LinearRegression_I_Class6-Completed

March 2, 2021

[61]:
```
import warnings
warnings.filterwarnings('ignore')


import pandas as pd
import numpy as np
from plotnine import *

from sklearn.linear_model import LinearRegression # Linear Regression Model
from sklearn.preprocessing import StandardScaler #Z-score variables
from sklearn.metrics import mean_squared_error, r2_score #model evaluation


%matplotlib inline
```

1 1. Simulating Data

In the following code block, I'm creating FAKE data in order to demonstrate how linear regression works. You won't be tested on how to write this kind of code directly (if you ever need to write it yourself, you'll be able to look it up and ask for help) but try to see if you understand what's going on.

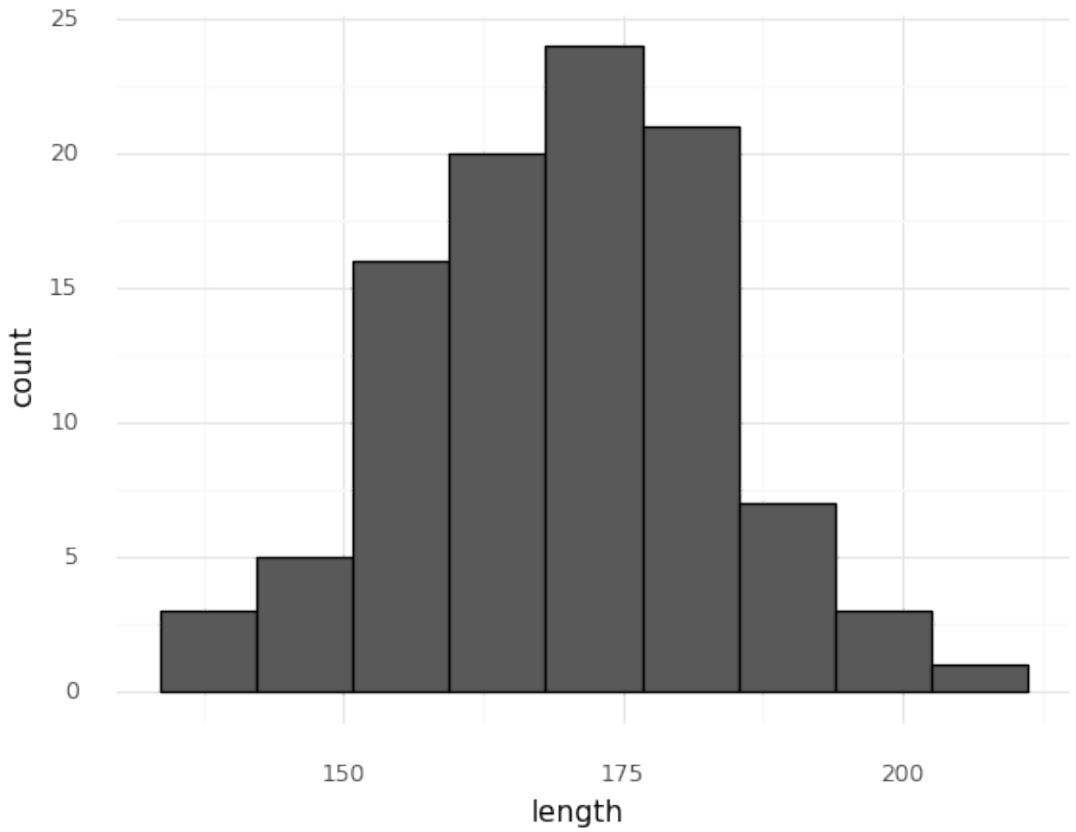
Each time you use a np.random.XXXX function, you'll get *different* fake data, because it's randomly generated. Try it out! If you want your data to come out the SAME each time, add the line `np.random.seed(XXXX)` at the top of the next cell, where `XXXX` is any integer you want like 42, 8675309, or 1234.

1.1 1.1 Linear Regression Data Generation

[62]:
```
# set population values for Cat length
np.random.seed(8675309)

# mean cat length in cm
mean_length_cm = 170

# standard deviation of cat length in cm
sd_length_cm = 15
```

```
# number of samples
n = 100

# generate random data for cat length that follows a normal distribution
length = np.random.normal(loc = mean_length_cm, scale = sd_length_cm, size = n)
```

[63]:
```
# plot cat length data----
(ggplot(pd.DataFrame({"length": length}),
       aes(x = "length")) +
geom_histogram(color = "black") +
theme_minimal())
```



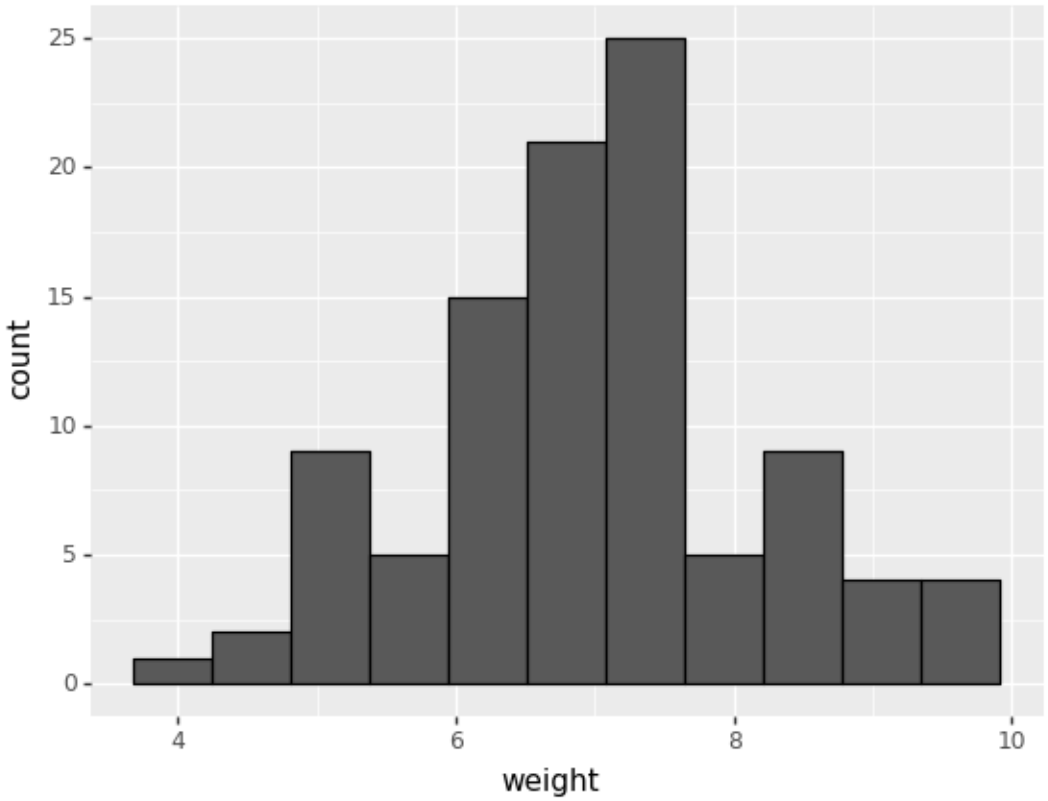
[63]:
```
<ggplot: (8795490170609)>
```

[64]:
```
# Set TRUE relationship between cat length and cat weight----

# true relationship between cat length and cat weight?
# i.e. As length increases by 1 cm, how much do we expect weight to increase?
true_coef = 0.04
```

```
# intercept: the expected weight of a cat that is 0 cm (doesn't make sense but␣
↪that's math for you)
intercept = 0.2
```

[65]:
```
# generate cat weights using array operations----

weight = intercept + length*true_coef + np.random.normal(0,1,n) # weight =␣
↪intercept + length*coefficient + random error
```

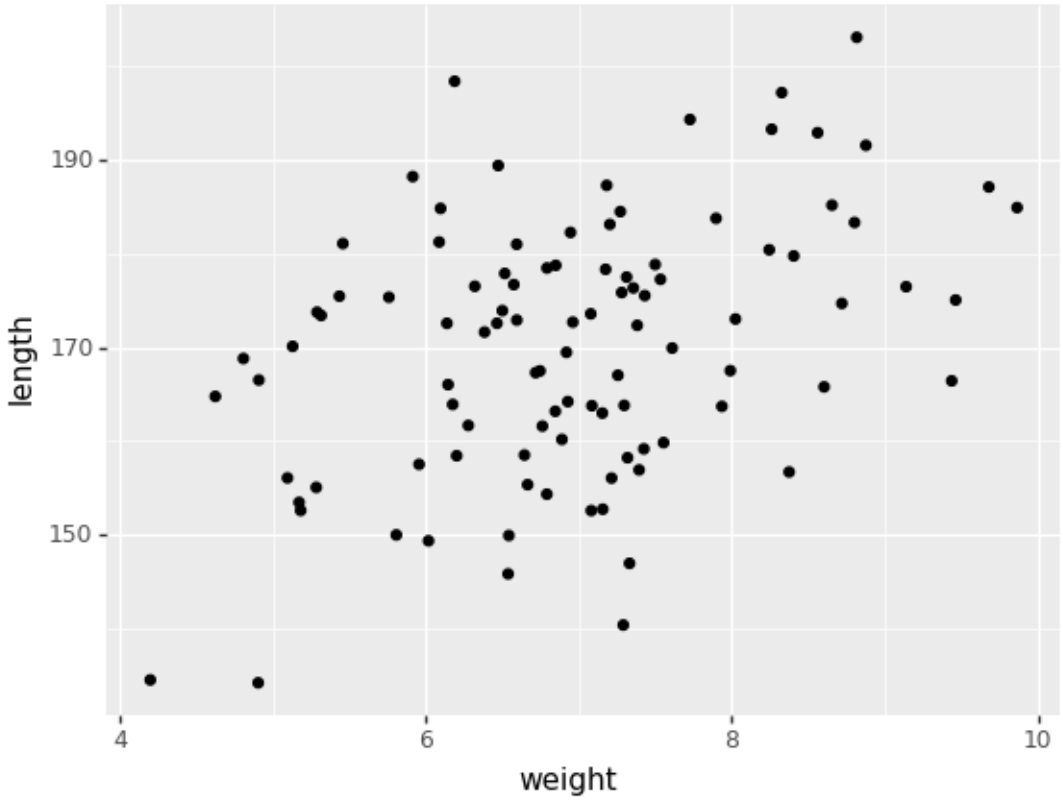
[66]:
```
# plot cat weight data as a histogram----
cats = pd.DataFrame({"length": length, "weight": weight})

################################################
(ggplot(cats, aes(x = "weight")) + geom_histogram(color = "black"))
################################################
```



[66]:
```
<ggplot: (8795493188244)>
```

[67]:
```
# plot cat weight by cat length----
################################################
```

```
(ggplot(cats, aes(x = "weight", y = "length")) + geom_point())
##############################################
```



[67]: <ggplot: (8795493951461)>

1.2 1.2 Fitting a Regression Model

[68]:
```
cats = pd.DataFrame({"length": length, "weight": weight})

features = ["length"]
X = cats[features]
y = cats[["weight"]] #if you don't have the extra brackets, y will be a series␣
 ↪instead of an array and throw an error



##############################################
# run a linear regression
lr = LinearRegression() ## create a linear regression model
lr.fit(X,y)
# fit the linear regression model using X and y
##############################################
```

[68]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

[69]:
```
# grab the coefficients from the model and store them in a data frame----

coef = pd.DataFrame({"Coef": lr.coef_[0], "Names": features})
coef = coef.append({"Coef": lr.intercept_[0], "Names": "intercept"},␣
 ↪ignore_index = True)
coef
```

[69]:

	Coef	Names
0	0.034702	length
1	1.056641	intercept

The coefficient of length is 0.034702, and the intercept is 1.056641. The coefficient of length indicates that as length increases by 1 cm, we expect weight to increase by 0.035 kg.

1.2.1 *Question*

Compare that to the TRUE population values we set above (hint: print `trueCoef` and `intercept`). How close are the numbers from *this* linear regression to the true values? What do you think will happen if we created another random sample? 100 random samples?

[70]:
```
print(true_coef, intercept)
```

```
0.04 0.2
```

The coef for length that I got were close the true value, but not exactly 0.04. The intercept was pretty off since the true value is 0.2 and my value is > 1. If we created a bunch more data sets, I would expect MOST coefs and intercepts to be close to the true values, but some (only a few) might be way off. The more different the estimate is from the true value, the less likely I expect it to be though.

2 2. Fitting Linear Regressions on Multiple Datasets

[71]:
```
def linearRegression(n = 100, trueCoef = 0.04, intercept = 0.2, error_sd = 1):

    # mean cat length in cm
    mean_length_cm = 170
    # standard deviation of cat length in cm
    sd_length_cm = 15

    # generate random data for cat length that follows a normal distribution
    length = np.random.normal(loc = mean_length_cm, scale = sd_length_cm, size␣
 ↪= n)
    weight = intercept + length*true_coef + np.random.normal(0,error_sd,n)
    # weight = intercept + length*coefficient + random error

    cats = pd.DataFrame({"length": length, "weight": weight})

    features = ["length"]
    X = cats[features]
    y = cats[["weight"]] #if you don't have the extra brackets, y will be a␣
 ↪series instead of an array and throw an error

    ################################################
    # run a linear regression
    lr = LinearRegression()

    # fit the linear regression
    lr.fit(X,y)
    ################################################



    # grab the coefficients from the model and store them in a data frame
    coef = pd.DataFrame({"Coef": lr.coef_[0], "Names": features})
    coef = coef.append({"Coef": lr.intercept_[0], "Names": "intercept"},␣
 ↪ignore_index = True)
    return({"coef": coef, "data": cats})
```

[72]:
```
#run regression simulation 500 times
iWouldRun500Regressions = [linearRegression(n = 100, trueCoef = 0.04, intercept␣
 ↪= 0.2) for x in range(0,500)]

# grab coefficients from 500 simulations
coef_df = pd.concat([x["coef"] for x in iWouldRun500Regressions])

# grab coefficients from 500 simulations
```

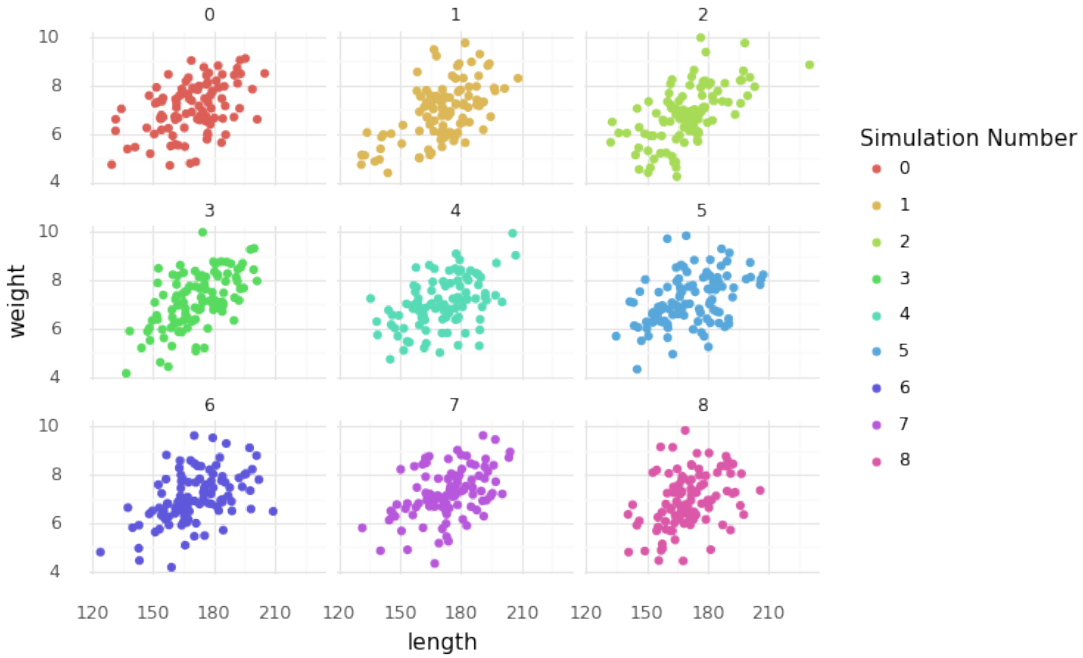
```
data_df = pd.concat([x["data"] for x in iWouldRun500Regressions])

# number simulations 0:499
data_df["simulation_no"] = sorted(list(range(0,500))*n)
coef_df["simulation_no"] = sorted(list(range(0,500))*2)
```

2.1 2.1 Let's Explore!

Now that we've run a bunch of simulations with the SAME true coefficient and intercept (but different random samples), let's look at the results of our 500 regression models.

First, let's just make some scatter plots to see some of the simulations. Notice how similar or different the simulations are from each other.

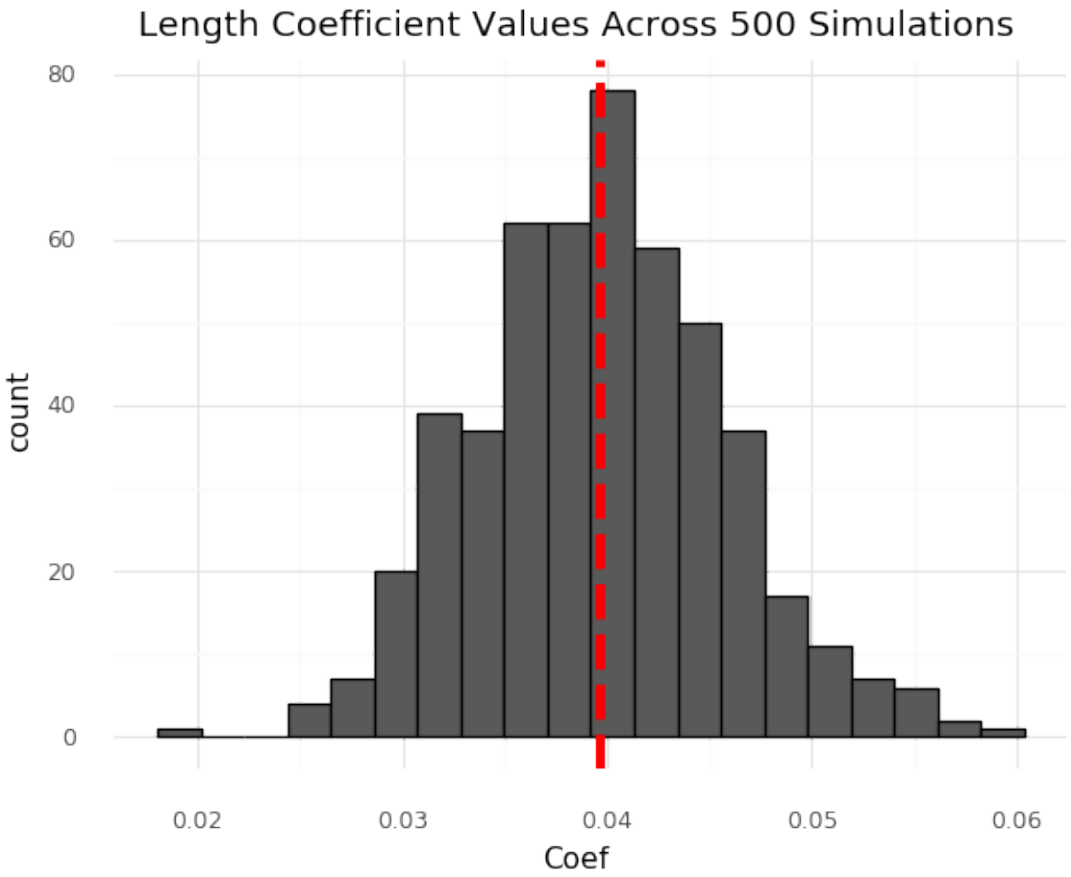
[73]:
```
n_plot = 9

chosen_datasets = data_df["simulation_no"] < 9

(ggplot(data_df.loc[chosen_datasets], aes(x = "length", y = "weight", color =␣
↪"factor(simulation_no)")) +
geom_point() +
facet_wrap("~simulation_no") +
theme_minimal() +
labs(color = "Simulation Number"))
```



[73]: <ggplot: (8795490272112)>

Let's look at the coefficient values from all the linear regressions we ran.

[74]:
```
# plot legnth coef values and mean length coef value (red line)----
coef_only = coef_df["Names"] == "length"


(ggplot(coef_df.loc[coef_only], aes(x = "Coef")) +
geom_histogram(color = "black") +
geom_vline(xintercept = coef_df.loc[coef_only, "Coef"].mean(), color = "red",␣
 ↪linetype = "dashed", size = 2) +
labs(title = "Length Coefficient Values Across 500 Simulations") +
theme_minimal())
```



[74]:
```
<ggplot: (8795493954042)>
```

[75]:
```
print("The mean coefficient for length across the 500 simulations is: " +␣
 ↪str(coef_df.loc[coef_only, "Coef"].mean()))
```

```
The mean coefficient for length across the 500 simulations is:
0.03964458623497787
```

2.1.1 *Question*

Look at the different values you got for the coefficient of length. We set the TRUE coefficient value to be 0.04, think about and describe how spread apart the estimates from our 500 regression models are. Does seeing how different our coefficient estimates can be *change* how you think about the coefficient estimates you get in regression models on real data?

OVer the 500 simulations, we saw a pretty wide range of values, but MOST were close to the true value of 0.04 as predicted! Some of them, though, were pretty different. As low as 0.02 and as high as 0.06. This shows me that when I run a regression on REAL data, my value will *probably* be close to the true value but there's a lot of range so I should take it with a grain of salt.

3 3. Fitting MORE Linear Regressions on Multiple Datasets

This is your chance to play around with the different values in our simulation. Play around with these values to get a better understanding of how different values effect our linear regression output.

3.0.1 *Question*

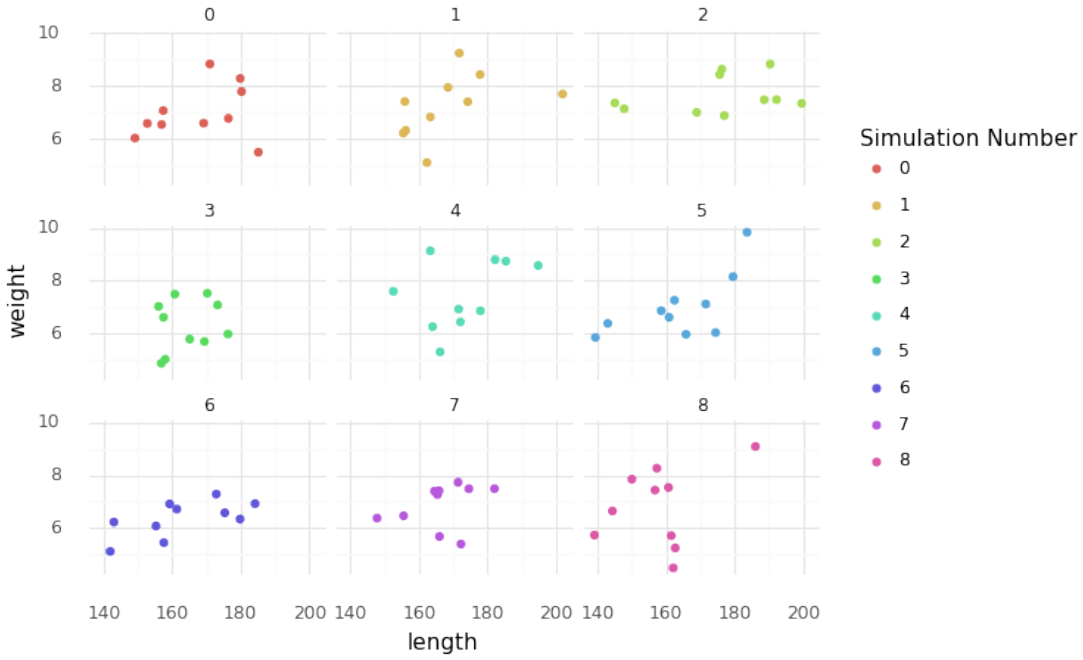
Here are some suggestions:

- Change `n`, the number of data points in each sample, to be very small (say 10), how does this change the results you saw?
- Change `n`, the number of data points in each sample, to be very large (say 1,000), how does this change the results you saw?
- Change the `error_sd` term, this is a measure of how much error is in the model. More error means that data is scattered tightly around the regression line, less error means that the data is scatters very loosely around the regression line. How does changing `error_sd` change the results you originally saw?

[76]:
```
#run regression simulation 500 times----

#--- play around with these numbers-------
n = 10
trueCoef = 0.04
intercept = intercept
error_sd = 1
#----------------------------------------

iWouldRun500More = [linearRegression(n = n, trueCoef = trueCoef, intercept =␣
 ↪intercept, error_sd = error_sd) for x in range(0,500)]

# grab coefficients from 500 simulations
coef_df2 = pd.concat([x["coef"] for x in iWouldRun500More])

# grab coefficients from 500 simulations
data_df2 = pd.concat([x["data"] for x in iWouldRun500More])

# number simulations 0:499
```

```
data_df2["simulation_no"] = sorted(list(range(0,500))*n)
coef_df2["simulation_no"] = sorted(list(range(0,500))*2)
```

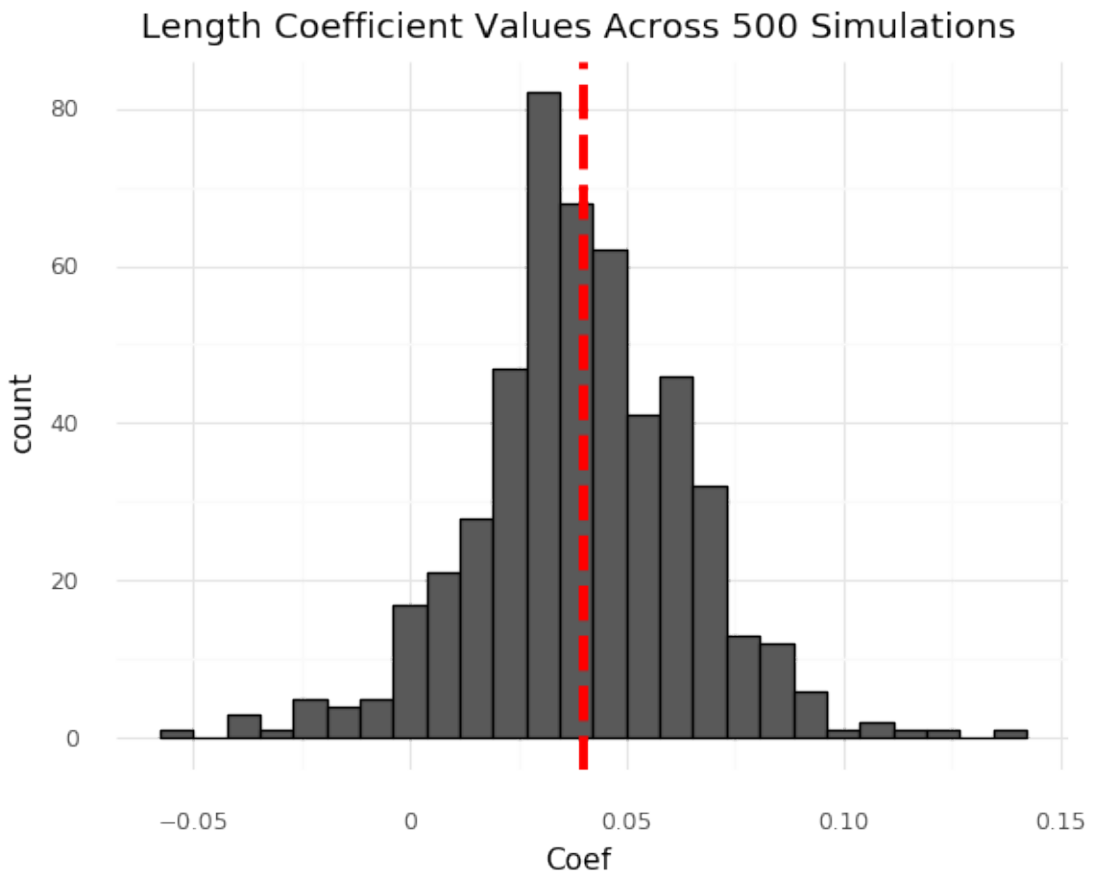
[77]:
```
# plot a few of the simulated data ----

n_plot = 9

chosen_datasets = data_df2["simulation_no"] < 9

(ggplot(data_df2.loc[chosen_datasets], aes(x = "length", y = "weight", color =␣
 ↪"factor(simulation_no)")) +
geom_point() +
facet_wrap("~simulation_no") +
theme_minimal() +
labs(color = "Simulation Number"))
```



[77]:
```
<ggplot: (8795489482575)>
```

[78]:
```
# plot the coefficients and EXPLORE! ----

# plot legnth coef values and mean length coef value (red line)----
coef_only = coef_df2["Names"] == "length"


(ggplot(coef_df2.loc[coef_only], aes(x = "Coef")) +
```

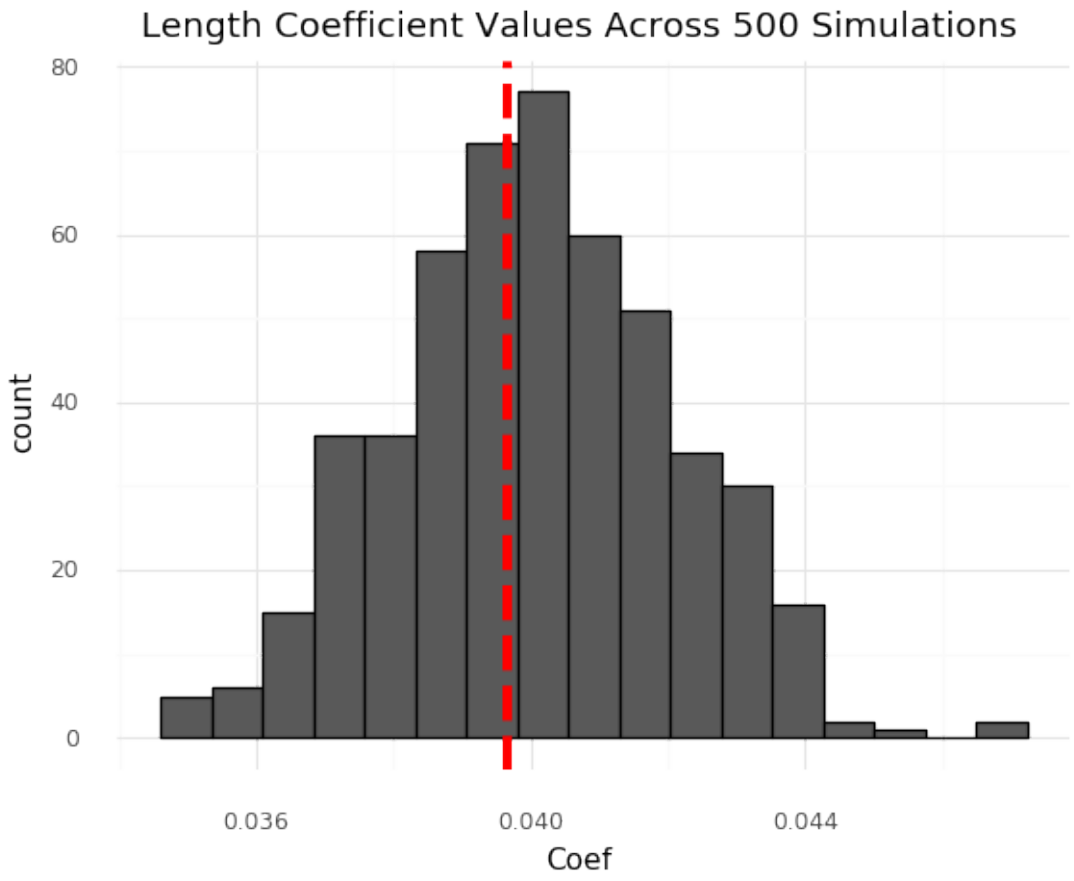
```
geom_histogram(color = "black") +
geom_vline(xintercept = coef_df.loc[coef_only, "Coef"].mean(), color = "red",␣
 ↪linetype = "dashed", size = 2) +
labs(title = "Length Coefficient Values Across 500 Simulations") +
theme_minimal())
```



[78]: <ggplot: (8795490091087)>

[79]:
```
#run regression simulation 500 times----

#--- play around with these numbers-------
n = 1000
trueCoef = 0.04
intercept = intercept
error_sd = 1
#----------------------------------------

iWouldRun500More = [linearRegression(n = n, trueCoef = trueCoef, intercept =␣
 ↪intercept, error_sd = error_sd) for x in range(0,500)]
```

```
# grab coefficients from 500 simulations
coef_df2 = pd.concat([x["coef"] for x in iWouldRun500More])

# grab coefficients from 500 simulations
data_df2 = pd.concat([x["data"] for x in iWouldRun500More])

# number simulations 0:499
data_df2["simulation_no"] = sorted(list(range(0,500))*n)
coef_df2["simulation_no"] = sorted(list(range(0,500))*2)

# plot a few of the simulated data ----

n_plot = 9

chosen_datasets = data_df2["simulation_no"] < 9

(ggplot(data_df2.loc[chosen_datasets], aes(x = "length", y = "weight", color =␣
 ↪"factor(simulation_no)")) +
geom_point() +
facet_wrap("~simulation_no") +
theme_minimal() +
labs(color = "Simulation Number"))

# plot the coefficients and EXPLORE! ----

# plot legnth coef values and mean length coef value (red line)----
coef_only = coef_df2["Names"] == "length"


(ggplot(coef_df2.loc[coef_only], aes(x = "Coef")) +
geom_histogram(color = "black") +
geom_vline(xintercept = coef_df.loc[coef_only, "Coef"].mean(), color = "red",␣
 ↪linetype = "dashed", size = 2) +
labs(title = "Length Coefficient Values Across 500 Simulations") +
theme_minimal())
```



[79]: <ggplot: (8795492715409)>

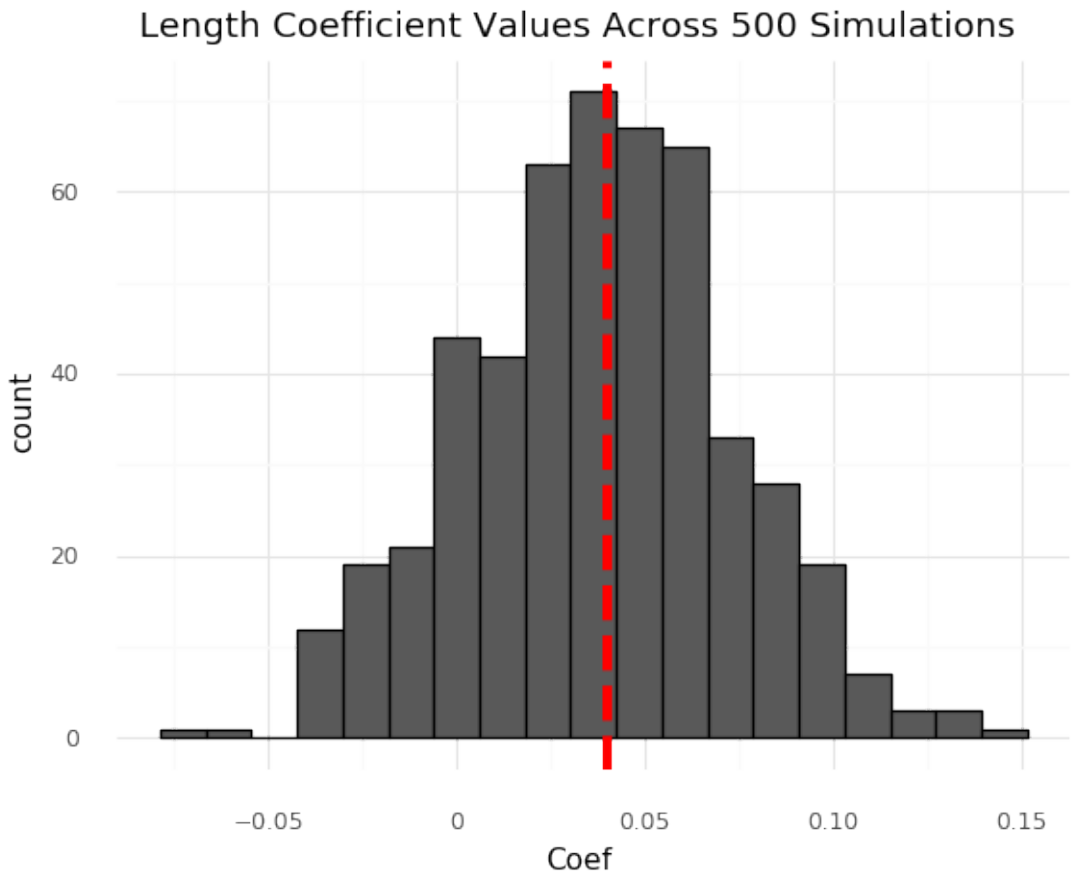
3.0.2 YOUR REFLECTIONS HERE

With less data, the estimates for the coefficients can be PRETTY wrong. Some of them were negative, and others were HUGE (0.15) compared to the true value. That shows that the estimates are more variable when we have less data.

Similarly, when running it with 1000 data points per sample, we see that even though there's some spread in the estimates of the coefficient, most are within 0.004 of the TRUE value which shows that estimates are more stable when we have large amounts of data.

[80]:
```
#run regression simulation 500 times----

#--- play around with these numbers-------
n = 100
trueCoef = 0.04
intercept = intercept
error_sd = 5
#----------------------------------------
```

```
iWouldRun500More = [linearRegression(n = n, trueCoef = trueCoef, intercept =␣
↪intercept, error_sd = error_sd) for x in range(0,500)]

# grab coefficients from 500 simulations
coef_df2 = pd.concat([x["coef"] for x in iWouldRun500More])

# grab coefficients from 500 simulations
data_df2 = pd.concat([x["data"] for x in iWouldRun500More])

# number simulations 0:499
data_df2["simulation_no"] = sorted(list(range(0,500))*n)
coef_df2["simulation_no"] = sorted(list(range(0,500))*2)

# plot a few of the simulated data ----

n_plot = 9

chosen_datasets = data_df2["simulation_no"] < 9

(ggplot(data_df2.loc[chosen_datasets], aes(x = "length", y = "weight", color =␣
↪"factor(simulation_no)")) +
geom_point() +
facet_wrap("~simulation_no") +
theme_minimal() +
labs(color = "Simulation Number"))

# plot the coefficients and EXPLORE! ----

# plot legnth coef values and mean length coef value (red line)----
coef_only = coef_df2["Names"] == "length"


(ggplot(coef_df2.loc[coef_only], aes(x = "Coef")) +
geom_histogram(color = "black") +
geom_vline(xintercept = coef_df.loc[coef_only, "Coef"].mean(), color = "red",␣
↪linetype = "dashed", size = 2) +
labs(title = "Length Coefficient Values Across 500 Simulations") +
theme_minimal())
```



[80]: <ggplot: (8795490839007)>

Increasing the error also increases the instability of the coefficient estimates. Less error means more accurate estimates.

3.0.3 *Question*

In prediction models, we often care most about the *accuracy* of the model. In Linear Regression we often measure that using the mean squared error. Using the `mean_squared_error()` function from sklean, and the other code in this notebook as an example, change the function `linearRegression()` below to calculate and store the mean squared error for each regression in a new dataframe, `mse`, and add it to the dictionary that the function returns. Then run the function 500 times, and plot a histogram of the mean squared errors.

What does this histogram tell you about the accuracy of our models?

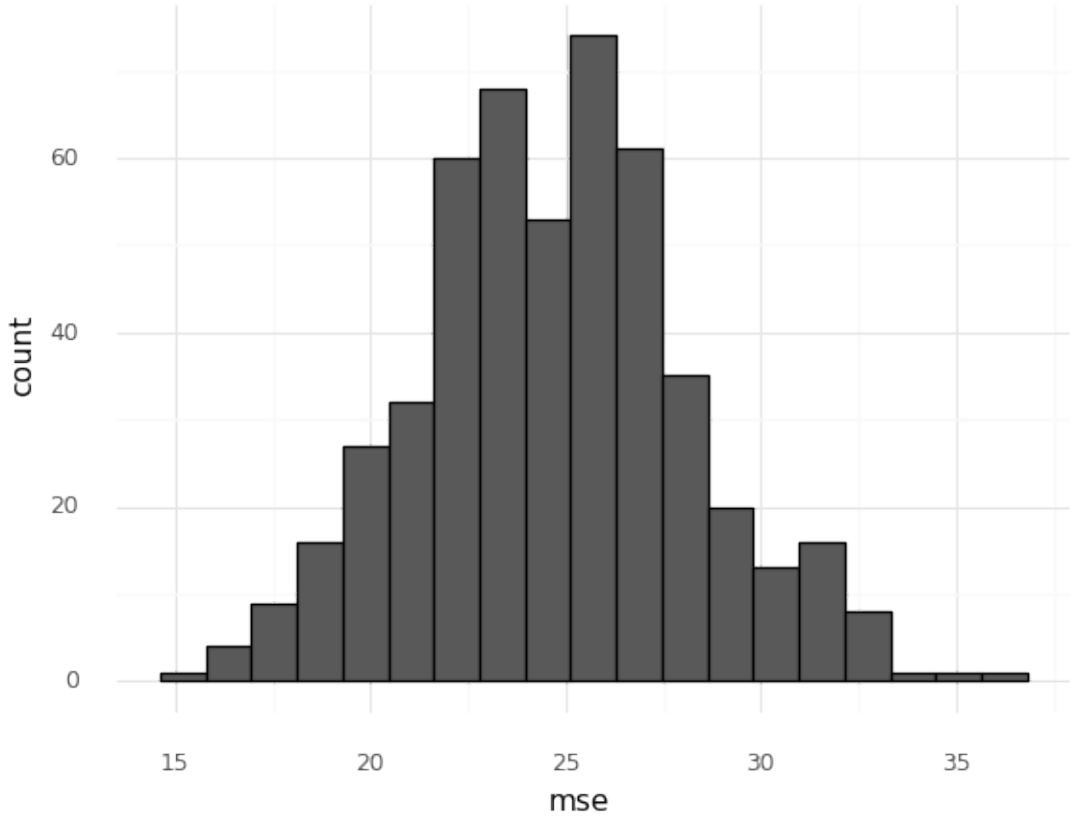
[81]:
```
def linearRegression(n = 100, trueCoef = 0.04, intercept = 0.2, error_sd = 1):

    # mean cat length in cm
    mean_length_cm = 170
```

```
    # standard deviation of cat length in cm
    sd_length_cm = 15

    # generate random data for cat length that follows a normal distribution
    length = np.random.normal(loc = mean_length_cm, scale = sd_length_cm, size␣
↪= n)

    # weight = intercept + length*coefficient + random error
    weight = intercept + length*true_coef + np.random.normal(0,error_sd,n)


    cats = pd.DataFrame({"length": length, "weight": weight})


    features = ["length"]
    X = cats[features]
    y = cats[["weight"]] #if you don't have the extra brackets, y will be a␣
↪series instead of an array and throw an error

    ################################################
    # run a linear regression
    lr = LinearRegression()
    lr.fit(X,y)
    # fit the model

    ################################################

    ##### ADD MSE CODE HERE ######
    mse = mean_squared_error(y, lr.predict(X))
    ################################

    # grab the coefficients from the model and store them in a data frame
    coef = pd.DataFrame({"Coef": lr.coef_[0], "Names": features})
    coef = coef.append({"Coef": lr.intercept_[0], "Names": "intercept"},␣
↪ignore_index = True)

    ################################################
    return({"coef": coef, "data": cats, "mse": mse}) ##### DON'T FORGET to add␣
↪the new mse data frame too! #####
    ################################################
```

[82]:
```
# Run your new simulations-----

iWouldRun500More2 = [linearRegression(n = n, trueCoef = trueCoef, intercept =␣
↪intercept, error_sd = error_sd) for x in range(0,500)]

# grab coefficients from 500 simulations
```

```
coef_df3 = pd.concat([x["coef"] for x in iWouldRun500More2])

# grab coefficients from 500 simulations
data_df3 = pd.concat([x["data"] for x in iWouldRun500More2])

# grab mse from 500 simulations
mse_df3 = pd.DataFrame({"mse":[x["mse"] for x in iWouldRun500More2]})

# number simulations 0:499
data_df3["simulation_no"] = sorted(list(range(0,500))*n)
coef_df3["simulation_no"] = sorted(list(range(0,500))*2)
mse_df3["simulation_no"] = list(range(0,500))
```

[83]:
```
# plot newly calculated mses ----
mean_mse = mse_df3.mse.mean()
(ggplot(mse_df3, aes(x = "mse")) +
 geom_histogram(color = "black") +
 theme_minimal())
```



[83]: <ggplot: (8795492199349)>

The MSE is also variable between samples, which means that even though the MSE estimate I get from my model is a good guess, it might truly be a little higher or a little lower than my estimate.

4 4. Interpreting Linear Regression Coefficients

While sometimes we just want the predictions from a linear regression model, we often will be asked to interpret the coefficients as well.

Use the table of coefficients you create below to answer the following questions.

4.0.1 *Question*

- Do you think the assumption of linearity is valid for this model?
- How will a 1 oz increase in weight change the predicted value for Amazon Price?
- What does the intercept represent?
- Using the values from the coefficient table, predict the Amazon Price for a book that has a list price of 15.99, a Weight of 4 oz, and 428 pages. Show your work using code.
- If we took another random sample of Amazon books from the same population, how do you expect the coefficients from the model would/could change?

[84]:
```
d = pd.read_table("https://raw.githubusercontent.com/cmparlettpelleriti/
 ↪CPSC392ParlettPelleriti/master/Data/amazon-books.txt")
predictors = ["List Price", "Weight (oz)", "NumPages"]
outcome = ["Amazon Price"]
###############################################

#get rid of missing values
d = d.dropna()

# z score predictors
X = d[predictors]
y = d[outcome]

z = StandardScaler()

Xz = z.fit_transform(X)

# create regression

lr = LinearRegression()

# fit model

lr.fit(Xz,y)

###############################################
```

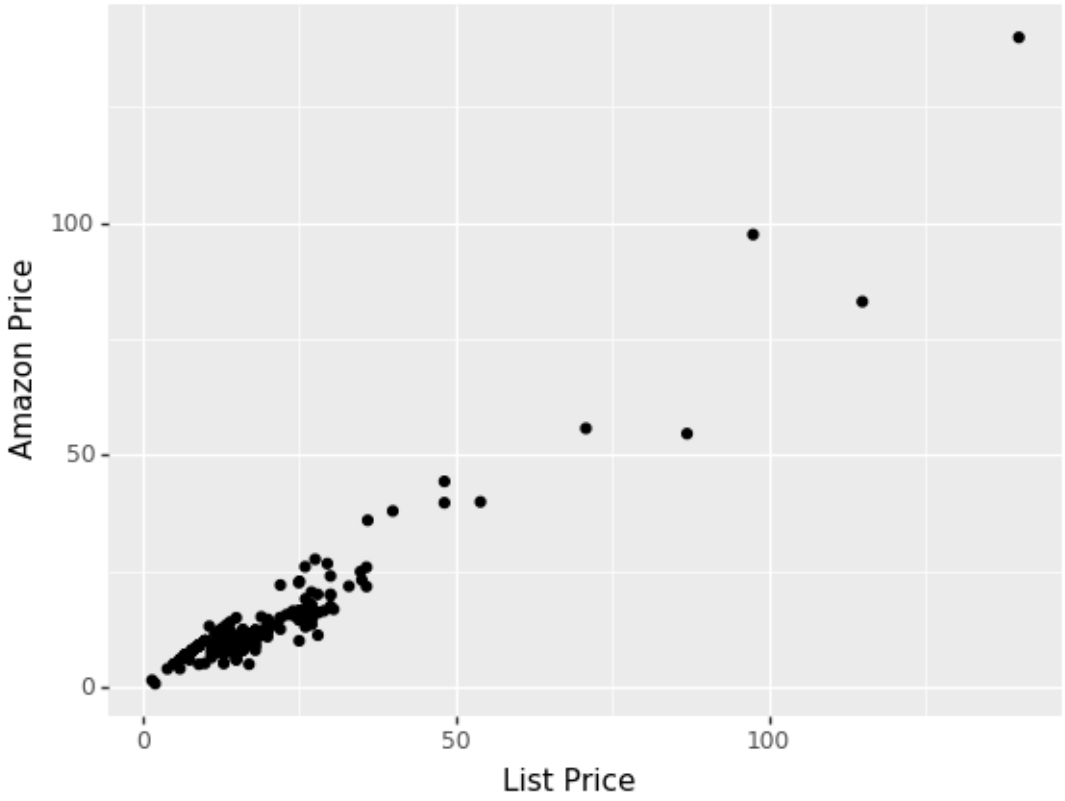
[84]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

[85]:
```
##############################################
# print out table of coefficients
coef = pd.DataFrame({"Coef": lr.coef_[0], "Names": predictors})
coef = coef.append({"Coef": lr.intercept_[0], "Names": "intercept"},␣
↪ignore_index = True)
coef
##############################################
```

[85]:

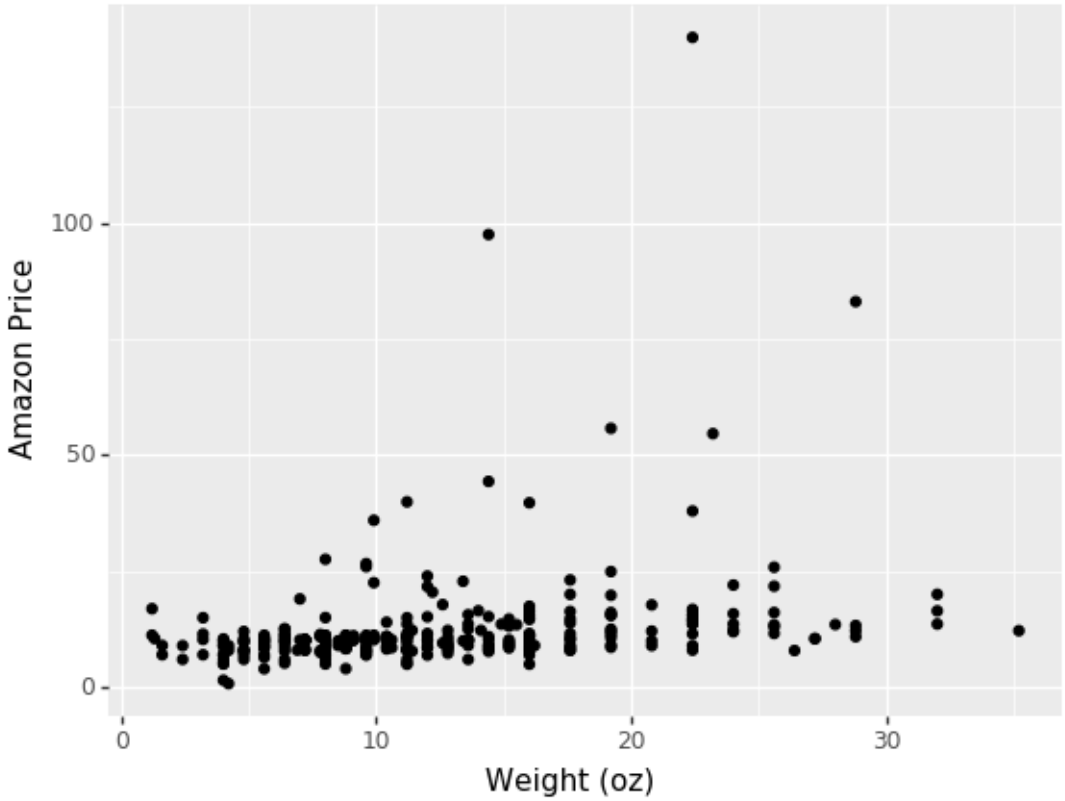
	Coef	Names
0	11.461940	List Price
1	-0.989749	Weight (oz)
2	-0.357650	NumPages
3	12.587645	intercept

[86]:
```
# linearity
ggplot(d, aes(x = "List Price", y = "Amazon Price")) + geom_point()
```



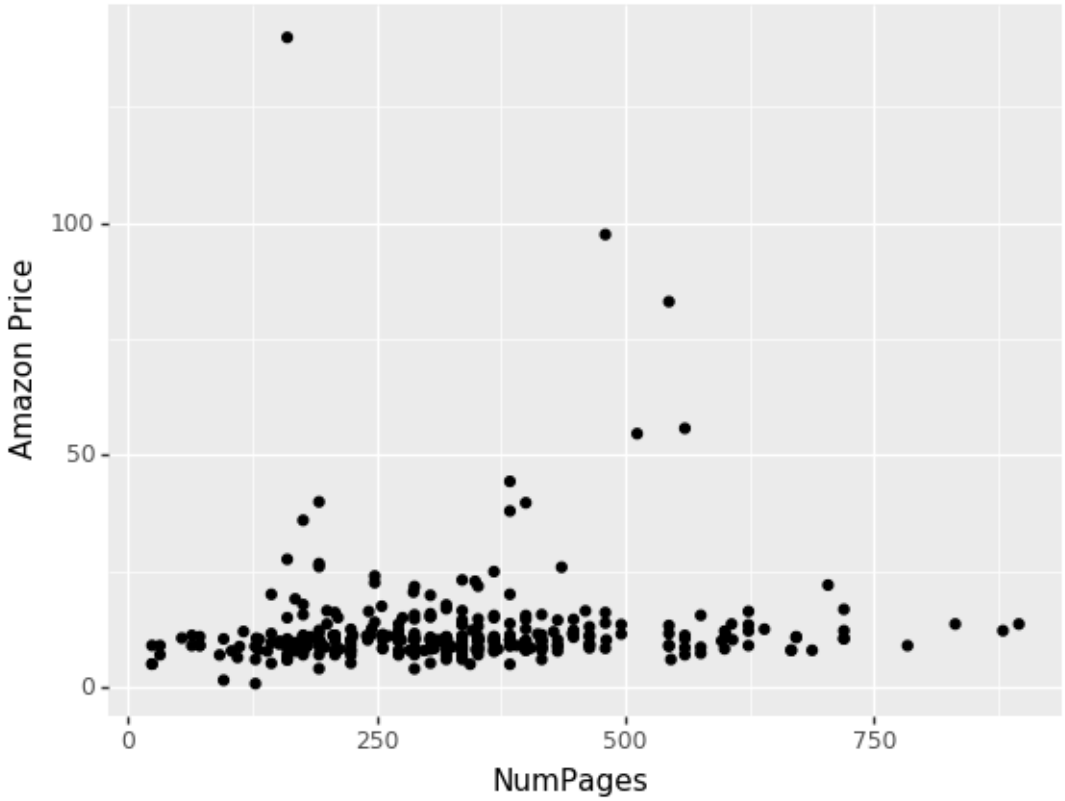
[86]: <ggplot: (8795491580115)>

[87]:
```
# linearity
ggplot(d, aes(x = "Weight (oz)", y = "Amazon Price")) + geom_point()
```



[87]: <ggplot: (8795492465309)>

[88]:
```
# linearity
ggplot(d, aes(x = "NumPages", y = "Amazon Price")) + geom_point()
```



[88]: <ggplot: (8795490596437)>

4.0.2 linearity

The relationships look roughly linear, although for numPages and Weight there are some outliers that might negatively affect the model. We should check whether those data points are valid, and see how they affect our model.

4.0.3 Coefficient Interpretations

A 1 oz increase in Weight would cause a -0.989749/sd(Weight) decrease in predicted amazon price. Remember when you z score, things are in terms of STANDARD DEVIATIONS not raw units (like oz)

The intercept for this model represents the predicted Amazon price when all the variables have the mean value (a 0 z-score represents a point at the mean).

[89]:
```
# Predictions

values = np.array([15.99,4,428]).reshape(1,-1)

valuesz = z.transform(values)
```

```
valuesz = np.append(valuesz, 1)

print("The predicted price is: " + str(np.sum(coef["Coef"] * valuesz)))
```

```
The predicted price is: 11.949664143636944
```

4.0.4 Another Sample

If we took another sample, I'd expect the values for the sample the be similar to the ones we got, but not exactly the same. There's always some variation in the sample. I'd be surprised if the estimates for the coefficients were SUPER different from the ones we got.

5 BONUS (Optional). Violations of Linearity

For some Optional data exploration, use the first cell (#nonLinReg) to simulate a dataset where the relationship between the predictor and the outcome are non-linear, meaning they violate the assumption of linearity. Use the second cell (#LinReg) to simulate a dataset where the relationship between the predictor and the outcome are linear. For both, create plots of 1) the predictor vs the outcome, 2) the predicted values vs the error and 3) a histogram of the errors.

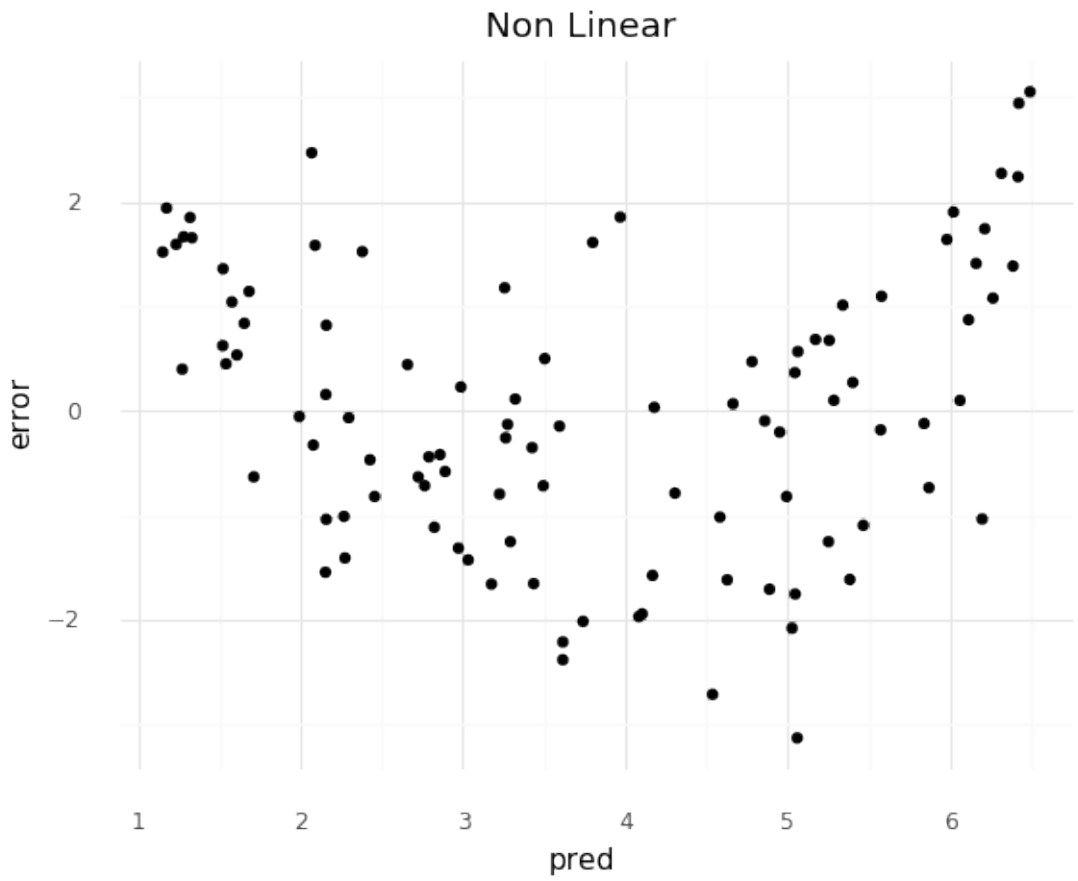
5.0.1 *Question*

- Compare patterns you see in the data that does *not* violate the assumption of linearity vs. the data that does. What's different?
- What are some consequenses of violating the assumption of linearity?

[90]:
```
#nonLinReg----------
x = np.random.uniform(low = -2, high = 2, size = 100)
y = 2 + np.exp(x) + np.random.normal(0,1,100)
# y = 2 + exp(x) + random error

df = pd.DataFrame({"x": x, "y": y})


# regression model
x = x.reshape(-1,1) #have to do this if there's only 1 predictor in the model

nlr = LinearRegression()
nlr.fit(x,y)

pred = nlr.predict(x)

df["pred"] = pred #add to df

df["error"] = y - pred
```

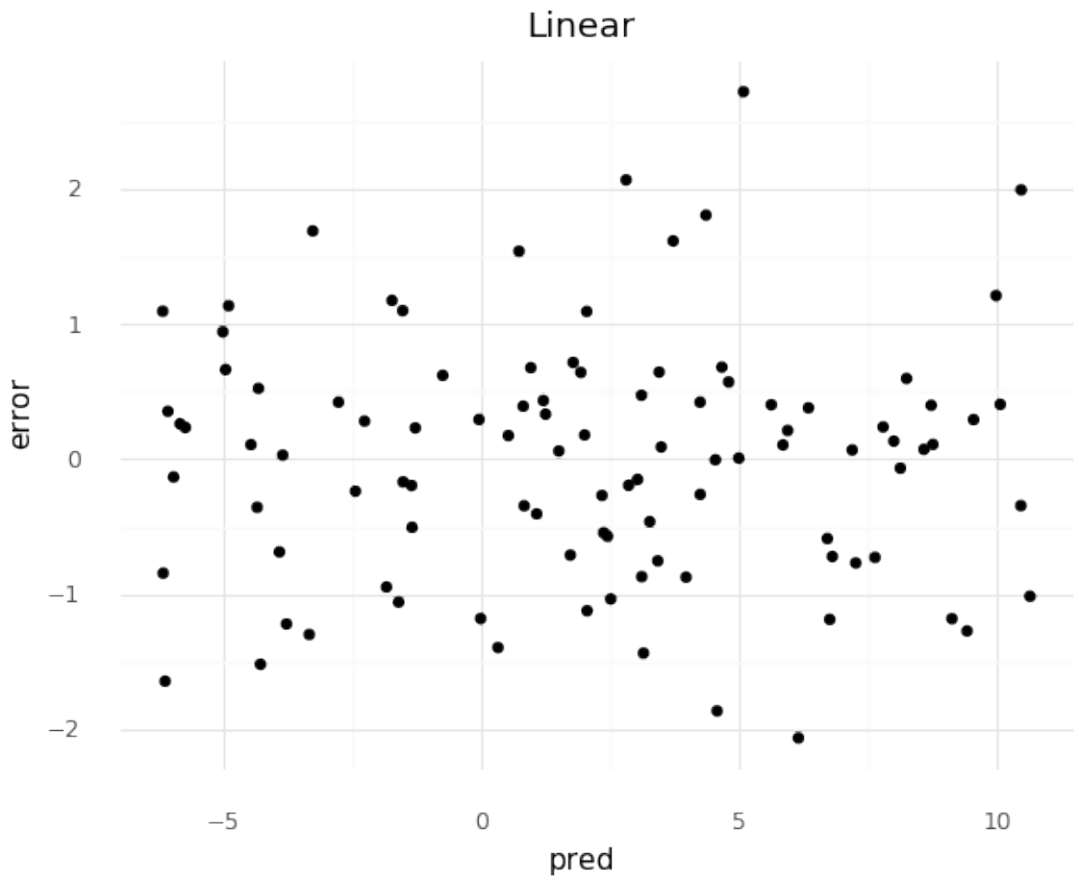
[91]:
```
#LinReg----------
x2 = np.random.uniform(low = -2, high = 2, size = 100)
y2 = 2 + 4.5*x2 + np.random.normal(0,1,100)

# y = 2 + 4.5x + random error

df2 = pd.DataFrame({"x": x2, "y": y2})


# regression model
x2 = x2.reshape(-1,1) #have to do this if there's only 1 predictor in the model

lr2 = LinearRegression()
lr2.fit(x2,y2)

pred2 = lr2.predict(x2)

df2["pred"] = pred2 #add to df

df2["error"] = y2 - pred2
```

[92]:
```
################################################
(ggplot(df, aes("pred", "error")) + geom_point() + theme_minimal() +␣
 ↪ggtitle("Non Linear"))



################################################
```



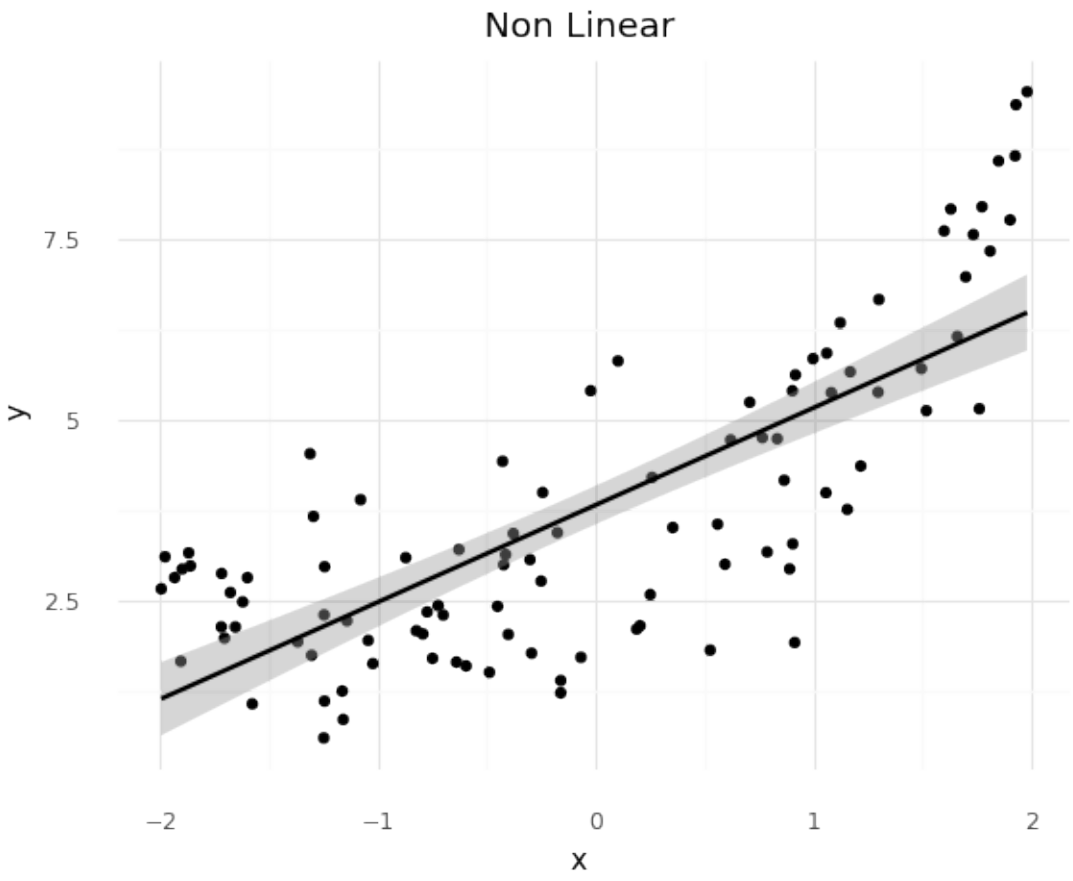
[92]: <ggplot: (8795491417902)>

[93]:
```
(ggplot(df2, aes("pred", "error")) + geom_point() + theme_minimal() +␣
↪ggtitle("Linear"))
```



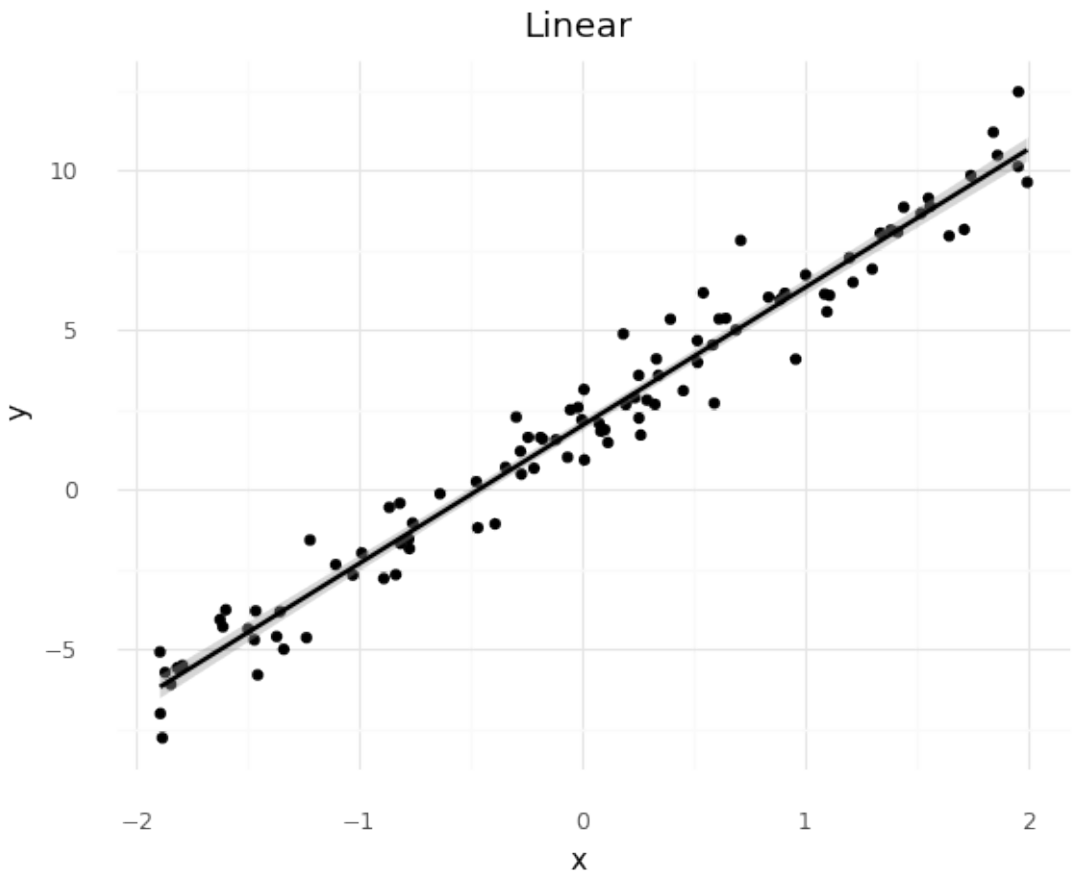
[93]: <ggplot: (8795490817654)>

[94]:
```
(ggplot(df, aes("x", "y")) + geom_point() + geom_smooth(method = "lm") +␣
↪theme_minimal() + ggtitle("Non Linear"))
```



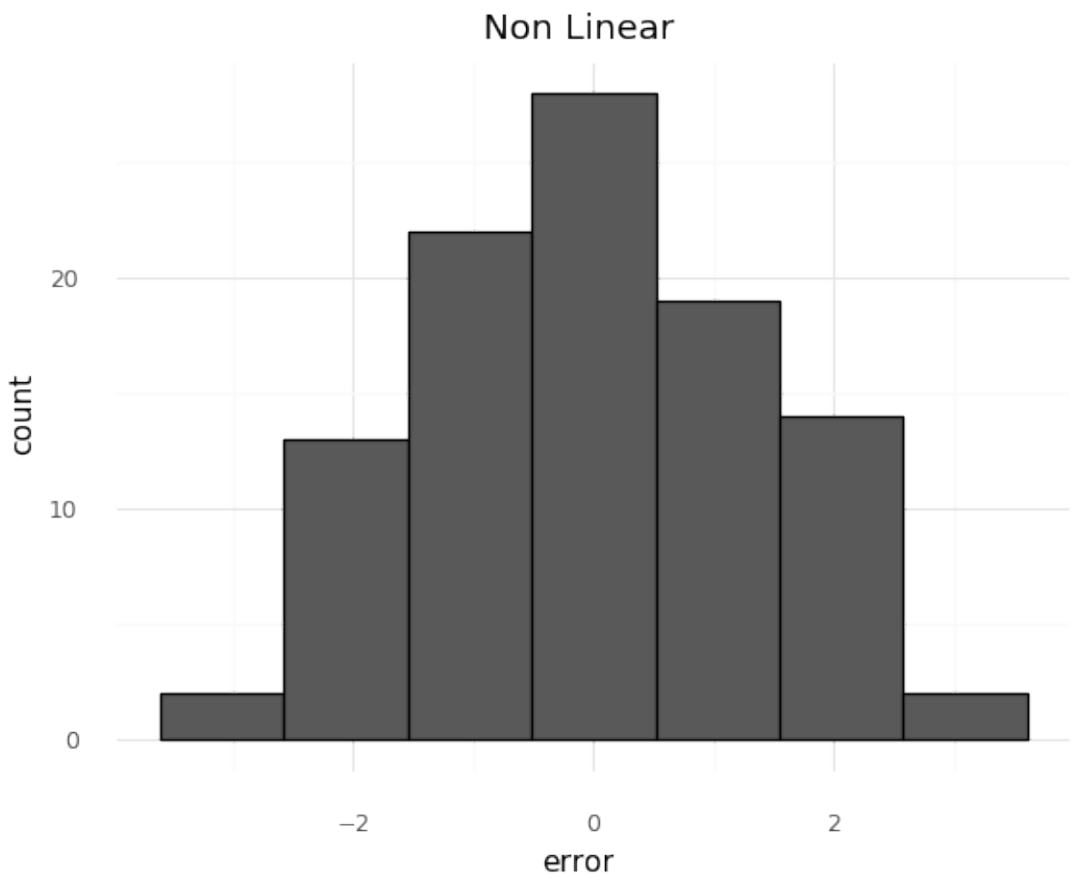
[94]: <ggplot: (8795490756744)>

[95]:
```
(ggplot(df2, aes("x", "y")) + geom_point() + geom_smooth(method = "lm")␣
↪+theme_minimal() + ggtitle("Linear"))
```



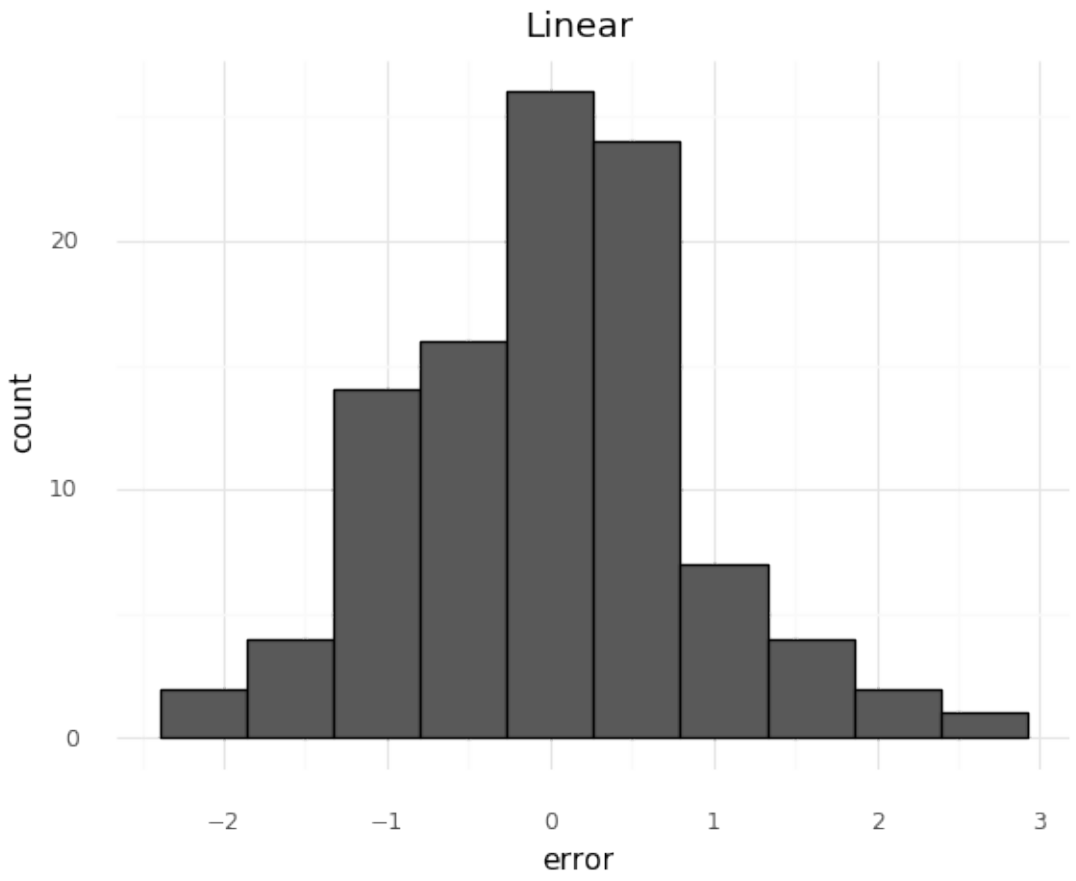
[95]: <ggplot: (8795491519681)>

[96]:
```
(ggplot(df, aes("error")) + geom_histogram(color = "black") + theme_minimal() +␣
↪ggtitle("Non Linear"))
```



[96]: <ggplot: (8795491519714)>

[97]:
```
(ggplot(df2, aes("error")) + geom_histogram(color = "black") + theme_minimal()␣
↪+ ggtitle("Linear"))
```



[97]: <ggplot: (8795489482623)>

5.0.2 Reflections

At the beginning and end of the model with non-linear data, we're systematically OVER predicting, and in the middle we're underpredicting. Depending on the context, we might be systematically disadvantaging people. What if this was using household income to predict college readiness. This would mean that we're consistently under predicting middle class students' college readiness. Disqualifying them from aid…admittance…

But we're also OVER estimating college readiness for low and high income people, possibly putting them in courses that are too difficult, or not giving them access to resources that would help them.

Non linearity often means that our model is systematically wrong in some way. The model OVER-ALL can still have a seemingly good fit, while exhibiting worrying bias in certain portions of the model.