lab09

November 14, 2024

[1]:
```
# Initialize Otter
import otter
grader = otter.Notebook("lab09.ipynb")
```

1 Lab 9: Regression

Welcome to Lab 9!

Today we will get some hands-on practice with linear regression. You can find more information about this topic in Chapter 15.2.

[2]:
```
# Run this cell, but please don't change it.

# These lines import the Numpy and Datascience modules.
import numpy as np
from datascience import *

# These lines do some fancy plotting magic.
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plots
plots.style.use('fivethirtyeight')
import warnings
warnings.simplefilter('ignore', FutureWarning)
```

2 Lab Warm Up!

We will work together as a class in the following coding cells to prepare you for all sections of this lab.

Make sure to come to lab on time so you don't miss points for this warm-up!

1. Review the steps of linear regression in your own words

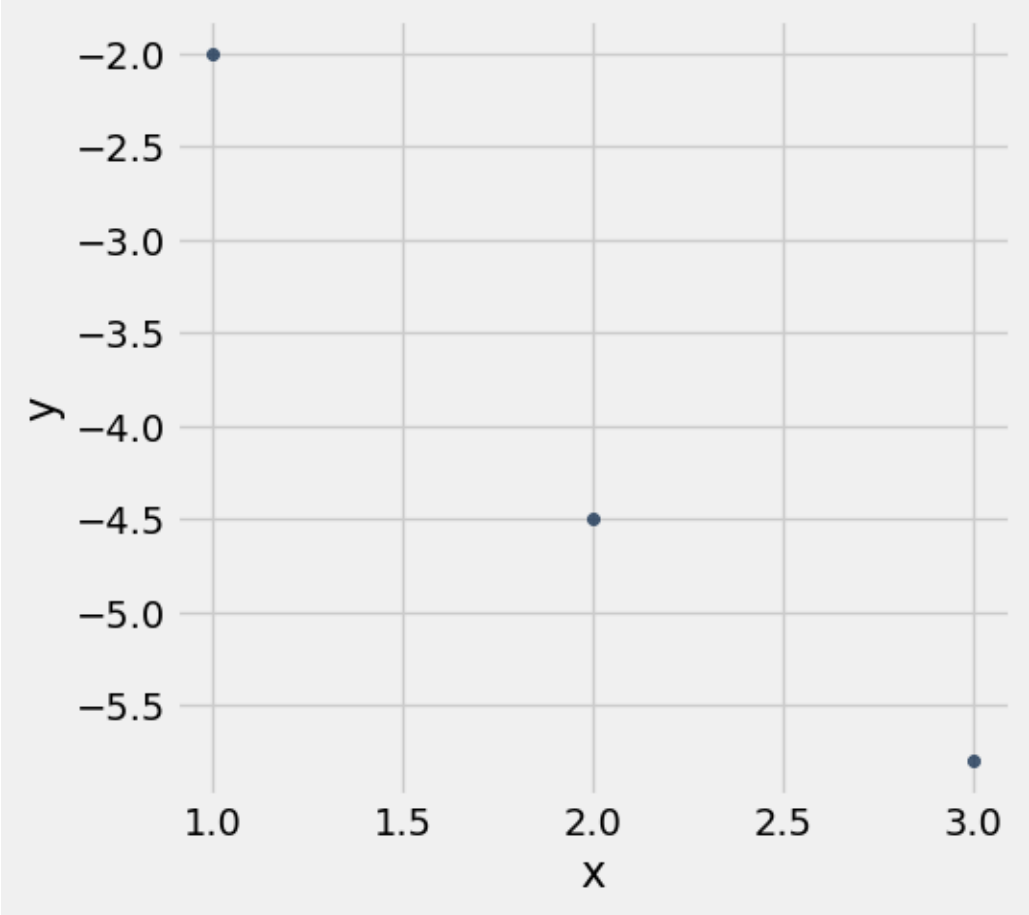
Add lab discussion notes here

[3]:
```
# Consider the following table
my_data = Table().with_columns('x', make_array(1,2,3), 'y', make_array(-2.0,-4.
↪5,-5.8))
```

```
my_data

# X is inc and y is decreasing so negatively correlated
```

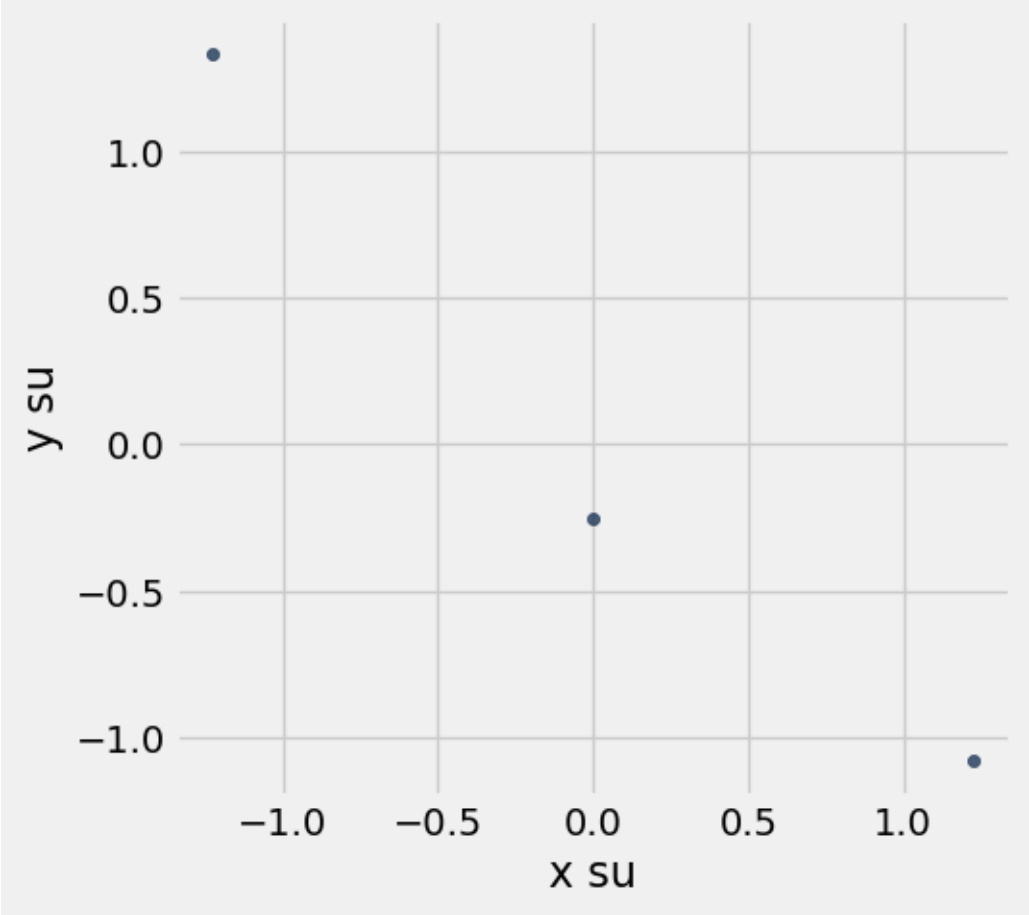
[3]:
```
x    | y
1    | -2
2    | -4.5
3    | -5.8
```

[4]:
```
my_data.scatter('x', 'y')
```



[5]:
```
# Calculating regression parameters
# step 1: standardize the units
x_mean = np.average(my_data.column('x'))
x_std = np.std(my_data.column('x'))
x_standard_units= (my_data.column('x') - x_mean) / x_std
y_mean = np.average(my_data.column('y'))
y_std = np.std(my_data.column('y'))
```

```
y_standard_units = (my_data.column('y') - y_mean) / y_std
su_table = Table().with_columns('x su', x_standard_units, 'y su',␣
 ↪y_standard_units)
su_table.scatter('x su', 'y su')
```



[6]:
```
# Step 2: compute the correlation coefficient, r
r = np.average(x_standard_units * y_standard_units)
r
```

[6]: -0.98378270884914565

[8]:
```
# Step 3: compute the slope and the intercept using r
slope = r * y_std / x_std
# keep in rise over run order for slope
intercept = y_mean - slope * x_mean
```

[9]:
```
# Making a prediction
# y = slope * new_x + intercept aka y = mx + b
```

```
new_x  = 2.5 #random guess not legit
# look at graphs at 2.5
prediction  = slope * new_x + intercept
prediction
# line tells us what we can predict
```

[9]:
```
-5.0500000000000007
```

[ ]:
```

```

3 1. How Faithful is Old Faithful?

Old Faithful is a geyser in Yellowstone National Park that is famous for eruption on a fairly regular schedule. Run the cell below to see Old Faithful in action!

[10]:
```
# For the curious: this is how to display a YouTube video in a
# Jupyter notebook.  The argument to YouTubeVideo is the part
# of the URL (called a "query parameter") that identifies the
# video.  For example, the full URL for this video is:
#   https://www.youtube.com/watch?v=wE8NDuzt8eg
from IPython.display import YouTubeVideo
YouTubeVideo("wE8NDuzt8eg")
```

[10]:

Some of Old Faithful's eruptions last longer than others. Whenever there is a long eruption, it is usually followed by an even longer wait before the next eruption. If you visit Yellowstone, you might want to predict when the next eruption will happen, so that you can see the rest of the park instead of waiting by the geyser.

Today, we will use a dataset on eruption durations and waiting times to see if we can make such predictions accurately with linear regression.

The dataset has one row for each observed eruption. It includes the following columns: - `duration`: Eruption duration, in minutes - `wait`: Time between this eruption and the next, also in minutes

Run the next cell to load the dataset.

[11]:
```
faithful = Table.read_table("faithful.csv")
faithful
```

[11]:
```
duration | wait
3.6      | 79
1.8      | 54
3.333    | 74
2.283    | 62
4.533    | 85
2.883    | 55
4.7      | 88
3.6      | 85
1.95     | 51
4.35     | 85
… (262 rows omitted)
```

Question 1.0. The following statements are the unordered steps of linear regression.

1. Compute the parameters of the regression line: the slope and the intercept.
2. Evaluate the regression line by computing the line's RMSE and analyzing the residuals plot.
3. Use the regression line to generate predictions for each x value.
4. Determine if linear regression is a reasonable method by visualizing your data and computing the correlation coefficient.

Make an array called `least_squares_order` that contains the correct order of a linear regression analysis, where the first item of the array is the first step of an linear regression analysis and the last item of the array is the last step of an linear regression analysis.

[12]:
```
least_squares_order = make_array(4, 1, 3, 2)
```

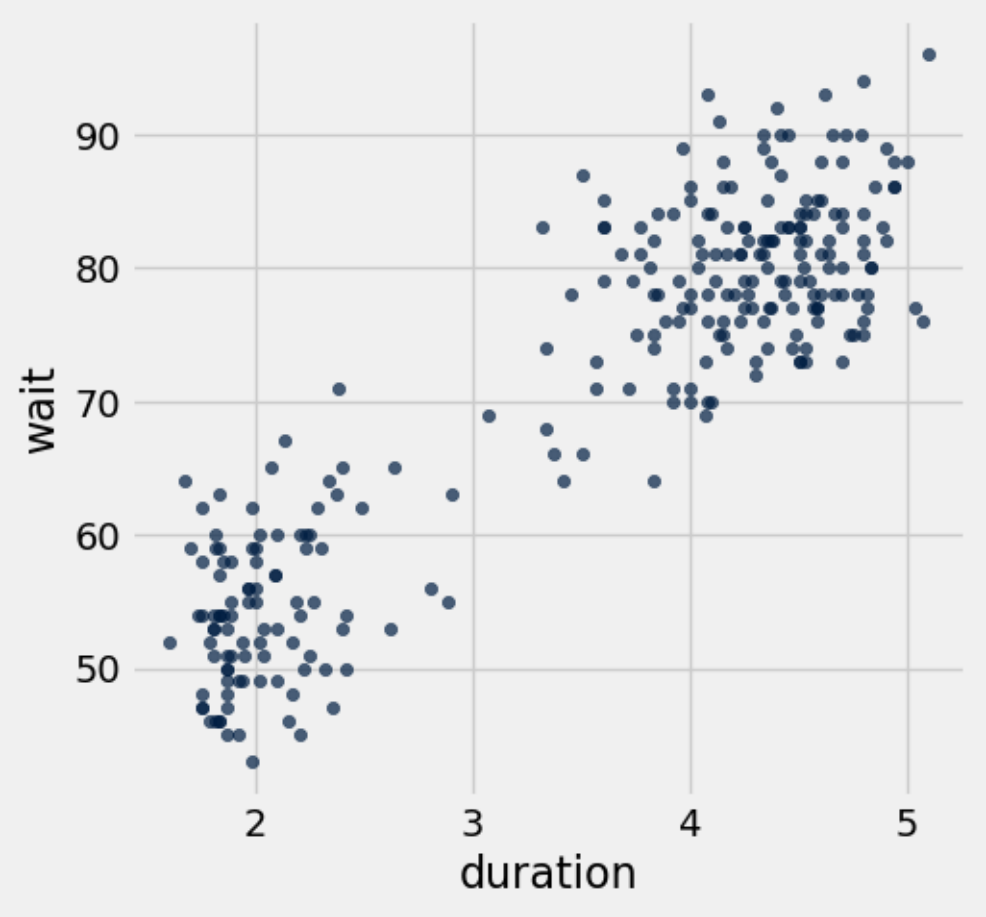
[13]:
```
grader.check("q1_0")
```

[13]:
```
q1_0 results: All test cases passed!
```

We would like to use linear regression to make predictions, but that won't work well if the data aren't roughly linearly related. To check that, we should look at the data.

Question 1.1. Make a scatter plot of the data (in the `faithful` table). It's conventional to put the column we want to predict on the vertical axis and the other column on the horizontal axis. In this case let's focus on trying to predict waiting times.

[14]:
```
faithful.scatter('duration', 'wait')
```



Question 1.2. Are eruption duration and waiting time roughly linearly related based on the scatter plot above? Is this relationship positive?

Based on the scatter plot eruption duration and waiting time are roughly linearly related. The points show a positive correlation meaning that as the eruption duration increases the waiting time tends to increase as well. This relationship appears to be positive.

We're going to continue with the assumption that they are linearly related, so it's reasonable to use linear regression to analyze this data.

We'd next like to plot the data in standard units. If you don't remember the definition of standard units, textbook section 14.2 might help!

Question 1.3. Compute the mean and standard deviation of the eruption durations and waiting times. **Then** create a table called `faithful_standard` containing the eruption durations and waiting times in standard units. The columns should be named `duration (standard units)` and `wait (standard units)`. Don't hesitate to add a couple of helper lines if it helps you get to the solution.

[49]:
```
duration_mean = np.mean(faithful.column('duration'))
duration_std = np.std(faithful.column('duration'))
wait_mean = np.mean(faithful.column('wait'))
wait_std = np.std(faithful.column('wait'))

def standard_units(any_numbers):
    "Convert any array of numbers to standard units."
    return (any_numbers - np.mean(any_numbers))/np.std(any_numbers)

faithful_standard = Table().with_columns(
    "duration (standard units)", standard_units(faithful.column('duration')),
    "wait (standard units)", standard_units(faithful.column('wait')))
faithful_standard
```

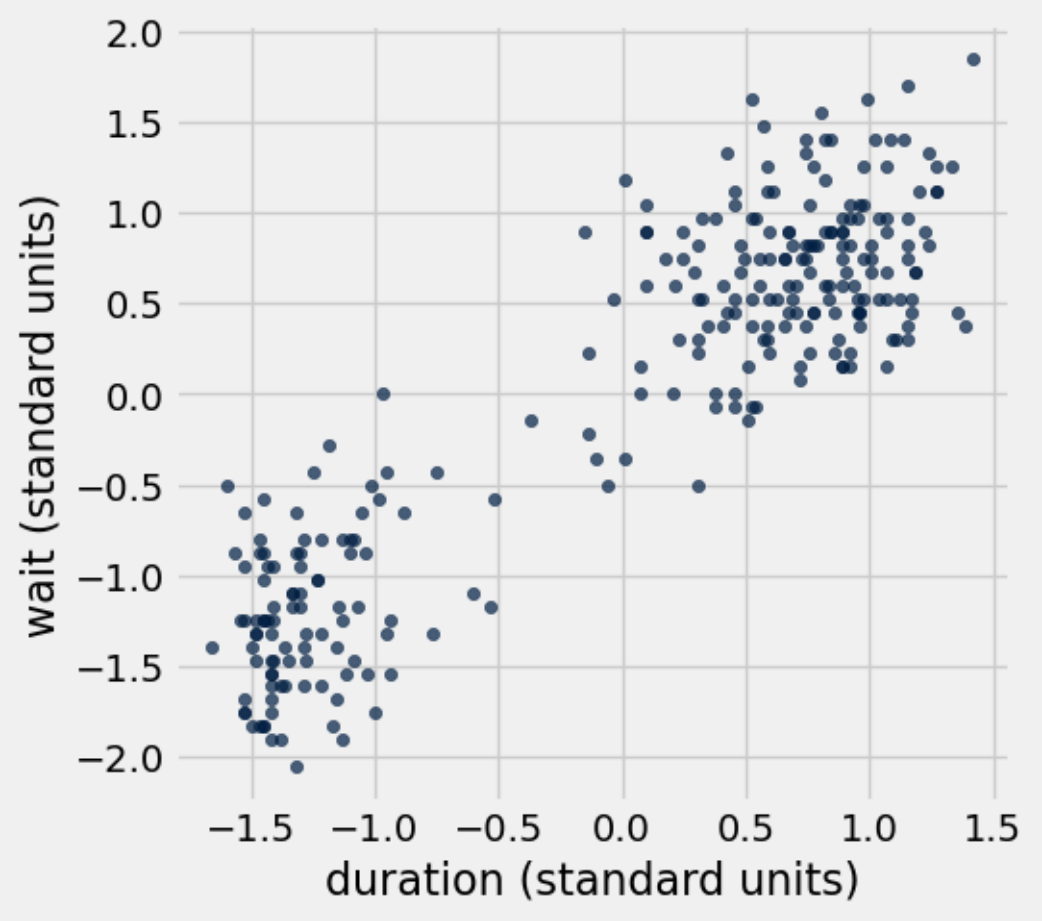
[49]:
```
duration (standard units) | wait (standard units)
0.0984989                 | 0.597123
-1.48146                  | -1.24518
-0.135861                 | 0.228663
-1.0575                   | -0.655644
0.917443                  | 1.03928
-0.530851                 | -1.17149
1.06403                   | 1.26035
0.0984989                 | 1.03928
-1.3498                   | -1.46626
0.756814                  | 1.03928
… (262 rows omitted)
```

[50]:
```
grader.check("q1_3")
```

[50]:
```
q1_3 results: All test cases passed!
```

Question 1.4. Plot the data again, but this time in standard units.

[51]:
```
faithful_standard.scatter("duration (standard units)", "wait (standard units)")
```



You'll notice that this plot looks the same as the last one! However, the data and axes are scaled differently. So it's important to read the ticks on the axes.

Question 1.5. Among the following numbers, which would you guess is closest to the correlation between eruption duration and waiting time in this dataset?

-1 , 0 or 1

Assign `correlation` to the number corresponding to your guess (either -1, 0 or 1).

[56]:
```
correlation = 1
```

[57]:
```
grader.check("q1_5")
```

[57]:
```
q1_5 results: All test cases passed!
```

Question 1.6. Compute the correlation coefficient: `r`, start by using the table `faithful_standard` from Question 1.3. If you don't remember how to compute `r`, section 15.1 explains how to do this.

[58]:
```
#HINT: Use the table faithful_standard that was created in question 1.3 (Note:␣
 ↪this table has the data in standard units)
r = np.mean(faithful_standard.column("duration (standard units)") *␣
 ↪faithful_standard.column("wait (standard units)"))
r
```

[58]:
```
0.90081116832181318
```

[59]:
```
grader.check("q1_6")
```

[59]:
```
q1_6 results: All test cases passed!
```

3.1 2. The regression line

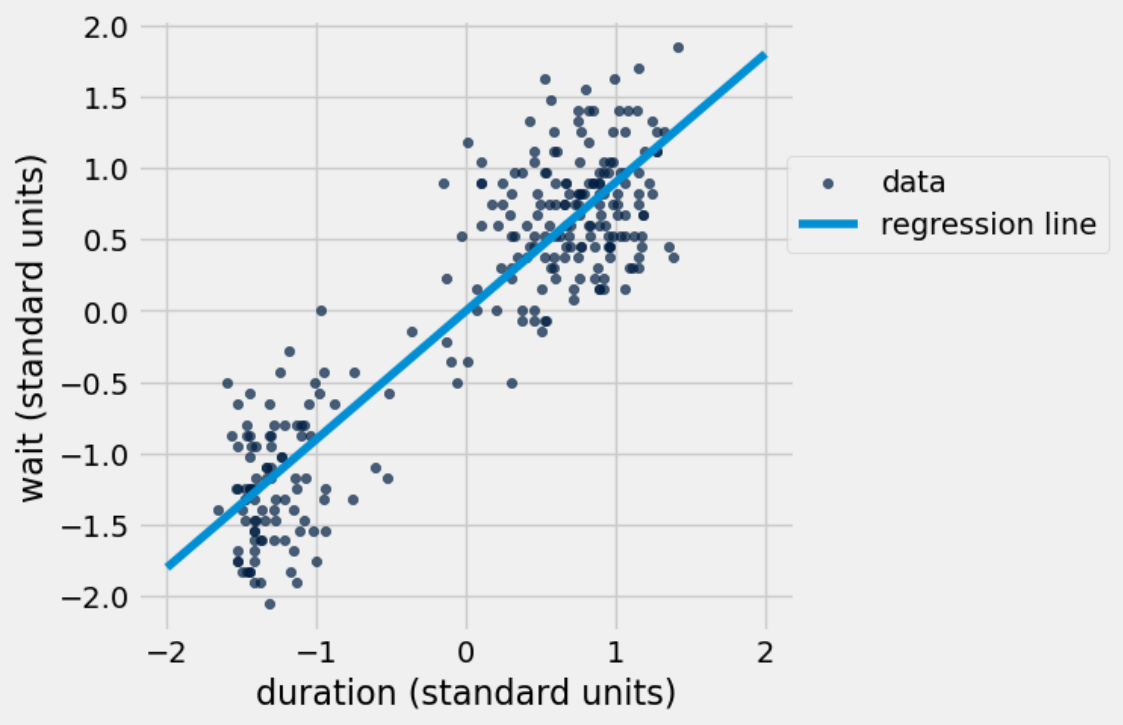
Recall that the **correlation** is the **slope of the regression line when the data are put in standard units**.

The next cell plots the regression line in standard units:

$$\text{waiting time in standard units} = r \times \text{eruption duration in standard units}$$

Then, it plots the data in standard units again, for comparison.

[60]:
```
def plot_data_and_line(dataset, x, y, point_0, point_1):
    """Makes a scatter plot of the dataset, along with a line passing through␣
 ↪two points."""
    dataset.scatter(x, y, label="data")
    xs, ys = zip(point_0, point_1)
    plots.plot(xs, ys, label="regression line")
    plots.legend(bbox_to_anchor=(1.5,.8))

plot_data_and_line(faithful_standard,
                   "duration (standard units)",
                   "wait (standard units)",
                   [-2, -2*r],
                   [2, 2*r])
```



How would you take a point in standard units and convert it back to original units? We'd have to "stretch" its horizontal position by `duration_std` and its vertical position by `wait_std`. That means the same thing would happen to the slope of the line.

Stretching a line horizontally makes it less steep, so we divide the slope by the stretching factor. Stretching a line vertically makes it more steep, so we multiply the slope by the stretching factor.

Question 2.1. Calculate the slope of the regression line in original units, and assign it to `slope`.

(If the "stretching" explanation is unintuitive, consult section 15.2 in the textbook.)

[61]:
```
# HINT: Use wait_std & duration_std from question 1.3, and r (correlation␣
↪coefficient) from question 1.6
slope = r * (wait_std / duration_std)
slope
```

[61]:
```
10.729641395133527
```

[62]:
```
grader.check("q2_1")
```

[62]:
```
q2_1 results: All test cases passed!
```

We know that the regression line passes through the point `(duration_mean, wait_mean)`. Recall that the equation of the regression line in the original units is:

$$\text{waiting time} = \text{slope} \times \text{eruption duration} + (-\text{slope} \times \text{duration\_mean} + \text{wait\_mean})$$

Question 2.2. Calculate the intercept in original units and assign it to `intercept`. Section 15.2.5 may be helpful.

[63]:
```
intercept = wait_mean - (slope * duration_mean)
intercept
```

[63]:
```
33.474397022753351
```

[64]:
```
grader.check("q2_2")
```

[64]:
```
q2_2 results: All test cases passed!
```

3.2 3. Investigating the regression line

The slope and intercept tell you exactly what the regression line looks like. To predict the waiting time for an eruption, multiply the eruption's duration by `slope` and then add `intercept`.

Question 3.1. Compute the predicted waiting time for an eruption that lasts 2 minutes, and for an eruption that lasts 5 minutes.

[65]:
```
two_minute_predicted_waiting_time = slope * 2 + intercept
five_minute_predicted_waiting_time = slope * 5 + intercept

# Here is a helper function to print out your predictions.
# Don't modify the code below.
def print_prediction(duration, predicted_waiting_time):
    print("After an eruption lasting", duration,
          "minutes, we predict you'll wait", predicted_waiting_time,
          "minutes until the next eruption.")

print_prediction(2, two_minute_predicted_waiting_time)
print_prediction(5, five_minute_predicted_waiting_time)
```

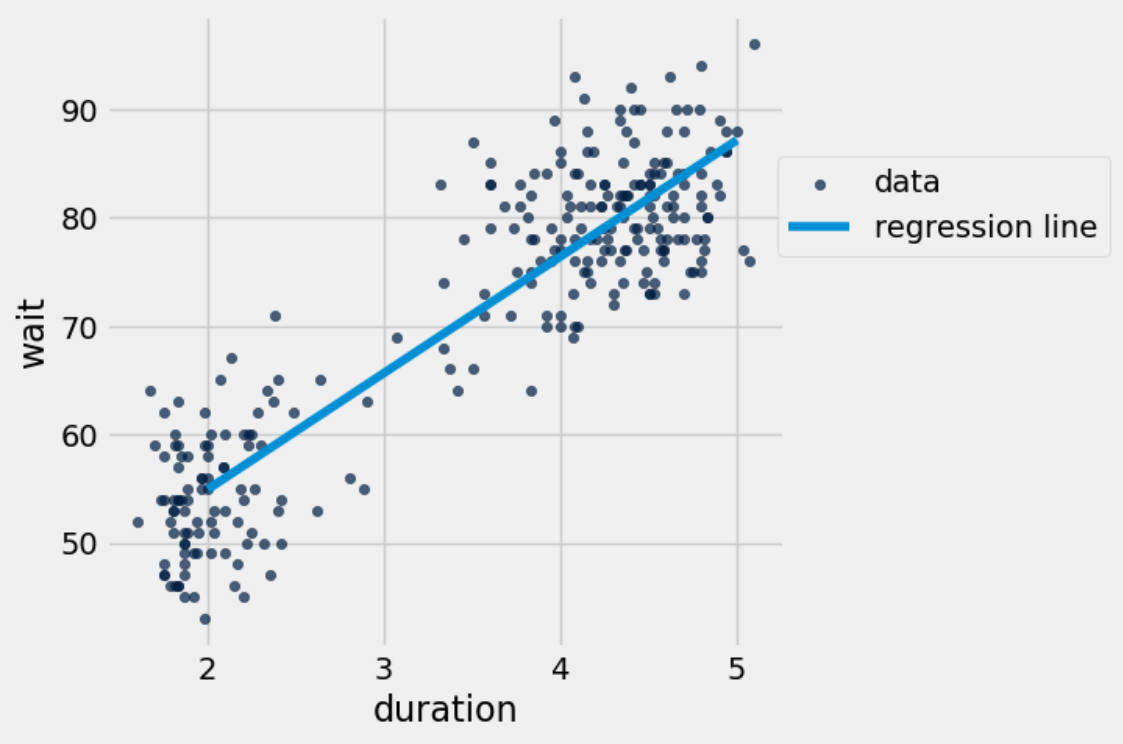
```
After an eruption lasting 2 minutes, we predict you'll wait 54.933679813 minutes
until the next eruption.
After an eruption lasting 5 minutes, we predict you'll wait 87.1226039984
minutes until the next eruption.
```

[66]:
```
grader.check("q3_1")
```

[66]:
```
q3_1 results: All test cases passed!
```

The next cell plots the line that goes between those two points, which is (a segment of) the regression line.

[67]:
```
plot_data_and_line(faithful, "duration", "wait",
                   [2, two_minute_predicted_waiting_time],
                   [5, five_minute_predicted_waiting_time])
```



Question 3.2. Make predictions for the waiting time after each eruption in the `faithful` table. (Of course, we know exactly what the waiting times were! We are doing this so we can see how accurate our predictions are.) Put these numbers into a column in a new table called `faithful_predictions`. Its first row should look like this:

duration	wait	predicted wait
3.6	79	72.1011

[68]:
```
#HINT: There is no need to use a for loop here, follow the steps below instead

#STEP 1: Save the predicted waiting times in an array predicted_waits for all␣
 ↪of the durations in the faithful table
predicted_waits = slope * faithful.column("duration") + intercept
#STEP 2: Add the new array as a column to the table faithful and save the new␣
 ↪table as faithful_predictions
# (make sure the new column is labeled "predicted wait")
faithful_predictions = faithful.with_column("predicted wait", predicted_waits)
faithful_predictions
```

[68]:
```
duration | wait | predicted wait
3.6      | 79   | 72.1011
1.8      | 54   | 52.7878
```

```
3.333    | 74   | 69.2363
2.283    | 62   | 57.9702
4.533    | 85   | 82.1119
2.883    | 55   | 64.408
4.7      | 88   | 83.9037
3.6      | 85   | 72.1011
1.95     | 51   | 54.3972
4.35     | 85   | 80.1483
… (262 rows omitted)
```

[69]:
```
grader.check("q3_2")
```

[69]:
```
q3_2 results: All test cases passed!
```

Question 3.3. How close were we? Compute the *residual* for each eruption in the dataset. The residual is the actual waiting time minus the predicted waiting time. Add the residuals to `faithful_predictions` as a new column called `residual` and name the resulting table `faithful_residuals`.

[70]:
```
# HINT: Just like in Question 3.2, your code will be much simpler if you don't␣
↪use a `for` loop.
# Your code can be just 1 line or you can add a helper line.
residuals = faithful_predictions.column("wait") - faithful_predictions.
↪column("predicted wait")
faithful_residuals = faithful_predictions.with_column("residual", residuals)
faithful_residuals
```

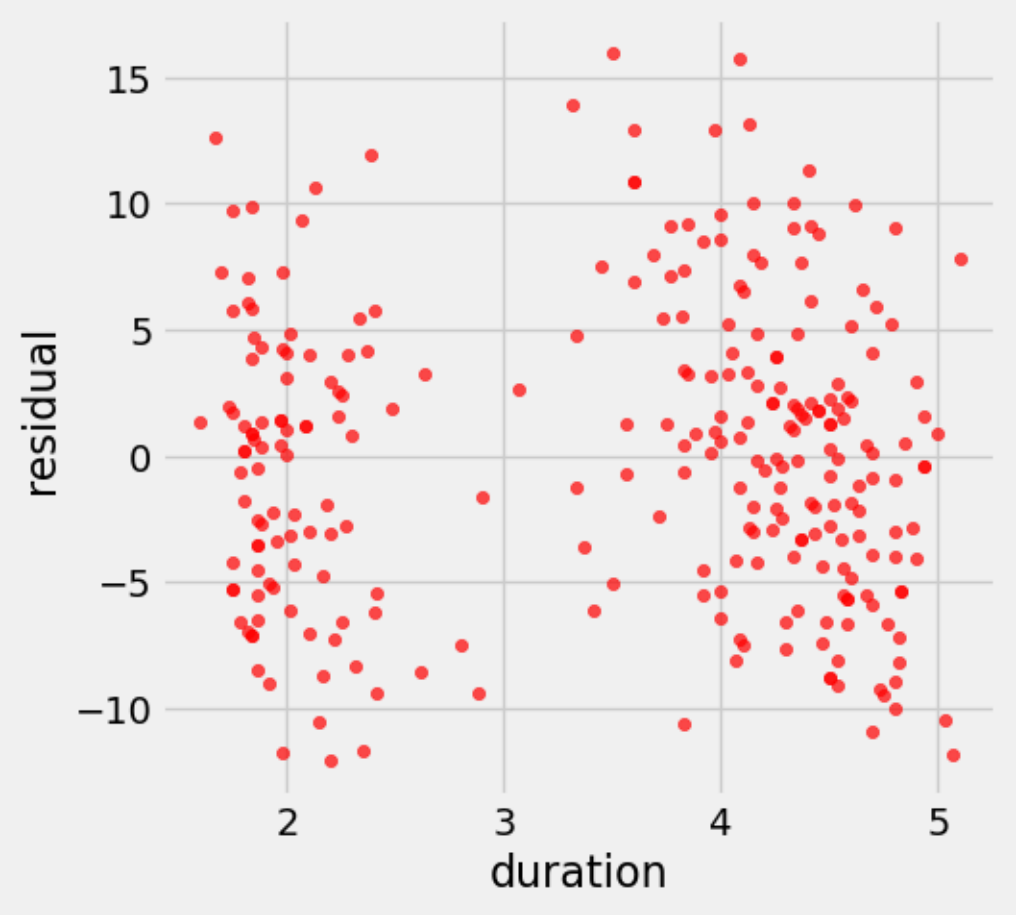
[70]:
```
duration | wait | predicted wait | residual
3.6      | 79   | 72.1011        | 6.89889
1.8      | 54   | 52.7878        | 1.21225
3.333    | 74   | 69.2363        | 4.76371
2.283    | 62   | 57.9702        | 4.02983
4.533    | 85   | 82.1119        | 2.88814
2.883    | 55   | 64.408         | -9.40795
4.7      | 88   | 83.9037        | 4.09629
3.6      | 85   | 72.1011        | 12.8989
1.95     | 51   | 54.3972        | -3.3972
4.35     | 85   | 80.1483        | 4.85166
… (262 rows omitted)
```

[71]:
```
grader.check("q3_3")
```

[71]:
```
q3_3 results: All test cases passed!
```

Here is a plot of the residuals you computed. Each point corresponds to one eruption. It shows how much our prediction over- or under-estimated the waiting time.

[72]:
```
faithful_residuals.scatter("duration", "residual", color="r")
```



There isn't really a pattern in the residuals, which confirms that it was reasonable to try linear regression. It's true that there are two separate clouds; the eruption durations seemed to fall into two distinct clusters. But that's just a pattern in the eruption durations, not a pattern in the relationship between eruption durations and waiting times.

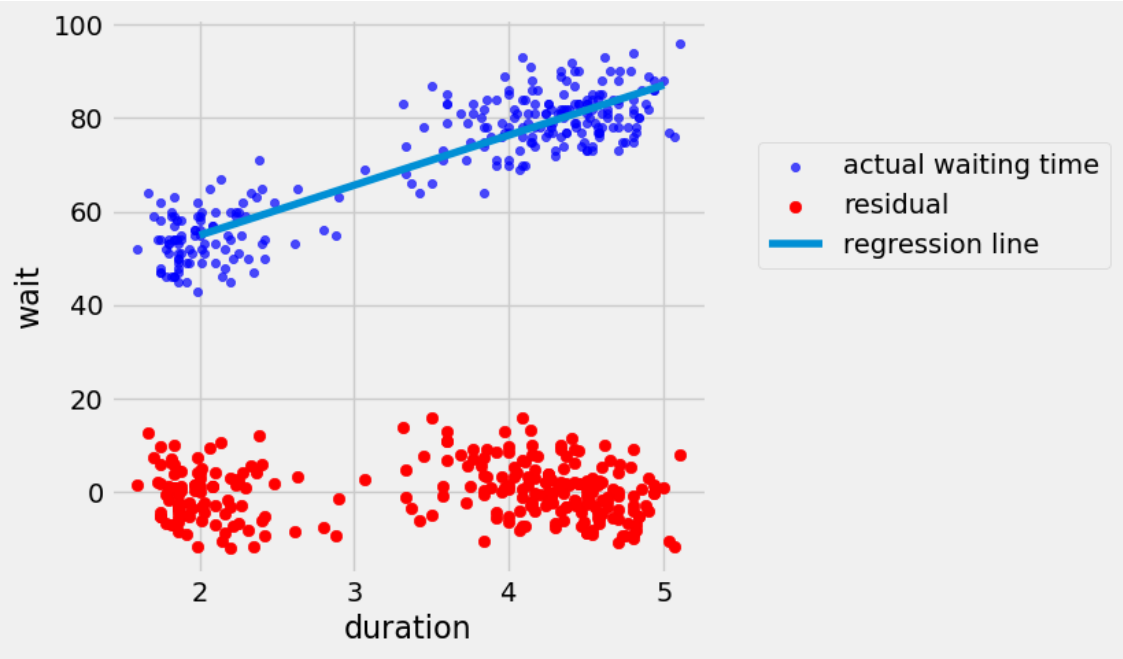
3.3 4. How accurate are different predictions?

Earlier, you should have found that the correlation is fairly close to 1, so the line fits fairly well on the training data. That means the residuals are overall small (close to 0) in comparison to the waiting times.

We can see that visually by plotting the waiting times and residuals together:

[73]:
```
# Just run this cell.
faithful_residuals.scatter("duration", "wait", label="actual waiting time",␣
  ↪color="blue")
plots.scatter(faithful_residuals.column("duration"), faithful_residuals.
  ↪column("residual"), label="residual", color="r")
plots.plot([2, 5], [two_minute_predicted_waiting_time,␣
  ↪five_minute_predicted_waiting_time], label="regression line")
```

```
plots.legend(bbox_to_anchor=(1.7,.8));
```



However, unless you have a strong reason to believe that the linear regression model is true, you should be wary of applying your prediction model to data that are very different from the training data.

Question 4.1. In `faithful`, no eruption lasted exactly 0, 2.5, or 60 minutes. Using this line, what is the predicted waiting time for an eruption that lasts 0 minutes? 2.5 minutes? An hour?

[74]:
```
zero_minute_predicted_waiting_time = slope * 0 + intercept
two_point_five_minute_predicted_waiting_time = slope * 2.5 + intercept
hour_predicted_waiting_time = slope * 60 + intercept

print_prediction(0, zero_minute_predicted_waiting_time)
print_prediction(2.5, two_point_five_minute_predicted_waiting_time)
print_prediction(60, hour_predicted_waiting_time)
```

```
After an eruption lasting 0 minutes, we predict you'll wait 33.4743970228
minutes until the next eruption.
After an eruption lasting 2.5 minutes, we predict you'll wait 60.2985005106
minutes until the next eruption.
After an eruption lasting 60 minutes, we predict you'll wait 677.252880731
minutes until the next eruption.
```

[75]:
```
grader.check("q4_1")
```

[75]:
```
q4_1 results: All test cases passed!
```

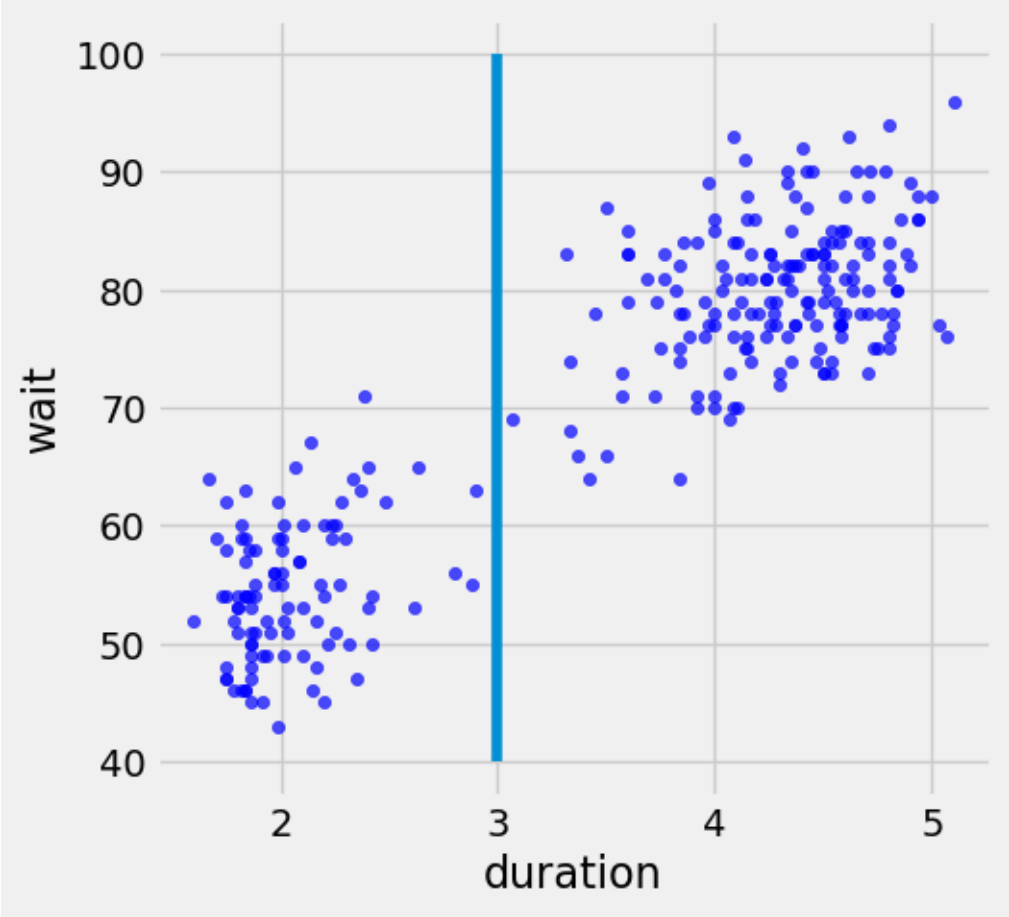
Question 4.2. For each prediction, state whether you think it's reliable and explain your reasoning.

The prediction for 0 minutes is not reliable because eruptions cannot last 0 minutes so it does not make sense. The prediction for 2.5 minutes is more reasonable because it is within the range of the data so the model works better there. The prediction for 60 minutes is not reliable because it is way outside the data we used to train the model so the result is not tgat trustworthy.

3.4 5. Divide and Conquer

It appears from the scatter diagram that there are two clusters of points: one for durations around 2 and another for durations between 3.5 and 5. A vertical line at 3 divides the two clusters.

[76]:
```
faithful.scatter("duration", "wait", label="actual waiting time", color="blue")
plots.plot([3, 3], [40, 100]);
```



The `standardize` function from lecture appears below, which takes in a table with numerical columns and returns the same table with each column converted into standard units.

[77]:
```
# Just run this cell.

def standard_units(any_numbers):
```

```
    "Convert any array of numbers to standard units."
    return (any_numbers - np.mean(any_numbers)) / np.std(any_numbers)

def standardize(t):
    """Return a table in which all columns of t are converted to standard units.
 ↪"""
    t_su = Table()
    for label in t.labels:
        t_su = t_su.with_column(label + ' (su)', standard_units(t.
 ↪column(label)))
    return t_su
```

Question 5.1. Separately compute the correlation coefficient r for all the points with a duration below 3 **and then** for all the points with a duration above 3. To do so, create a function that computes `r` from a table, and then pass it two different tables of points, called `below_3` and `above_3`.

Hint: You can assume that the table does not have any duration values that are exactly 3.

[78]:
```
def corr_coeff(t):
    """Return the regression coefficient for columns 0 & 1."""
    t_su = standardize(t)
    # STEP 1: complete the function to return the regression coefficient for␣
 ↪column 0 & 1
    x_su = t_su.column(0)
    y_su = t_su.column(1)
    return np.mean(x_su * y_su)

# STEP 2: split the original table faithful into 2 tables below_3 and above_3
below_3 = faithful.where("duration", are.below(3))
above_3 = faithful.where("duration", are.above(3))

# The lines below will use the new function created above to calculate r for␣
 ↪both tables below_3 and above_3
below_3_r = corr_coeff(below_3)
above_3_r = corr_coeff(above_3)

# The line below will print the values you obtained for r below 3 and r above 3
print("For points below 3, r is", below_3_r, "; for points above 3, r is",␣
 ↪above_3_r)
```

```
For points below 3, r is 0.290189526493 ; for points above 3, r is
0.372782225571
```

[79]:
```
grader.check("q5_1")
```

[79]:
```
q5_1 results: All test cases passed!
```

Question 5.2. Complete the functions `slope_of` and `intercept_of` below.

When you're done, the functions `wait_below_3` and `wait_above_3` should each use a different regression line to predict a wait time for a duration. The first function should use the regression line for all points with duration below 3. The second function should use the regression line for all points with duration above 3.

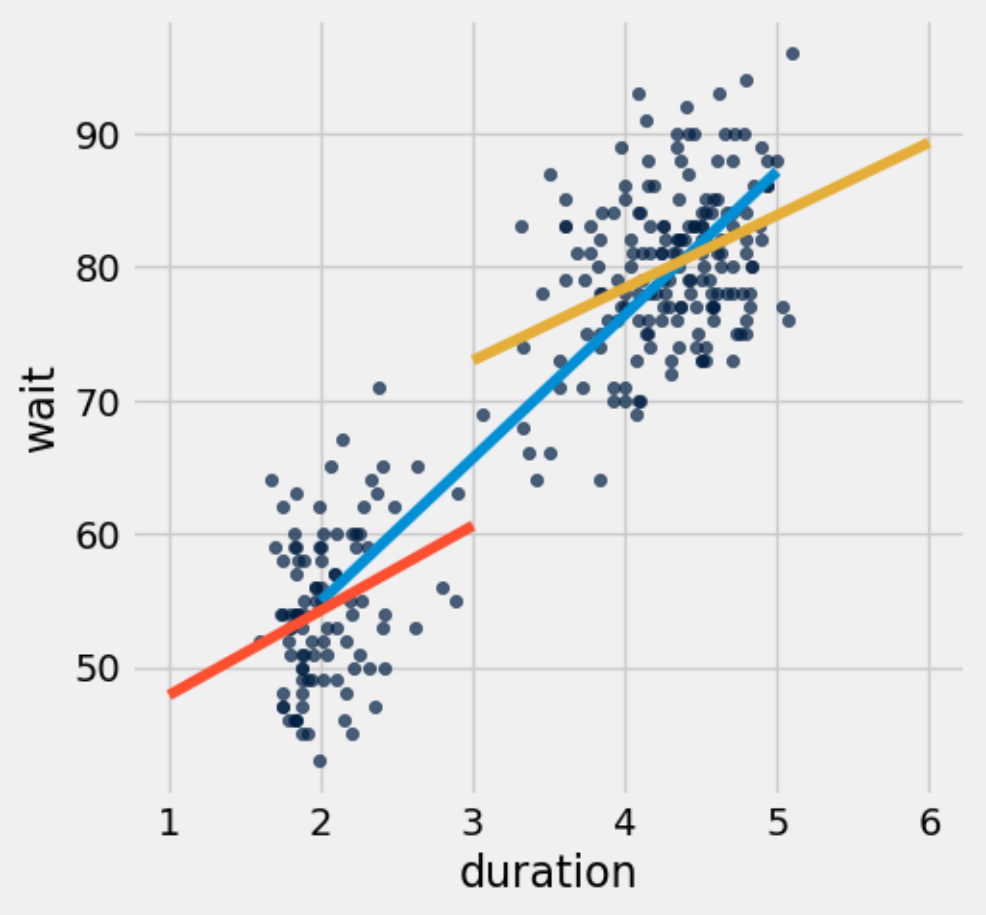
[83]:
```
# Create generic functions for slope and intercept
# HINT: If you don't remember how to calculate slope and intercept, refer back␣
 ↪to section 2.1 and 2.2
def slope_of(table, r):
    """Return the slope of the regression line for table in original units.

    Assume that column 0 contains x values and column 1 contains y values.
    r is the regression coefficient for x and y.
    """
    x_std = np.std(table.column(0))
    y_std = np.std(table.column(1))
    return r * (y_std / x_std)

def intercept_of(table, r):
    """Return the intercept of the regression line for table in original units.

    Assume again that column 0 contains x values and column 1 contains y values.
    r is the regression coefficient for x and y.
    """
    slope = slope_of(table, r)
    x_mean = np.mean(table.column(0))
    y_mean = np.mean(table.column(1))
    return y_mean - (slope * x_mean)

# Below we use the functions you created above
below_3_slope = slope_of(below_3, below_3_r)
below_3_intercept = intercept_of(below_3, below_3_r)
above_3_slope = slope_of(above_3, above_3_r)
above_3_intercept = intercept_of(above_3, above_3_r)

def wait_below_3(duration):
    return below_3_slope * duration + below_3_intercept

def wait_above_3(duration):
    return above_3_slope * duration + above_3_intercept
```

[84]:
```
grader.check("q5_2")
```

[84]:
```
q5_2 results: All test cases passed!
```

The plot below shows the two different regression lines, one for each cluster, along with the original regression line!

[86]:
```
faithful.scatter(0, 1)
plots.plot([2, 5], [two_minute_predicted_waiting_time,␣
 ↪five_minute_predicted_waiting_time])
plots.plot([1, 3], [wait_below_3(1), wait_below_3(3)])
plots.plot([3, 6], [wait_above_3(3), wait_above_3(6)]);
```



Question 5.3. Write a function `predict_wait` that takes a `duration` and returns the predicted wait time using the appropriate regression line, depending on whether the duration is below 3 or greater than (or equal to) 3.

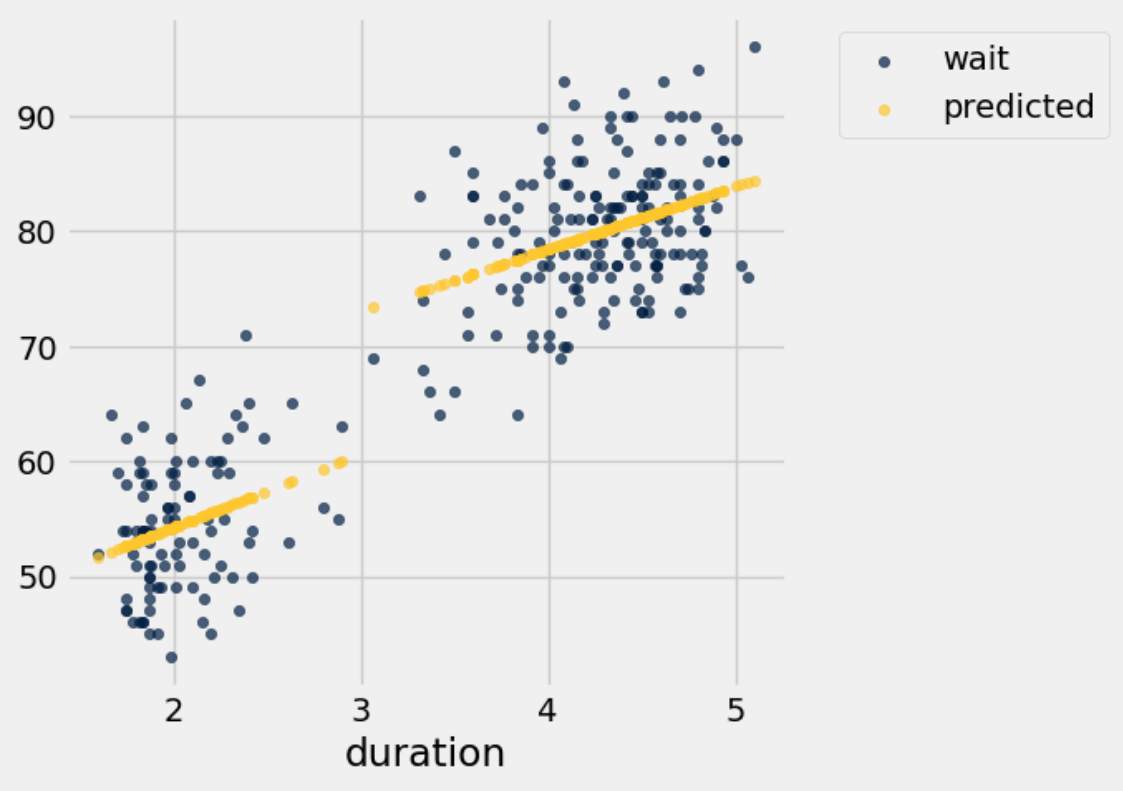
[87]:
```
# HINT: You can use the functions wait_below_3 and wait_above_3 from 5.2
def predict_wait(duration):
    if duration < 3:
        return wait_below_3(duration)
    else:
        return wait_above_3(duration)
```

[88]:
```
grader.check("q5_3")
```

[88]:
```
q5_3 results: All test cases passed!
```

The predicted wait times for each point appear below.

[89]:
```
faithful_pred_split = faithful.with_column('predicted', faithful.
  ↪apply(predict_wait, 'duration'))
faithful_pred_split.scatter(0)
```



Question 5.4. Do you think the predictions produced by `predict_wait` would be more or less accurate than the predictions from the regression line you created in section 2? How could you tell?

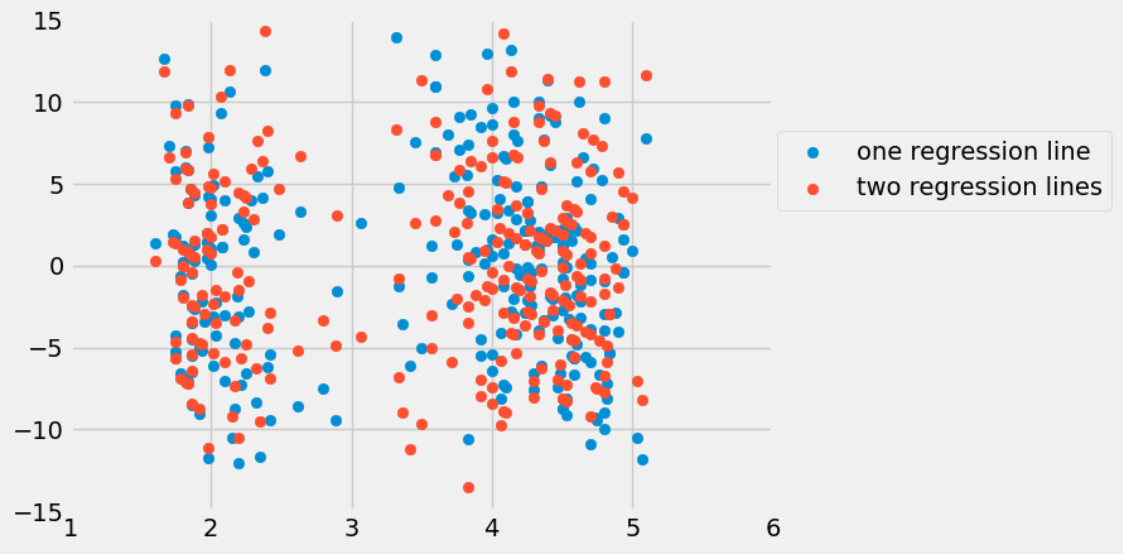
The predictions from predict_wait would be more accurate because they use two separate regression lines for the two clusters of points. This makes the predictions better fit the data since the single regression line from section 2 does not account for the two groups. You can tell because the predicted points in yellow follow the actual data points more closely.

The following cell will plot the residuals for each eruption in the dataset when we have one regression line and two regression lines. We also see the average magnitude of the residual values.

[90]:
```
# Just run this cell
faithful_pred_split_residuals = faithful_pred_split.with_column('residual',␣
  ↪faithful_pred_split.column(1) - faithful_pred_split.column(2))
```

```
plots.scatter(faithful_residuals.column('duration'), faithful_residuals.
 ↪column('residual'), label='one regression line')
plots.scatter(faithful_pred_split_residuals.column('duration'),␣
 ↪faithful_pred_split_residuals.column('residual'), label='two regression␣
 ↪lines');
plots.axis([1, 6, -15, 15])
plots.legend(bbox_to_anchor=(1.5,.8));
print("Average Magnitude of Residual Values for One Regression Line: ", np.
 ↪mean(abs(faithful_residuals.column('residual'))))
print("Average Magnitude of Residual Values for Two Regression Lines: ", np.
 ↪mean(abs(faithful_pred_split_residuals.column('residual'))))
```

```
Average Magnitude of Residual Values for One Regression Line:  4.77872005877
Average Magnitude of Residual Values for Two Regression Lines:  4.57575882719
```



The residual plot for the wait times when they are predicted by two regression lines (red) doesn't really have a pattern, which confirms that it was also appropriate to use linear regression in our "Divide and Conquer" scenario. How do the two residual plots compare?

3.5 6. Submission

Congratulations, you're done with Lab 9!

Be sure to...

- run the tests and verify that they all pass,
- choose **Download as PDF** from the **File** menu
- submit the .pdf file on **canvas**.