Data Prediction

Regression with Statsmodels in Python

Python packages for regression

- statsmodels
  - Optimized for insight
- scikit-learn
  - Optimized for prediction

Linear regression and logistic regression

Linear regression

- The response variable is numeric.

Logistic regression

- The response variable is logical. That is, it takes True or False values.

Swedish motor insurance data

- Each row represents one geographic region in Sweden.
- There are 63 rows.

n_claims	total_payment_sek
108	392.5
19	46.2
13	15.7
124	422.2
40	119.4
...	...

Descriptive statistics

```python
import pandas as pd
print(swedish_motor_insurance.mean())
```

```
n_claims            22.904762
total_payment_sek   98.187302
dtype: float64
```

```python
print(swedish_motor_insurance['n_claims'].corr(swedish_motor_insurance['total_payment_sek']))
```

```
0.9128782350234068
```

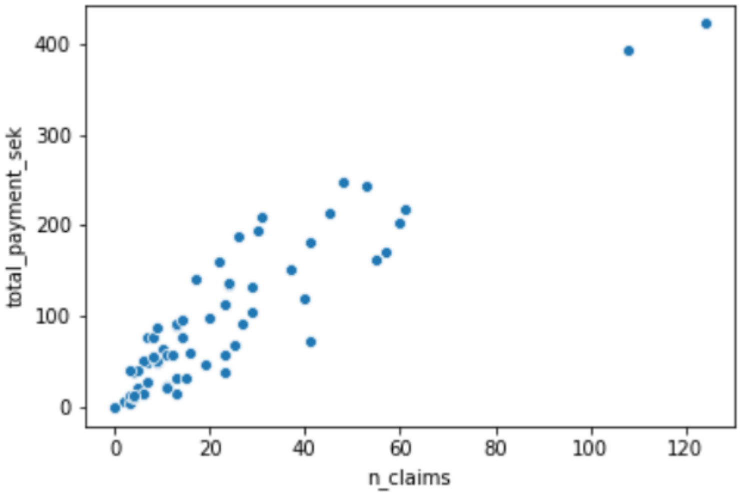
What is regression?

- Statistical models to explore the relationship between a response variable and some explanatory variables.
- Given values of explanatory variables, you can predict the values of the response variable.
- **Response variable** (a.k.a. dependent variable) The variable that you want to predict.
- **Explanatory variables** (a.k.a. independent variables) The variables that explain how the response variable will change.

n_claims	total_payment_sek
108	3925
19	462
13	157
124	4222
40	1194
200	**???**

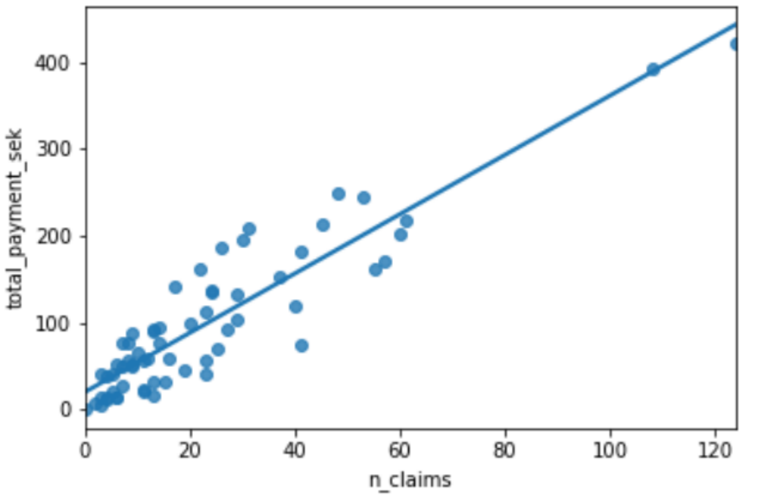
Visualizing pairs of variables

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.scatterplot(x="n_claims",
                y="total_payment_sek",
                data=swedish_motor_insurance)

plt.show()
```



Adding a linear trend line

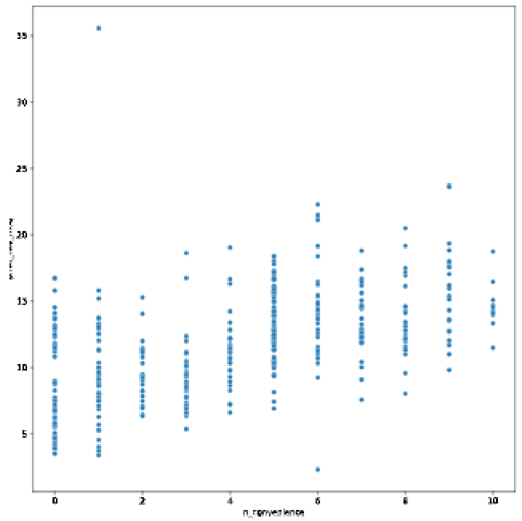
```
sns.regplot(x="n_claims",
            y="total_payment_sek",
            data=swedish_motor_insurance,
            ci=None)
```



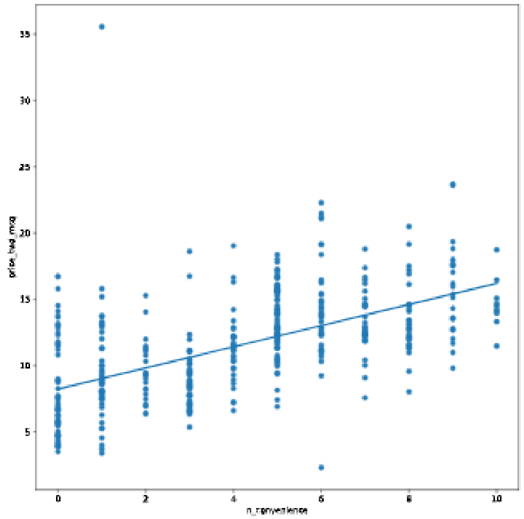
Visualizing two numeric variables
Before you can run any statistical models, it's usually a good idea to visualize your dataset. Here, you'll look at the relationship between house price per area and the number of nearby convenience stores using the Taiwan real estate dataset.

One challenge in this dataset is that the number of convenience stores contains integer data, causing points to overlap. To solve this, you will make the points transparent.

taiwan_real_estate is available from Taiwan_real_estate2.csv



- Import the seaborn and matplotlib packages
- Using taiwan_real_estate, draw a scatter plot of "price_twd_msq" (y-axis) versus "n_convenience" (x-axis).



- Draw a trend line calculated using linear regression. Omit the confidence interval ribbon. Note: The scatter_kws argument in regplot, scatter_kws={'alpha': 0.5}, makes the data points 50% transparent.

Straight lines are defined by two things

Intercept

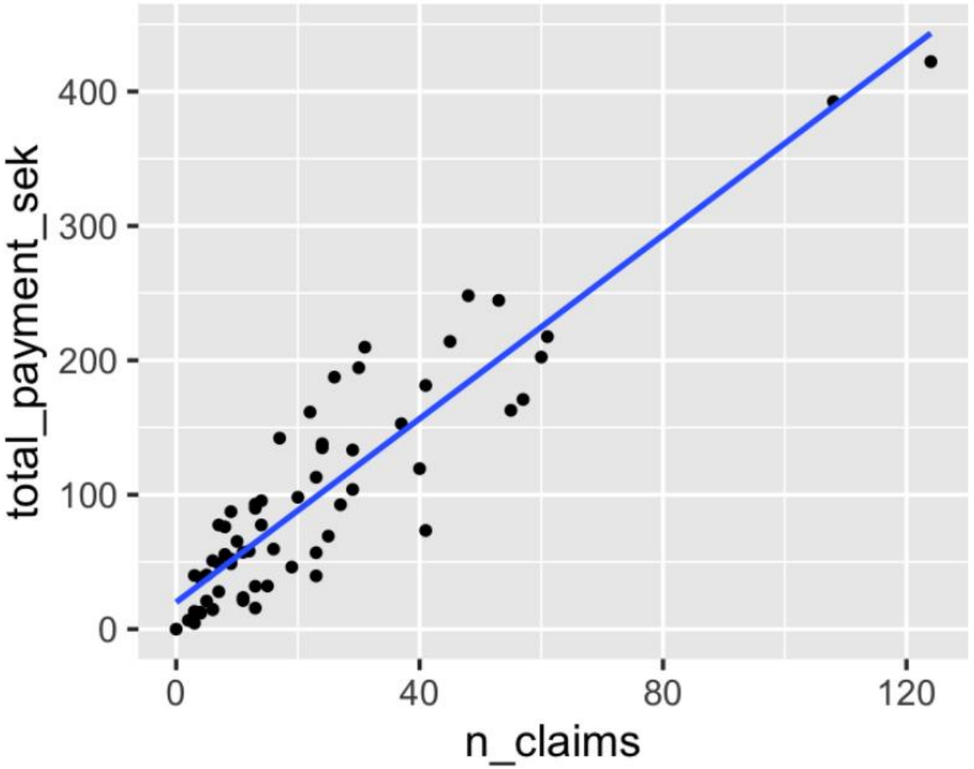
The y value at the point when x is zero.

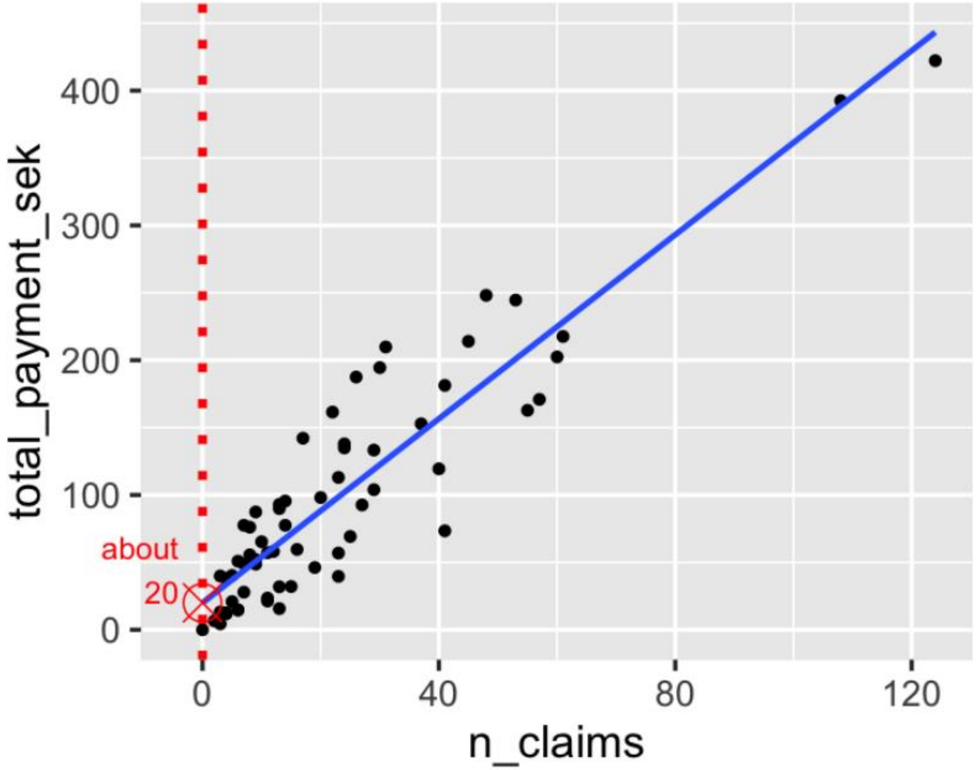
Slope

The amount the y value increases if you increase x by one.

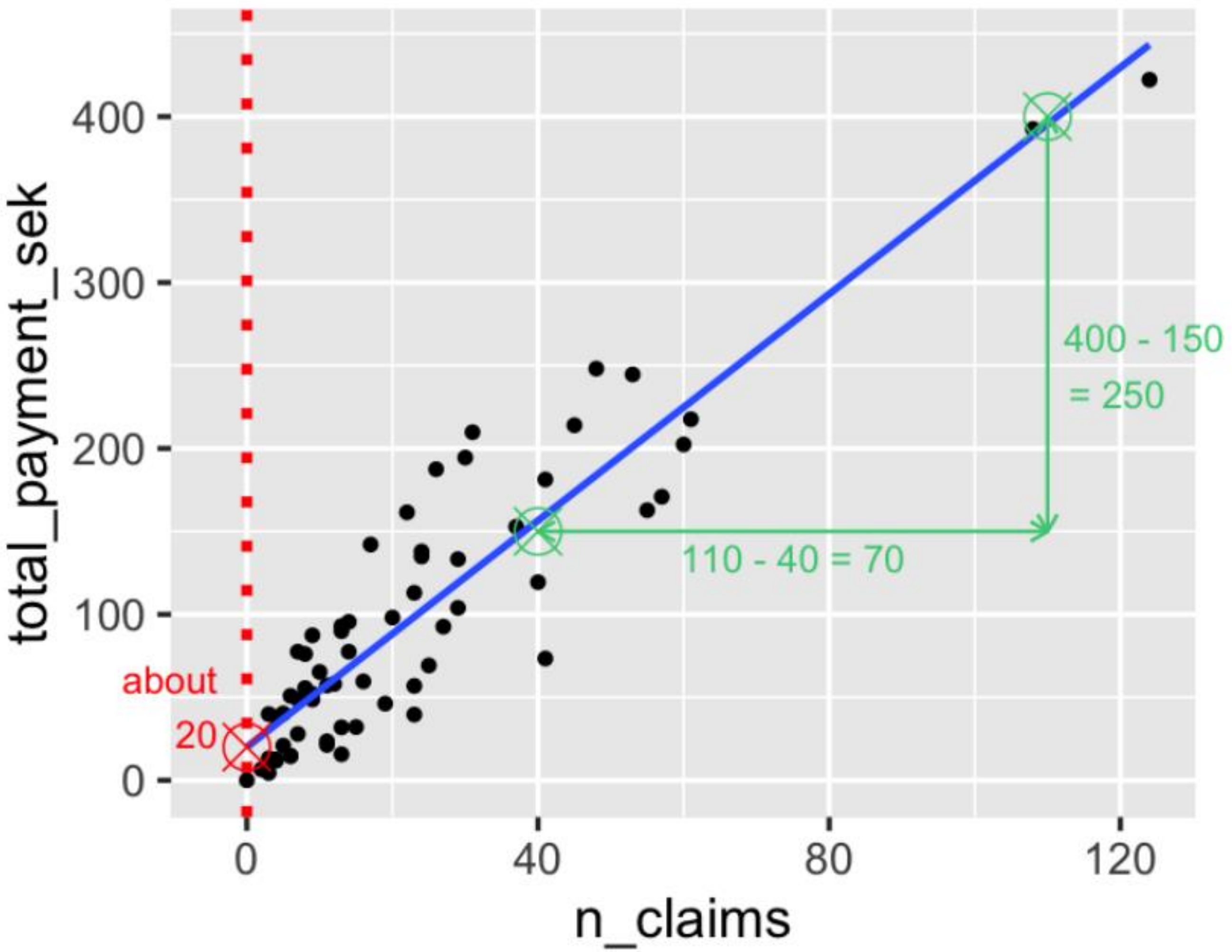
Equation

$y = \text{intercept} + \text{slope} * x$

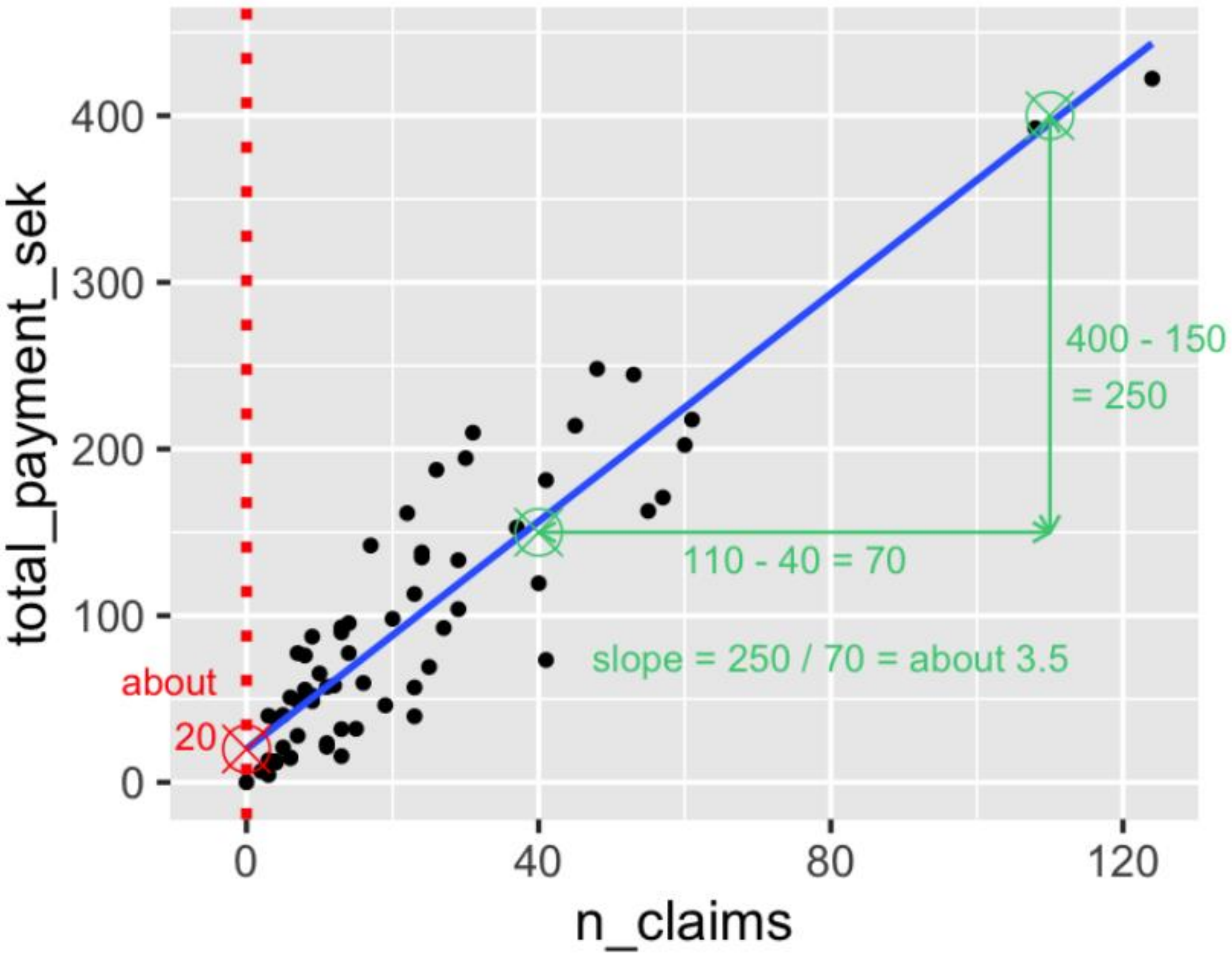




Estimating the slope



Estimating the slope



Running a model

```
from statsmodels.formula.api import ols
mdl_payment_vs_claims = ols("total_payment_sek ~ n_claims",
                            data=swedish_motor_insurance)

mdl_payment_vs_claims = mdl_payment_vs_claims.fit()
print(mdl_payment_vs_claims.params)
```

```
Intercept    19.994486
n_claims      3.413824
dtype: float64
```

Interpreting the model coefficients

```
Intercept    19.994486
n_claims      3.413824
dtype: float64
```

Equation

$$\text{total\_payment\_sek} = 19.99 + 3.41 * \text{n\_claims}$$

Linear regression with ols()
While sns.regplot() can display a linear regression trend line, it doesn't give you access to the intercept and slope as variables, or allow you to work with the model results as variables. That means that sometimes you'll need to run a linear regression yourself.

Time to run your first model!

taiwan_real_estate is available. TWD is an abbreviation for Taiwan dollars.

- Import the ols() function from the statsmodels.formula.api package.
- Run a linear regression with price_twd_msq as the response variable, n_convenience as the explanatory variable, and taiwan_real_estate as the dataset. Name it mdl_price_vs_conv.
- Fit the model.
- Print the parameters of the fitted model.

```
Intercept        8.224237
n_convenience    0.798080
dtype: float64
```

Question

The model had an Intercept coefficient of 8.2242. What does this mean?

- On average, houses had a price of 8.2242 TWD per sqr.m.
- On average, a house with zero convenience stores nearby had a price of 8.2242 TWD per sqr.m.
- The minimum house price was 8.2242 TWD per sqr.m.
- The minimum house price with zero convenience stores nearby was 8.2242 TWD per sqr.m.
- The intercept tells you nothing about house prices

Question

The model had an n_convenience coefficient of 0.7981. What does this mean?

- If you increase the number of nearby convenience stores by one, then the expected increase in house price is 0.7981 TWD per sqr.m.
- If you increase the house price by 0.7981 TWD per sqr.m., then the expected increase in the number of nearby convenience stores is one.
- If you increase the number of nearby convenience stores by 0.7981, then the expected increase in house price is one TWD per sqr.m.
- If you increase the house price by oneTWD per sqr.m., then the expected increase in the number of nearby convenience stores is 0.7981
- The n_convenience coefficient tells you nothing about house prices.

Categorial explanatory variables

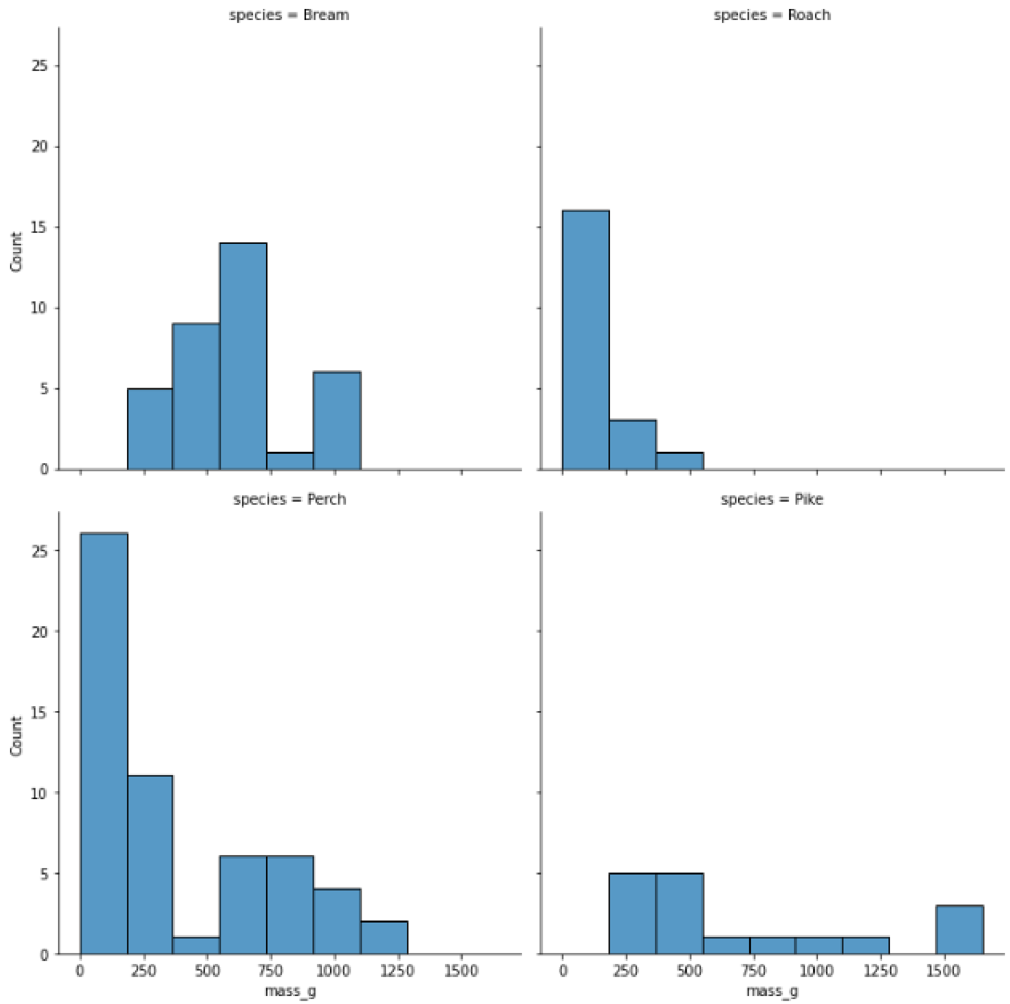
Fish dataset

- Each row represents one fish.
- There are 128 rows in the dataset.
- There are 4 species of fish:
  - Common Bream
  - European Perch
  - Northern Pike
  - Common Roach

species	mass_g
Bream	242.0
Perch	5.9
Pike	200.0
Roach	40.0
...	...

Visualizing 1 numeric and 1 categorical variable

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.displot(data=fish,
            x="mass_g",
            col="species",
            col_wrap=2,
            bins=9)

plt.show()
```



Plot histogram using displot in seaborn.

Summary statistics: mean mass by species

```
summary_stats = fish.groupby("species")["mass_g"].mean()
print(summary_stats)
```

```
species
Bream    617.828571
Perch    382.239286
Pike     718.705882
Roach    152.050000
Name: mass_g, dtype: float64
```

You can see that the mean mass of a bream is approximately six hundred and eighteen grams. The mean mass for a perch is three hundred and eighty two grams, and so on.

Let's run a linear regression using mass as the response variable and species as the explanatory variable. The syntax is the same: you call ols(), passing a formula with the response variable on the left and the explanatory variable on the right, and setting the data argument to the DataFrame. We fit the model using the fit method, and retrieve the parameters using .params on the fitted model.

```python
from statsmodels.formula.api import ols
mdl_mass_vs_species = ols("mass_g ~ species", data=fish).fit()
print(mdl_mass_vs_species.params)
```

```
Intercept            617.828571
species[T.Perch]    -235.589286
species[T.Pike]      100.877311
species[T.Roach]    -465.778571
```

This time we have four values: an intercept, and one coefficient for three of the fish species. A coefficient for bream is missing, but the number for the intercept looks familiar. The intercept is the mean mass of the bream that you just calculated.

Model with or without an intercept

```
species
Bream      617.828571
Perch      382.239286
Pike       718.705882
Roach      152.050000
Name: mass_g, dtype: float64
```

From previous slide, model with intercept

```
mdl_mass_vs_species = ols(
  "mass_g ~ species", data=fish).fit()
print(mdl_mass_vs_species.params)
```

```
Intercept            617.828571
species[T.Perch]    -235.589286
species[T.Pike]      100.877311
species[T.Roach]    -465.778571
```

The coefficients are relative to the intercept: $617.83 - 235.59 = 382.24!$

Model without an intercept

```
mdl_mass_vs_species = ols(
  "mass_g ~ species + 0", data=fish).fit()
print(mdl_mass_vs_species.params)
```

```
species[Bream]    617.828571
species[Perch]    382.239286
species[Pike]     718.705882
species[Roach]    152.050000
```

In case of a single, categorical variable, coefficients are the means.

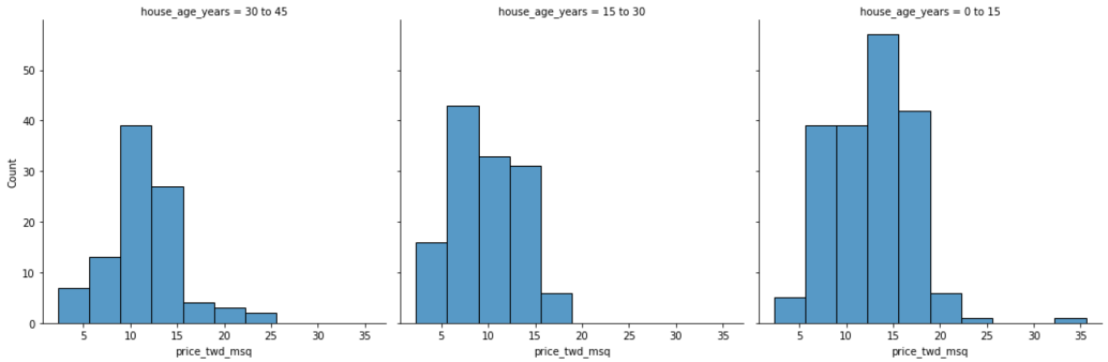
Visualizing numeric vs. categorical

If the explanatory variable is categorical, the scatter plot that you used before to visualize the data doesn't make sense. Instead, a good option is to draw a histogram for each category.

The Taiwan real estate dataset has a categorical variable in the form of the age of each house. The ages have been split into 3 groups: 0 to 15 years, 15 to 30 years, and 30 to 45 years.

taiwan_real_estate is considered in this exercise.

- Using taiwan_real_estate, plot a histogram of price_twd_msq with 10 bins. Split the plot by house_age_years to give 3 panels.



- Group taiwan_real_estate by house_age_years and calculate the mean price (price_twd_msq) for each age group. Assign the result to mean_price_by_age.
- Print the result and inspect the output.

To run a linear regression model with categorical explanatory variables, you can use the same code as with numeric explanatory variables. The coefficients returned by the model are different, however. Here you'll run a linear regression on the Taiwan real estate dataset.

- Run and fit a linear regression with price_twd_msq as the response variable, house_age_years as the explanatory variable, and taiwan_real_estate as the dataset. Assign to mdl_price_vs_age.
- Print its parameters.

```
Intercept                       12.637471
house_age_years[T.15 to 30]     -2.760728
house_age_years[T.30 to 45]     -1.244207
dtype: float64
```

- Update the model formula so that no intercept is included in the model. Assign to mdl_price_vs_age0.
- Print its parameters.

```
house_age_years[0 to 15]     12.637471
house_age_years[15 to 30]     9.876743
house_age_years[30 to 45]    11.393264
dtype: float64
```

Making Predictions

The fish dataset: bream

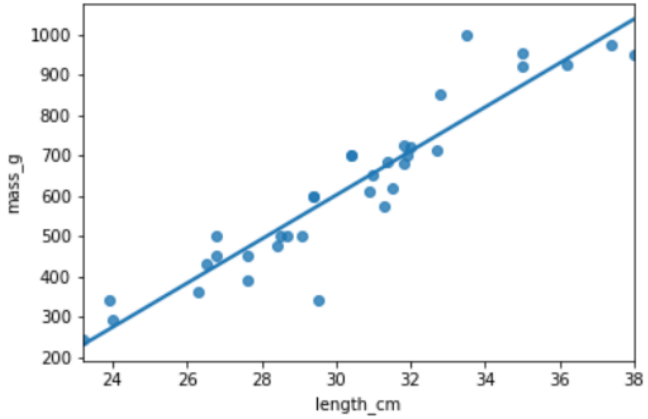
```
bream = fish[fish["species"] == "Bream"]
print(bream.head())
```

```
  species  mass_g  length_cm
0   Bream   242.0       23.2
1   Bream   290.0       24.0
2   Bream   340.0       23.9
3   Bream   363.0       26.3
4   Bream   430.0       26.5
```

This time, we'll look only at the bream data. There's a new explanatory variable too: the length of each fish, which we'll use to predict the mass of the fish.

Plotting mass vs. length

```
sns.regplot(x="length_cm",
            y="mass_g",
            data=bream,
            ci=None)

plt.show()
```



Running the model

```
mdl_mass_vs_length = ols("mass_g ~ length_cm", data=bream).fit()
print(mdl_mass_vs_length.params)
```

```
Intercept   -1035.347565
length_cm      54.549981
dtype: float64
```

Before we can make predictions, we need a fitted model. As before, we call ols with a formula and the dataset, after which we add .fit(). The response, mass in grams, goes on the left-hand side of the formula, and the explanatory variable, length in centimeters, goes on the right.

We need to assign the result to a variable to reuse later on. To view the coefficients of the model, we use the .params attribute.

Data on explanatory values to predict

- If I set the explanatory variables to these values, what value would be the response variable have?

```
explanatory_data = pd.DataFrame({"length_cm": np.arange(20, 41)})
```

```
   length_cm
0         20
1         21
2         22
3         23
4         24
5         25
     ...
```

To predict the values, the next step is to call predict on the model, passing the DataFrame of explanatory variables as the argument. The predict function returns a Series of predictions, one for each row of the explanatory data.

Call predict()

```
print(mdl_mass_vs_length.predict(explanatory_data))
```

```
0        55.652054
1       110.202035
2       164.752015
3       219.301996
4       273.851977
    ...
16      928.451749
17      983.001730
18     1037.551710
19     1092.101691
20     1146.651672
Length: 21, dtype: float64
```

Having a single column of predictions isn't that helpful to work with. It's easier to work with if the predictions are in a DataFrame alongside the explanatory variables. To do this, you can use the pandas assign method. It returns a new object with all original columns in addition to new ones.

You start with the existing column, explanatory_data. Then, you use .assign to add a new column, named after the response variable, mass_g. You calculate it with the same predict code from the previous slide. The resulting DataFrame contains both the explanatory variable and the predicted response.

Now we can answer questions like "how heavy would we expect a bream with length twenty-three centimeters to be?", even though the original dataset didn't include a bream of that exact length. Looking at the prediction data, you can see that the predicted mass is two hundred and nineteen grams.

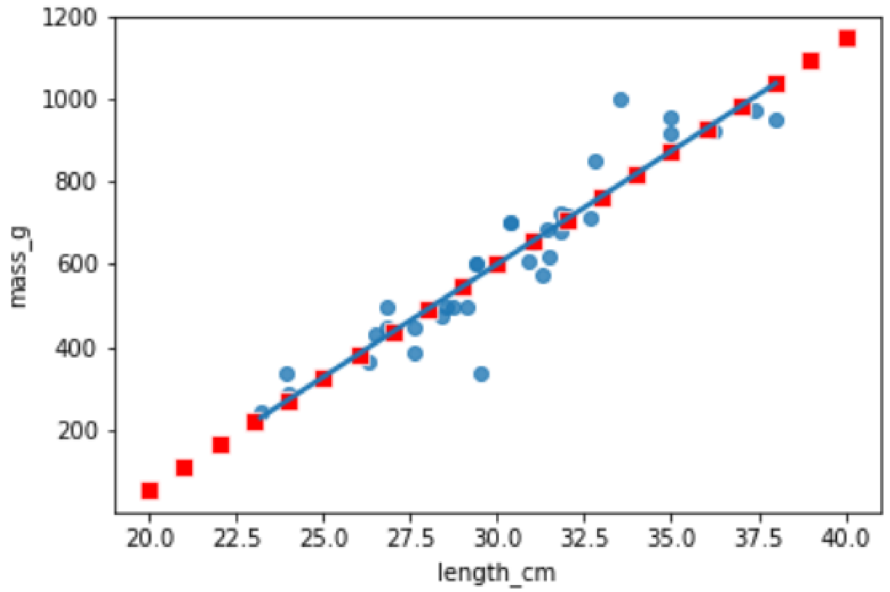
Predicting inside a DataFrame

```
explanatory_data = pd.DataFrame(
  {"length_cm": np.arange(20, 41)}
)
prediction_data = explanatory_data.assign(
    mass_g=mdl_mass_vs_length.predict(explanatory_data)
)
print(prediction_data)
```

```
    length_cm       mass_g
0          20    55.652054
1          21   110.202035
2          22   164.752015
3          23   219.301996
4          24   273.851977
..        ...          ...
16         36   928.451749
17         37   983.001730
18         38  1037.551710
19         39  1092.101691
20         40  1146.651672
```

Showing predictions

```
import matplotlib.pyplot as plt
import seaborn as sns
fig = plt.figure()
sns.regplot(x="length_cm",
            y="mass_g",
            ci=None,
            data=bream,)
sns.scatterplot(x="length_cm",
                y="mass_g",
                data=prediction_data,
                color="red",
                marker="s")
plt.show()
```

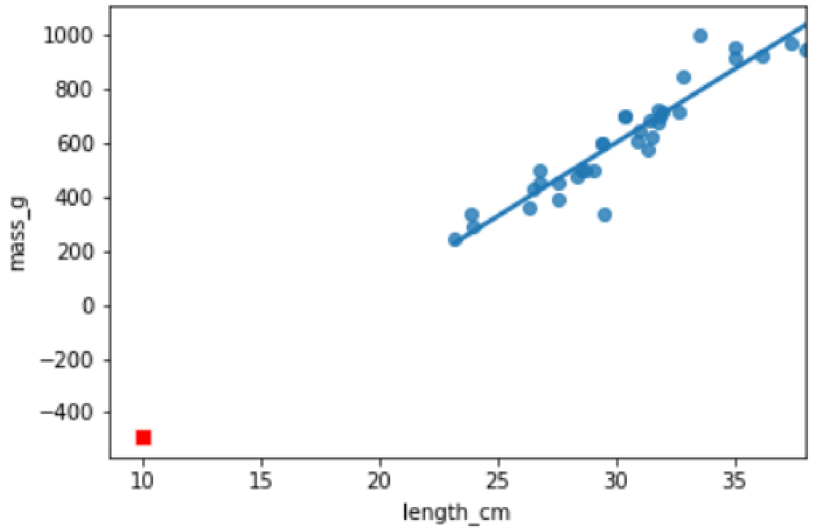


Extrapolating

- Extrapolating means making predictions outside the range of observed data.

```
little_bream = pd.DataFrame({"length_cm": [10]})

pred_little_bream = little_bream.assign(
    mass_g=mdl_mass_vs_length.predict(little_bream))

print(pred_little_bream)
```

```
   length_cm      mass_g
0         10 -489.847756
```



Predicting house prices

Perhaps the most useful feature of statistical models like linear regression is that you can make predictions. That is, you specify values for each of the explanatory variables, feed them to the model, and get a prediction for the corresponding response variable. Here, you'll make predictions for the house prices in the Taiwan real estate dataset.

- Create a DataFrame of explanatory data, where the number of convenience stores, n_convenience, takes the integer values from zero to ten.
- Print explanatory_data.

- Use the model mdl_price_vs_conv to make predictions from explanatory_data and store it as price_twd_msq.
- Print the predictions.

- Create a DataFrame of predictions named prediction_data. Start with explanatory_data, then add an extra column, price_twd_msq, containing the predictions you created in the previous step.

```
    n_convenience  price_twd_msq
0               0       8.224237
1               1       9.022317
2               2       9.820397
3               3      10.618477
4               4      11.416556
5               5      12.214636
6               6      13.012716
7               7      13.810795
8               8      14.608875
9               9      15.406955
10             10      16.205035
```

```
explanatory_data = pd.DataFrame(
  {"length_cm": np.arange(20, 41)}
)
prediction_data = explanatory_data.assign(
    mass_g=mdl_mass_vs_length.predict(explanatory_data)
)
print(prediction_data)
```

.fittedvalues attribute

Fitted values: predictions on the original dataset

```
print(mdl_mass_vs_length.fittedvalues)
```

or equivalently

```
explanatory_data = bream["length_cm"]

print(mdl_mass_vs_length.predict(explanatory_data))
```

```
0       230.211993
1       273.851977
2       268.396979
3       399.316934
4       410.226930
    ...
30      873.901768
31      873.901768
32      939.361745
33     1004.821722
34     1037.551710
Length: 35, dtype: float64
```

.resid attribute

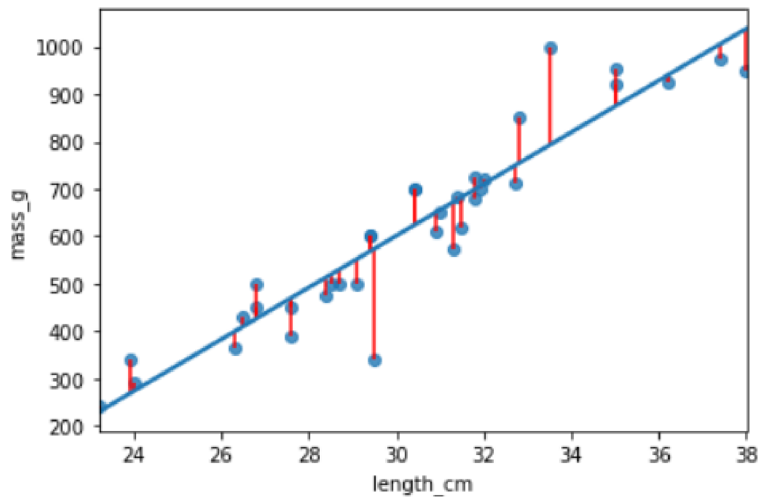
Residuals: actual response values minus predicted response values

```
print(mdl_mass_vs_length.resid)
```

or equivalently

```
print(bream["mass_g"] - mdl_mass_vs_length.fittedvalues)
```

```
0     11.788007
1     16.148023
2     71.603021
3    -36.316934
4     19.773070
   ...
```



"Residuals" are a measure of inaccuracy in the model fit, and are accessed with the .resid attribute. Like fitted values, there is one residual for each row of the dataset. Each residual is the actual response value minus the predicted response value. In this case, the residuals are the masses of breams, minus the fitted values. Here we illustrated the residuals as red lines on the regression plot. Each vertical line represents a single residual.

Manually predicting house prices

You can manually calculate the predictions from the model coefficients. When making predictions in real life, it is better to use .predict(), but doing this manually is helpful to reassure yourself that predictions aren't magic - they are simply arithmetic.

In fact, for a simple linear regression, the predicted value is just the intercept plus the slope times the explanatory variable.

$$\text{response} = \text{intercept} + \text{slope} * \text{explanatory}$$

Here, explanatory_data is a number of convenience store from 0 to10.

- Get the coefficients/parameters of mdl_price_vs_conv, assigning to coeffs.
- Get the intercept, which is the first element of coeffs, assigning to intercept.
- Get the slope, which is the second element of coeffs, assigning to slope.
- Manually predict price_twd_msq using the formula, specifying the intercept, slope, and explanatory_data.
- Run the code to compare your manually calculated predictions to the results from .predict().

Transforming variables

Transforming variables

- Sometimes, the relationship between the explanatory variable and the response variable may not be a straight line. To fit a linear regression model, you may need to transform the explanatory variable or the response variable, or both of them.

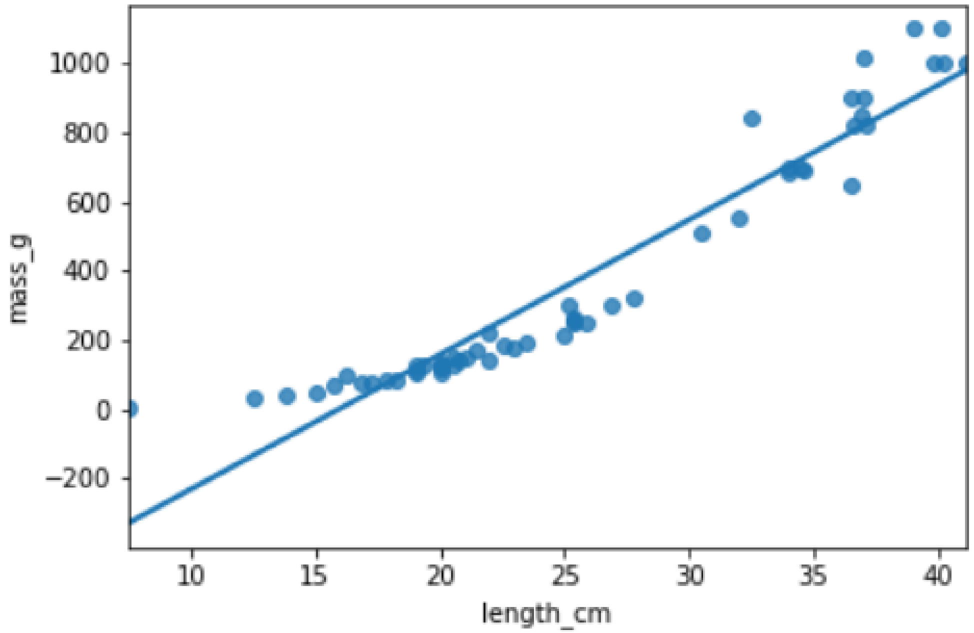
Perch dataset

```
perch = fish[fish["species"] == "Perch"]
print(perch.head())
```

```
   species  mass_g  length_cm
55   Perch     5.9        7.5
56   Perch    32.0       12.5
57   Perch    40.0       13.8
58   Perch    51.5       15.0
59   Perch    70.0       15.7
```

It's not a linear relationship

```
sns.regplot(x="length_cm",
            y="mass_g",
            data=perch,
            ci=None)

plt.show()
```



Bream vs. perch

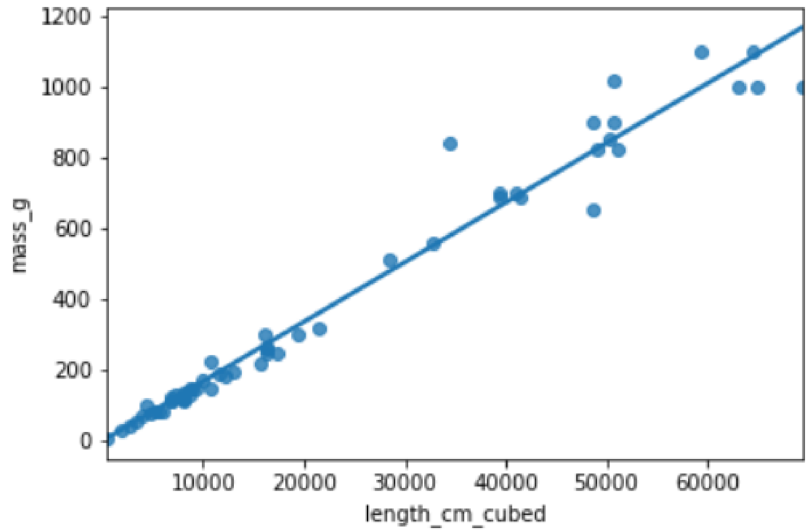
 

To understand why the bream had a strong linear relationship between mass and length, but the perch didn't, you need to understand your data. By looking at the picture of the bream on the left, it has a very narrow body. It is a guess that as bream get bigger, they mostly get longer and not wider. By contrast, the perch on the right has a round body, so it is a guess that as it grows, it gets fatter and taller as well as longer. Since the perches are growing in three directions at once, maybe the length cubed will give a better fit.

Plotting mass vs. length cubed

```
perch["length_cm_cubed"] = perch["length_cm"] ** 3
```

```
sns.regplot(x="length_cm_cubed",
            y="mass_g",
            data=perch,
            ci=None)
plt.show()
```



Here's an update to the previous plot. The only change is that the x-axis is now length to the power of three. To do this, first create an additional column where you calculate the length cubed. Then replace this newly created column in your regplot call. The data points fit the line much better now, so we're ready to run a model.

Modeling mass vs. length cubed

```
perch["length_cm_cubed"] = perch["length_cm"] ** 3

mdl_perch = ols("mass_g ~ length_cm_cubed", data=perch).fit()
mdl_perch.params
```

```
Intercept         -0.117478
length_cm_cubed    0.016796
dtype: float64
```

To model this transformation, we replace the original length variable with the cubed length variable. We then fit the model and extract its coefficients.

Predicting mass vs. length cubed

```
explanatory_data = pd.DataFrame({"length_cm_cubed": np.arange(10, 41, 5) ** 3,
                                 "length_cm": np.arange(10, 41, 5)})
```

```
prediction_data = explanatory_data.assign(
  mass_g=mdl_perch.predict(explanatory_data))
print(prediction_data)
```

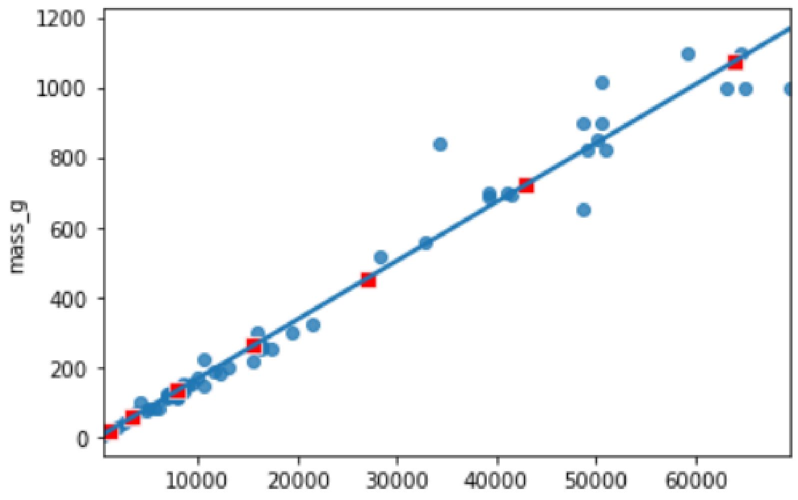
```
   length_cm_cubed  length_cm       mass_g
0             1000         10    16.678135
1             3375         15    56.567717
2             8000         20   134.247429
3            15625         25   262.313982
4            27000         30   453.364084
5            42875         35   719.994447
6            64000         40  1074.801781
```

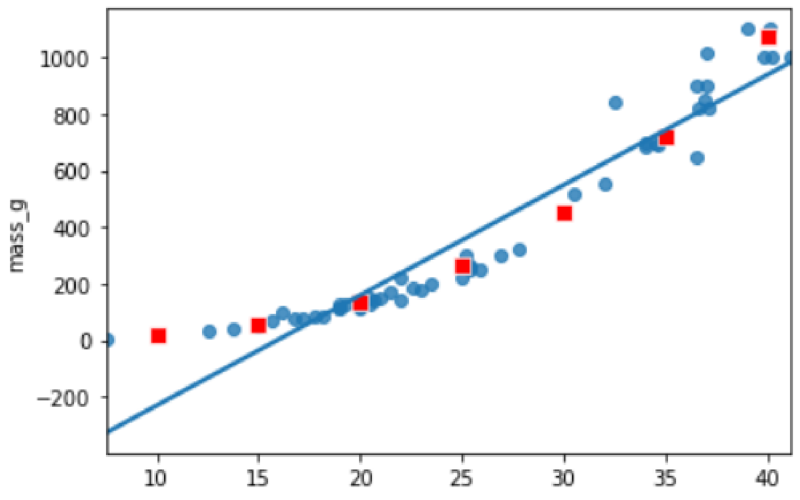
We create the explanatory DataFrame in the same way as usual. Notice that you specify the lengths cubed. We can also add the untransformed lengths column for reference. The code for adding predictions is the same assign and predict combination as you've seen before.

Plotting mass vs. length cubed

```
fig = plt.figure()
sns.regplot(x="length_cm_cubed", y="mass_g",
            data=perch, ci=None)
sns.scatterplot(data=prediction_data,
                x="length_cm_cubed", y="mass_g",
                color="red", marker="s")
```

```
fig = plt.figure()
sns.regplot(x="length_cm", y="mass_g",
            data=perch, ci=None)
sns.scatterplot(data=prediction_data,
                x="length_cm", y="mass_g",
                color="red", marker="s")
```





The predictions have been added to the plot of mass versus length cubed as red points. As you might expect, they follow the line drawn by regplot. It gets more interesting on the original x-axis. Notice how the red points curve upwards to follow the data. Your linear model has non-linear predictions, after the transformation is undone.

```
perch = fish[fish['species'] == 'Perch']
print(perch.head())

sns.regplot(x='length_cm', y = 'mass_g', data = perch, ci=None)
plt.show()

perch['length_cm_cubed'] = perch['length_cm']**3
sns.regplot(x='length_cm_cubed', y = 'mass_g', data = perch, ci=None)
plt.show()

mdl_perch = ols('mass_g ~ length_cm_cubed', data=perch).fit()
print(mdl_perch.params)

explanatory_data = pd.DataFrame({'length_cm_cubed': np.arange(10,41,5)**3,
                                 'length_cm': np.arange(10,41,5)})
prediction_data = explanatory_data.assign(mass_g=mdl_perch.predict(explanatory_data))
print(prediction_data)

fig = plt.figure()
sns.regplot(x='length_cm_cubed', y = 'mass_g', data = perch, ci=None)
sns.scatterplot(data=prediction_data, x = 'length_cm_cubed', y = 'mass_g’, color='red', marker='s')
plt.show()
```

Transforming the explanatory variable

If there is no straight-line relationship between the response variable and the explanatory variable, it is sometimes possible to create one by transforming one or both of the variables. Here, you'll look at transforming the explanatory variable.

You'll take another look at the Taiwan real estate dataset, this time using the distance to the nearest MRT (metro) station as the explanatory variable. You'll use code to make every commuter's dream come true: shortening the distance to the metro station by taking the square root.

- Look at the regplot (dist_to_mrt_m vs price_twd_msq).
- Add a new column to taiwan_real_estate called sqrt_dist_to_mrt_m that contains the square root of dist_to_mrt_m.
- Create the same scatter plot (with regplot) as the first one but use the new transformed variable on the x-axis instead.
- Look at the new plot. Notice how the numbers on the x-axis have changed. This is a different line to what was shown before. Do the points track the line more closely?

- Run a linear regression of price_twd_msq versus the square root of dist_to_mrt_m using taiwan_real_estate.
- Print the parameters.

- Create a DataFrame of predictions named prediction_data by adding a column of predictions called price_twd_msq to explanatory_data.
- Predict using mdl_price_vs_dist and explanatory_data.
- Print the predictions.

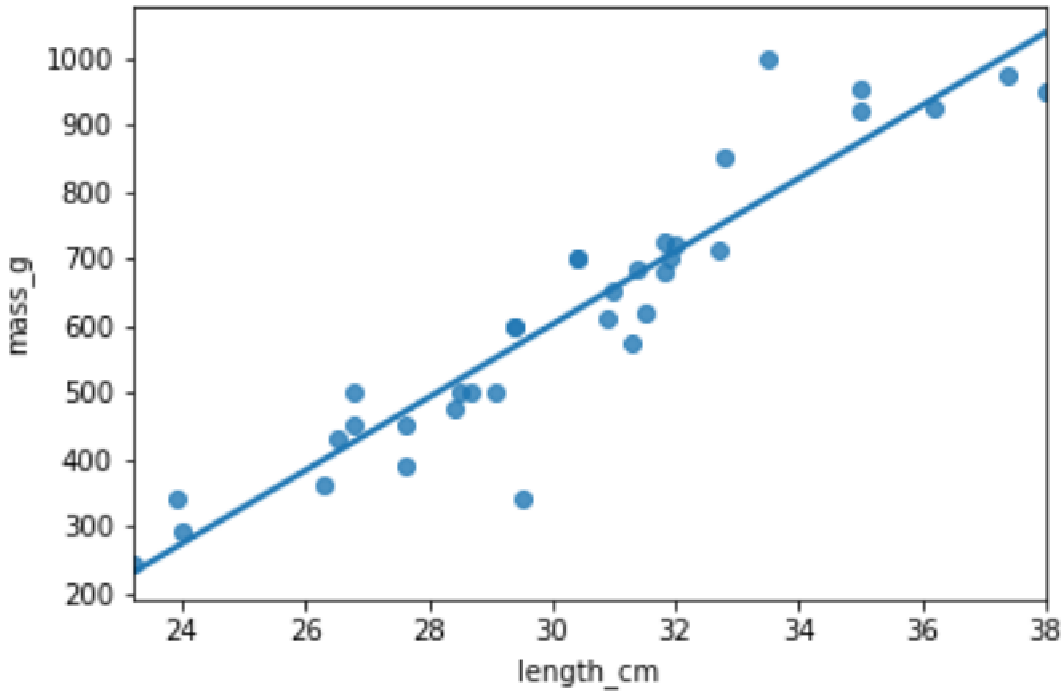
Add a layer to your regplot containing points from prediction_data, colored "red". Both with squared root and original values.

Quantifying Model Fit

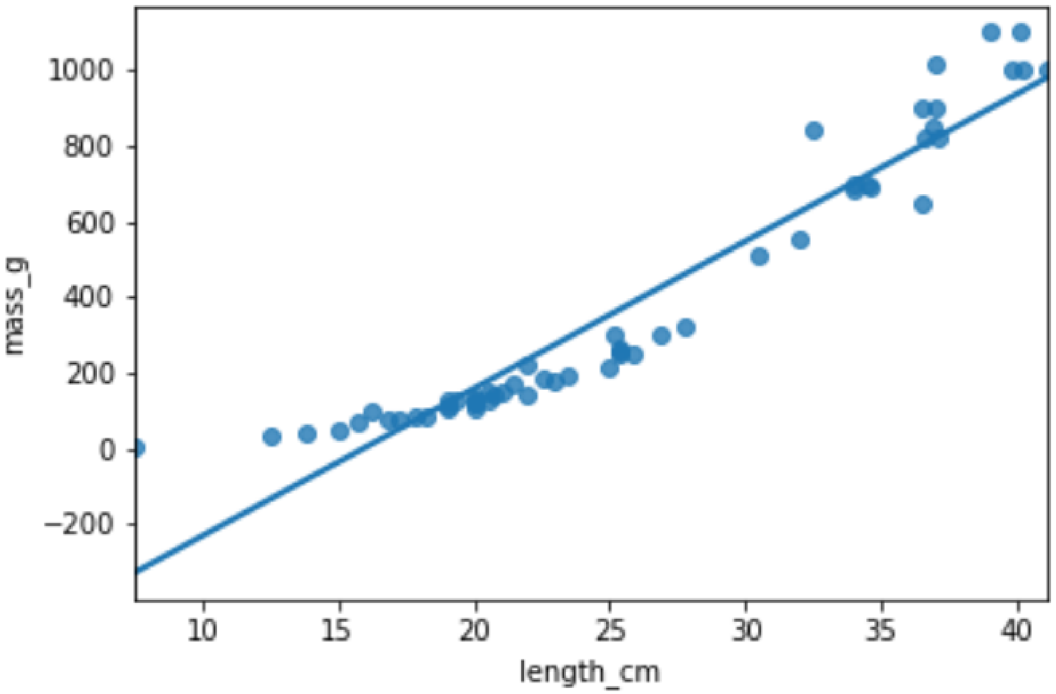
How good or bad model is

Bream and perch models

Bream



Perch



Coefficient of Determination

- Sometimes called "r-squared" or "R-squared".
- The proportion of the variance in the response variable that is predictable from the explanatory variable
  - 1 means a perfect
  - 0 means the worst possible

.summary()

Look at the value titled "R-Squared"

```
mdl_bream = ols("mass_g ~ length_cm", data=bream).fit()
```

```
print(mdl_bream.summary())
```

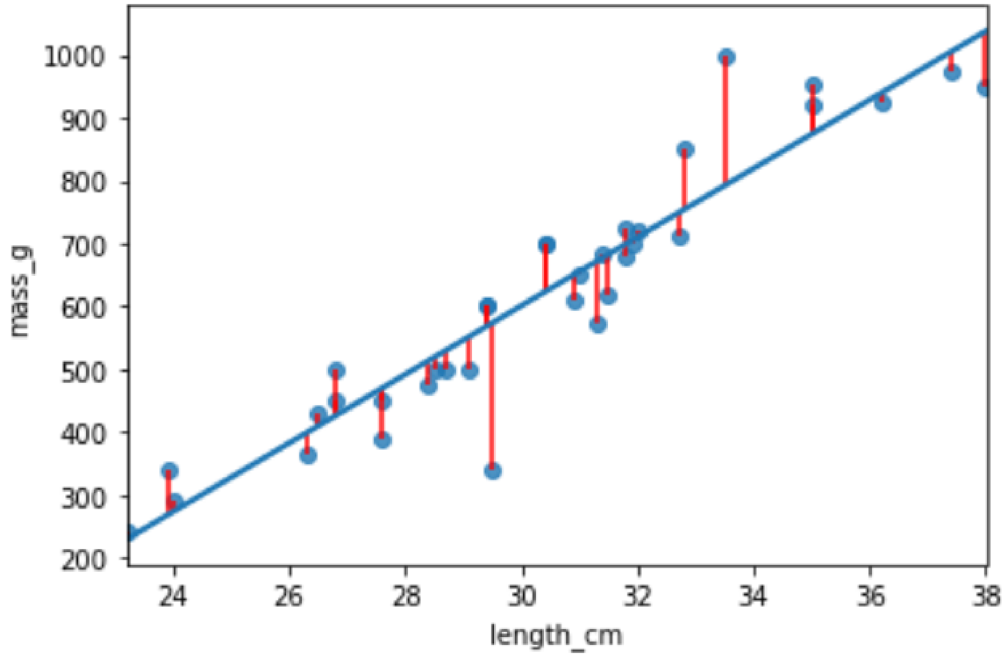
```
# Some lines of output omitted


                            OLS Regression Results
Dep. Variable:                 mass_g   R-squared:                       0.878
Model:                            OLS   Adj. R-squared:                  0.874
Method:                 Least Squares   F-statistic:                     237.6
```

.rsquared attribute

```
print(mdl_bream.rsquared)
```

```
0.8780627095147174
```

Residual standard error (RSE)



- A "typical" difference between a prediction and an observed response
- It has the same unit as the response variable.
- MSE = RSE^2

.mse_resid attribute

```
mse = mdl_bream.mse_resid
print('mse: ', mse)
```

```
mse:  5498.555084973521
```

```
rse = np.sqrt(mse)
print("rse: ", rse)
```

```
rse:  74.15224261594197
```

Interpreting RSE

`mdl_bream` has an RSE of `74`.

> The difference between predicted bream masses and observed bream masses is typically about 74g.

Q&A

HOMEWORK – Mini Project # 1 (Deadline: 11 Sep 2023)
Transforming both response and explanatory variables.

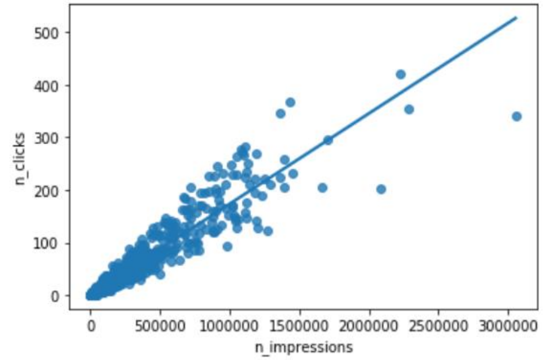
The response variable can be transformed too, but this means you need an extra step at the end to undo that transformation. That is, you "back transform" the predictions.

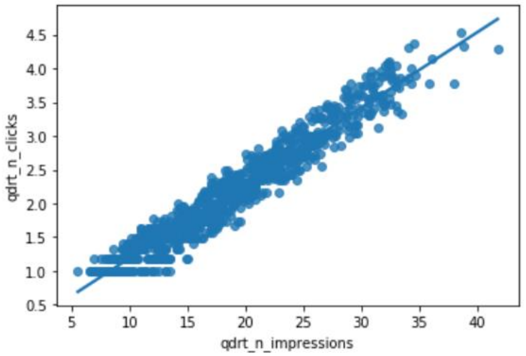
Here you will be working with data on the digital advertising workflow: spending money to buy ads, and counting how many people see them (the "impressions"). The next step is determining how many people click on the advert after seeing it.

ad_conversion can be loaded from ad_conversion.csv.

Complete the following tasks.

- Look at the scatter plot (using regplot for n_impressions vs n_clicks)
- Create a qdrt_n_impressions column using n_impressions raised to the power of 0.25.
- Create a qdrt_n_clicks column using n_clicks raised to the power of 0.25.
- Create a regression plot using the transformed variables. Do the points track the line more closely?





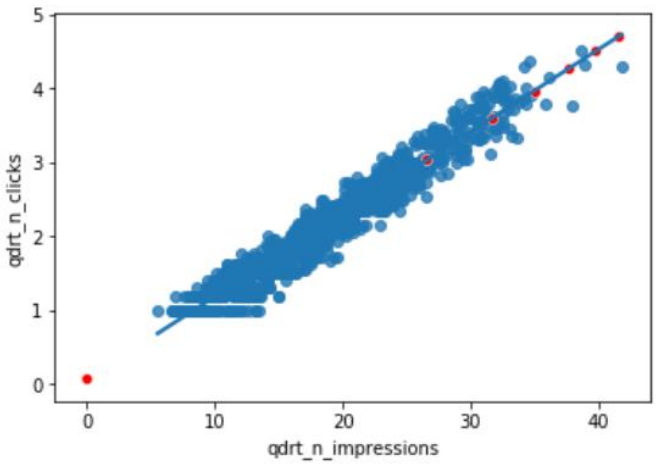
- Run a linear regression of qdrt_n_clicks versus qdrt_n_impressions using ad_conversion and assign it to mdl_click_vs_impression.
- Print the model parameters

```
Intercept             0.071748
qdrt_n_impressions    0.111533
dtype: float64
```

- Create explanatory data for predictions
  - qdrt_n_impressions (for a range of 0 to 3000000, with a step of 500000) raised to a power of 0.25
  - Also have n_impressions as a reference (no need to raise a power)
  - Then predict data for qdrt_n_clicks and assign it to prediction_data.
  - Print the prediction data.

	qdrt_n_impressions	n_impressions	qdrt_n_clicks
0	0.000000	0.0	0.071748
1	26.591479	500000.0	3.037576
2	31.622777	1000000.0	3.598732
3	34.996355	1500000.0	3.974998
4	37.606031	2000000.0	4.266063
5	39.763536	2500000.0	4.506696
6	41.617915	3000000.0	4.713520

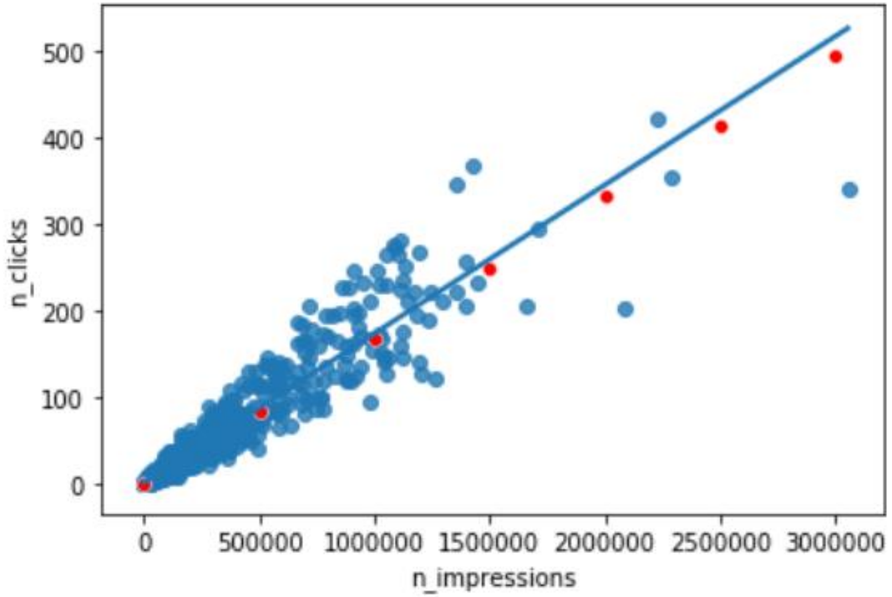
  - Do scatter plot (using regplot) and add a layer of your prediction points



Back transformation

Up until now, you transformed the response variable, ran a regression, and made predictions. But you're not done yet! In order to correctly interpret and visualize your predictions, you'll need to do a back-transformation.

- From prediction_data, create n_clicks by raising qdrt_n_clicks to a power of 4
- Edit the plot to add a layer of points from prediction_data (n_impressions vs. n_clicks), colored "red".
- Determine whether this model is a good fit.
- Also determine RSE and interpret the result.



- Save your work with the following naming convention.
    - YourID_YourName_Section_MiniProject1_LinearRegression.ipynb
- You must submit both .ipynb and .pdf of your work.
- Have your name and ID included in the first and the last cell of your Jupyte