In [2]:
```
# initializing otter-grader
import otter
grader = otter.Notebook()
```

Lab 8: Multiple Linear Regression

In this lab, you will be working with the diamond dataset. You will fit a linear model to predict the price of a diamond using its characteristics. You will get experience with extracting and creating features using techniques such as one-hot encoding or log transformation to improve the accuracy of your model. At the end, you will get a chance to create your own features for the linear model!

This lab should be completed and submitted by 11:59 PM on Friday May 22, 2020.

Collaboration Policy

Data science is a collaborative activity. While you may talk with others about the labs, we ask that you **write your solutions individually** and do not copy them from others.

By submitting your work in this course, whether it is homework, a lab assignment, or a quiz/exam, you agree and acknowledge that **this submission is your own work and that you have read the policies regarding Academic Integrity**: https://studentconduct.sa.ucsb.edu/academic-integrity (https://studentconduct.sa.ucsb.edu/academic-integrity). The Office of Student Conduct has policies, tips, and resources for proper citation use, recognizing actions considered to be cheating or other forms of academic theft, and students' responsibilities. You are required to read the policies and to abide by them.

List collaborators here

In [3]:
```
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import sklearn
import altair as alt
```

Preliminary

First, we load the diamond dataset and look at the fields in this dataset.

In [4]:
```
data = pd.read_csv("diamonds.csv.zip", index_col=0)
data.head()
```

Out[4]:

	carat	cut	color	clarity	depth	table	price	x	y	z
1	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
2	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
3	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
4	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
5	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

Each record in the dataset corresponds to a single diamond. The fields are

1. **carat**: The weight of the diamonds.
2. **cut**: The quality of the cut. This is an *ordinal* variable which takes on a value in the set: { `Fair` , `Good` , `Very Good` , `Premium` , and `Ideal` }.
3. **color**: The color of the diamond. This is an *ordinal* variable which takes on a value from the set of characters between `J` (worst) and `D` (best).
4. **clarity**: How obvious inclusions are within the diamond. This is an *ordinal* variable that takes on a value from the set: { `I1` (worst), `SI2` , `SI1` , `VS2` , `VS1` , `VVS2` , `VVS1` , `IF` (best)}.
5. **depth**: The height of a diamond, measured from the culet to the table, divided by its average girdle diameter.
6. **table**: The width of the diamond's table expressed as a percentage of its average diameter.
7. **price**: Price of the diamond in USD.
8. **x**: Length of the diamond measured in mm.
9. **y**: Width of the diamond measured in mm.
10. **z**: Depth of the diamond measured in mm.

We are interested in **predicting the price of a diamond given it's characteristics**. Mathematically, we would like to fit a linear model with parameters θ corresponding to features $\mathbf{x}$ to best capture the price of the diamonds:

$$f_\theta(\mathbf{x}) \rightarrow \text{Price}.$$

Part 1

For the first part of the lab, we will be focusing on diamond's **carat**, **depth**, and **table** characteristics. Hence $\mathbf{x} = [$ **carat**, **depth**, **table** $]$ for a given diamond.

We are interested in using a linear model with a bias term as our model. We could express the model mathematically as:

$$f_\theta(\mathbf{x}) = f_\theta\left(\mathbf{carat}, \mathbf{depth}, \mathbf{table}\right) = \theta_0 + \theta_1 * \mathbf{carat} + \theta_2 * \mathbf{depth} + \theta_3 * \mathbf{table}.$$

Question 1a

Set the variable `data1` to be a subset of the original dataframe `data` such that `data1` only contains the columns `carat` , `depth` , `table` and `price` . (Note that the order of the columns in dataframe `data1` should follow the order `carat` , `depth` , `table` , `price` in order to pass the autograder test.)

In [5]:
```
data1 = data[['carat', 'depth', 'table', 'price']]
```

In the following code, we split `data1` into two variables:

(1) Target values `y` : this consists of the prices of the diamonds.

(2) Set of features `X_features` : this is a data frame where each row is a feacture vector consisting of features $[$ **carat**, **depth**, **table** $]$ (without the bias term).

In [6]:
```
Y = data1['price']
X_features = data1[['carat', 'depth', 'table']]
```

Question 1b

We defined a function `add_bias` which takes in a dataframe and adds a column of 1's to the left of the input dataframe. This function should modify the input dataframe in place. Please fill in this function with your solution. **Please name this extra column 'ones' in the dataframe.** After calling the function on `X_features` you will get a dataframe whose first five rows of `X_features` will look like the following:

	ones	carat	depth	table
0	1.0	0.23	61.5	55.0
1	1.0	0.21	59.8	61.0
2	1.0	0.23	56.9	65.0
3	1.0	0.29	62.4	58.0
4	1.0	0.31	63.3	58.0

Hint: You might find `pd.insert` method to be useful as you can specify the column index for the newly-added column: [https://www.geeksforgeeks.org/python-pandas-dataframe-insert/ (https://www.geeksforgeeks.org/python-pandas-dataframe-insert/)](https://www.geeksforgeeks.org/python-pandas-dataframe-insert/)

In [22]:
```
def add_bias(data):
    x = pd.DataFrame(np.ones(len(data) + 1))
    return data.insert(0, "ones", x, True)

X = X_features.copy()
add_bias(X)
X.head()
```

Out[22]:

	ones	carat	depth	table
1	1.0	0.23	61.5	55.0
2	1.0	0.21	59.8	61.0
3	1.0	0.23	56.9	65.0
4	1.0	0.29	62.4	58.0
5	1.0	0.31	63.3	58.0

Question 1c

We need a loss function to evaluate how good our model approximates the prices of the diamonds. In the cell below, complete the function `avg_squared_loss` which returns the average squared loss between true target values `y` and our predictions `y_hat` . Note that both inputs, `y` and `y_hat` , to the function are arrays. You can assume that they have the same length.

Recall that the average squared loss is defined as:

$$Avg\ Squared\ Loss(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

In [23]:
```
def avg_squared_loss(y, y_hat):
    return (1/len(y))*sum(((y-y_hat)**2))
```

Now we are ready to build our linear model. We saw that the **predictions** for the entire data set, $\hat{\mathbb{Y}}$, with a linear model can be computed as:

$$\hat{\mathbb{Y}} = \mathbb{X}\theta$$

The **covariate matrix** $\mathbb{X} \in \mathbb{R}^{n \times (d+1)}$ consists of n rows where each row corresponds to a record in the dataset and the $d + 1$ columns correspond to the d features extracted from the data plus an additional bias term.

The following function `linear_model` computes the prediction $\hat{\mathbb{Y}}$ given parameters θ and covariate matrix $\mathbb{X}$.

In [24]:
```
def linear_model(theta, X):
    return X @ theta # The @ symbol is matrix multiplication
```

Here the `@` symbol is the matrix multiply operation and is equivalent to writing `X.dot(theta)` .

Question 1d

In the cell below, choose any `theta` you would like (please note that the dimension of the `theta` you choose should match the number of columns of the covariate matrix) and make predictions for `Y` using the linear model defined above given the `theta` you chose. Assign the variable `Y_hat` with the predictions and the variable `loss` with the average squared loss of your predictions based on the `theta` you chose.

In [33]:
```
theta = np.random.normal(scale=3, size=(4))
Y_hat = linear_model(theta, X)
loss = avg_squared_loss(Y, Y_hat)
```

You might notice the loss of the predictions for an arbitrary choice of `theta` is quite big. We can find the optimal `theta` by minimizing the mean square loss:

$$
\begin{aligned}
L(\theta) &= \frac{1}{n}\sum_{i=1}^{n}\left(\mathbb{Y}_i - (\mathbb{X}\theta)_i\right)^2 \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(\mathbb{Y}_i - \mathbb{X}_i\theta\right)^2 \\
&= \frac{1}{n}||\mathbb{Y} - \mathbb{X}\theta||_2^2 \\
&= \frac{1}{n}\left(\mathbb{Y} - \mathbb{X}\theta\right)^T\left(\mathbb{Y} - \mathbb{X}\theta\right)
\end{aligned}
$$

By taking derivative with respect to θ and set the derivative equal to 0. We can get the normal equation:

$$(\mathbb{X}^T\mathbb{X})\hat{\theta} = \mathbb{X}^T\mathbb{Y}$$

Solving for $\hat{\theta}$ in the above equation gives us the minimizer of the squared loss with respect to our data.

If $\mathbb{X}^T\mathbb{X}$ is invertible (full rank), $\hat{\theta}$ can be computed analytically as:

$$\hat{\theta} = \left(\mathbb{X}^T\mathbb{X}\right)^{-1}\mathbb{X}^T\mathbb{Y}.$$

We will not use the above analytic approach for solving $\hat{\theta}$ in this lab. Instead, we will use the `sklearn` library to fit our model and find the optimal θ.

In [34]:
```
# Import the LinearRegression model from sklearn
from sklearn.linear_model import LinearRegression
```

Question 1e

In lab7 we have learned how to use the sklearn package to create a linear regression model, as well as using it to fit on the data and get the predicted values. Today we are going to use it again. In case you are not familiar with the syntax, check lab7 to get refreshed! In the cell below,

1. Fit a linear model `model1` using `X` and `Y` defined earlier in the lab.
2. Make predictions `Y_hat1` for `Y` using the fitted model.
3. Calculate the average squared loss `loss1` of your prediction.

In [53]:
```
X.shape, Y.shape
```

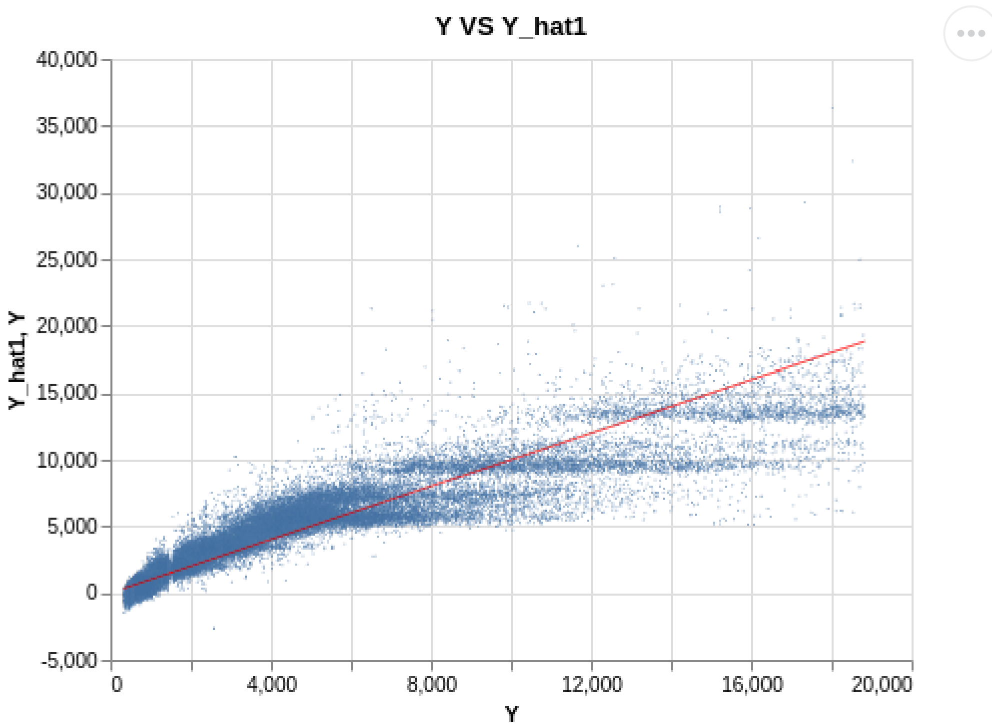
Out[53]:
```
((53940, 4), (53940,))
```

In [75]:
```
model1 = LinearRegression()
model1.fit(X = X, y = Y)
Y_hat1 = model.predict(X)
loss1 = avg_squared_loss(Y, Y_hat1)
```

In the cell below, we create a scatter plot by plotting (`Y` , `Y_hat1`). The red line is the identity line where each point on the line has the same values for the variables representing the x-axis and y-axis. If our model is very accurate, we would expect $Y \approx Y_{hat1}$, and thus all the points should be very close to the identity line. However, what do you observe in the current plot?

In [55]:
```
alt.data_transformers.disable_max_rows()

source = pd.DataFrame({
    'Y': Y,
    'Y_hat1': Y_hat1
})

layer1 = alt.Chart(source).mark_circle(size=1).encode(
    x='Y',
    y='Y_hat1'
).properties(
    title='Y VS Y_hat1'
)

layer2 = alt.Chart(source).mark_line(size=1).encode(
    x='Y',
    y='Y',
    color = alt.value("red")
)

layer1 + layer2
```

Out[55]:



In [56]:
```
# as the value of Y increases, it deviates from Y_hat
```

Part 2

For part 1, we only used the quantitative features `carat` , `depth` , `table` . As you can see from Question 1(e), the loss seems to be big. Is there a way to fit a better model by incorporating other features?

In this second part of the lab, we explore incorporating qualitative features into our model.

Recall our dataframe looks like the following:

In [57]:
```
data.head()
```

Out[57]:

	carat	cut	color	clarity	depth	table	price	x	y	z
1	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
2	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
3	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
4	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
5	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

We only incorporated information about `carat` , `depth` , `table` in our previous features. Do `cut` , `color` , and `clarity` matter when it comes to predicting the prices of the diamonds?

Based on this online article [https://www.pricescope.com/diamond-prices (https://www.pricescope.com/diamond-prices)](https://www.pricescope.com/diamond-prices), these characteristics should matter! So let's try to incoporate these into our model!

Recall from the lecture, to include qualitative variables as features, we may use one-hot encoding. The idea of one-hot encoding is to vectorize the variables with 1's and 0's. For example, suppose we have a qualitative variable `smoking` and the variable can take on either 'smoker' or 'non-smoker' like what we show below:

	smoking
0	smoker
1	non-smoker
2	smoker
3	non-smoker
4	non-smoker

After one-hot encoding, the resulting dataframe will look like:

	smoker	non-smoker
0	1	0
1	0	1
2	1	0
3	0	1
4	0	1

For this lab, we will use the [`DictVectorizer` (https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.DictVectorizer.html)](https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.DictVectorizer.html) method from `sklearn` package to implement one-hot encoding.

Let's first examine how the model will behave if we include the `cut` feature. In the cell below, we created a new dataframe `X_char_w_cut` which adds one more column `cut` to the features defined in Part 1.

In [39]:
```
X_features_with_cut = data[['carat', 'cut', 'depth', 'table']]
add_bias(X_features_with_cut)
X_features_with_cut
```

Out[39]:

	ones	carat	cut	depth	table
1	1.0	0.23	Ideal	61.5	55.0
2	1.0	0.21	Premium	59.8	61.0
3	1.0	0.23	Good	56.9	65.0
4	1.0	0.29	Premium	62.4	58.0
5	1.0	0.31	Good	63.3	58.0
...	...	...	...	...	...
53936	1.0	0.72	Ideal	60.8	57.0
53937	1.0	0.72	Good	63.1	55.0
53938	1.0	0.70	Very Good	62.8	60.0
53939	1.0	0.86	Premium	61.0	58.0
53940	1.0	0.75	Ideal	62.2	55.0

53940 rows × 5 columns

Question 2a

In the cell below, complete the code so that `X_with_cut` is the new feature matrix after one-hot encoding. There are a few things we need to do:

1. Review the notebook example in the lecture on how to convert a categorical variable into a one-hot encoding matrix.
2. Adjust the index issue (Done for you already).
3. Combine the other features (except cut), and the one-hot encoding matrix together to form `X_with_cut` . You can use `pd.concat` .

In [72]:
```
from sklearn.feature_extraction import DictVectorizer

# one-hot encoding


cuts = X_features_with_cut[['cut']].to_dict(orient='records')

encoder = DictVectorizer(sparse=False)
cuts_df = pd.DataFrame(
    data = encoder.fit_transform(cuts),
    columns = encoder.feature_names_
)


# adjusting the index inconsistency issue
X_features_with_cut.reset_index(drop=True, inplace=True)
cuts_df.reset_index(drop=True, inplace=True)

# Combine the features together with pd.concat
X_with_cut = pd.concat([X_features_with_cut, cuts_df], axis=1).drop(col
umns=['cut'])
```

In [73]:
```
X_with_cut.shape
```

Out[73]:
```
(53940, 9)
```

In [ ]:
```

```

Question 2b

Now please fit a linear model using our new covariate matrix `X`. Compute the average squared loss of the predictions. Compare this loss with the loss you computed in Question 1(e) without using the `cut` feature.

In [80]:
```
model2 = LinearRegression()
model2.fit(X_with_cut, Y)
Y_hat2 = model2.predict(X_with_cut)
loss2 = avg_squared_loss(Y, Y_hat2)
```

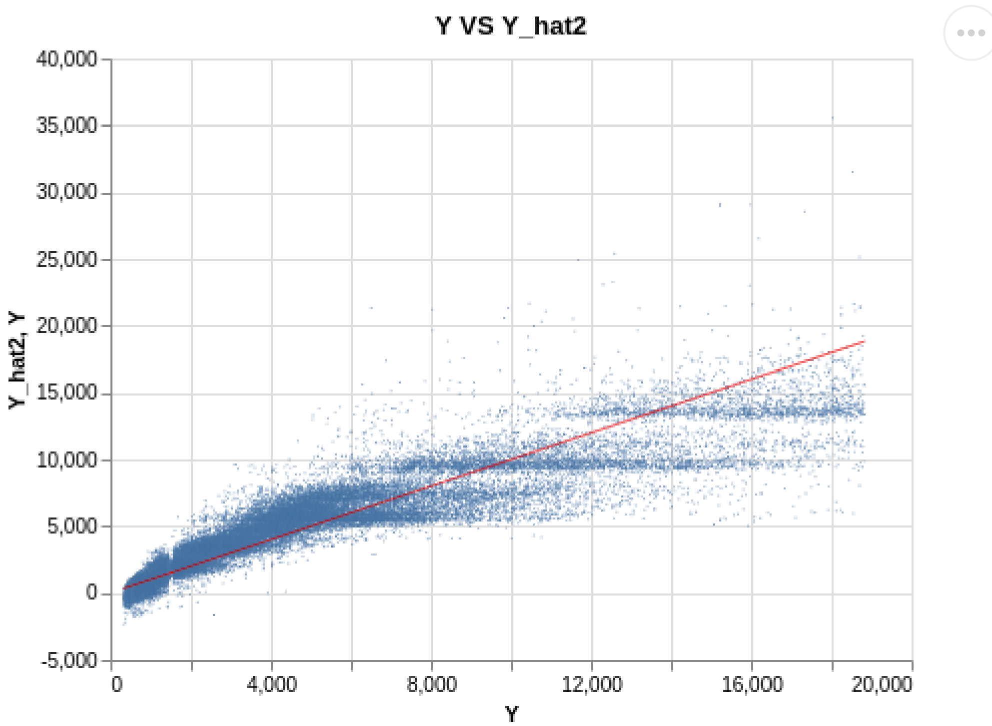
Let us see the proportion that the loss decreases after incorporating the `cut` feature.

In [81]:
```
loss2 < loss1
```

Out[81]:
```
True
```

In [82]:
```
alt.data_transformers.disable_max_rows()

source = pd.DataFrame({
    'Y': Y,
    'Y_hat2': Y_hat2
})

layer1 = alt.Chart(source).mark_circle(size=1).encode(
    x='Y',
    y='Y_hat2'
).properties(
    title='Y VS Y_hat2'
)

layer2 = alt.Chart(source).mark_line(size=1).encode(
    x='Y',
    y='Y',
    color = alt.value("red")
)

layer1 + layer2
```

Out[82]:



The plot looks similar to the earlier one we have. Can we do better?

Yeap. Probably quadratic/exponential transformation / model woud help)

Question 2c

In the cell below, we consider adding `color` and `clarity` as features. Please fill in the relevant code below to fit a model with the covariate matrix `X_features_with_cut_color_clarity` .

In [89]:
```
X_features_with_cut_color_clarity = data[['carat', 'cut', 'color', 'cla
rity', 'depth', 'table']] # Do not change this line
X_features_with_cut_color_clarity.head()
```

Out[89]:

	carat	cut	color	clarity	depth	table
1	0.23	Ideal	E	SI2	61.5	55.0
2	0.21	Premium	E	SI1	59.8	61.0
3	0.23	Good	E	VS1	56.9	65.0
4	0.29	Premium	I	VS2	62.4	58.0
5	0.31	Good	J	SI2	63.3	58.0

In [103]:
```
# extract the columns 'carat', 'cut', 'color', 'clarity', 'depth', 'tab
le'
X_features_with_cut_color_clarity = data[['carat', 'cut', 'color', 'cla
rity', 'depth', 'table']] # Do not change this line
add_bias(X_features_with_cut_color_clarity)

cut_color_clarity = X_features_with_cut_color_clarity[['cut', 'color',
'clarity']].to_dict(orient='records')
encoder = DictVectorizer(sparse=False)

cut_color_clarity_df = pd.DataFrame(
    data = encoder.fit_transform(cut_color_clarity),
    columns = encoder.feature_names_
)

# adjusting the index inconsistency issue. Uncomment the following two
 lines
X_features_with_cut_color_clarity.reset_index(drop=True, inplace=True)
cut_color_clarity_df.reset_index(drop=True, inplace=True)

# Combine the features together
X_with_cut_color_clarity = pd.concat([X_features_with_cut_color_clarity
, cut_color_clarity_df], axis=1).drop(columns=['cut', 'color', 'clarit
y'])

model3 = LinearRegression()
model3.fit(X_with_cut_color_clarity, Y)
Y_hat3 = model3.predict(X_with_cut_color_clarity)
loss3 = avg_squared_loss(Y, Y_hat3)

loss3
```

Out[103]: 1336023.5269157814

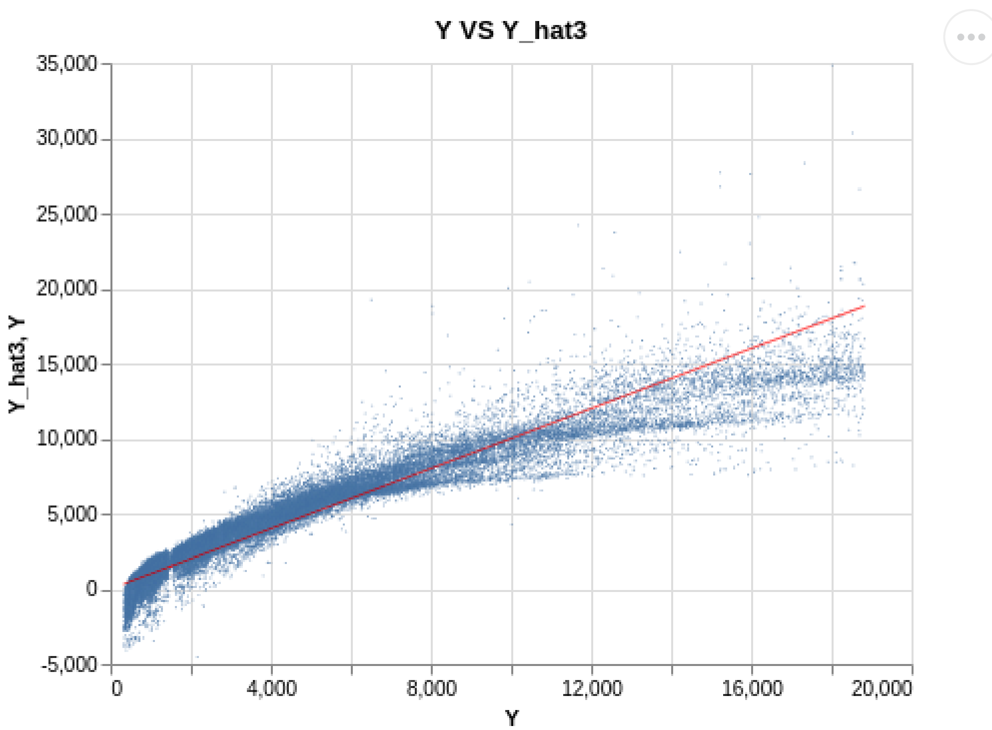
Compare `loss3` with `loss2` to check if our model works better.

In [104]:
```
loss3 < loss2
```

Out[104]:
```
True
```

In [105]:
```
alt.data_transformers.disable_max_rows()

source = pd.DataFrame({
    'Y': Y,
    'Y_hat3': Y_hat3
})

layer1 = alt.Chart(source).mark_circle(size=1).encode(
    x='Y',
    y='Y_hat3'
).properties(
    title='Y VS Y_hat3'
)

layer2 = alt.Chart(source).mark_line(size=1).encode(
    x='Y',
    y='Y',
    color = alt.value("red")
)

layer1 + layer2
```

Out[105]:



In [116]:
```
X_with_cut_color_clarity.head()
```

Out[116]:

	ones	carat	depth	table	clarity=I1	clarity=IF	clarity=SI1	clarity=SI2	clarity=VS1	clarity=VS
0	1.0	0.23	61.5	55.0	0.0	0.0	0.0	1.0	0.0	0
1	1.0	0.21	59.8	61.0	0.0	0.0	1.0	0.0	0.0	0
2	1.0	0.23	56.9	65.0	0.0	0.0	0.0	0.0	1.0	0
3	1.0	0.29	62.4	58.0	0.0	0.0	0.0	0.0	0.0	1
4	1.0	0.31	63.3	58.0	0.0	0.0	0.0	1.0	0.0	0

5 rows × 24 columns

Question 3

Try coming up with more features to make the model perform even better! Some suggestions are: include a `log(carat)` feature with the logarithmic values of `carat` or the characteristics `x` , `y` , `z` in the feature set. Write your code in the cell below.

In [125]:
```
X_with_cut_color_clarity.iloc[:, ]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-125-ab7a76c2956a> in <module>
----> 1 X_with_cut_color_clarity[['cut=ideal']]

/opt/conda/lib/python3.7/site-packages/pandas/core/frame.py in __getitem__(self, key)
   2999             if is_iterator(key):
   3000                 key = list(key)
-> 3001             indexer = self.loc._convert_to_indexer(key, axis=1, raise_missing=True)
   3002 
   3003         # take() does not accept boolean indexers

/opt/conda/lib/python3.7/site-packages/pandas/core/indexing.py in _convert_to_indexer(self, obj, axis, is_setter, raise_missing)
   1283                 # When setting, missing keys are not allowed, even with .loc:
   1284                 kwargs = {"raise_missing": True if is_setter else raise_missing}
-> 1285                 return self._get_listlike_indexer(obj, axis, **kwargs)[1]
   1286         else:
   1287             try:

/opt/conda/lib/python3.7/site-packages/pandas/core/indexing.py in _get_listlike_indexer(self, key, axis, raise_missing)
   1090 
   1091         self._validate_read_indexer(
-> 1092             keyarr, indexer, o._get_axis_number(axis), raise_missing=raise_missing
   1093         )
   1094         return keyarr, indexer

/opt/conda/lib/python3.7/site-packages/pandas/core/indexing.py in _validate_read_indexer(self, key, indexer, axis, raise_missing)
   1175                 raise KeyError(
   1176                     "None of [{key}] are in the [{axis}]".format(
-> 1177                         key=key, axis=self.obj._get_axis_name(axis)
   1178                     )
   1179                 )

KeyError: "None of [Index(['cut=ideal'], dtype='object')] are in the [columns]"
```

In [129]:

```
log_car = X_with_cut_color_clarity['carat'].transform([np.log])
cut_ideal_max = X_with_cut_color_clarity['cut=Ideal'].transform(lambda x: np.multiply(x, 100))
cut_premium_max = X_with_cut_color_clarity['cut=Premium'].transform(lambda x: np.multiply(x, 10))
X_new = X_with_cut_color_clarity.copy()
X_new['cut=Ideal'] = cut_ideal_max
X_new['cut=Premium'] = cut_premium_max

model4 = model
```

Congratulations! You have completed this assignment. Hope you enjoyed it!

Running Built-in Tests

1. All tests are in `tests` directory
2. Each python file in `tests` is a test
3. `grader.check('testname')` runs test `'testname'`, e.g. `'q1'`
4. `grader.check_all()` runs all visible tests

In [106]:

```
# Run built-in checks
grader.check_all()
```

q1a

All tests passed!

q1b

All tests passed!

q1c

All tests passed!

q1d

0 of 1 tests passed

Tests failed:

- **./tests/q1d.py**

  Test code:

```
>>> np.isclose(Y_hat[1], linear_model(theta, X)[1])
True
```

  Test result:

```
Trying:
    np.isclose(Y_hat[1], linear_model(theta, X)[1])
Expecting:
    True
*******************************************************************
***
Line 1, in ./tests/q1d.py 1
Failed example:
    np.isclose(Y_hat[1], linear_model(theta, X)[1])
Expected:
    True
Got:
    False
```

q1e

All tests passed!

q2a

All tests passed!

q2b

All tests passed!

q2c

All tests passed!

In [107]:
```
# Generate pdf in classic notebook (does not work in JupyterLab)
import nb2pdf
nb2pdf.convert('lab8.ipynb')

# To generate pdf using command-line, run in terminal,
# nb2pdf lab8.ipynb
```

Submission Checklist

1. Check filename is 'lab8.ipynb'
2. Save file to confirm all changes are on disk
3. Run *Kernel* > *Restart & Run All* to execute all code from top to bottom
4. Check `grader.check_all()` output
5. Save file again to write any new output to disk
6. Check generated pdf that all responses are displayed correctly
7. Submit to Gradescope