Comedy Show Lab

Imagine that you are the producer for a comedy show at your school. We need you to use knowledge of linear regression to make predictions as to the success of the show.

Working through a linear regression

The comedy show is trying to figure out how much money to spend on advertising in the student newspaper. The newspaper tells the show that

- For every two dollars spent on advertising, three students attend the show.
- If no money is spent on advertising, no one will attend the show.

Write a linear regression function called `attendance` that shows the relationship between advertising and attendance expressed by the newspaper.

In [1]:
```
def attendance(advertising):
    return (advertising/2)*3
    pass
```

In [2]:
```
attendance(100) # 150
```

Out[2]:
```
150.0
```

In [3]:
```
attendance(50) # 75
```

Out[3]:
```
75.0
```

As the old adage goes, "Don't ask the barber if you need a haircut!" Likewise, despite what the student newspaper says, the comedy show knows from experience that they'll still have a crowd even without an advertising budget. Some of the comedians in the show have friends (believe it or not), and twenty of those friends will show up. Write a function called `attendance_with_friends` that models the following:

- When the advertising budget is zero, 20 friends still attend
- Three additional people attend the show for every two dollars spent on advertising

In [4]:
```
def attendance_with_friends(advertising):
    return (advertising/2)*3 + 20
    pass
```

In [5]:
```
attendance_with_friends(100) # 170
```

Out[5]:
```
170.0
```

In [6]:
```
attendance_with_friends(50) # 95
```

Out[6]:
```
95.0
```

Plot it

Let's help plot this line so you can get a sense of what your m and b values look like in graph form.

Our x values can be a list of `initial_sample_budgets`, equal to a list of our budgets. And we can use the outputs of our `attendance_with_friends` function to determine the list of `attendance_values`, the attendance at each of those x values.

In [7]:
```
initial_sample_budgets = [0, 50, 100]
attendance_values = [20, 95, 170]
```

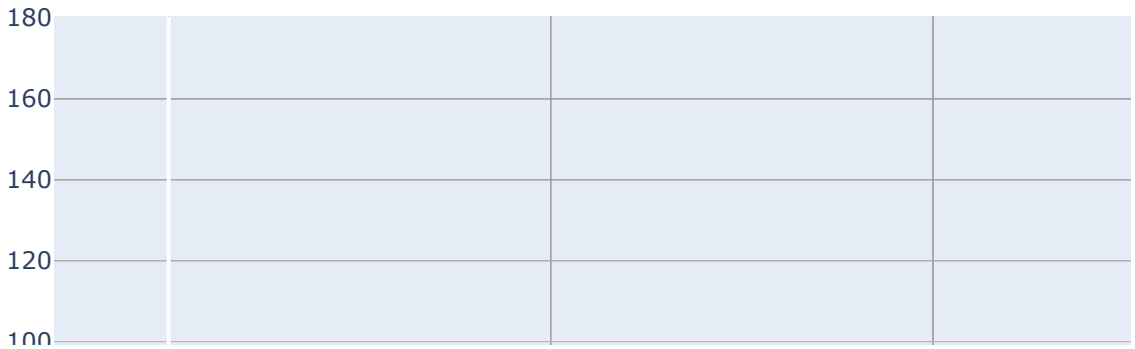
First we import the necessary plotly library, and `graph_obs` function, and setup `plotly` to be used without uploading our plots to its website.

Finally, we plot out our regression line using our `attendance_with_friends` function. Our x values will be the budgets. For our y values, we need to use our `attendance_with_friends` function to create a list of y-value attendances for every input of x.

In [8]:
```
import plotly
from plotly import graph_objs
plotly.offline.init_notebook_mode(connected=True)

trace_of_attendance_with_friends = graph_objs.Scatter(
    x=initial_sample_budgets,
    y=attendance_values,
)

plotly.offline.iplot([trace_of_attendance_with_friends])
```



In [9]:
```
trace_of_attendance_with_friends
```

Out[9]:
```
Scatter({
    'x': [0, 50, 100], 'y': [20, 95, 170]
})
```

Now let's write a couple functions that we can use going forward. We'll write a function called `m_b_data` that given a slope of a line, m, a y-intercept, b, will return a dictionary that has a key of `x` pointing to a list of `x_values`, and a key of `y` that points to a list of `y_values`. Each y value should be the output of a regression line for the provided m and b values, for each of the provided `x_values`.

In [10]:
```
def m_b_data(m, b, x_values):
    y_values = []
    line = dict()
    for x in x_values:
        print(m*x+b)
        y_values.append(m*x + b)
    line.update({'x':x_values,'y':y_values})
    return line
    pass
```

In [11]:
```
m_b_data(1.5, 20, [0, 50, 100]) # {'x': [0, 50, 100], 'y': [20.0, 95.0, 170
```

```
20.0
95.0
170.0
```

Out[11]:
```
{'x': [0, 50, 100], 'y': [20.0, 95.0, 170.0]}
```

Now let's write a function called `m_b_trace` that uses our `m_b_data` function to return a dictionary that includes keys of `name` and `mode` in addition to `x` and `y`. The values of `mode` and `name` are provided as arguments. When the `mode` argument is not provided, it has a default value of `lines` and when `name` is not provided, it has a default value of `line function`.

In [12]:
```
def m_b_trace(m, b, x_values, mode = 'lines', name = 'line function'):
    line_update = dict()
    line_update['mode'] = mode
    line_update['name'] = name
    line = m_b_data(m,b,x_values)
    line_update.update({k:v for k,v in line.items()})
    return line_update
    pass
```

In [13]:
```
m_b_trace(1.5, 20, [0, 50, 100])
# {'mode': 'line', 'name': 'line function', 'x': [0, 50, 100], 'y': [20.0,
```

```
20.0
95.0
170.0
```

Out[13]:
```
{'mode': 'lines',
 'name': 'line function',
 'x': [0, 50, 100],
 'y': [20.0, 95.0, 170.0]}
```

Calculating lines

The comedy show decides to advertise for two different shows. The attendance looks like the following.

Budgets (dollars)	Attendance
200	400

Budgets (dollars)	Attendance
400	700

In code, we represent this as the following:

In [14]:
```
first_show = {'budget': 200, 'attendance': 400}
second_show = {'budget': 400, 'attendance': 700}
```

Write a function called `marginal_return_on_budget` that returns the expected amount of increase per every dollar spent on budget.

> The function should use the formula for calculating the slope of a line provided two points.

In [15]:
```
def marginal_return_on_budget(first_show, second_show):
    if (first_show['attendance'] > second_show['attendance']):
        return (first_show['attendance'] - second_show['attendance'])/(firs
    else:
        return (second_show['attendance'] - first_show['attendance'])/(secc
    pass
```

In [16]:
```
marginal_return_on_budget(first_show, second_show) # 1.5
```

Out[16]:
```
1.5
```

In [17]:
```
first_show
```

Out[17]:
```
{'budget': 200, 'attendance': 400}
```

Just to check, let's use some different data to make sure our `marginal_return_on_budget` function calculates the slope properly.

In [18]:
```
imaginary_third_show = {'budget': 300, 'attendance': 500}
imaginary_fourth_show = {'budget': 600, 'attendance': 900}
marginal_return_on_budget(imaginary_third_show, imaginary_fourth_show) # 1.
```

Out[18]:
```
1.3333333333333333
```

Great! Now we'll begin to write functions that we can use going forward. The functions will calculate attributes of lines in general and can be used to predict the attendance of the comedy show.

Take the following data. The comedy show spends zero dollars on advertising for the next show. The attendance chart now looks like the following:

Budgets (dollars)	Attendance
0	100

Budgets (dollars)	Attendance
200	400
400	700

In [19]:
```
budgets = [0, 200, 400]
attendance_numbers = [100, 400, 700]
```

To get you started, we'll provide a function called `sorted_points` that accepts a list of x values and a list of y values and returns a list of point coordinates sorted by their x values. The return value is a list of sorted tuples.

In [20]:
```
def sorted_points(x_values, y_values):
    values = list(zip(x_values, y_values))
    sorted_values = sorted(values, key=lambda value: value[0])
    return sorted_values
```

In [21]:
```
sorted_points([4, 1, 6], [4, 6, 7])
```

Out[21]:
```
[(1, 6), (4, 4), (6, 7)]
```

build_starting_line

In this section, we'll write a function called `build_starting_line` . The function that we end up building simply draws a line between our points with the highest and lowest x values. We are selecting these points as an arbitrary "starting" point for our regression line.

> As John von Neumann said, "truth … is much too complicated to allow anything but approximations." All models are inherently wrong, but some are useful. In future lessons, we will learn how to build a regression line that accurately matches our dataset. For now, we will focus on building a useful "starting" line using the first and last points along the x-axis.

First, write a function called `slope` that, given a list of x values and a list of y values, will use the points with the lowest and highest x values to calculate the slope of a line.

In [22]:
```
def slope(x_values, y_values):
    data_list = []
    data_list = sorted_points(x_values, y_values)
    x_hi = 0
    x_lo = 0
    y_hi = 0
    y_lo = 0
    for i in range(0,len(data_list)):
        x_hi = data_list[len(data_list)-1][0]
        x_lo = data_list[0][0]
        y_hi = data_list[len(data_list)-1][1]
        y_lo = data_list[0][1]
        #return (data_list[len(data_list)-1][1]-data_list[0][1])/(data_list
    return (y_hi-y_lo)/(x_hi-x_lo)
    pass
```

In [23]:
```
slope([200, 400], [400, 700]) # 1.5
```

Out[23]:
```
1.5
```

Now write a function called `y_intercept`. Use the `slope` function to calculate the slope if it isn't provided as an argument. Then we will use the slope and the values of the point with the highest x value to return the y-intercept.

In [24]:
```
def y_intercept(x_values, y_values, m = None):
    slope_value = slope(x_values, y_values)
    data_list = sorted_points(x_values, y_values)
    return data_list[len(data_list)-1][1] - (slope_value * data_list[len(da
    pass
```

In [25]:
```
y_intercept([200, 400], [400, 700]) # 100
```

Out[25]:
```
100.0
```

In [26]:
```
y_intercept([0, 200, 400], [10, 400, 700]) # 10
```

Out[26]:
```
10.0
```

Now write a function called `build_starting_line` that given a list of `x_values` and a list of `y_values` returns a dictionary with a key of `m` and a key of `b` to return the `m` and `b` values of the calculated regression line. Use the `slope` and `y_intercept` functions to calculate the line.

In [27]:
```
def build_starting_line(x_values, y_values):
    para_dict = dict()
    slope_value = slope(x_values, y_values)
    intercept_value = y_intercept(x_values, y_values, m = None)
    para_dict.update({'b':intercept_value,'m':slope_value})
    return para_dict
    pass
```

In [28]:
```
build_starting_line([0, 200, 400], [10, 400, 700]) # {'b': 10.0, 'm': 1.725
```

Out[28]:
```
{'b': 10.0, 'm': 1.725}
```

Finally, let's write a function called `expected_value_for_line` that returns the expected attendance given the m, b, and $x\ value$.

In [29]:
```
first_show = {'budget': 300, 'attendance': 700}
second_show = {'budget': 400, 'attendance': 900}

shows = [first_show, second_show]
```

In [30]:
```
def expected_value_for_line(m, b, x_value):
    return m*x_value+b
    pass
```

In [31]:
```
expected_value_for_line(1.5, 100, 100) # 250
```

Out[31]:
```
250.0
```

Using our functions

Now that we have built these functions, we can use them on our dataset. Uncomment and run the lines below to see how we can use our functions going forward.

In [32]:
```
first_show = {'budget': 200, 'attendance': 400}
second_show = {'budget': 400, 'attendance': 700}
third_show = {'budget': 300, 'attendance': 500}
fourth_show = {'budget': 600, 'attendance': 900}

comedy_shows = [first_show, second_show, third_show, fourth_show]

show_x_values = list(map(lambda show: show['budget'], comedy_shows))
show_y_values = list(map(lambda show: show['attendance'], comedy_shows))
```

In [33]:
```
def trace_values(x_values, y_values, mode = 'markers', name="data"):
    return {'x': x_values, 'y': y_values, 'mode': mode, 'name': name}
```

In [34]:
```
def plot(traces):
    plotly.offline.iplot(traces)
```

In [35]:
```
comedy_show_trace = trace_values(show_x_values, show_y_values, name = 'come
comedy_show_trace
```

Out[35]:
```
{'x': [200, 400, 300, 600],
 'y': [400, 700, 500, 900],
 'mode': 'markers',
 'name': 'comedy show data'}
```

In [36]:
```
show_starting_line = build_starting_line(show_x_values, show_y_values)
show_starting_line
```

Out[36]:
```
{'b': 150.0, 'm': 1.25}
```

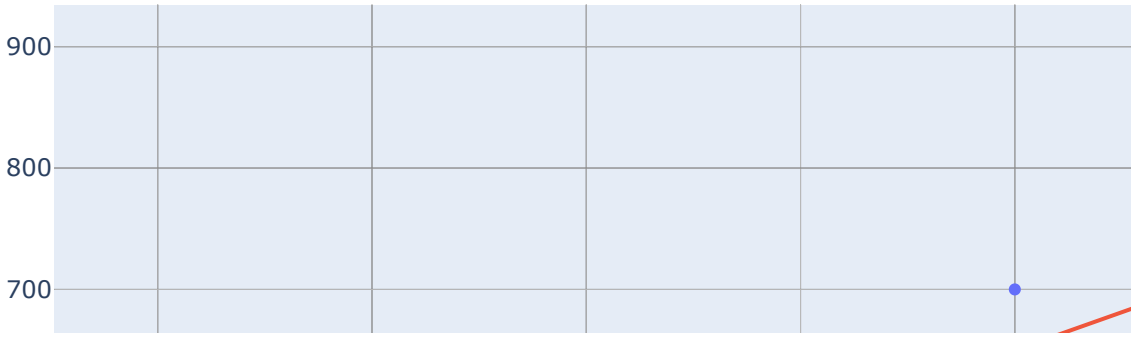
In [37]:
```
trace_show_line = m_b_trace(show_starting_line['m'], show_starting_line['b'
```

```
400.0
650.0
525.0
900.0
```

In [38]:
```
trace_show_line
```

Out[38]:
```
{'mode': 'lines',
 'name': 'starting line',
 'x': [200, 400, 300, 600],
 'y': [400.0, 650.0, 525.0, 900.0]}
```

In [39]:
```
plot([comedy_show_trace, trace_show_line])
```



As we can see above, we built a "starting" regression line out of the points with the lowest and highest x values. We will learn in future lessons how to improve our line so that it becomes the

"best fit" given all of our dataset, not just the first and last points. For now, this approach sufficed since our goal was to practice working with and plotting line functions.