Lab_7_Linear_Regression

November 17, 2021

1 Linear Regression

In 1991, Orley Ashenfelter, an economics professor at Princeton University, stunned the wine world with a bold prediction. He predicted that the 1990 vintage of Bordeaux wines would be the "wine of the century,′′ even better than the prized 1961 vintage. Furthermore, he made this prediction without tasting even a drop of the wine, which had been placed in oak barrels just months earlier.

How did Ashenfelter predict the quality of the wine without tasting it? He used data on past vintages to come up with the following formula for predicting wine quality:

$$\begin{aligned}\widehat{\text{wine quality}} = -7.8 &+ 0.62 \cdot (\text{average summer temperature}) \\ &+ 0.0012 \cdot (\text{winter rainfall}) \\ &- 0.0037 \cdot (\text{harvest rainfall}) \\ &+ 0.024 \cdot (\text{age of the wine}) \end{aligned} \tag{1}$$

The variable on the left-hand side of this expression, wine quality, is what we are trying to predict and is called the *target* (or *label*). (The hat symbol over "wine quality" indicates that the values are predicted instead of observed.) The variables on the right-hand side, such as "average summer temperature" and "harvest rainfall," are called *features* and are the inputs used to predict the target. Although Ashenfelter had no way of knowing the quality of the 1990 wines, he did have the values of the features in 1990, so to make a prediction, all he had to do was plug those values into the equation above. In this way, he arrived at the following prediction for the quality of the 1990 Bordeaux, after they had been aged for 31 years (like the 1961 Bordeaux had been at the time):

$$\begin{aligned}-7.8 &+ 0.62 \cdot (18.7) \\ &+ 0.0012 \cdot (468) \\ &- 0.0037 \cdot (80) \\ &+ 0.024 \cdot (31) = 4.8. \end{aligned} \tag{2}$$

For comparison, the quality of the prized 1961 vintage was 4.6.

You can imagine the uproar from wine experts, who had spent years refining their palates to distinguish good wines from bad. Robert Parker, the most influential wine critic in America, called Ashenfelter's predictions "ludicrous and absurd", comparing him to a "movie critic who never goes to see the movie but tells you how good it is based on the actors and the director." It did not help that Ashenfelter had also openly challenged Parker's rating of the 1986 Bordeaux. Parker thought they would be "very good and sometimes exceptional." But according to Ashenfelter's formula, the low summer temperatures and high harvest rainfalls in 1986 doomed the vintage.

Who was right? Thirty years later, Robert Parker ranks the 1986 Bordeaux well, but the 1990 Bordeaux wines are exceptional, with three of the six wines scoring a 98 on a 100-point scale.

We will reproduce Ashenfelter's analysis, which is an example of *machine learning*. Machine learning is concerned with the general problem of how to use data to make predictions. The process of producing a model like Ashenfelter's from data is called *fitting* a model (although the terms *training* or *learning* are also used), and the data that is used to fit the model is the *training data*.

1.1 Getting Familiar with the Data

First, we read in the historical data that Ashenfelter used. The observational unit in this data set is the vintage, so we index this `DataFrame` by the year.

[1]:
```
import pandas as pd
data_dir = ""
bordeaux_df = pd.read_csv("bordeaux.csv",index_col="year")
bordeaux_df.head()
```

[1]:

year	price	summer	har	sep	win	age
1952	37.0	17.1	160	14.3	600	40
1953	63.0	16.7	80	17.3	690	39
1955	45.0	17.1	130	16.8	502	37
1957	22.0	16.1	110	16.2	420	35
1958	18.0	16.4	187	19.1	582	34

The **price** column is in 1981 dollars, normalized so that the 1961 Bordeaux has a price of 100. Price is a reasonable proxy for the quality of the wine. The **summer** column contains the average summer temperature (in degrees Celsius), while the **har** and **win** columns contain the harvest and winter rainfalls (in millimeters). The **sep** column stores the average temperature in September, which Ashenfelter did not include in his model.

Let us also take a peek at the end of this `DataFrame`.

[2]:
```
bordeaux_df.tail()
```

[2]:

year	price	summer	har	sep	win	age
1987	NaN	17.0	115	18.9	452	5
1988	NaN	17.1	59	16.8	808	4
1989	NaN	18.6	82	18.4	443	3
1990	NaN	18.7	80	19.3	468	2
1991	NaN	17.7	183	20.4	570	1

We see that the `DataFrame` also contains data for vintages where the price is missing (including 1990, the vintage for which Ashenfelter made his prediction). In fact, prices are only available up to 1980, as it takes several years before wine quality can be estimated with much reliability), so only part of the `DataFrame` can be used for training. The rest of the data, where the features are known but the target is not, is called the *test data*. Machine learning fits a model to the training data, which is then used to predict the targets in the test data. The following code splits the `DataFrame` into the training and test sets.

[3]:
```
bordeaux_train = bordeaux_df.loc[:1980].copy()
bordeaux_test = bordeaux_df.loc[1981:].copy()
```

1.2 Warm-Up: A Model with One Feature

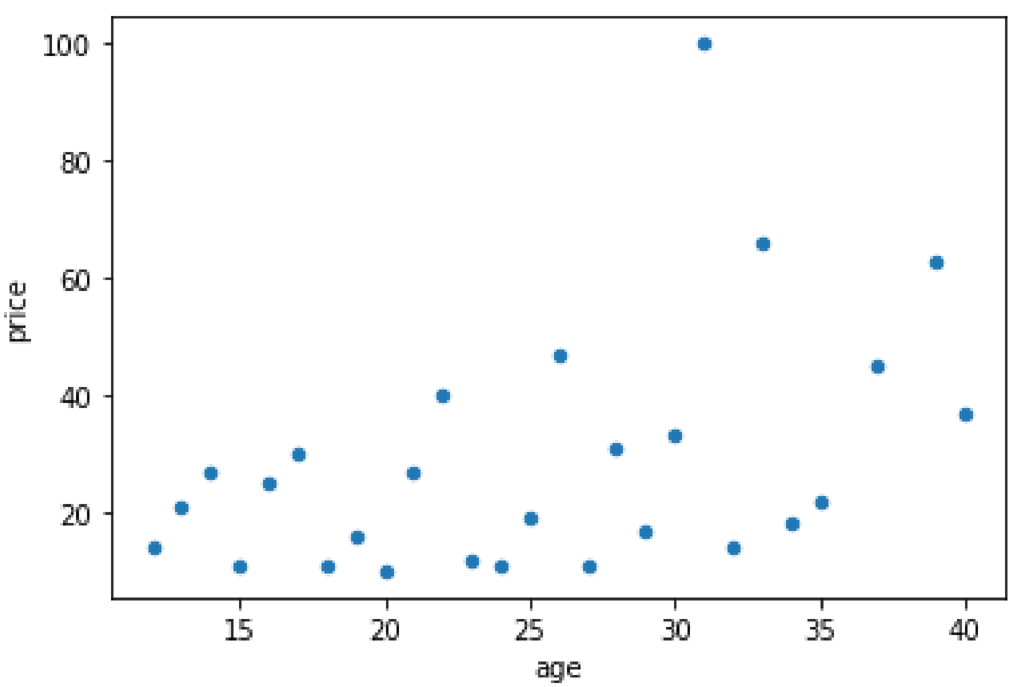
Before fitting a model that uses all of the features, we first consider a model that uses only the age of the wine to predict the price. That is, we fit a model of the form

$$\widehat{\text{price}} = b + c \cdot \text{age}, \tag{3}$$

where b and c are numbers that we will learn from the training data. Models of the form above are called *linear regression* models. (The way in which this model is "linear" will become apparent in a moment.) This model only involves two variables, **age** and **price**, so we can visualize the data easily using a scatterplot (see Chapter 3).

[4]:
```
bordeaux_train.plot.scatter(x="age", y="price")
```

[4]: `<AxesSubplot:xlabel='age', ylabel='price'>`



Now, to fit models like the above to the training data, we use the scikit-learn package, which was used in Chapter 3 for transforming variables and calculating distances. However, its main purpose is to fit machine learning models, including linear regression. All models in scikit-learn are used in essentially the same way, following the three-step pattern:

1. Declare the model.
2. Fit the model to training data.
3. Use the model to predict on test data.

In the case of the linear regression model above, the code is as follows.

[5]:
```
from sklearn.linear_model import LinearRegression

X_train = bordeaux_train[["age"]]
X_test = bordeaux_test[["age"]]
y_train = bordeaux_train["price"]

model = LinearRegression()
model.fit(X=X_train, y=y_train)
model.predict(X=X_test)
```

[5]:
```
array([12.41648163, 11.26046336, 10.1044451 ,  8.94842683,  7.79240856,
        6.6363903 ,  5.48037203,  4.32435376,  3.1683355 ,  2.01231723,
        0.85629897])
```

The parameters of `.fit()` are `X` for the features and `y` for the targets, which are assumed to be 2-D and 1-D arrays of numbers, respectively. So even when there is only one feature, as in this case, we still need to supply a 2-D array with one column—hence, the double brackets around `"age"` when defining `X_train` and `X_test`.

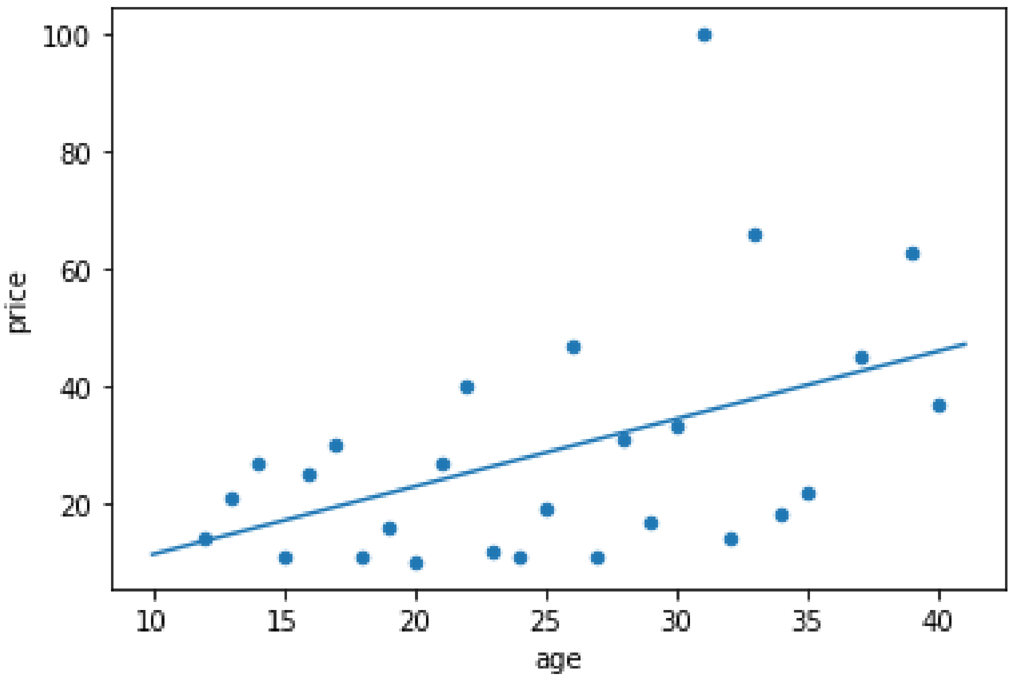
By contrast, `.predict()` only has one parameter, `X` for the features. That is because its job is to predict the targets `y` for the given features. Note that the predictions will always be returned in the form of `numpy` arrays, no matter the type of the input data—so although we supplied `pandas` objects, `sklearn` still returned the predicted values as `numpy` arrays. The predictions are in the same order as the rows of `X`.

Because there are only two variables involved, the model above is a rare example of a machine learning model we can visualize. A general way to do this is to generate a fine grid of `X` values using `np.linspace()` and call `model.predict()` to get the predicted target at each of these values. We can then use these predictions to draw a curve which depicts the predicted value of `y` at each value of `X`. In the code below, we put the predictions in a `pandas` `Series`, indexed by the `X` values, and then call `.plot.line()`.

[6]:
```
import numpy as np

X_new = pd.DataFrame()
# create a sequence of 200 evenly spaced numbers from 10 to 41
X_new["age"] = np.linspace(10, 41, num=200)

# create a Series out of the predicted values
```

```python
# (trailing underscore indicates fitted values)
y_new_ = pd.Series(
    model.predict(X_new), # y values in Series.plot.line()
    index=X_new["age"]     # x values in Series.plot.line()
)

# plot the data, then the model
bordeaux_train.plot.scatter(x="age", y="price")
y_new_.plot.line()
```

[6]: <AxesSubplot:xlabel='age', ylabel='price'>



The resulting plot is shown above. Notice that the curve is a straight line, which is why this model is called *linear* regression. In hindsight, this is obvious from the model equation: b is simply the intercept and c the slope of this line. All linear regression does is choose the intercept and slope to minimize the total squared distance between the points and the line—that is, between the observed and predicted prices. In mathematical terms, b and c are chosen to minimize

$$\begin{aligned} \text{sum of } (\text{price} - \widehat{\text{price}})^2 \quad &= \quad \text{sum of } (\text{price} - (b + c \cdot \text{age}))^2 && (4)\\ \text{over training data} \quad & \quad\quad \text{over training data.} && (5) \end{aligned}$$

Since `sklearn` does this optimization for us, it is not necessary to understand the details of this process to extract useful insights out of linear regression. However, the math is explained in the appendix of this lesson for those who are curious.

1.3 What to Do about Nonlinearity

One question is whether the relationship between age and price is truly linear. In the graph above, it seems that the points deviate more from the line when prices are high than when they are low. To correct this, we need to spread out low prices and rein in high prices. Previously, we learned that this can be achieved by applying a log transformation to the prices. Let's add a column to the training data for the log-price.

[7]:
```
bordeaux_train["log(price)"] = np.log(bordeaux_train["price"])
```

Now, we will fit a linear regression model to predict this new target. That is, in contrast to the previous model, we now fit the model

$$\widehat{\log(\text{price})} = b + c \cdot \text{age}, \tag{6}$$

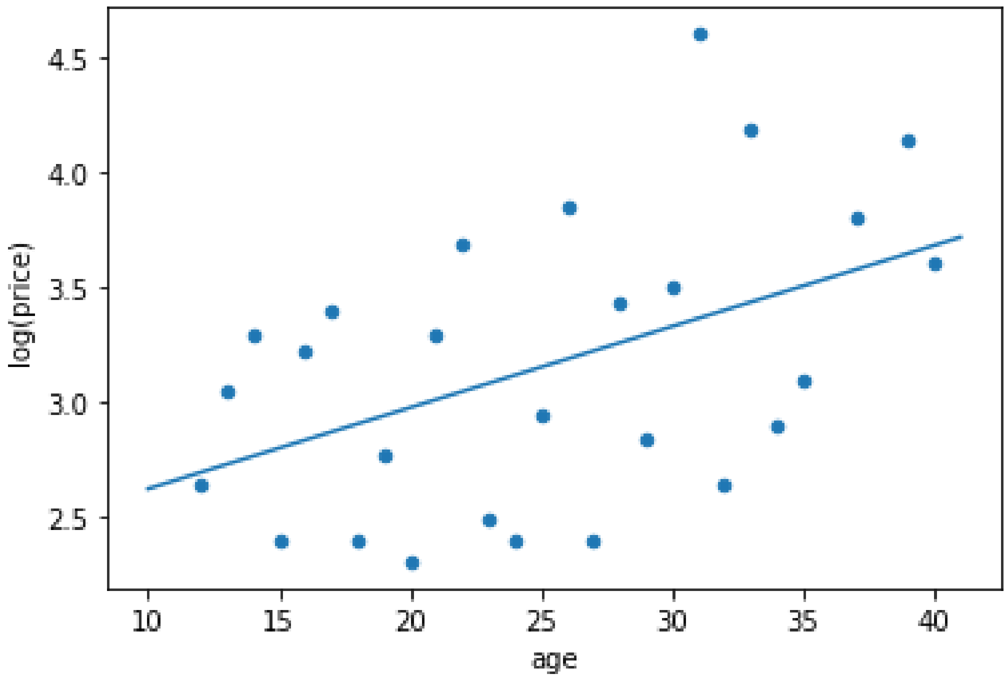
where b and c are chosen to minimize

$$\text{sum of } (\log(\text{price}) - \widehat{\log(\text{price})})^2 \text{ over training data} \tag{7}$$

over the training data. The code below fits this model.

[8]:
```
log_price_model = LinearRegression()
log_price_model.fit(X=bordeaux_train[["age"]],
                    y=bordeaux_train["log(price)"])

X_new = pd.DataFrame()
X_new["age"] = np.linspace(10, 41, num=200)
y_new_ = pd.Series(
    log_price_model.predict(X_new),
    index=X_new["age"]
)

bordeaux_train.plot.scatter(x="age", y="log(price)")
y_new_.plot.line()
```

[8]:
```
<AxesSubplot:xlabel='age', ylabel='log(price)'>
```



The points are more evenly spread out when the target is log-price instead of price. For this reason, Ashenfelter chose log-price to be the measure of "wine quality" in his linear regression model.

1.4 Fitting Ashenfelter's Model

We are now ready to reproduce Ashenfelter's analysis. To do so, we will need to fit a linear regression model that predicts the log-price from the average summer temperature, winter rainfall, harvest rainfall, and the age of the wine. In other words, the model is of the form

$$\begin{aligned}\widehat{\log(\text{price})} = b &+ c_1 \cdot (\text{average summer temperature}) \\ &+ c_2 \cdot (\text{winter rainfall}) \\ &+ c_3 \cdot (\text{harvest rainfall}) \\ &+ c_4 \cdot (\text{age of the wine}),\end{aligned} \tag{8}$$

where b, c_1, c_2, c_3, c_4 are chosen to minimize

$$\text{sum of } (\log(\text{price}) - \widehat{\log(\text{price})})^2 \text{ over training data.} \tag{9}$$

This is still a *linear regression* model, albeit a more complicated one.

The code to fit this model is the natural extension of the code we wrote to fit the earlier models in this lesson. Instead of passing `bordeaux_train[["age"]]` for `X`, we now supply a `DataFrame` containing all of the features we want to be in the model.

[9]:
```
ashen_model = LinearRegression()
ashen_model.fit(
```

```
    X=bordeaux_train[["summer", "win", "har", "age"]],
    y=bordeaux_train["log(price)"]
)
```

[9]:
```
LinearRegression()
```

This model is much harder to visualize, since it involves five variables: four features, plus the target. Nevertheless, we can obtain predictions from it just as we did with the simpler models above. We just need to supply the values of all of the features in the model, in the same order as in the training data.

[10]:
```
ashen_model.predict(
    X=bordeaux_test[["summer", "win", "har", "age"]]
)
```

[10]:
```
array([3.17926885, 3.4231464 , 3.71919787, 2.83391541, 3.48195778,
       2.4330387 , 2.91879638, 3.5924235 , 3.97294747, 4.04789338,
       3.14087609])
```

1.5 Communication Corner: Interpreting the Model

Even though we cannot visualize Ashenfelter's model, we can still interpret the model by examining the values of the *intercept* b and the *coefficients* c_1, c_2, c_3, c_4.

The coefficients are saved in the `.coef_` attribute, after the model has been fitted. (As above, the trailing underscore in `.coef_` reminds us that these are fitted values.)

[11]:
```
ashen_model.coef_
```

[11]:
```
array([ 0.61871092,  0.00119721, -0.00374825,  0.02435187])
```

These coefficients are in the same order as the columns of `X`. So 0.61871092 is the coefficient for **summer**, 0.00119721 the coefficient for **win**, and so on. If you compare these values with the model at the beginning of this lesson, you will see that they are exactly the coefficients that Ashenfelter obtained.

A positive coefficient means that the predicted target *increases* as that feature increases, while a negative coefficient means that it *decreases* as that feature increases. Since **win** has a positive coefficient (0.0012) and **har** has a negative coefficient (-0.0037), we conclude from the model that Bordeaux wines tend to be best when winter rainfall is high and harvest rainfall is low.

Another essential component of a linear regression model is the *intercept*, which is stored in the `.intercept_` attribute, separately from the coefficients.

[12]:
```
ashen_model.intercept_
```

[12]:
```
-7.831137841446707
```

In principle, the intercept is the predicted value when all of the features are equal to 0. However, this interpretation is often purely hypothetical, since it may be impossible for some features to be 0. For example, to interpret the intercept of -7.8 in the model above, we would have to set

summer equal to 0. That is, we would have to imagine a summer in Bordeaux, France where the average temperature was 0°C (i.e., freezing), which would be so catastrophic that the quality of red wine would be the least of our worries!

2 Exercises

Exercises 1-3 ask you to fit linear regression models to the Ames housing data set (AmesHousing.txt), which contains information about homes in Ames, Iowa.

Gr Liv Area1. Fit a linear regression model that predicts the price of a home (**SalePrice**) using square footage (**Gr Liv Area**) as the only feature. Then, make a graph of the fitted model (this is possible because there is only one feature in this model). Do this the way we did it in the lesson, by creating a grid of `X` values and calling `model.predict()` on those `X` values.

[13]:
```python
import pandas as pd

df_housing = pd.read_csv("AmesHousing.txt", sep='\t')
##bordeaux_df = pd.read_csv("bordeaux.csv",index_col="year")
df_housing
```

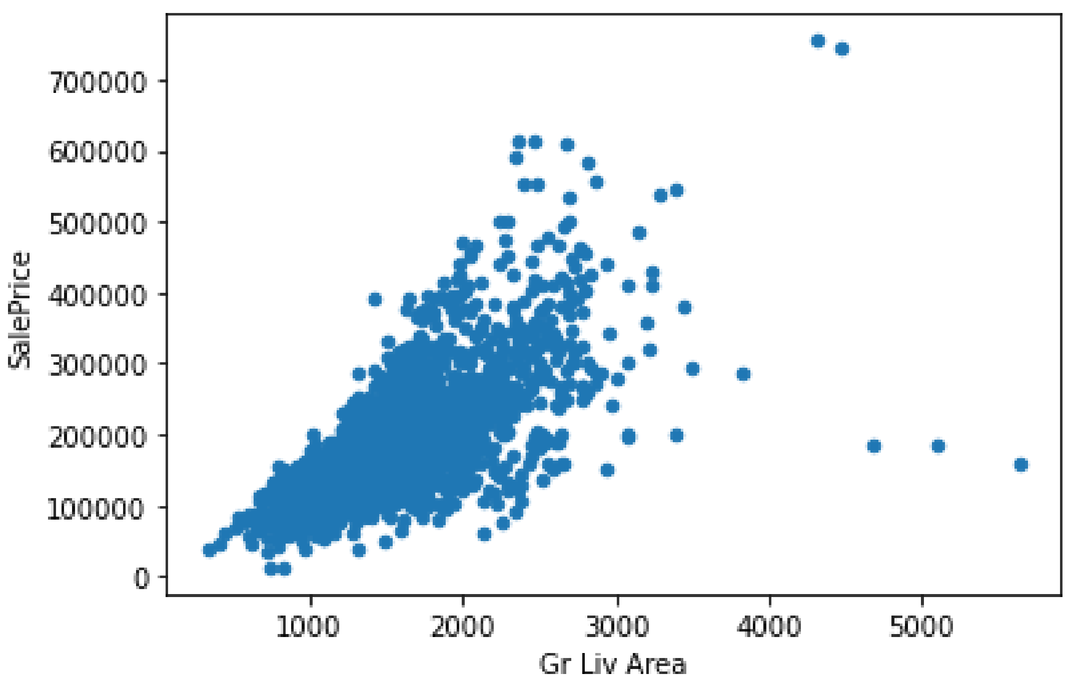
[13]:
```
      Order        PID  MS SubClass MS Zoning  Lot Frontage  Lot Area Street  \
0         1  526301100           20        RL         141.0     31770   Pave
1         2  526350040           20        RH          80.0     11622   Pave
2         3  526351010           20        RL          81.0     14267   Pave
3         4  526353030           20        RL          93.0     11160   Pave
4         5  527105010           60        RL          74.0     13830   Pave
...     ...        ...          ...       ...           ...       ...    ...
2925   2926  923275080           80        RL          37.0      7937   Pave
2926   2927  923276100           20        RL           NaN      8885   Pave
2927   2928  923400125           85        RL          62.0     10441   Pave
2928   2929  924100070           20        RL          77.0     10010   Pave
2929   2930  924151050           60        RL          74.0      9627   Pave

     Alley Lot Shape Land Contour  ... Pool Area Pool QC  Fence Misc Feature  \
0      NaN       IR1          Lvl  ...         0     NaN    NaN          NaN
1      NaN       Reg          Lvl  ...         0     NaN  MnPrv          NaN
2      NaN       IR1          Lvl  ...         0     NaN    NaN         Gar2
3      NaN       Reg          Lvl  ...         0     NaN    NaN          NaN
4      NaN       IR1          Lvl  ...         0     NaN  MnPrv          NaN
...    ...       ...          ...  ...       ...     ...    ...          ...
2925   NaN       IR1          Lvl  ...         0     NaN  GdPrv          NaN
2926   NaN       IR1          Low  ...         0     NaN  MnPrv          NaN
2927   NaN       Reg          Lvl  ...         0     NaN  MnPrv         Shed
2928   NaN       Reg          Lvl  ...         0     NaN    NaN          NaN
2929   NaN       Reg          Lvl  ...         0     NaN    NaN          NaN

     Misc Val Mo Sold Yr Sold Sale Type  Sale Condition  SalePrice
0           0       5    2010       WD           Normal     215000
```

```
1              0      6   2010       WD            Normal     105000
2          12500      6   2010       WD            Normal     172000
3              0      4   2010       WD            Normal     244000
4              0      3   2010       WD            Normal     189900
...          ...    ...    ...      ...               ...        ...
2925           0      3   2006       WD            Normal     142500
2926           0      6   2006       WD            Normal     131000
2927         700      7   2006       WD            Normal     132000
2928           0      4   2006       WD            Normal     170000
2929           0     11   2006       WD            Normal     188000

[2930 rows x 82 columns]
```

[14]:
```python
housing_train = df_housing.loc[:2400].copy() ## :2400
housing_test = df_housing.loc[2401:].copy()  ## 2401:
```

[15]:
```python
housing_train.plot.scatter(x="Gr Liv Area", y="SalePrice")
```

[15]:
```
<AxesSubplot:xlabel='Gr Liv Area', ylabel='SalePrice'>
```



[16]:
```python
from sklearn.linear_model import LinearRegression

X_train = housing_train[["Gr Liv Area"]]
X_test = housing_test[["Gr Liv Area"]]
y_train = housing_train["SalePrice"]
```

```python
model = LinearRegression()
model.fit(X=X_train, y=y_train)
model.predict(X=X_test)
```

```
[16]: array([227417.98142733, 228536.26097545, 167030.88582862, 189508.30474592,
       167478.19764787, 167478.19764787, 203039.48727822, 200802.92818197,
       171615.83197593, 161663.14399763, 177319.05767137, 181344.86404461,
       181792.17586386, 222833.03528002, 162781.42354575, 177319.05767137,
       279865.29223435, 181344.86404461, 167030.88582862, 188613.68110742,
       177319.05767137, 183469.59518605, 172622.28356925, 187607.22951411,
       154058.84307038, 142205.07986027, 182910.45541199, 189731.96065555,
       182239.48768311, 220037.33640971, 183245.93927643, 172622.28356925,
       193198.62725473, 230996.47598133, 183245.93927643, 200579.27227235,
       225740.56210514, 225405.07824071, 201362.06795604, 274162.06653892,
       270695.39993973, 281990.02337579, 235469.59417382, 264321.20651543,
       420097.54756912, 333766.36645394, 229654.54052358, 267564.21720499,
       319340.56028313, 405895.39730794, 293396.47476665, 296974.96932065,
       327504.00098444, 234127.65871607, 201250.24000122, 249000.77670613,
       200131.9604531 , 195211.53044135, 182686.79950236, 157413.68171476,
       309723.35616927, 181009.38018018, 231331.95984576, 188166.36928817,
       206394.3259226 , 190738.41224886, 155624.43443776, 156071.74625701,
       159985.72467544, 142205.07986027, 171056.69220187, 140974.97235733,
       142093.25190546, 143994.32713727, 203039.48727822, 191744.86384217,
       219925.5084549 , 193757.76702879, 208071.74524478, 170833.03629225,
       150592.1764712 , 162110.45581688, 132587.8757464 , 122411.53185847,
       124983.57481916, 207736.26138034, 199460.99272423, 149921.20874233,
       150033.03669714, 124536.26299991, 116149.16638898, 117938.41366597,
       129009.3811924 , 119056.6932141 , 116484.65025341, 152716.90761264,
       119727.66094297, 310394.32389814, 284785.7222461 , 330523.35576438,
       296304.00159178, 271142.71175898, 196106.15407985, 142316.90781508,
       153947.01511557, 148691.10123939, 151263.14420007, 151263.14420007,
       148691.10123939, 184923.35859861, 176871.74585212, 169043.78901525,
       135271.7466619 , 203934.11091672, 189843.78861036, 181568.51995424,
       149585.72487789, 152605.07965782, 130351.31665015, 151598.62806451,
       151598.62806451, 220484.64822896, 147908.3055557 , 229207.22870433,
       255598.62604006, 172845.93947887, 311288.94753664, 214669.59457871,
       163676.04718425, 144665.29486614, 124088.95118066, 132028.73597234,
       212433.03548247, 123641.63936141, 132140.56392715, 114919.05888604,
       132028.73597234, 142093.25190546, 148691.10123939, 237370.66940563,
       131916.90801753, 111116.90842242, 205947.01410335, 189508.30474592,
       154394.32693482, 180897.55222536, 175529.81039437, 136949.16598409,
       132252.39188196, 154058.84307038, 105749.16659142, 127220.1339154 ,
       131134.11233384, 152045.93988376, 111676.04819648,  98033.03770936,
       154506.15488963, 174747.01471068, 196106.15407985, 150480.34851639,
       253362.06694381, 190962.06815848, 207736.26138034, 119168.52116891,
       191633.03588736, 153835.18716076, 152605.07965782, 255710.45399487,
```

```
140639.48849289, 362506.15084073, 127220.1339154 , 147572.82169126,
142428.73576989, 167030.88582862, 169155.61697006, 163899.70309388,
167701.8535575 , 136613.68211965, 140192.17667365, 152716.90761264,
119951.3168526 , 131916.90801753, 168260.99333156, 127220.1339154 ,
190626.58429404, 142540.56372471, 126884.65005097, 132587.8757464 ,
113353.46751867, 242514.75532701, 169267.44492487, 309723.35616927,
161215.83217838, 160992.17626875,  94230.88724574, 107650.24182323,
134488.95097821,  94454.54315537,  95013.68292943, 140863.14440252,
192863.14339029, 211314.75593434, 150256.69260676, 284450.23838166,
178437.33721949, 162557.76763613, 135942.71439077, 142652.39167952,
208407.22910922, 149473.89692308, 116484.65025341, 148020.13351051,
111116.90842242, 105749.16659142, 212768.5193469 , 179108.30494837,
288252.38884528, 193422.28316435, 184587.87473417, 147796.47760089,
130798.6284694 , 130798.6284694 , 130798.6284694 , 130798.6284694 ,
130798.6284694 , 130798.6284694 , 188501.85315261, 143099.70349877,
143099.70349877, 165241.63855163, 126213.68232209, 131357.76824346,
111116.90842242, 129121.20914722, 239719.05645669, 163005.07945538,
201026.5840916 , 199796.47658866, 172845.93947887, 198007.22931166,
174187.87493662, 126772.82209616, 105413.68272698, 232673.89530351,
129904.0048309 , 108880.34932617, 210196.47638622, 111116.90842242,
139521.20894477, 168372.82128637, 121852.39208441, 189620.13270073,
 69516.9092322 , 146678.19805277, 144553.46691133, 108433.03750692,
230101.85234283, 212880.34730172, 212209.37957284, 249783.57238981,
175529.81039437, 203486.79909747, 167478.19764787, 211538.41184397,
148691.10123939, 417972.81642769, 154058.84307038, 121852.39208441,
128562.06937315, 102617.98385667, 190850.24020367, 218807.22890677,
200579.27227235, 139521.20894477, 146007.23032389, 164794.32673238,
144441.63895652, 128002.92959909, 116484.65025341, 190738.41224886,
162557.76763613, 137843.78962258,  83383.57562894, 131693.2521079 ,
105749.16659142, 292501.85112816, 121852.39208441, 197336.26158279,
 91994.32814949, 174411.53084624, 163228.735365  , 106084.65045586,
168372.82128637, 166583.57400938, 134936.26279746,  98815.83339305,
124536.26299991, 173516.90720774, 229319.05665914, 197895.40135685,
165688.95037088, 129904.0048309 , 112794.3277446 , 134488.95097821,
169043.78901525, 159426.58490138, 133929.81120415, 133482.4993849 ,
162110.45581688, 118944.86525929, 132923.35961084, 144106.15509208,
171056.69220187, 177542.71358099, 206953.46569666, 130798.6284694 ,
234463.14258051, 232450.23939389, 229207.22870433, 216123.35799128,
213439.48707578, 112011.53206092, 121852.39208441, 175194.32652993,
219142.71277121, 126325.51027691, 270248.08812048, 105078.19886255,
114471.74706679, 204605.0786456 , 176983.57380693, 165465.29446125,
143099.70349877, 239048.08872782, 260630.88400662, 239159.91668263,
425129.80553568, 188949.16497186, 180897.55222536, 132699.70370121,
221938.41164152, 194428.73475767, 142876.04758914, 140192.17667365,
177319.05767137, 159650.24081101, 300665.29182946, 228200.77711102,
175529.81039437, 224286.79869258, 149585.72487789, 150927.66033564,
159202.92899176, 194987.87453173, 182015.83177349, 233904.00280645,
```

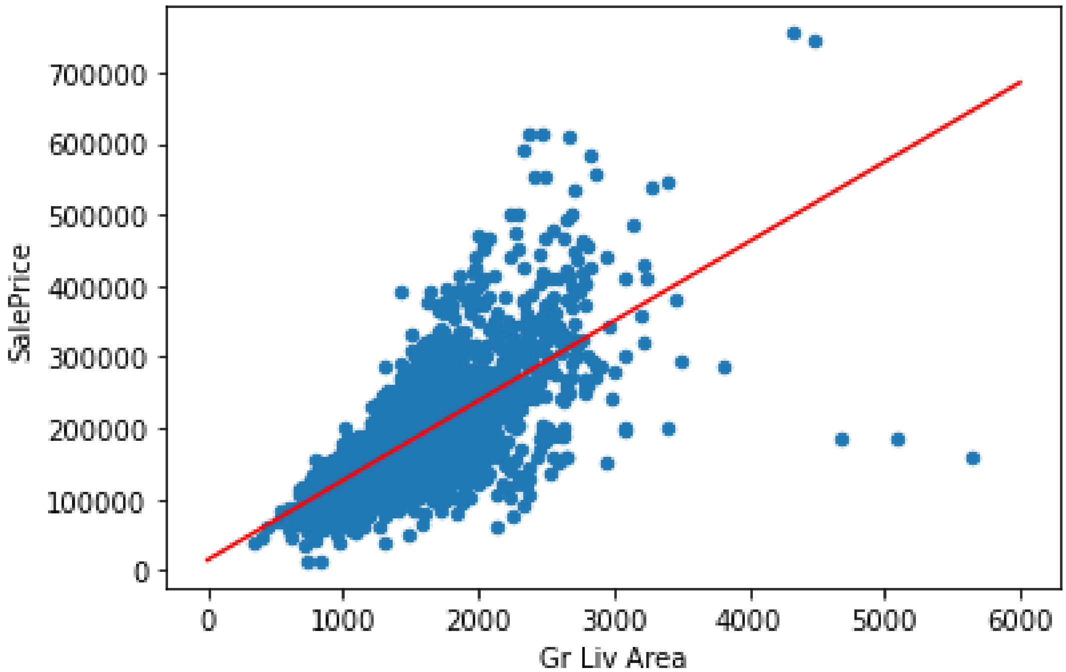
```
       220931.96004821, 208407.22910922, 192304.00361623, 216794.32572015,
       194987.87453173, 192974.97134511, 228648.08893027, 191409.37997773,
       124983.57481916, 185929.81019192, 169714.75674412, 116708.30616304,
       137508.30575815, 182127.6597283 , 173852.39107218, 125207.23072878,
       185035.18655342, 181568.51995424, 192080.34770661, 210084.6484314 ,
       213439.48707578, 222609.3793704 , 213104.00321134, 186824.43383042,
       121181.42435553, 169714.75674412, 157860.99353401, 123082.49958735,
       111116.90842242, 114471.74706679, 111116.90842242, 162110.45581688,
       236923.35758638, 211538.41184397, 210755.61616028, 166359.91809975,
       214334.11071428, 227977.12120139, 199460.99272423, 262084.64741918,
       243633.03487513, 231331.95984576, 237594.32531526, 249000.77670613,
       219813.68050008, 109327.66114542, 109327.66114542, 229095.40074952,
       169938.41265375, 232338.41143908, 204605.0786456 , 208742.71297366,
       194652.39066729, 188054.54133336, 206841.63774184, 163899.70309388,
       182239.48768311, 177319.05767137, 160992.17626875, 175082.49857512,
       178437.33721949, 150927.66033564, 127667.44573465, 124088.95118066,
       146007.23032389, 207288.94956109, 116708.30616304, 273267.44290042,
       134377.1230234 , 134377.1230234 , 134377.1230234 , 205611.53023891,
       205611.53023891, 205611.53023891, 205611.53023891, 205499.7022841 ,
       119168.52116891, 164123.3590035 , 201138.41204641, 146901.85396239,
       151039.48829045, 111116.90842242, 180226.58449649, 154282.49898001,
       172622.28356925, 263873.89469618,  70187.87696107, 156854.5419407 ,
       230101.85234283, 192863.14339029, 187942.71337855, 164347.01491313,
       179891.10063205, 213998.62684984, 183357.76723124, 184699.70268899,
       197448.0895376 , 216011.53003646, 178325.50926468, 158531.96126288,
       206506.15387741, 187830.88542373, 184140.56291492, 206506.15387741,
       146454.54214314, 244527.65851363, 201362.06795604, 276734.1094996 ,
       194428.73475767, 215899.70208165, 217017.98162977, 198678.19704054,
       273379.27085523, 163899.70309388, 220708.30413858, 123641.63936141,
       158196.47739844,  89645.94109843,  86179.27449925, 120845.9404911 ,
       197671.74544723, 163899.70309388,  96020.13452274,  68174.97377445,
       170609.38038262, 119168.52116891, 213327.65912096, 198566.36908573,
       196217.98203467, 200579.27227235, 181233.0360898 , 126772.82209616,
       149809.38078751, 233568.51894201, 296192.17363696, 159314.75694657,
       199013.68090498, 205387.87432928, 141645.94008621, 164235.18695831,
       295633.0338629 , 171392.17606631, 215787.87412684, 233680.34689682,
       241284.64782407, 220037.33640971, 193422.28316435, 167478.19764787,
       150480.34851639, 164011.53104869, 141981.42395064, 160321.20853988,
       179443.7888128 , 296415.82954659, 112235.18797054, 199237.3368146 ,
        84949.16699631, 136613.68211965, 166583.57400938,  84949.16699631,
       136613.68211965, 136613.68211965, 136613.68211965, 136613.68211965,
       207736.26138034, 207736.26138034, 140415.83258327, 151374.97215489,
       126660.99414134, 115366.37070529, 122970.67163253, 169826.58469894,
       238153.46508932])
```

[17]:

```
X_new = pd.DataFrame()
X_new["Gr Liv Area"] = np.linspace(0, 6000, num=200)
```

```
y_new = pd.Series(model.predict(X_new), index= X_new['Gr Liv Area'])
housing_train.plot.scatter(x="Gr Liv Area", y="SalePrice")
y_new.plot.line(color='r')
```

[17]: <AxesSubplot:xlabel='Gr Liv Area', ylabel='SalePrice'>



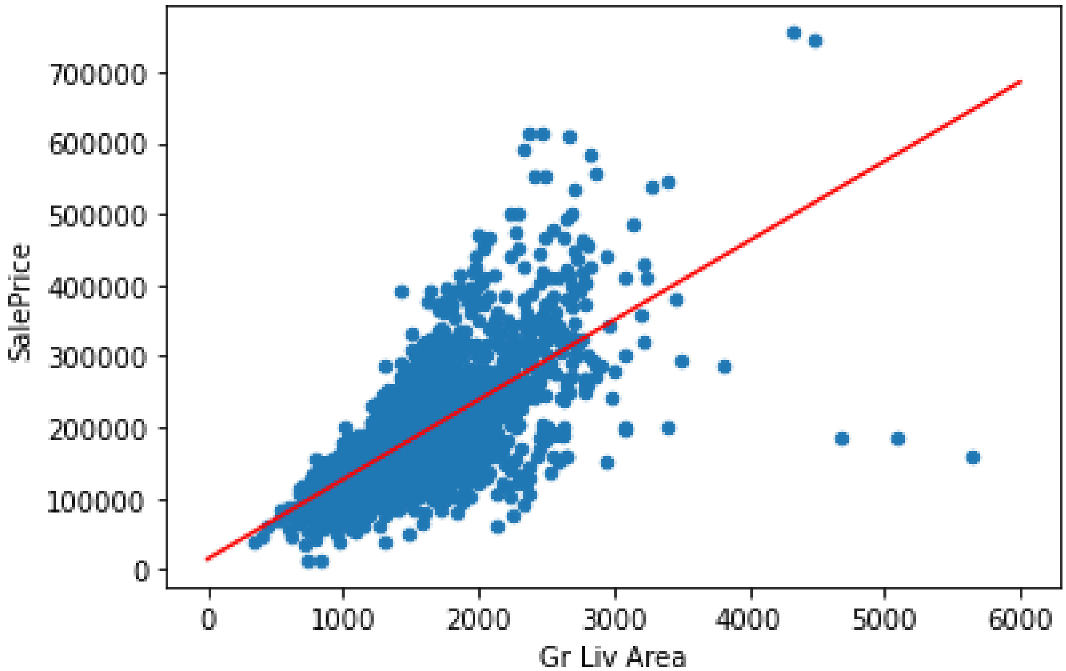
2. There is another way to graph a fitted linear regression model: extract the intercept and coefficient and draw a line with that intercept and slope. Verify that this gives the same graph as Exercise 2.

[18]:
```
B1 = model.coef_
print(model.coef_) ## x
B0 = model.intercept_  ## y
print(model.intercept_)
```

```
[111.82795481]
14497.55546449023
```

[19]:
```
import matplotlib.pyplot as plt
X_new = pd.DataFrame()
X_new["Gr Liv Area"] = np.linspace(0, 6000, num=200)
Y_new = B0 + B1*X_new
housing_train.plot.scatter(x="Gr Liv Area", y="SalePrice")
#y_new.plot.line(color='r')
plt.plot(X_new, Y_new, 'r')
```

[19]: [<matplotlib.lines.Line2D at 0x7fa469218280>]



3. Fit a linear regression model that predicts the price of a home using square footage, number of bedrooms (**Bedroom AbvGr**), number of full bathrooms (**Full Bath**), and number of half bathrooms (**Half Bath**). Interpret the coefficients. Then, use your fitted model to predict the price of a home that is 1500 square feet, with 3 bedrooms, 2 full baths, and 1 half bath.

[20]:
```
X_train = housing_train[["Gr Liv Area", "Bedroom AbvGr", "Full Bath", "Half␣
↪Bath"]]
X_test = housing_test[["Gr Liv Area", "Bedroom AbvGr", "Full Bath", "Half Bath"]]
y_train = housing_train["SalePrice"]

model = LinearRegression()
model.fit(X=X_train, y=y_train)
model.predict(X=X_test)

print(model.coef_)
```

```
[   117.40192734 -30031.61418122  27880.42583583    157.06658563]
```

Sale Prices increase as square footage, number of bathrooms, and number of half bathrooms increase.

Sale Price decreases as number of bedroom increases.

[21]:
```
price_1500 = model.intercept_ + (model.coef_[0] * 1500) + (model.coef_[1] * 3) +
  ↪(model.coef_[2] * 2) + (model.coef_[3] * 1)
print("Predicted price: " + str(price_1500))
```

```
Predicted price: 189716.93231060842
```

Exercises 4-5 ask you to fit linear regression models to the tips data (tips.csv), which contains information about tips collected by a waiter.

4. Suppose you want to predict how much a male diner will tip on a Sunday bill of $40.00. Fit a linear regression model to the tips data to answer this question. (Hint: You will need to convert categorical variables to quantitative variables. asZaqAZ)

[22]:
```
import pandas as pd

df = pd.read_csv("tips.csv")
new = {'sex': {'F': 0, 'M': 1}}
df=df.replace(new)
new = {'day': {'Sun': 1, 'Sat': 0, 'Thu': 0, 'Fri': 0}}
df=df.replace(new)
df
```

[22]:

	obs	totbill	tip	sex	smoker	day	time	size
0	1	16.99	1.01	0	No	1	Night	2
1	2	10.34	1.66	1	No	1	Night	3
2	3	21.01	3.50	1	No	1	Night	3
3	4	23.68	3.31	1	No	1	Night	2
4	5	24.59	3.61	0	No	1	Night	4
..	...	...	...	...	...	...	...	...
239	240	29.03	5.92	1	No	0	Night	3
240	241	27.18	2.00	0	Yes	0	Night	2
241	242	22.67	2.00	1	Yes	0	Night	2
242	243	17.82	1.75	1	No	0	Night	2
243	244	18.78	3.00	0	No	0	Night	2

[244 rows x 8 columns]

[23]:
```
tips_train = df.loc[:190].copy()
tips_test  = df.loc[191:].copy()

# Feature extraction.
X_train = tips_train[["totbill", "sex", "day" ]]
X_test  = tips_test[["totbill", "sex", "day"]]
y_train = tips_train["tip"]

# Fit the model.
model = LinearRegression()
model.fit(X=X_train, y=y_train)
model.predict(X=X_test)
```

```
predicted_tip = model.intercept_ + (model.coef_[0] * 40) + (model.coef_[1] * 1)␣
 ↪+ (model.coef_[2] * 1)

print("Predicted tip: " + str(predicted_tip))
```

```
Predicted tip: 5.213677819551327
```

5. Fit a linear regression model, with no intercept, that predicts the tip from the total bill. That is, we want our predictions to be of the form

$$\widehat{\text{tip}} = c \cdot (\text{total bill}).$$

where c is some coefficient to be learned from the training data.

(*Hint:* `LinearRegression()` has a parameter, `fit_intercept=`, which is `True` by default.)

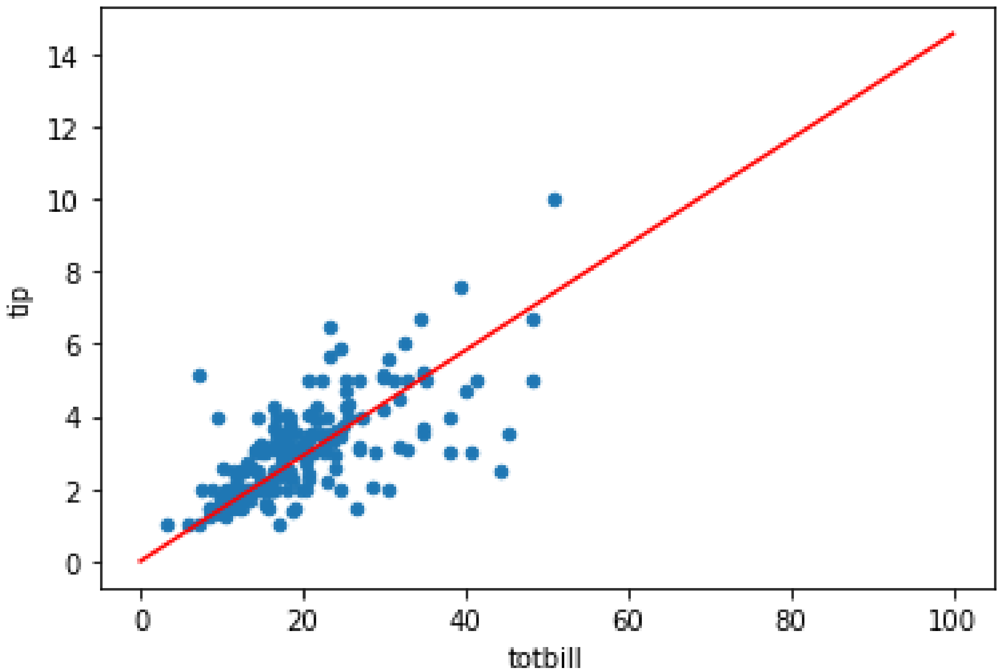
Plot the data and the fitted model. In practical terms, what assumption is being made when we fit a model with no intercept?

[24]:
```
X_train = tips_train[["totbill"]]
X_test = tips_test[["totbill"]]
y_train = tips_train["tip"]

model = LinearRegression(fit_intercept=False)
model.fit(X=X_train, y=y_train)
model.predict(X=X_test)

print(model.coef_)
```

```
[0.14556033]
```

[25]:
```
X_new = pd.DataFrame()
X_new["totbill"] = np.linspace(0,100, num=50)
Y_new_ = pd.Series(model.predict(X_new), index=X_new["totbill"])

tips_train.plot.scatter(x="totbill", y="tip")
Y_new_.plot.line(color='r')
```

[25]:
```
<AxesSubplot:xlabel='totbill', ylabel='tip'>
```



If there is no Y-intercept then that means we have no X-axis data that is negative or 0. The bill is never a negative number since you don't pay a negative amount in a bill. At the Y-intercept, the totalbill would be $0 which means a free meal.

```
[ ]:
```