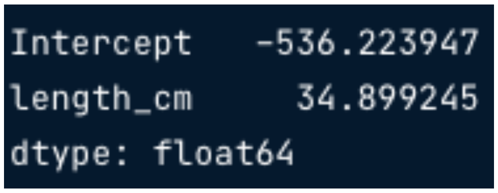
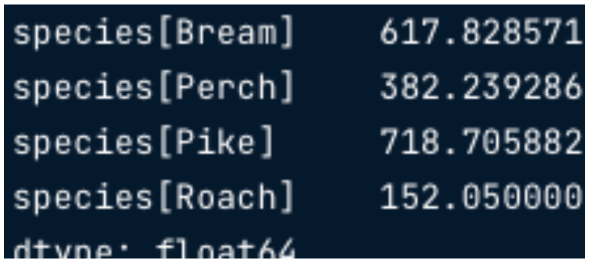
Intermediate Regression with statsmodel
by datacamp

multiple regression is a regression model with more than one explanatory variable
more explanatory variables can give more insight into the relationship between the explanatory variables and the response and provide better predictions
example - using the fish dataset (designating mass_g as response variable and length_cm as numeric explanatory variable, and species as categorical explanatory variable
from statsmodels.formula.api import ols
#**place the response varialbe on the left and the explanatory variable on the right
mdl_mass_vs_length = ols(‘mass_g ~ length_cm’, data=fish).fit()
#show the model coefficients using the params attribute
print(mdl_mass_vs_length.params)
**with a single numeric explanatory variable, you get one intercept coefficient and one slope coefficient



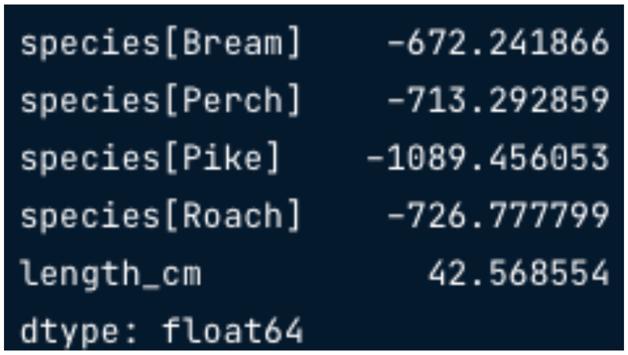
```
Intercept   -536.223947
length_cm     34.899245
dtype: float64
```

#change explanatory variable to ‘species’
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
dtype: float64
```

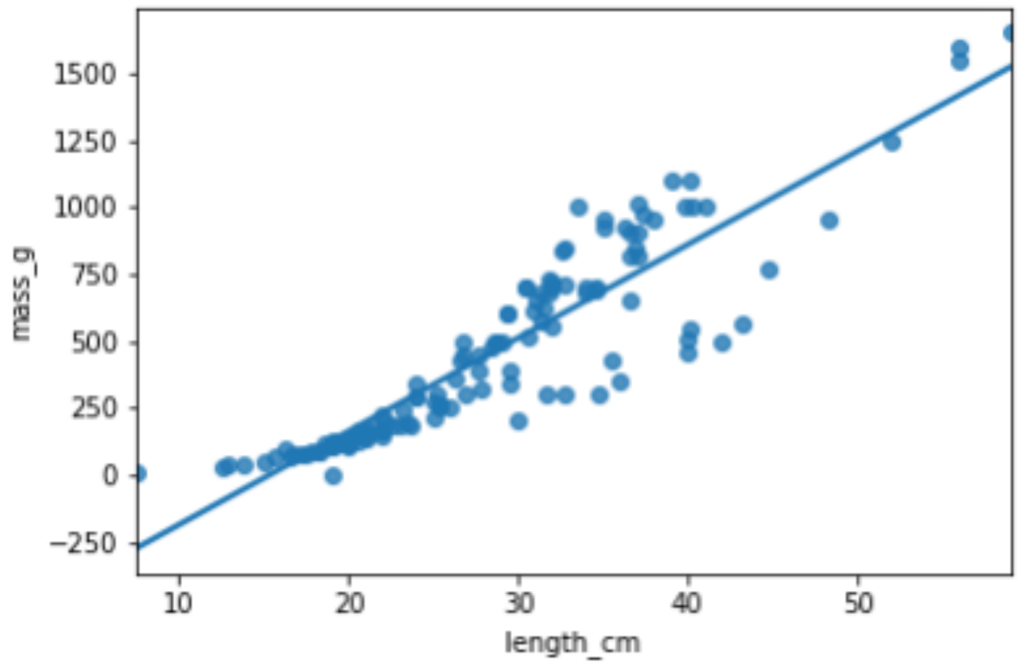
**when you have a categorical explanatory variable, the coefficients are a little easier to understand if you add ‘+ 0’
‘+ 0’ tells statsmodels not to include an intercept in the model
**output is one intercept for each category instead of one for the model
get one intercept coefficient for each category, ie. one coefficient for each species of fish
#now add both explanatory variables to the model
mdl_mass_vs_both = ols(‘mass_g~ length_com + species + 0’, data=fish).fit()
print(mdl_mass_vs_both.params)



```
species[Bream]     -672.241866
species[Perch]     -713.292859
species[Pike]     -1089.456053
species[Roach]     -726.777799
length_cm            42.568554
dtype: float64
```

output is one slope coefficient and an intercept coefficient for each category in the categorical variable
**above outputs show how drastically things can change as you add in additional features

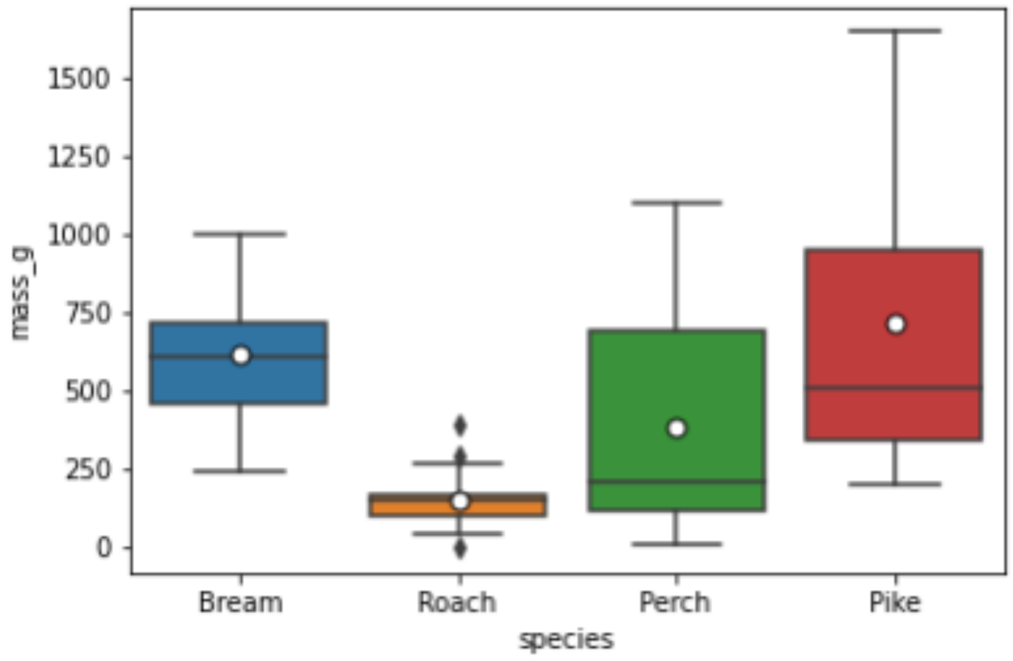
Visualize with 1 numeric explanatory variable
import matplotlib.pyplot as plt
import seaborn as sns
sns.regplot(x='length_cm', y='mass_g', data=fish, ci=None)
plt.show()



standard visualization for a linear regression with a numeric explanatory variable

Visualization for 1 categorical variable
few possible plots, the simplest is to draw a box plot for each category
**the model coefficients are the means of each category
#can show these on our boxplot with the 'showmeans' argument

sns.boxplot(x='species', y='mass_g', data=fish, showmeans=True)
plt.show()



Visualization of both the numerical and categorical explanatory variables
*seaborn doesn't have an easy way to do this

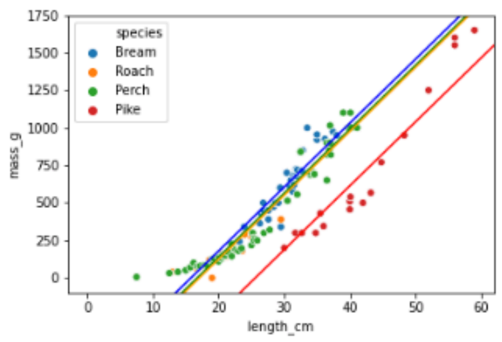
```
coeffs = mdl_mass_vs_both.params
print(coeffs)
```

```
species[Bream]    -672.241866
species[Perch]    -713.292859
species[Pike]    -1089.456053
species[Roach]    -726.777799
length_cm           42.568554
```

```
ic_bream, ic_perch, ic_pike, ic_roach, sl = coeffs
```

```
sns.scatterplot(x="length_cm",
                y="mass_g",
                hue="species",
                data=fish)
```

```
plt.axline(xy1=(0, ic_bream), slope=sl, color="blue")
plt.axline(xy1=(0, ic_perch), slope=sl, color="green")
plt.axline(xy1=(0, ic_pike), slope=sl, color="red")
plt.axline(xy1=(0, ic_roach), slope=sl, color="orange")
```



first extract the model coefficients into separate intercepts and the slope
*hue argument is invaluable when working with a categorical or continuous variable and want to color by its values
need to use plt.axline four times for each intercept
*axline draws a straightline defined by at least one point and the slope
'xy1' argument specifies the intercept with x at 0 and y as each species intercept
'slope' argument is set to the slope coefficient which is the same for all species
specify a 'color' arugment to distinguish each species

**since all slopes are equal in all plt.axline call, the trend lines run parallel to each other
this is why this type of regression is called ‘parallel slopes regression’

example

```
# Import ols from statsmodels.formula.api
from statsmodels.formula.api import ols

# Fit a linear regression of price_twd_msq vs. n_convenience
mdl_price_vs_conv = ols("price_twd_msq ~ n_convenience",
                    data=taiwan_real_estate).fit()

# Fit a linear regression of price_twd_msq vs. house_age_years, no intercept
mdl_price_vs_age = ols("price_twd_msq ~ house_age_years + 0",
data=taiwan_real_estate).fit()

# Fit a linear regression of price_twd_msq vs. n_convenience plus
house_age_years, no intercept
mdl_price_vs_both = ols('price_twd_msq ~ n_convenience + house_age_years + 0',
data=taiwan_real_estate).fit()

# Print the coefficients
print(mdl_price_vs_both.params)
```

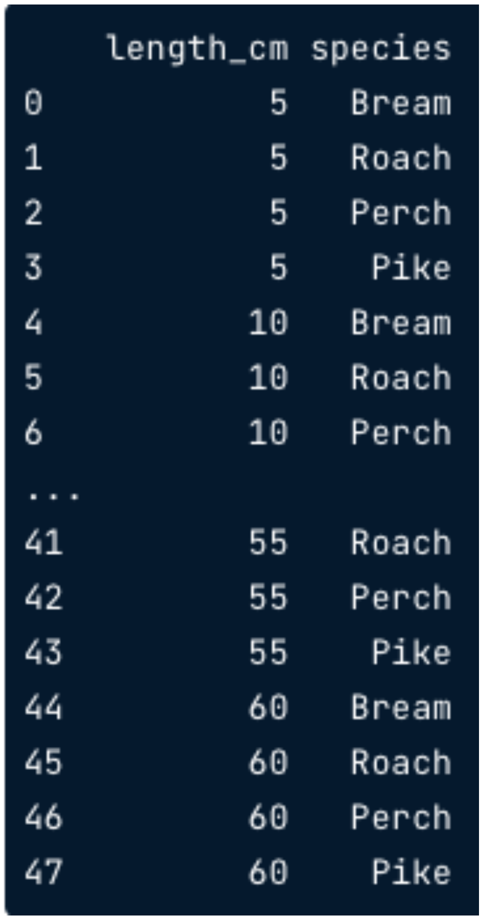
Predicting parallel slopes
the prediction workflow

```
import pandas as pd
import numpy as np
#key difference > for a single explanatory variable, the DataFrame has one column
#here we set it as a range
#remember arange(start, finish, interval)
expl_data_length = pd.DataFrame({length_cm’: np.arange(5, 61, 5)})
print(expl_data_length)
```

**for multiple explanatory variables you need to define multiple columns
**example - create a DataFrame that holds all combinations
to do this use the product() function in the itertools module
*product() returns a Cartesian product of your input variables
Cartesian product outputs all combinations of its inputs

```
from intertools import product
product([‘A’, ‘B’, ‘C’], [1, 2])
length_cm = np.arange(5, 61, 5)
species = fish[‘species’].unique()
#.unique() used for categorical variables and extracts the unique values of your
```

categorical variable
p = product(length_cm, species)
#transform the outp of the product function into a pandas DataFrame
expl_data_both = pd.DataFrame(p, columns=['length_cm', 'species'])
print(expl_data_both)



```
    length_cm species
0           5   Bream
1           5   Roach
2           5   Perch
3           5    Pike
4          10   Bream
5          10   Roach
6          10   Perch
...
41         55   Roach
42         55   Perch
43         55    Pike
44         60   Bream
45         60   Roach
46         60   Perch
47         60    Pike
```

next add a column of predictions to the DataFrame
predict mass_g from length_cm (single explanatory variable)
prediction_data_length = expl_data_length.assign(mass_g =
mdl_mass_vs_length.predict(expl_data))
predict mass_g from both explanatory variables
mass_g = expl_data_both.assign(mass_g = mdl_mass_vs_both.predict(expl_data))

Sidebar: .assign() method
In the context of data manipulation using libraries like Pandas in Python, the `assign()` method is used to create new columns in a DataFrame by applying specified functions to existing columns or adding constant values. It is a convenient way to add calculated or transformed columns to a DataFrame without modifying the original DataFrame in place.

The `assign()` method takes one or more keyword arguments, where each argument represents the name of the new column and the corresponding value or function to generate that column.

Here's the basic syntax of the `assign()` method:

```python
new_dataframe = old_dataframe.assign(new_column_name=value_or_function, ...)
```

Here's an example of how you might use the `assign()` method in Pandas:

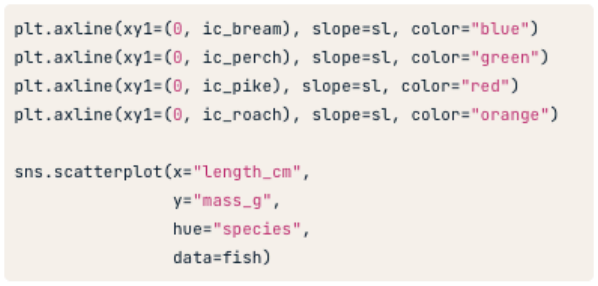
```python
import pandas as pd

# Sample DataFrame
data = {'A': [1, 2, 3, 4, 5],
        'B': [10, 20, 30, 40, 50]}

df = pd.DataFrame(data)

# Using assign() to create a new column
new_df = df.assign(C = df['A'] * 2, D = df['B'] / 2)

print(new_df)
```

In this example, we have a DataFrame `df` with columns 'A' and 'B'. We use the `assign()` method to create two new columns 'C' and 'D'. Column 'C' is created by doubling the values in column 'A', and column 'D' is created by dividing the values in column 'B' by 2.

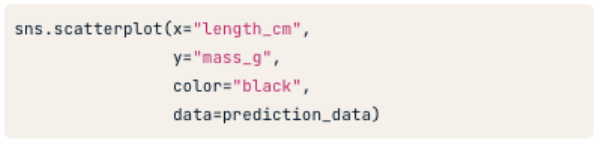
The `assign()` method returns a new DataFrame with the added columns, leaving the original DataFrame `df` unchanged.

Keep in mind that the `assign()` method is particularly useful for chaining operations and building more complex data transformations in a readable and concise manner.

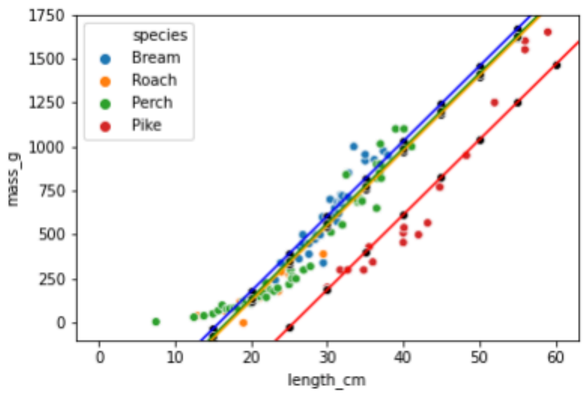
Visualize



```
plt.axline(xy1=(0, ic_bream), slope=sl, color="blue")
plt.axline(xy1=(0, ic_perch), slope=sl, color="green")
plt.axline(xy1=(0, ic_pike), slope=sl, color="red")
plt.axline(xy1=(0, ic_roach), slope=sl, color="orange")

sns.scatterplot(x="length_cm",
                y="mass_g",
                hue="species",
                data=fish)
```



```
sns.scatterplot(x="length_cm",
                y="mass_g",
                color="black",
                data=prediction_data)
```



**need to call two scatterplots
color argument to black to see prediction values

Manually calculating predictions for linear regression (repeat for practice)
example - single explanatory variable
#params attribute contains the coefficients of the model
coeffs = mdl_mass_vs_length.params
print(coeffs)
intercept, slope = coeffs
#response value is the intercept plus the slope times the explanatory variable
explanatory_data = pd.DataFrame({'length_cm': np.arange(5, 61, 5)})
prediction_data = explanatory_data.assign(mass_g = intercept + slope *
explanatory_data)
print(prediction_data)

Manually predicting multiple regression becomes clunky because you need to
label the different intercepts for each categorical variable
NumPy has a function called select() that simplifies the process of getting values
based on conditions
takes two arguments > list of conditions and a list of choices
*both lists have to be of the same length
example
conditions = [condition1, condition2, … condition_n]
choices = [choice1, choice2, … choice_n]
np.select(conditions, choices)
**read as follows 'if condition 1 is met, take the first element in choices, if
condition 2 is met, take the second element in choices, and so on'
example with fish dataset
conditions = [explanatory_data['species'] == 'Bream', explanatory_data['species']
== 'Perch', explanatory_data['species'] == 'Pike', explanatory_data['species'] ==
"Roach']
#condition list contains a condition statement so it returns a True or False

#choices list is the collection of intercepts extracted from the model coefficients
choices = [ic_bream, ic_perch, ic_pike, ic_roach]
intercept = np.select(conditions, choices)
print(intercept)
#our explanatory dataset contains 48 rows of data (12 for each species)
output contains 48 intercepts, 4 intercepts, each repeated 12 times
#next step
#response variable is the intercept plus the slope times the numeric explanatory variable (y + mb)
#assign function - creates new DataFrame by using old DataFrame and creating new columns
#new DF name, oldDF.assign(new col name = new col contents, **can put mult cols in)
prediction_data = explanatory_data.assign(intercept = np.select(conditions, choices), mass_g = intercept + slope * explanatory_data['length_cm'])
print(prediction_data)

example
# Create n_convenience as a range of numbers from 0 to 10
n_convenience = np.arange(0, 11)

# Extract the unique values of house_age_years
house_age_years = taiwan_real_estate["house_age_years"].unique()

# Create p as all combinations of values of n_convenience and house_age_years
p = product(n_convenience, house_age_years)

# Transform p to a DataFrame and name the columns
explanatory_data = pd.DataFrame(p, columns=['n_convenience', 'house_age_years'])

# Add predictions to the DataFrame
prediction_data = explanatory_data.assign(price_twd_msq = mdl_price_vs_both.predict(explanatory_data))

print(prediction_data)

Assessing model performance
coefficient of determination also called R-squared shows how well the linear regression line fits the observed values
**a larger number is better (range is 0 to 1 with 0 being the worst possible fit and 1 being a perfect fit
can get the coefficient of determination by using the rsquared attribute

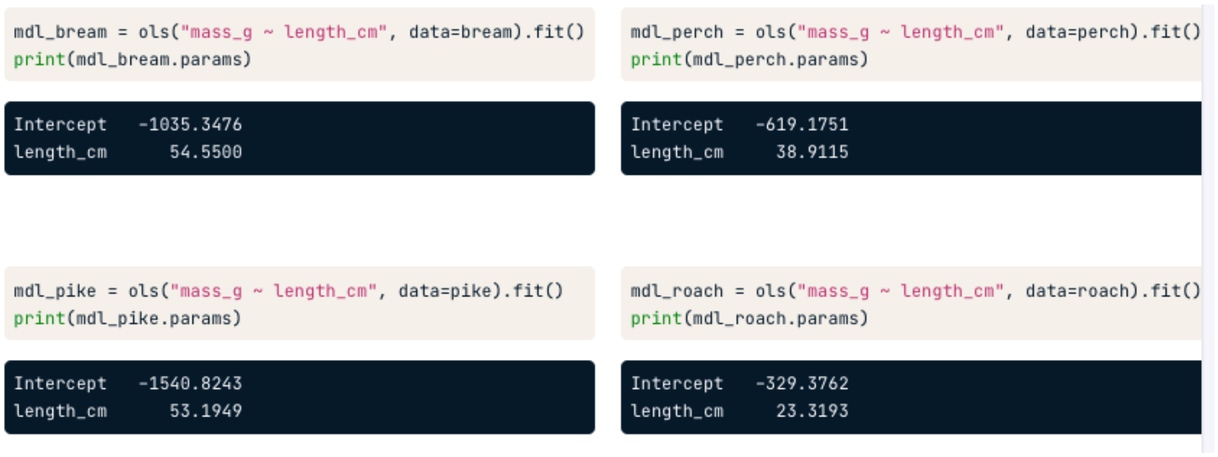
example
print(mdl_mass_vs_length.rsquared)
adding more explanatory variables often increases the coefficient of determination (R^2)
**however, just like everything else there is a 'sweet spot'
too many explanatory variables can cause overfitting
meaning that the model works well on 'this' dataset but poorly on 'that' dataset (ie the general population)
**adjusted coefficient of determination can help compensate this effect

$$\bar{R}^2 = 1 - (1 - R^2)\frac{n_{obs}-1}{n_{obs}-n_{var}-1}$$

the adjusted coefficient of determination penalizes more explanatory variables
noticeable with R^2 is small
or if nvar is a large fraction of nobs (number of explanatory variables is a sizable fraction of the number of observations)
get this metric in statsmodel with the rsquared_adj attribute

residual standard error (RSE) shows the typical size of the residuals
**a smaller number is better
RSE is not directly available as an attribute
get it this way:
rse = np.sqrt(model.mse_resid)
print('rse)length: ', rse_length)

Remember that parallel slopes enforces a common slope for each category
not always the best option
sometimes it is worth evaluating a different slope and to run separate models
example - again with the fish dataset
first split the dataset
bream = fish[fish['species'] == 'Bream']
perch = fish[fish['species'] == 'Perch']
pike = fish[fish['species'] == 'Pike']
roach = fish[fish['species'] == 'Roach']
now run four models
all prediciting mass based on length



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

```
Intercept   -1035.3476
length_cm      54.5500
```

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

```
Intercept   -619.1751
length_cm     38.9115
```

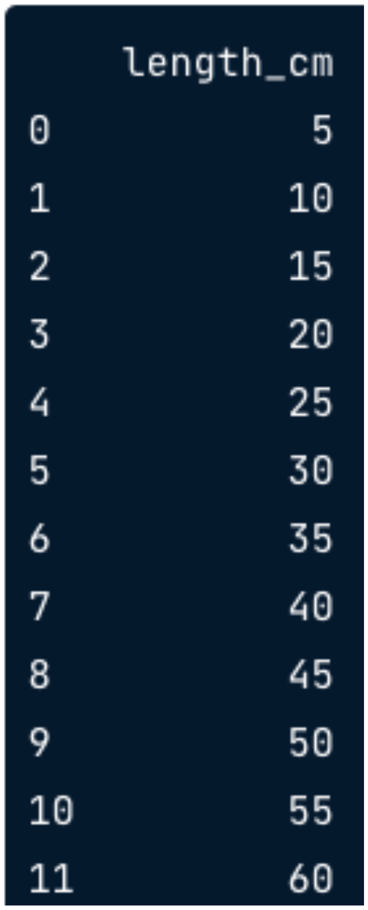
```
mdl_pike = ols("mass_g ~ length_cm", data=pike).fit()
print(mdl_pike.params)
```

```
Intercept   -1540.8243
length_cm      53.1949
```

```
mdl_roach = ols("mass_g ~ length_cm", data=roach).fit()
print(mdl_roach.params)
```

```
Intercept   -329.3762
length_cm     23.3193
```

this gives us four different intercepts and slopes
to make predictions on these models, we need to create a DataFrame of explanatory variables
**since the explanatory variables are the same for each model, we only need to make one DataFrame
explanatory_data = pd.DataFrame({'length_cm: np.arange(5, 61, 5)})
print(explanatory_data)



```
    length_cm
0           5
1          10
2          15
3          20
4          25
5          30
6          35
7          40
8          45
9          50
10         55
11         60
```

now make predictions
make a new DF using the old DF and the assign() method, adding a new column

and name it after the response variable
make four of these for our example which has four models (separate model coefficients for each species)

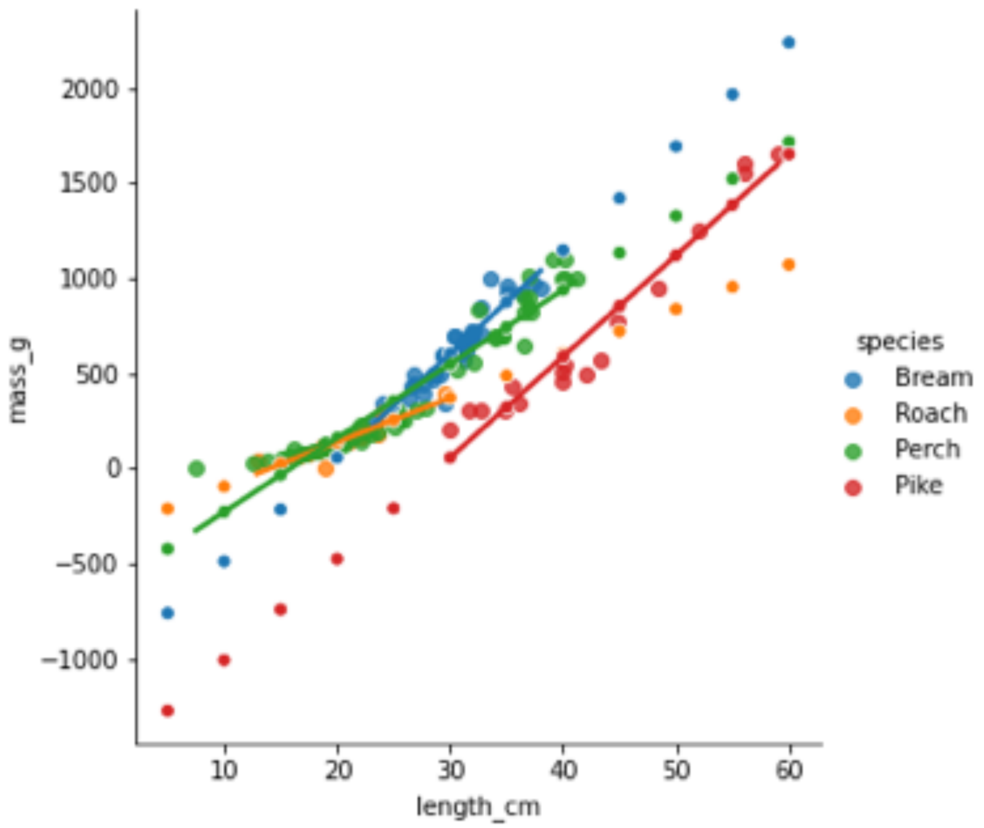
```
prediction_data_bream = explanatory_data.assign(
    mass_g = mdl_bream.predict(explanatory_data),
    species = "Bream")
```

```
prediction_data_perch = explanatory_data.assign(
    mass_g = mdl_perch.predict(explanatory_data),
    species = "Perch")
```

```
prediction_data_pike = explanatory_data.assign(
    mass_g = mdl_pike.predict(explanatory_data),
    species = "Pike")
```

```
prediction_data_roach = explanatory_data.assign(
    mass_g = mdl_roach.predict(explanatory_data),
    species = "Roach")
```

**you don't have to include 'species = ' '' in above code but it does help with plotting
rather than work with all DFs separately, concat them for convenience
predcition_data = pd.concat([prediction_data_bream, prediction_data_roach, prediction_data_perch, prediction_data_pike])
time to visualize
**you can not use regplot to visualize regression models across subsets of a dataset
use lmplot instead
sns.lmplot(x='length_cm', y='mass_g', data=fish, hue='species', ci=None)
x=explanatory variable, y=response variable, dataset, use hue to subset said data
here each species has its own line an slope
also sanity check our concatenated predictions and see if they align with our lmplot
sns.scatterplot(x='length_cm', y='mass_g', data=prediction_data, hue='species', ci=None, legend=False)
plt.show()



**as predicted, each line of prediction points follows seaborn's trend lines
Remember separating out models isn't always better than running a single model on the whole dataset
always compare coefficient of determination
mdl_fish = ols('mass_g ~ length_cm + species', data=fish).fit()
print(mdl_fish.rsquared_adj)
output > 0.917
compare this against

```
print(mdl_bream.rsquared_adj)
```

```
0.874
```

```
print(mdl_perch.rsquared_adj)
```

```
0.917
```

```
print(mdl_pike.rsquared_adj)
```

```
0.941
```

```
print(mdl_roach.rsquared_adj)
```

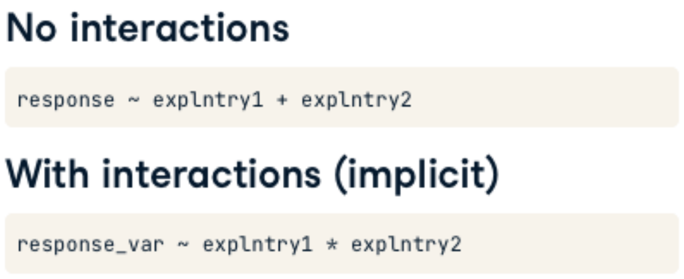
```
0.815
```

a mixed bag which is often the case
do the same for residual standard error
 np.sqrt(model.mse_resid))

One model with an interaction
not ideal to be using different models for different bits of your dataset
better to specify a single model that contains intercepts and slopes ofr each category
this can be achieved through specifying 'interactions' between explanatory variables

**this is key, these 'interactions' depend on the dataset
they require knowledge and expertise of the dataset
**the effect of one explanatory variable on the expected response changes depending on the value of another explanatory variable, then those two explanatory variables interact
example - fish dataset
fish have different mass to length ratios
effect of length on the expected mass is different for different species

How do we specify this?

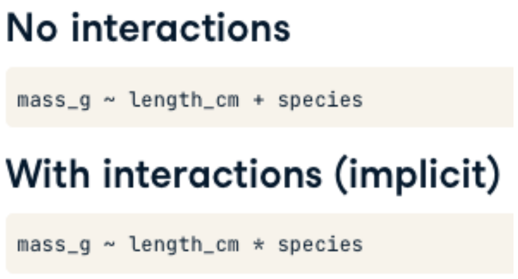


No interactions

```
response ~ explntry1 + explntry2
```

With interactions (implicit)

```
response_var ~ explntry1 * explntry2
```



No interactions

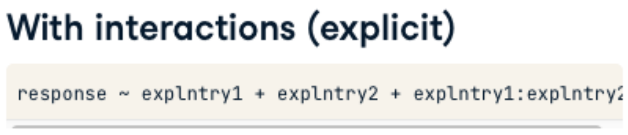
```
mass_g ~ length_cm + species
```

With interactions (implicit)

```
mass_g ~ length_cm * species
```

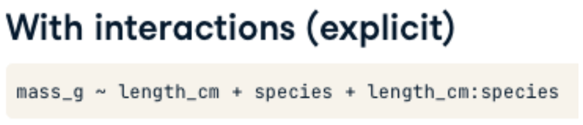
*to include an interaction between the variables, just swap the + for a *
statsmodels figures out what interactions are needed on its own

**however to make your output easier to read it is better to explicitly document which interactions are included in the model
how to do this?
example
mdl_mass_vs_both_inter = ols('mass_g ~ species + species:length_cm + 0', data=fish).fit()



With interactions (explicit)

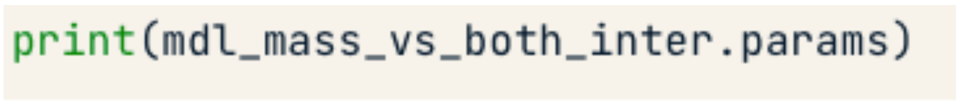
```
response ~ explntry1 + explntry2 + explntry1:explntry
```



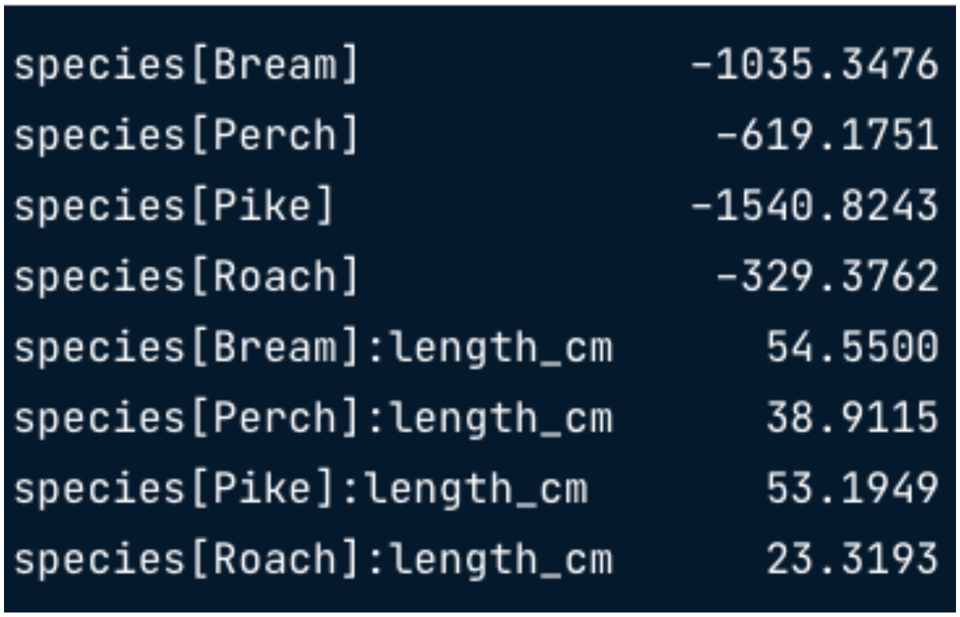
With interactions (explicit)

```
mass_g ~ length_cm + species + length_cm:species
```

remember the zero removes the global intercept
we now get this:



```
print(mdl_mass_vs_both_inter.params)
```



```
species[Bream]                -1035.3476
species[Perch]                 -619.1751
species[Pike]                 -1540.8243
species[Roach]                 -329.3762
species[Bream]:length_cm         54.5500
species[Perch]:length_cm         38.9115
species[Pike]:length_cm          53.1949
species[Roach]:length_cm         23.3193
```

**this gives us what we got from above above when we made four separate models but nice thing here is you get it in one set of code vs four

The code for the prediction flow is just as it was above

```
from itertools import product
length_cm = np.arange(5, 61, 5)
species = fish[‘species’].unique()
p = product(length_cm, species)
#remember that ‘product’ from itertools gives up Cartesian product which gets us all the combinations of selected variables
explanatory_data = pd.DataFrame(p, columns=[‘length_cm’, ‘species’])
prediction_data = explanatory_data.assign(mass_g = mdl_mass_vs-both_inter.predict(explanatory_data))
```

visualize as above above

then manually calculate the predictions

```
coeffs = mdl_mass_vs_both_inter.params
```

unpack the coefficients into four intercepts and four slopes

```
ic_bream, ic_perch, ic_pike, ic_roach, slope_bream, slope_perch, slope_pike, slope_roach = coeffs
```

now back to our friend np.select

```
conditions = [explanatory_data[‘species’] == ‘Bream’, explanatory_data[‘species’] == ‘Perch’, explanatory_data[‘species’] == ‘Pike’, explanatory_data[‘species’] == ‘Roach’]
ic_choices = [ic_bream, ic_perch, ic_pike, ic_roach]
intercept = np.select(conditions, ic_choices)
slope_choices = [slope_bream, slope_perch, slope_pike, slope_roach]
slope = np.select(condtions, slope_choices)
```

np.select reads ‘if species is Bream pick Bream intercept’ and so on

```
prediction_data = explanatory_data.assign(mass_g = intercept + slope * explanatory_data[‘length_cm’]
print(prediction_data)
```

Simpson’s Paradox

occurs when the trend of a model on the whole dataset is very different from the trends shown by models on subsets of the dataset

trend = slope coefficient

example

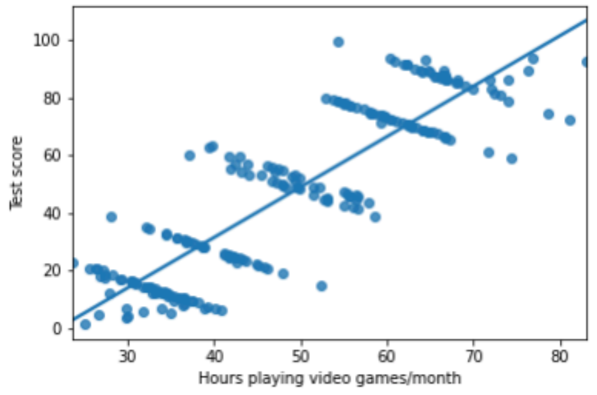
model of the whole dataset is positive slope but a model for each group shows a negative slope

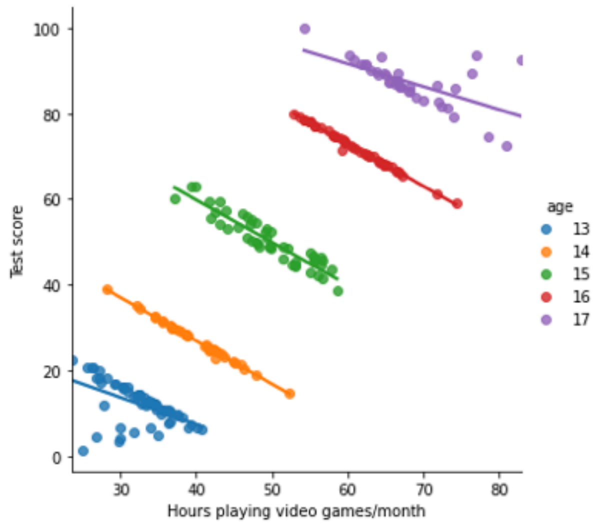
How to reconcile this difference?

if possible, try to plot the dataset

articulate a question before you start modeling

visual example:

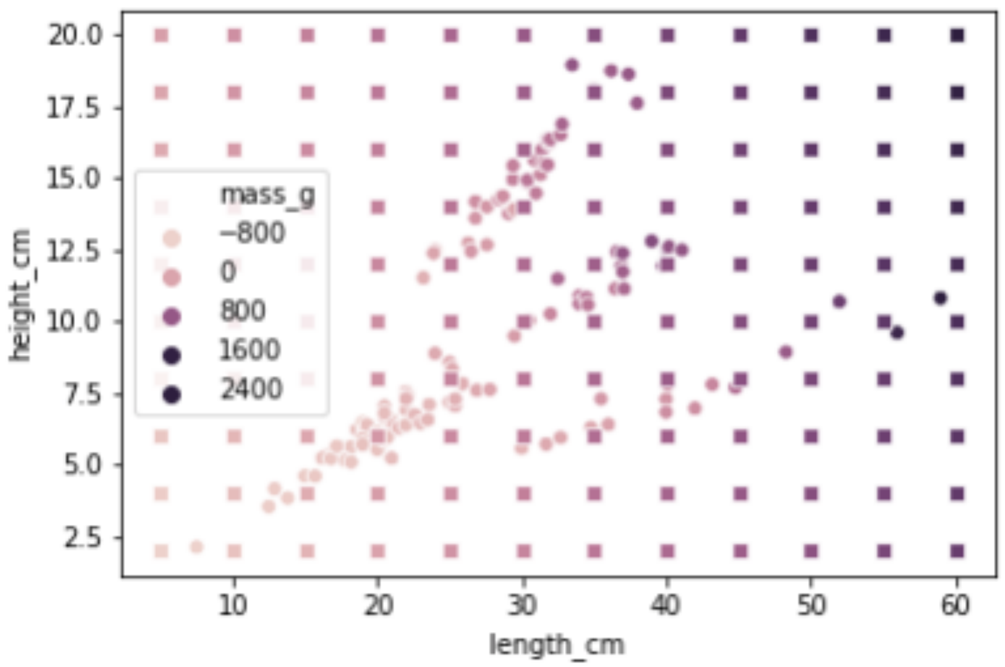




the first graph suggests that playing more games increases test score
in the second we reveal that each grouping is an age group
this changes the interpretation
showing that older children in general score higher
and playing lots of games actually is related to lower scores

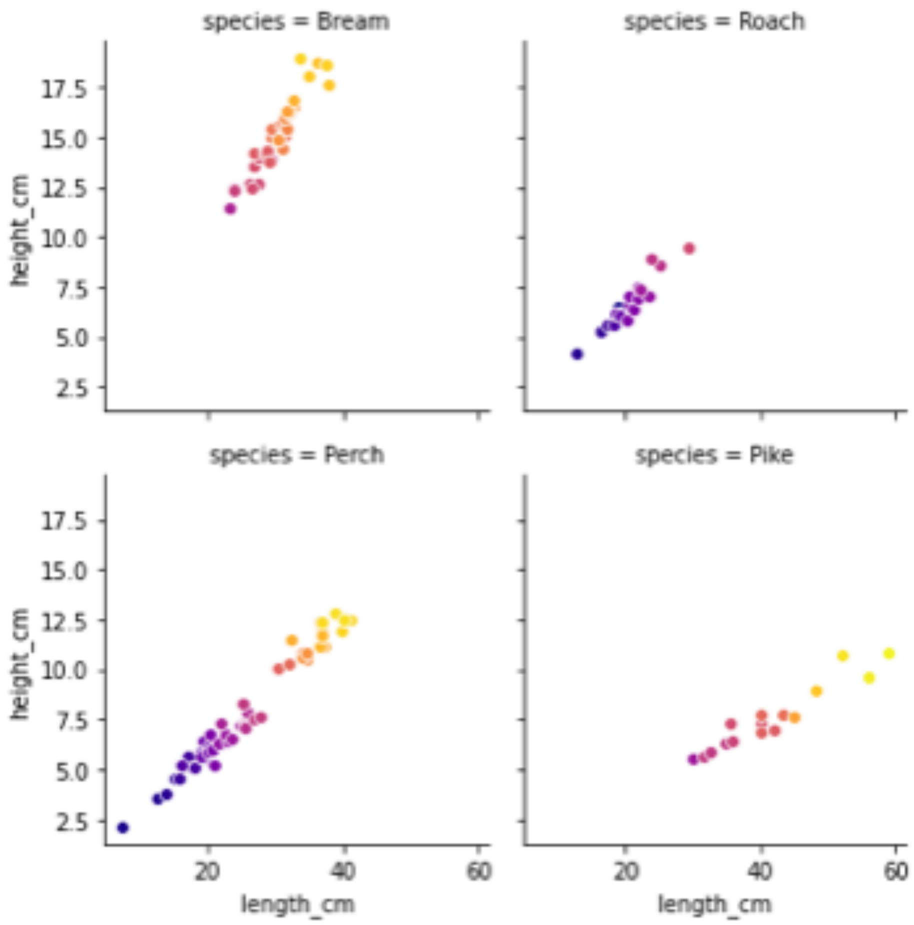
Again reconciling the difference
resolving this is messy
often, the models including the goups will contain insight that you'd miss other wise
*disagreements may reveal tha you need even more explanatory variables
*context is important

Two numeric explanatory variables
*this means visualizing three numeric variables (2 exp, 1 response)
3D scatter plot or 2D scatter plot with response as color
3D scatter plot needs to be interactive so that the audience can rotate the data to explore from different angles
2D with response as color is better
use the same modeling and prediction flow as seen above above
plotting is slightly different
example
sns.scatterplot(x='length_cm', y='height_cm', data=fish, hue='mass_g')
*add second scatter plot
sns.scatterplot(x='length_cm', y='height_cm', data=prediction_data, hue='mass_g', legend=False, marker='s')
*to avoid duplication remove the legend of one of the scatter plots
and change the markers
ouput >



color grid gives a nice overview of how the response variable changes over the plane of the explanatory variables

What if there is even more than two explanatory variables?
can do this by 'faceting', ie giving each group its own panel
example with fish dataset
grid = sns.FacetGrid(data=fish, col='species', hue='mass_g', col_wrap=2, palette='plasma')
grid.map(sns.scatterplot, 'length_cm', 'height_cm')



it becomes tricky to include more than three numeric variables in a scatter plot
*you can include as many categorical variables as you like using faceting
*visualizing and plotting become harder with increasing number of variables

Modeling with more than two explanatory variables

No interactions

```
ols("mass_g ~ length_cm + height_cm + species + 0", data=fish).fit()
```

two-way interactions between pairs of variables

```
ols(
  "mass_g ~ length_cm + height_cm + species +
  length_cm:height_cm + length_cm:species + height_cm:species + 0", data=fish).fit()
```

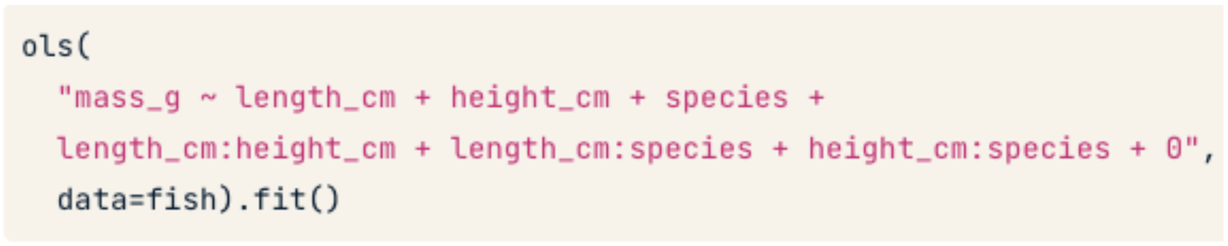
three-way interaction between all three variables

```
ols(
  "mass_g ~ length_cm + height_cm + species +
  length_cm:height_cm + length_cm:species + height_cm:species + length_cm:height_cm:species + 0", data=fish).fit()
```

simpler syntax

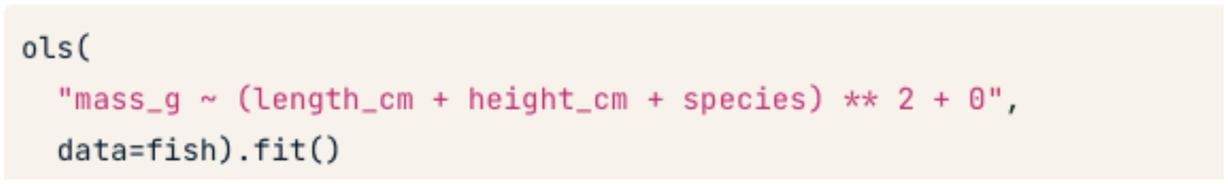
ols('mass_g ~ length_cm * height_cm * species + 0', data=fish).fit()

Only two-way interaction:



```
ols(
  "mass_g ~ length_cm + height_cm + species +
  length_cm:height_cm + length_cm:species + height_cm:species + 0",
  data=fish).fit()
```

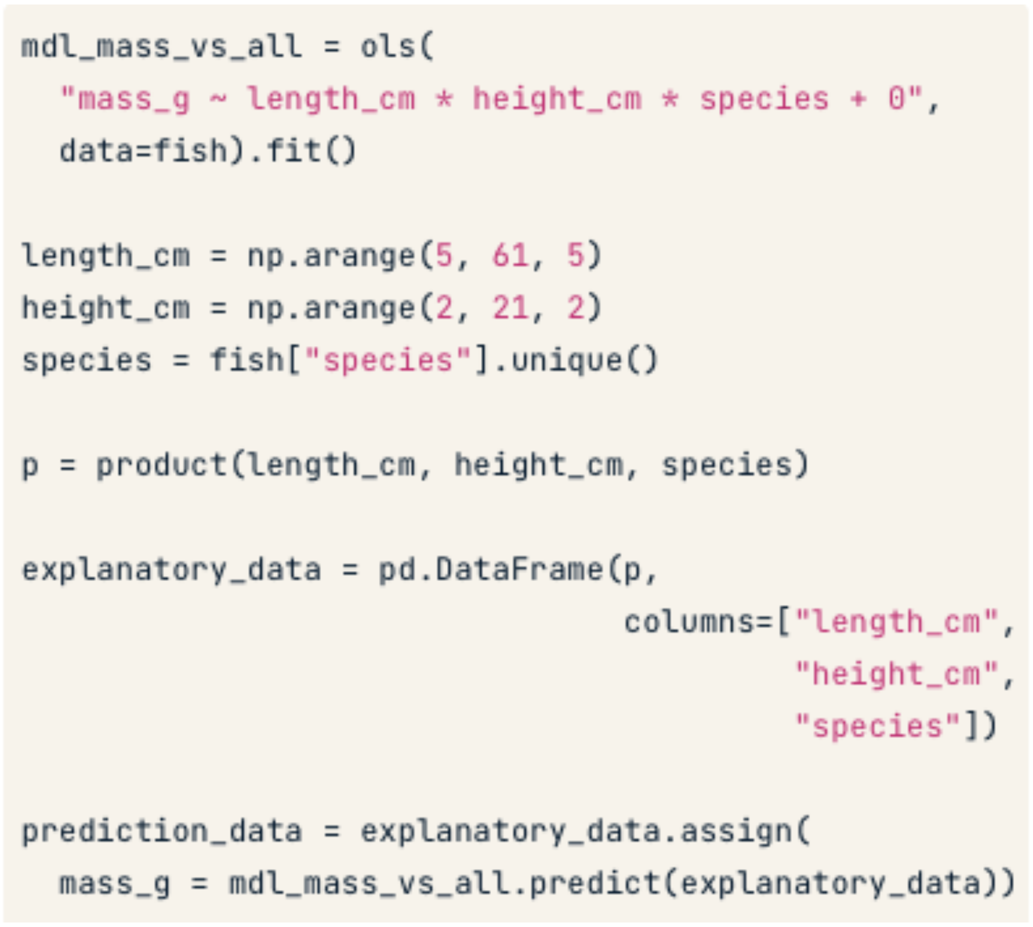
same as



```
ols(
  "mass_g ~ (length_cm + height_cm + species) ** 2 + 0",
  data=fish).fit()
```

for 2-way interaction but not 3-way interaction

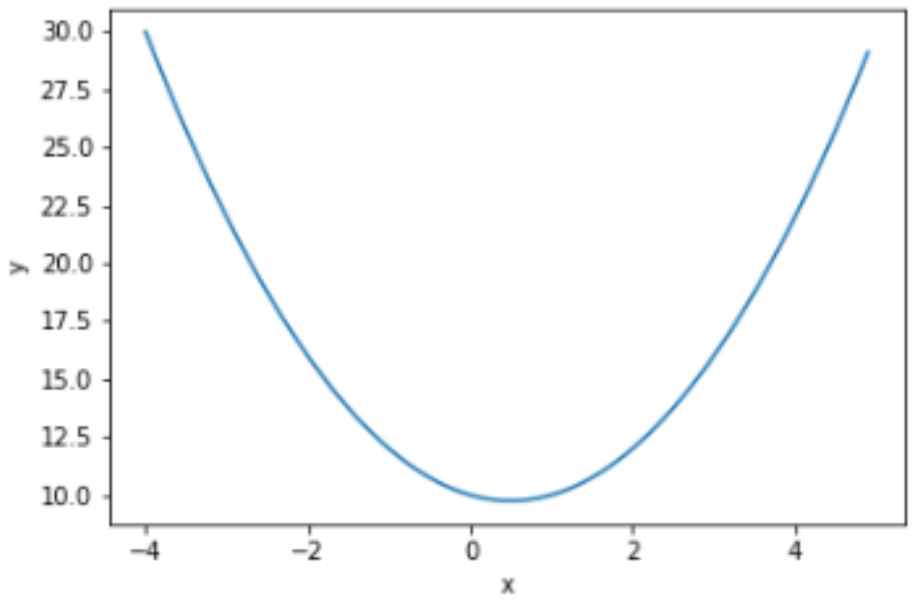
Prediction flow as expected



```
mdl_mass_vs_all = ols(
  "mass_g ~ length_cm * height_cm * species + 0",
  data=fish).fit()

length_cm = np.arange(5, 61, 5)
height_cm = np.arange(2, 21, 2)
species = fish["species"].unique()

p = product(length_cm, height_cm, species)

explanatory_data = pd.DataFrame(p,
                                columns=["length_cm",
                                         "height_cm",
                                         "species"])

prediction_data = explanatory_data.assign(
  mass_g = mdl_mass_vs_all.predict(explanatory_data))
```

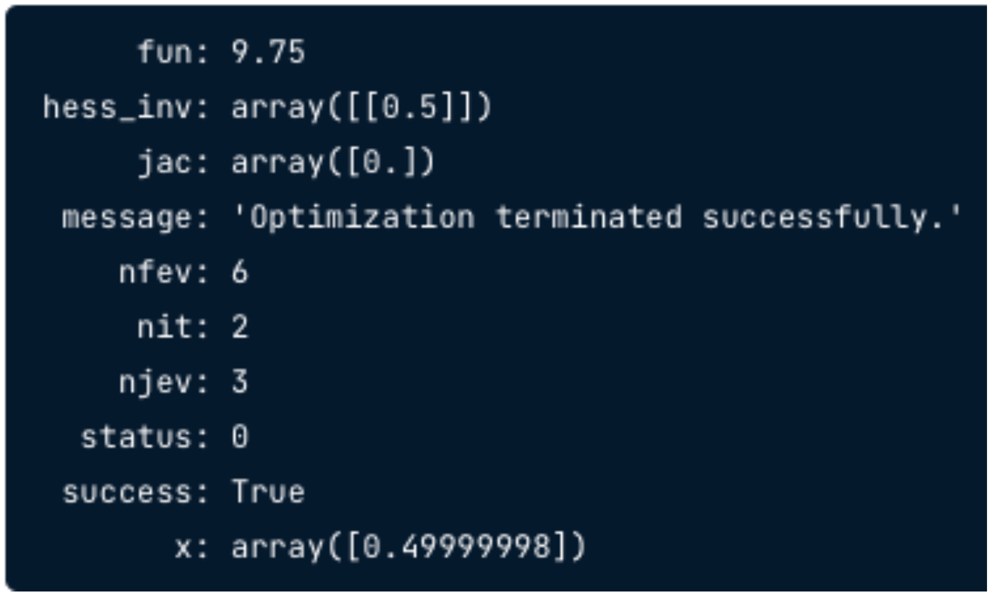
*modeling scales nicely with more variables
not the rapid increase in dimensionality
at this point visualizing is beyond the limit of visual interpretation

How linear regression works
goal is to find the best fit
residual is actual response minus predicted response
want those residuals to be as short as possible
sum of squares metric measures all of these residuals and squares them to avoid negatives
the real question:
how to determine which intercept and slope coefficients will result in the smallest sum of squares?
how to solve this?
we need to take a detour into numerical optimization
which means finding the minimum point of a function
consider this quadratic equation
x = np.arange(-4, 5, 0.1)
y = x ** 2 - x + 10
xy_data = pd.DataFrame({‘x’: x, ‘y;: y})
sns.lineplot(x=‘x’, y=‘y’, data=xy_data)



plot shows that the minimum point of the function occurs when x is a little above 0 and y is a little above 10
but how can we find it exactly?
can use calculus
y = x^2 - x + 10
derivative y / derivative x = 2x - 1

set derivative to 0
0 = 2x - 1
x = 0.5
y = 0.5^2 - 0.5 + 10 = 9.75
*however not all equations can be solved like this
Python also makes this easier for us with the minimize() function
performs numerical optimization in Python
minimize() takes x as an input and returns y as x-squared minus x + 10
first argument is the function to call without parentheses
second argument is an initial guess at the answer (*sometimes important for more complicated functions, but here you could pick anything)
from scipy.optimize import minimize
def calc_quadratic(x):
    y = x ** 2 - x + 10
    return y
minimize(fun=calc_quadratic, x0=3)
output (compare to above):



```
      fun: 9.75
 hess_inv: array([[0.5]])
      jac: array([0.])
  message: 'Optimization terminated successfully.'
     nfev: 6
      nit: 2
     njev: 3
   status: 0
  success: True
        x: array([0.49999998])
```

Linear regression algorithm for simple linear regression
define a function to calculate the sum of squares metric
def calc_sum_of_squares(coeffs):
    intercept, slope = coeffs
…skipping some calculations (not explained in tutorial)
call minimize() to find coefficients that minimize this function
minimize(fun=calc_sum_of_squares, x0=0)

example
# Complete the function

```
def calc_sum_of_squares(coeffs):
    # Unpack coeffs
    intercept, slope = coeffs
    # Calculate predicted y-values
    y_pred = intercept + slope * x_actual
    # Calculate differences between y_actual and y_pred
    y_diff = y_pred - y_actual
    # Calculate sum of squares
    sum_sq = np.sum(y_diff ** 2)
    # Return sum of squares
    return sum_sq

# Call minimize on calc_sum_of_squares
print(minimize(fun=calc_sum_of_squares,
               x0=[0, 0]))

# Compare the output with the ols() call.
print(ols("price_twd_msq ~ n_convenience", data=taiwan_real_estate).fit().params)

output > x: array([8.22423741, 0.79807971])
    Intercept        8.224
    n_convenience    0.798
```

**pretty darn comparable

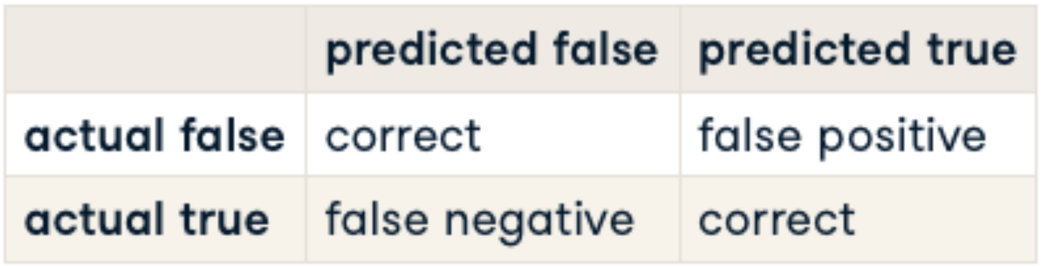
Multiple logisitic regression
no new syntax here
same as for linear regression
execept with logit() instead of ols()

```
from statsmodels.formula.api import logit

logit("response ~ explanatory", data=dataset).fit()
```

```
logit("response ~ explanatory1 + explanatory2", data=dataset).fit()
```

```
logit("response ~ explanatory1 * explanatory2", data=dataset).fit()
```

Recall that when the response variable has two possible values, there are four outcomes for the model



	predicted false	predicted true
actual false	correct	false positive
actual true	false negative	correct

quantify and visualize using a confusion matrix
conf_matrix = mdl_logit.pred_table()
remember the confusion matrix lets you calculate accuracy, sensitivity, and specificity

Prediction flow (also the same as above)

```
from itertools import product
explanatory1 = some_values
explanatory2 = some_values
p = product(explanatory1, explanatory2)
explanatory_data = pd.DataFrame(p, columns=[‘explanatory1’, ‘explanatory2’])
prediction_data = explanatory_data.assign(mass_g =
mdl_logit.predict(explanatory_data))
#for visualization purposes also creat a column with most likely outcomes
prediction_data[‘most_likely_outcome’] = np.
round(predicition_data[‘has_churned’])
#then two scatter plots, one for data and one for prediction data
```

Example

```
# Create conf_matrix
conf_matrix = mdl_churn_vs_both_inter.pred_table()

# Extract TN, TP, FN and FP from conf_matrix
TN = conf_matrix[0,0]
TP = conf_matrix[1,1]
FN = conf_matrix[1,0]
FP = conf_matrix[0,1]

# Calculate and print the accuracy
accuracy = (TP + TN) / (TP + FP + FN + TN)
print("accuracy", accuracy)

# Calculate and print the sensitivity
sensitivity = TP / (TP + FN)
print("sensitivity", sensitivity)
```

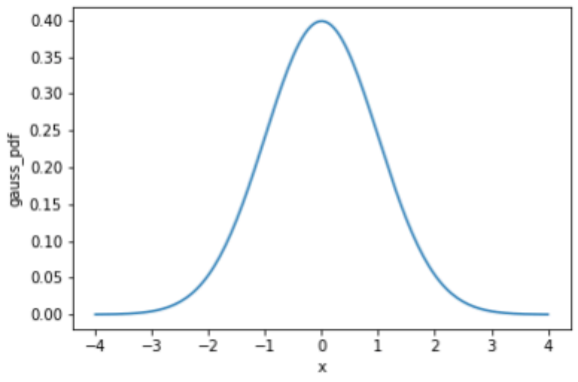
# Calculate and print the specificity
specificity = TN / (TN + FP)
print("specificity", specificity)

Logisitic distribution
just a reminder on normal distribution

Gaussian probability density function (PDF)

```
from scipy.stats import norm

x = np.arange(-4, 4.05, 0.05)

gauss_dist = pd.DataFrame({
  "x": x,
  "gauss_pdf": norm.pdf(x)}
)
```

```
sns.lineplot(x="x",
             y="gauss_pdf",
             data=gauss_dist)
```

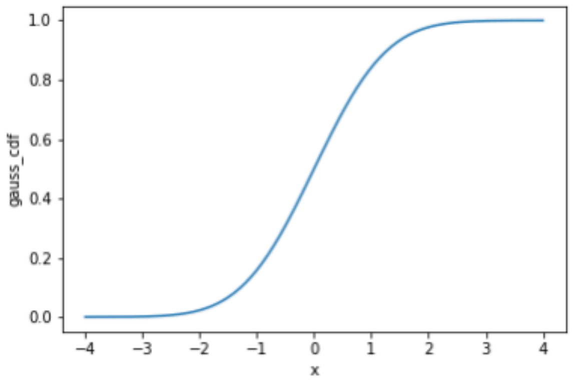


for the purposes of regression we care more about the area under this curve
by integrating the norm.pdf function (ie calculating the area underneath) we get another curve, the cummulative distribution curve

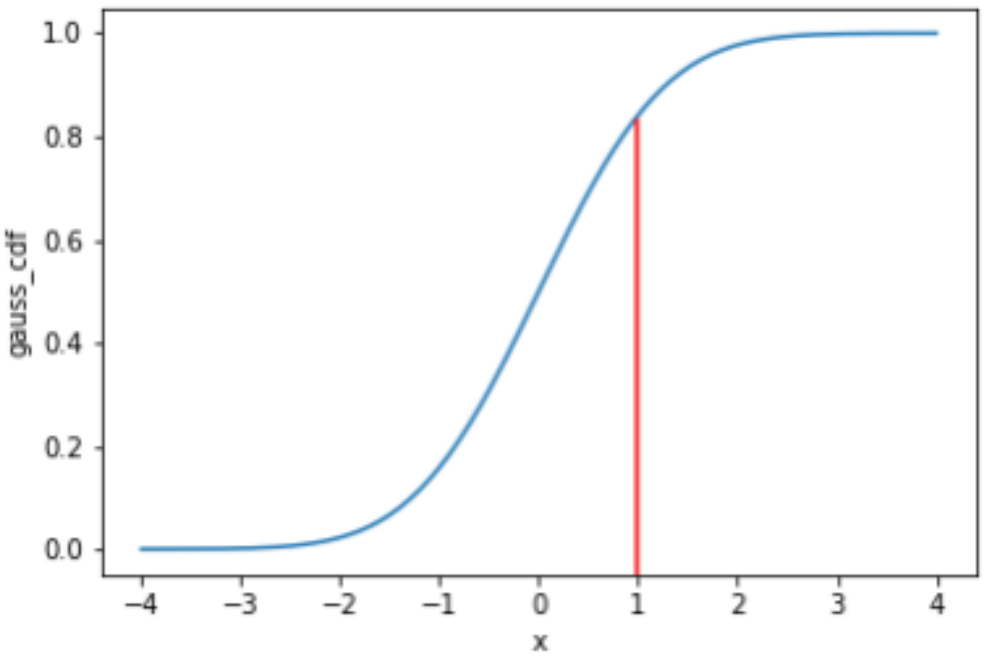
Gaussian cumulative distribution function (CDF)

```
x = np.arange(-4, 4.05, 0.05)

gauss_dist = pd.DataFrame({
  "x": x,
   "gauss_pdf": norm.pdf(x),
   "gauss_cdf": norm.cdf(x)}
)
```

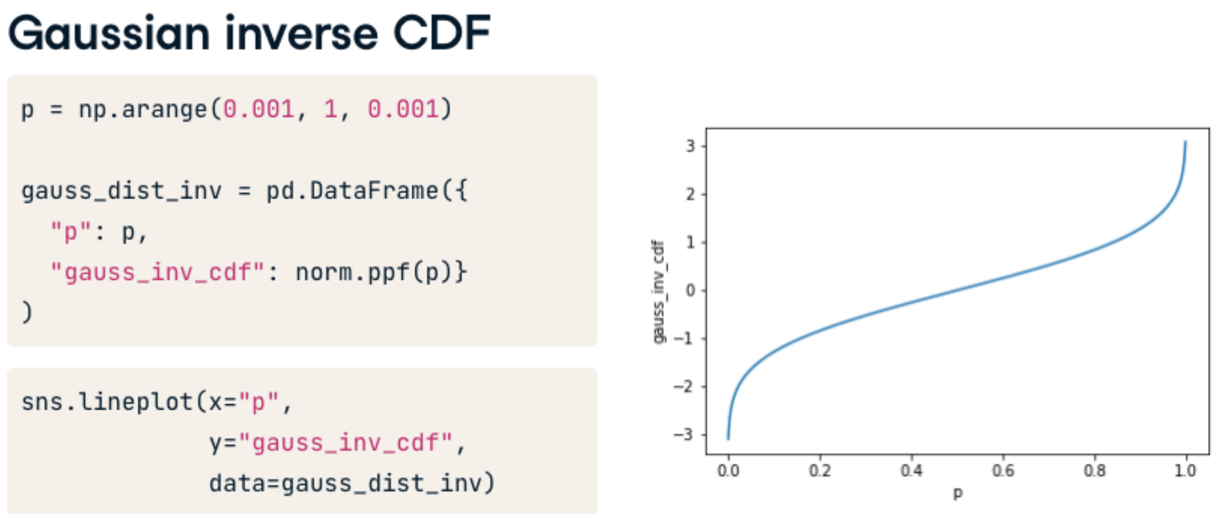
```
sns.lineplot(x="x",
             y="gauss_cdf",
             data=gauss_dist)
```



CDF curve for all distributions when x has its minimum possible value y will be 0
when x has its maximum possible value y will be 1
**think of the CDF as a transformation from the values of x to probabilities



what this tells us is when x is 1, the CDF curve is at 0.84
tells us for a normally distributed variable x, the probability that x is less the 1 is 84%



Gaussian inverse CDF

```
p = np.arange(0.001, 1, 0.001)

gauss_dist_inv = pd.DataFrame({
  "p": p,
  "gauss_inv_cdf": norm.ppf(p)}
)
```

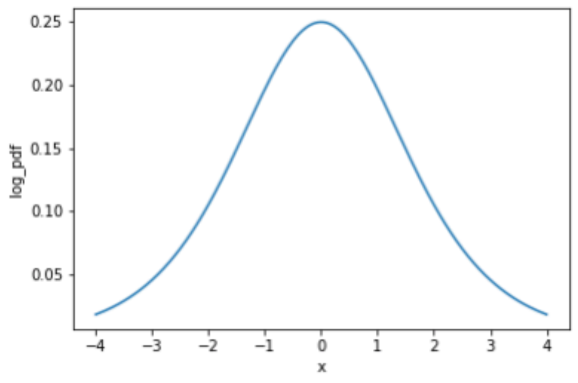
```
sns.lineplot(x="p",
             y="gauss_inv_cdf",
             data=gauss_dist_inv)
```

**need a way to go back from probabilities to x-values
inverse CDF is also known as percent point function or PPF or quantile function
**calculated from norm.ppf
**same curve but with x and y axes flipped

Logistic PDF

```python
from scipy.stats import logistic

x = np.arange(-4, 4.05, 0.05)

logistic_dist = pd.DataFrame({
  "x": x,
  "log_pdf": logistic.pdf(x)}
)
```

```python
sns.lineplot(x="x",
             y="log_pdf",
             data=logistic_dist)
```



Logisitic PDF similar to the Gaussian PDF but tails at the extreme left and right of the plot are fatter

Logistic distribution
logistic distribution CDF is also called the logistic function

$$\mathrm{cdf}(x) = \frac{1}{(1+exp(-x))}$$

the inverse CDF is also called the logit function
*logit is also known as the log odds ratio for describing predictions

$$\mathrm{inverse\_cdf}(p) = log(\frac{p}{(1-p)})$$

# Import logistic
from scipy.stats import logistic

# Create x ranging from minus ten to ten in steps of 0.1
x = np.arange(-10, 10.1, 0.1)

# Create logistic_dist
logistic_dist = pd.DataFrame({"x": x,
                              "log_cdf": logistic.cdf(x),
                              "log_cdf_man": 1 / (1 + np.exp(-x))})

# Using logistic_dist, plot log_cdf vs. x
sns.lineplot(x='x', y='log_cdf', data=logistic_dist)

```
# Show the plot
plt.show()

# Create p ranging from 0.001 to 0.999 in steps of 0.001
p = np.arange(0.001, 1, 0.001)

# Create logistic_dist_inv
logistic_dist_inv = pd.DataFrame({"p": p,
                                  "logit": logistic.ppf(p),
                                  "logit_man": np.log(p / (1 - p))})

# Using logistic_dist_inv, plot logit vs. p
sns.lineplot(x='p', y='logit', data=logistic_dist_inv)

# Show the plot
plt.show()
```

How logistic regression works
sum of squares does not work
y_actual is always 0 or 1
y_pred is always between 0 or 1
here we use the metric ‘Likelihood’

‘Likelihood’ unlike sum of squares where the goal is to find the minimum possible value
with likelihood the goal is to find the maximum possible value
take the product of the predicted and actual responese
np.sum(y_pred * y_actual + (1 - y_pred) * (1 - y_actual))
since y_actual only has two possible values, you get either:
y_actual = 1
y_pred * 1 + (1 - y_pred) * (1 - 1) = y_pred
how this works?
as y_pred increases, the metric increases too, and the maximum likelihood occurs when y_pred is one, the same as the actual value
or
y_actual = 0
y_pred * 0 + (1 - y_pred) * (1 - 0) = 1 - y_pred
how this works?
as y_pred decreases, the metric decreases too, and the maximum likelihood occurs when y_pred is zero, the same as the actual value
**key is in either case, you get a higher likelihood score when the predicted response is close to the actual response

Log-likelihood
computing likelihood involves adding many very small numbers, leading to numerical error
log-likelihood is easier to compute
log_likelihood = np.log(y_pred) * y_actual + np.log(1 - y_pred) * (1 - y_actual)
**both equations give the same answer

Negative log-likelihood
maximizing log-likelihood is the same as minimizing negative log-likelihood
why do this?
the optimize package can only minimize functions
this tweak lets us get around that
-np.sum(log_likelihoods)

Nice example
```
# Complete the function
def calc_neg_log_likelihood(coeffs):
   # Unpack coeffs
   intercept, slope = coeffs
   # Calculate predicted y-values
   y_pred = logistic.cdf(intercept + slope * x_actual)
   # Calculate log-likelihood
   log_likelihood = np.log(y_pred) * y_actual + np.log(1 - y_pred) * (1 - y_actual)
   # Calculate negative sum of log_likelihood
   neg_sum_ll = -np.sum(log_likelihood)
   # Return negative sum of log_likelihood
   return neg_sum_ll

# Call minimize on calc_sum_of_squares
print(minimize(fun=calc_neg_log_likelihood,
               x0=[0,0]))

# Compare the output with the logit() call.
print(logit("has_churned ~ time_since_last_purchase", data=churn).fit().params)
```