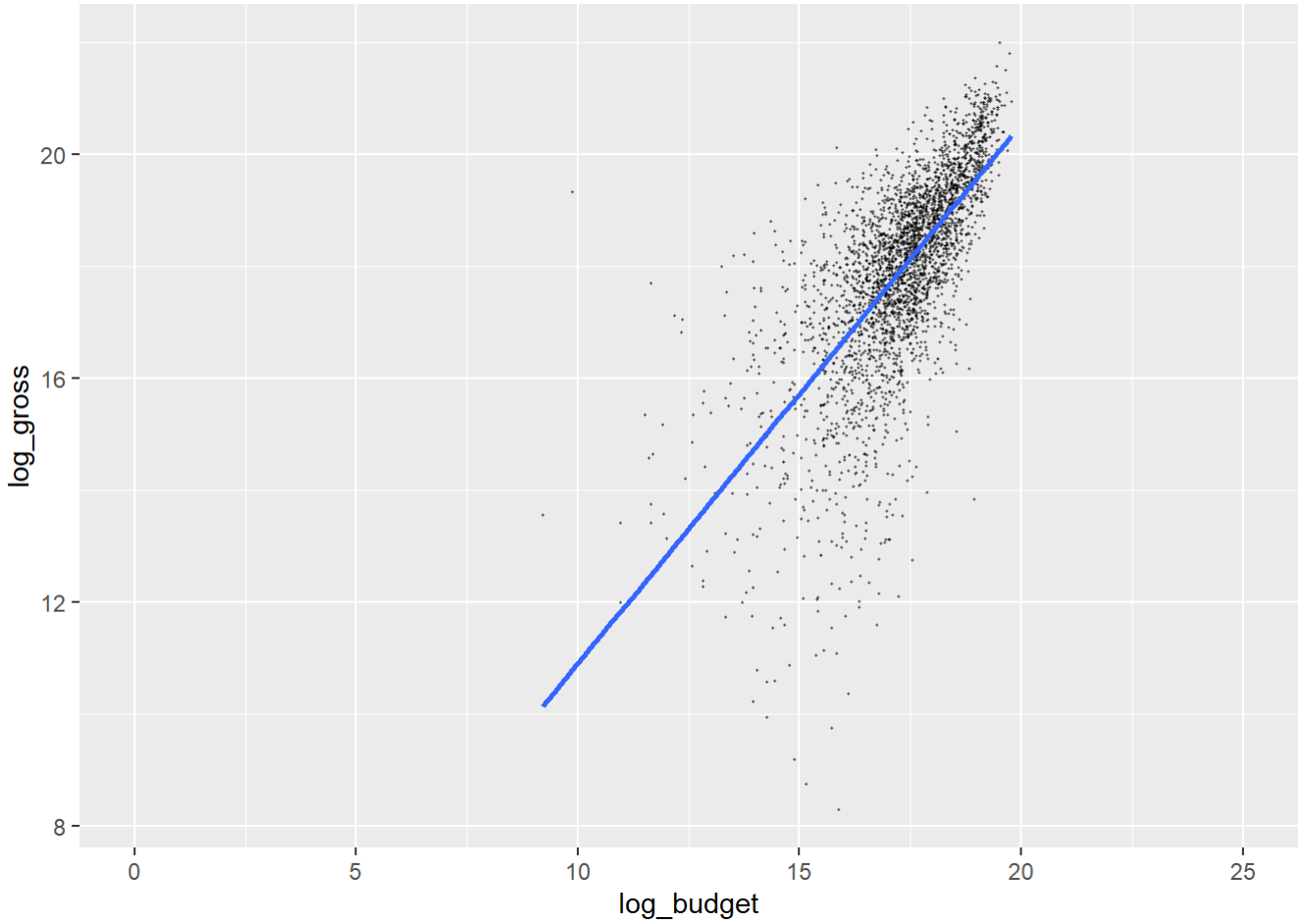
To make things easier, I'm going to create a new variable that is just the log of gross revenues

```
mv<-mv%>%
  mutate(log_gross=log(gross))
```

Basic Linear Model

This plot shows a line fit to the data, with the log of budget on the x axis and the log of gross budget on the y axis.

```
mv%>%
  mutate(log_budget=log(budget))%>%
  ggplot(aes(y=log_gross,x=log_budget))+
  geom_point(size=.25,alpha=.5)+
  geom_smooth(method="lm",se=FALSE)+
  xlim(0,25)
```



Does an assumption of linearity work in this case? Why or why not? Can we summarize this whole dataset in just two numbers?

Note that the `geom_smooth(method = 'lm')` command is running the model for us! Recall from above where we defined the **theoretical** regression as $Y = \alpha + \beta_1 X + \epsilon$. The figure above is now calculating $\hat{Y} = \hat{\alpha} + \hat{beta}_1 X + \hat{\epsilon}$.

How is it doing this? It is using the "linear model" regression function that is included in `R`! This function takes the form `lm(formula,data)` where `formula` is the regression equation and `data` is just the data. For example, if `Y` is logged gross and `X` is logged budget, we would write: `lm(formula = log_gross ~ log_budget,data = mv)`. We can save this model to an object `m1` with the assignment operator `<-` and then look at the results with `summary()`.

```
m1 <- lm(log_gross ~ log_budget,mv %>%
           mutate(log_gross = log(gross),log_budget = log(budget)))

summary(m1)
```

```
##
## Call:
## lm(formula = log_gross ~ log_budget, data = mv %>% mutate(log_gross = log(gross),
##     log_budget = log(budget)))
##
## Residuals:
##     Min      1Q  Median
## -8.2672 -0.6354  0.1648
##      3Q     Max
##  0.7899  8.5599
##
## Coefficients:
##             Estimate
## (Intercept)  1.26107
## log_budget   0.96386
##             Std. Error
## (Intercept)    0.30953
## log_budget     0.01786
##             t value Pr(>|t|)
## (Intercept)   4.074 4.73e-05
## log_budget   53.971  < 2e-16
##
## (Intercept) ***
## log_budget  ***
## ---
## Signif. codes:
##   0 '***' 0.001 '**' 0.01
##   '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.281 on 3177 degrees of freedom
##   (12 observations deleted due to missingness)
## Multiple R-squared:  0.4783, Adjusted R-squared:  0.4782
## F-statistic:  2913 on 1 and 3177 DF,  p-value: < 2.2e-16
```

The results tell us that the `(Intercept)` (which is the same as α in our theory equation, and $\hat{\alpha}$ in our results) is equal to 1.26, and that the `log_budget` (which is the same as β_1 in our theory equation, and $\hat{\beta_1}$ in our results) is equal to 0.96. Thus, for every one unit increase in the logged budget, there is a little less than a 1 unit increase in the gross.

When we are comparing the relationship between a logged outcome and a logged predictor, we can interpret the coefficient as a percent change. Specifically, we would say that a one percent change in the budget corresponds to a 0.96 percent change in gross. For the full breakdown of how to talk about regression interpretations when it comes to logged data, see this website: https://sites.google.com/site/curtiskephart/ta/econ113/interpreting-beta (https://sites.google.com/site/curtiskephart/ta/econ113/interpreting-beta).

Evaluating a Model

The `summary()` function tells us the predicted values for the right hand side of the regression equation: $\hat{\alpha}$ and $\hat{\beta}_1$. But we will also want to know the left-hand side: how good are our predictions for $\hat{Y}$? To answer this question, we want to measure the "errors" that our model makes. In other words, how off are our predicted values compared to the true values?

There are two ways to answer this question. The first is to just **look** at the errors, both as a univariate visualization and as a comparison between the errors and the predictor. The second is to calculate something called the **R**oot **M**ean **S**quared **E**rror, or **RMSE**.

But in either case, we need to calculate the errors themselves first, which are defined as the difference between the true values of Y and the predicted values, denoted $\hat{Y}$. To calculate $\hat{Y}$, we can run the `predict()` function on the model object.

```
predY <- predict(m1)
predY %>% head()
```

```
##        1        2        3
## 18.94904 16.87826 19.46990
##        4        5        6
## 16.45239 17.12049 19.33986
```

These values are exactly corresponding to the original data, meaning we can add them as a new column to our dataset in order to evaluate how good our model is. The more similar the predicted values to the actual values of the movie gross, the better our model!

NB: the number of predictions is not the same as the total number of rows in our data. This is because several of the rows contained missing data in either the outcome (`gross`) or in the predictor (`budget`). We can either create a new dataset that removes these missing values ahead of time, or we can use the information contained in the `m1` object which tells us which rows were dropped due to missing data.

```
mvEval1 <- mv %>%
  slice(-m1$na.action) %>% # Drop any observations that were not used in the regression
  mutate(pred_gross = predY,
         log_gross = log(gross),
         log_budget = log(budget)) %>%
  select(title,log_gross,pred_gross,log_budget)

mvEval1 %>%
  head()
```

```
## # A tibble: 6 × 4
##   title   log_gross pred_gross
##   <chr>       <dbl>      <dbl>
## 1 Almost…      18.1       18.9
## 2 Americ…      17.8       16.9
## 3 Gladia…      20.4       19.5
## 4 Requie…      16.3       16.5
## 5 Memento      17.9       17.1
## 6 Cast A…      20.3       19.3
## # … with 1 more variable:
## #   log_budget <dbl>
```

Now that we have our true Y (`log_gross`) and our predicted $\hat{Y}$ (`pred_gross`), we can calculate the errors (denoted ε) by just subtracting the predicted values from the true values.

$$\varepsilon = Y - \hat{Y}$$

```
mvEval1 <- mvEval1 %>%
  mutate(errors = log_gross - pred_gross)
mvEval1 %>%
  head()
```

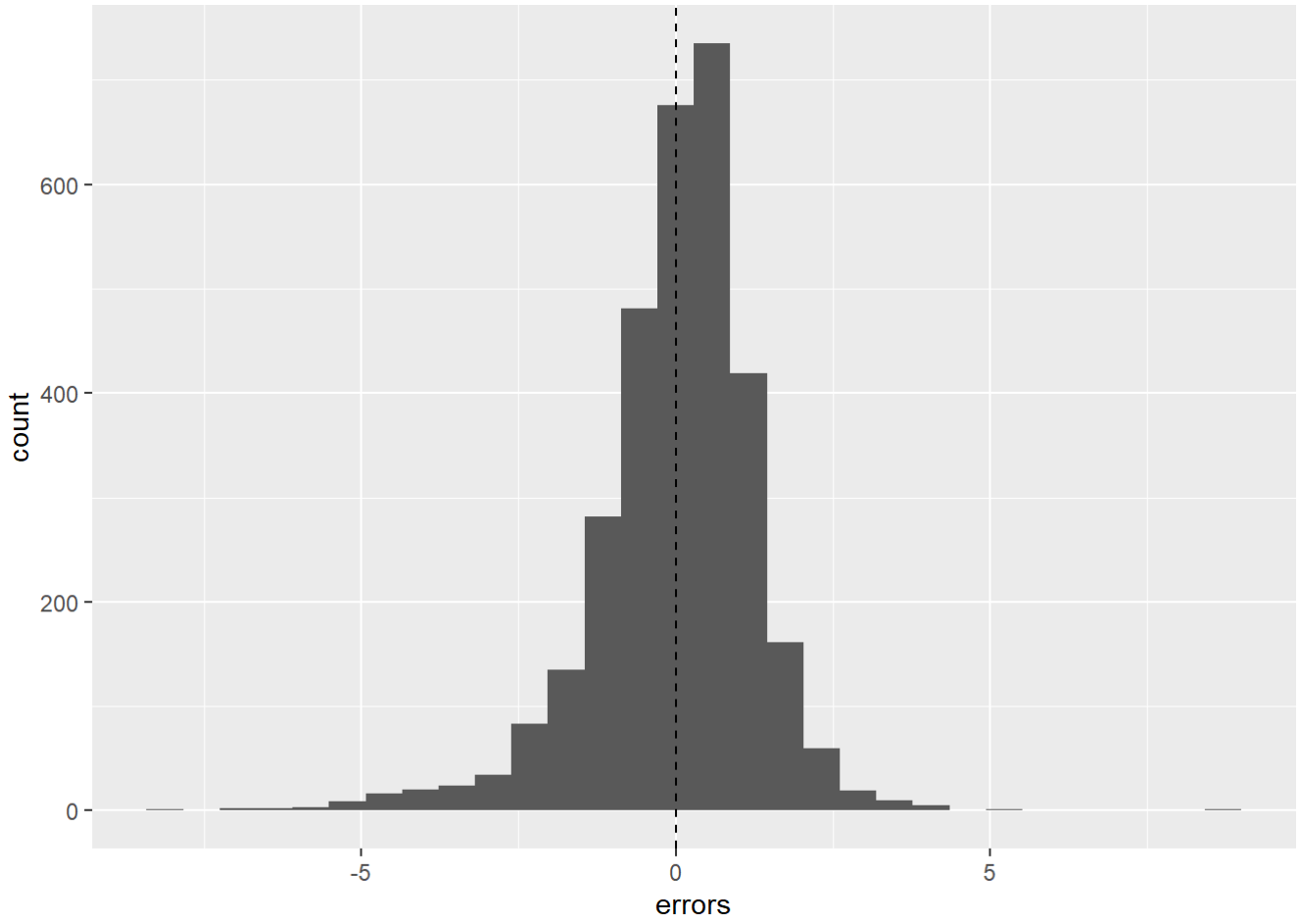
```
## # A tibble: 6 × 5
##   title   log_gross pred_gross
##   <chr>       <dbl>      <dbl>
## 1 Almost…      18.1       18.9
## 2 Americ…      17.8       16.9
## 3 Gladia…      20.4       19.5
## 4 Requie…      16.3       16.5
## 5 Memento      17.9       17.1
## 6 Cast A…      20.3       19.3
## # … with 2 more variables:
## #   log_budget <dbl>,
## #   errors <dbl>
```

As we can see, sometimes our model **overpredicts** the log gross, which corresponds to the negative errors for Almost Famous and Requiem for a Dream. In other cases, it **underpredicts** the log cross, corresponding to positive errors for American Psycho, Gladiator, Memento, and Cast Away.

Visualizing Errors

The first thing to do is to just look at these errors. We do this in two ways. First we just plot them as a histogram or density. By construction, the errors will be centered around zero. However, a “good” model will have errors that are symmetrically distributed, taking the appearance of a bell curve. As we can see, our model does a pretty decent job. There is very mild skew, where we **overpredict** more than we **underpredict** (see that the distribution goes further out to the left below zero than to the right). Nevertheless, the distribution looks pretty good overall.

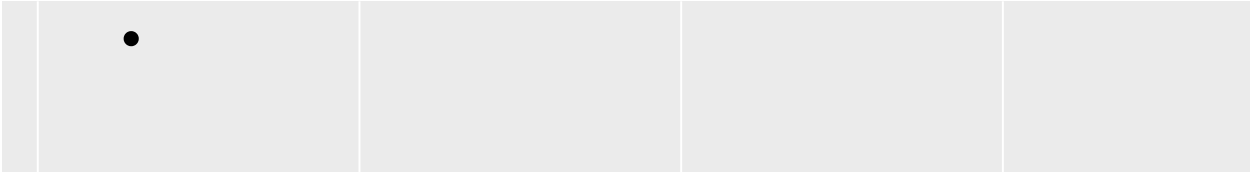
```
mvEval1 %>%
  ggplot(aes(x = errors)) +
  geom_histogram() +
  geom_vline(xintercept = 0,linetype = 'dashed')
```

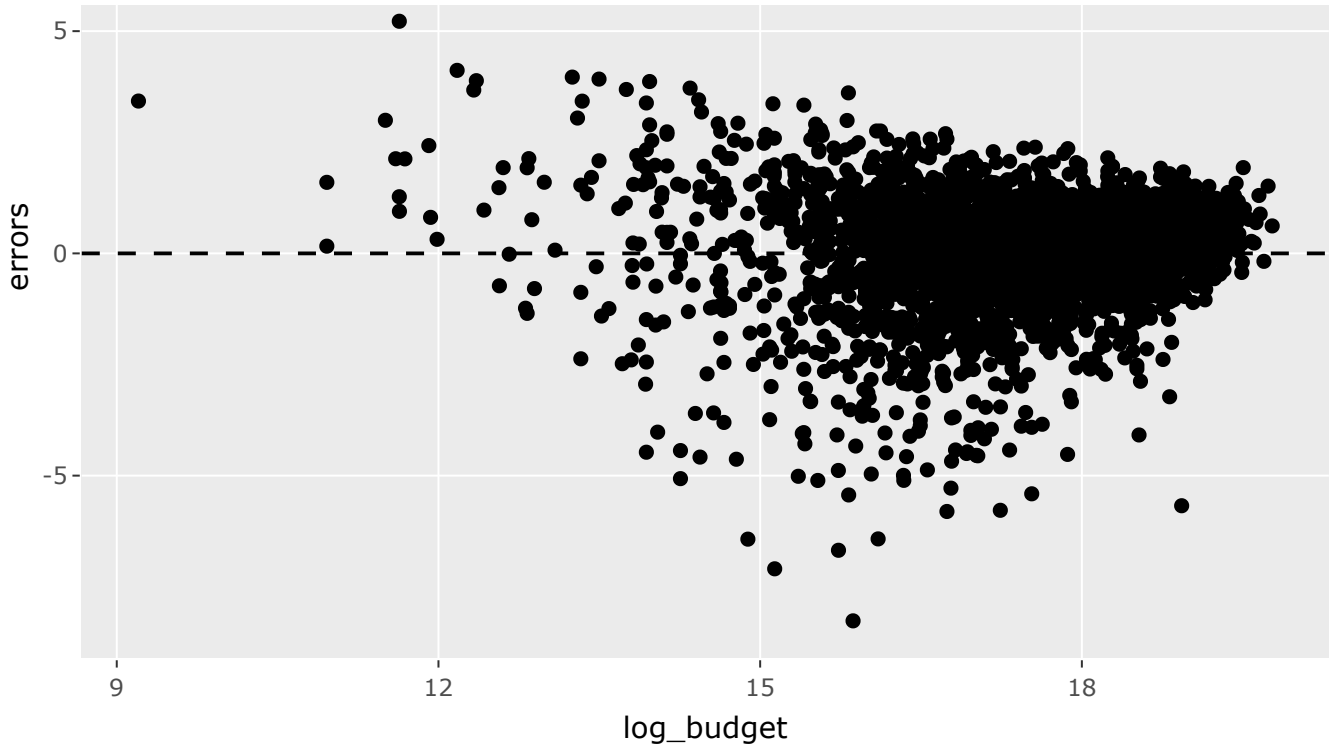


The second thing to do is to visualize the errors as a scatter plot, where the errors are on the y-axis and the X predictor (in this case, `log_budget`) is on the x-axis. Again, the center of this plot will be around zero on the y-axis by design. In this plot, a "good" model will look like a rectangular cloud of errors, where we miss above and below roughly equally, regardless of the budget. We can add a `geom_smooth()` to help visualize the performance. Ideally, we want the line to be flat and zero.

```
p <- mvEval1 %>%
  ggplot(aes(x = log_budget,y = errors,text = title)) +
  geom_point() +
  geom_smooth() +
  geom_hline(yintercept = 0,linetype = 'dashed')

ggplotly(p)
```





As we can see, our model does poorly, as indicated by the curved `geom_smooth()` line. Substantively, this means that we are dramatically **underestimating** lower budget movies and, to a lesser extend, **underestimating** big budget movies. In both cases, our errors are positive.

Recall also that the perfect model is a uniform cloud of data. Meanwhile, our result has much larger mistakes for mid-budget movies compared to big budget movies. In the worst cases, we dramatically **underestimate** Paranormal Activity (the outlier in the top left of the plot), and **overestimate** Ginger Snaps (the outlier in the bottom center of the plot).

Based on these analyses, we would say that our model could be improved.

Root Mean Squared Error

It is always essential to look at the errors visually, with both univariate and multivariate plots. However, we also might want to save a single summary statistic that captures the overall performance, without having to rely on these somewhat subjective visualizations.

To do so, we will use a very standard method, Root Mean Squared Error, or RMSE. To calculate the RMSE, we just follow the recipe that is in its name. We square the errors (**S**), take the average (**M**), then take the square root of the result (**R**). The name RMSE is exactly what RMSE is– neat, huh?

$$RMSE(\hat{Y}) = \sqrt{1/n \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2}$$

We can do this manually, and I think it's instructive to do so at least once for learning. This might seem like a scary equation with a lot of greek letters, but let's break it down into its separate pieces:

1. Let's start with the errors $Y_i - \hat{Y}_i$. Y_i are the true values and $\hat{Y}_i$ are the predictions. To calculate their difference, we just use a `mutate()` command and subtract the predicted values from the true values.

```
mvEval1 <- mvEval1 %>% mutate(diff = log_gross - pred_gross) .
```

2. Now we square this: `mvEval1 <- mvEval1 %>% mutate(sqDiff = diff^2)`.
3. And now we take it's average (note that $\frac{1}{n}\sum_{i=1}^{n} X_i$ is just the way we write average…we literally sum up every value $\sum_{i=1}^{2} X_i$ and then divide this by n, which is the same as multiplying by $\frac{1}{n}$).
   `msEval1 <- mvEval1 %>% summarise(mse = mean(sqDiff))`.
4. Finally, we just take the square root! `sqrt(msEval1$mse)` and we're done!

```
mvEval1 %>%
  mutate(errors = log_gross - pred_gross) %>% # calculate the errors (E)
  mutate(sqErrors = diff^2) %>%               # square the errors (S)
  summarise(mse = mean(sqDiff)) %>%           # take the mean of the errors (M)
  summarise(rmse = sqrt(mse))                 # take the square root the mean (R)
```

```
## Error in `mutate()`:
## ! Problem while
##   computing `sqErrors =
##   diff^2`.
## Caused by error in `diff^2`:
## ! non-numeric argument to binary operator
```

That is a lot of work to calculate this! We can make it easier by calculating the errors (aka "residuals") directly with the `resid()` function.

```
e <- resid(m1)
se <- e^2
mse <- mean(se)
rmse <- sqrt(mse)
rmse
```

```
## [1] 1.280835
```

Calculating rmse from the data we used to fit the line is actually not correct, because it doesn't help us learn about out of sample data. To solve this problem, we need another concept: training and testing.

Training and Testing

The essence of prediction is discovering the extent to which our models can predict outcomes for data that *does not come from our sample*. Many times this process is temporal. We fit a model to data from one time period, then take predictors from a subsequent time period to come up with a prediction in the future. For instance, we might use data on team performance to predict the likely winners and losers for upcoming soccer games.

This process does not have to be temporal. We can also have data that is out of sample because it hadn't yet been collected when our first data was collected, or we can also have data that is out of sample because we designated it as out of sample.

The data that is used to generate our predictions is known as *training* data. The idea is that this is the data used to train our model, to let it know what the relationship is between our predictors and our outcome. So far, we have only worked with training data.

That data that is used to validate our predictions is known as *testing* data. With testing data, we take our trained model and see how good it is at predicting outcomes using out of sample data.

One very simple approach to this would be to cut our data. We could then train our model on half the data, then test it on the other portion. This would tell us whether our measure of model fit (e.g. rmse) is similar or different when we apply our model to out of sample data. That's what we're going to do in this lesson. We'll split the data randomly in two, with one part used to train our models and the other part used to test the model against out of sample data.

You will notice that this approach is very similar to the bootstrapping we have already done.

Note: the `set.seed` command ensures that your random split should be the same as my random split.

Training and Testing Data

The core idea is to separate the data into two random subsets: a training set (where you will estimate a model) and a testing set (where you will test the model).

```
set.seed(123)

# First I'm going to add two columns of logged gross and logged budget to the dataset
mv <- mv %>%
  mutate(log_gross = log(gross),
         log_budget = log(budget))

# Get a list of row numbers for the training set
inds <- sample(1:nrow(mv),size = round(nrow(mv)/2),replace = F) # NB: we set replace = F
for cross validation

# Now use these indices to create two data frames, the first that includes them, and the
second that doesn't include them
train <- mv %>%
  slice(inds)

test <- mv %>%
  slice(-inds)
```

Now we estimate our model on the training set!

```
mTrain <- lm(log_gross ~ log_budget,train)
```

Instead of calculating the RMSE directly on this model though, we want to *use* this model to *predict* outcomes on the test dataset. To do this, we will use the `predict()` command again, but tell it to use a new dataset. To save a few steps, I'm going to add it directly to the `test` dataset.

```
test$predGross <- predict(mTrain,newdata = test)
```

Now we can calculate the RMSE here!

```
e <- test$predGross - test$log_gross
se <- e^2
mse <- mean(se,na.rm=T)
rmse <- sqrt(mse)
```

Great! Now let's put it in a loop, just like with bootstrapping!

```
cvRes <- NULL

for(i in 1:100) {
  # Get a list of row numbers for the training set
  inds <- sample(1:nrow(mv),size = round(nrow(mv)/2),replace = F) # NB: we set replace = F for cross validation

  # Now use these indices to create two data frames, the first that includes them, and the second that doesn't include them
  train <- mv %>%
    slice(inds)

  test <- mv %>%
    slice(-inds)

  mTrain <- lm(log_gross ~ log_budget,train)

  test$predGross <- predict(mTrain,newdata = test)

  e <- test$predGross - test$log_gross
  se <- e^2
  mse <- mean(se,na.rm=T)
  rmse <- sqrt(mse)

  cvRes <- c(cvRes,rmse)
}
```

And now we have a vector of crossvalidated measures of model fit! Take the average to see how well we're doing!

```
mean(cvRes,na.rm=T)
```

```
## [1] 1.286128
```

```
summary(cvRes)
```

```
##    Min. 1st Qu.  Median
##   1.235   1.270   1.290
##    Mean 3rd Qu.    Max.
##   1.286   1.303   1.346
```

Evaluate Predictions in the Testing Data

Is this good? Who knows! RMSE is very context dependent. One way to think about it for us is that the model predicts that a 10 million dollar investment will generate about 18 million: sounds good, right?

```
summary(m1)
```

```
##
## Call:
## lm(formula = log_gross ~ log_budget, data = mv %>% mutate(log_gross = log(gross),
##     log_budget = log(budget)))
##
## Residuals:
##     Min      1Q  Median
## -8.2672 -0.6354  0.1648
##      3Q     Max
##  0.7899  8.5599
##
## Coefficients:
##             Estimate
## (Intercept)  1.26107
## log_budget   0.96386
##             Std. Error
## (Intercept)    0.30953
## log_budget     0.01786
##             t value Pr(>|t|)
## (Intercept)   4.074 4.73e-05
## log_budget   53.971  < 2e-16
##
## (Intercept) ***
## log_budget  ***
## ---
## Signif. codes:
##   0 '***' 0.001 '**' 0.01
##   '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.281 on 3177 degrees of freedom
##   (12 observations deleted due to missingness)
## Multiple R-squared:  0.4783, Adjusted R-squared:  0.4782
## F-statistic:  2913 on 1 and 3177 DF,  p-value: < 2.2e-16
```

```
quick_est<-exp(1.26+ (.96* log(1e7)))
dollar(quick_est)
```

```
## [1] "$18,501,675"
```

Except the rmse says that gross could be between 66 million (yay) and 5.1 million. Right now, that's our prediction– a 10 million dollar investment, base on this model, will make somewhere between 5.1 million and 66 million. Your investors are probably not thrilled and are thinking about just putting the money in T Bills.

```
quick_upper_bound<-exp(1.26+ .96*log(1e7) +1.28)

dollar(quick_upper_bound)
```

```
## [1] "$66,543,859"
```

```
quick_lower_bound<-exp(1.26+ .96* log(1e7) -1.28)

dollar(quick_lower_bound)
```

```
## [1] "$5,144,156"
```