# Evaluating Regression Lines Lab

## Introduction

In the previous lesson, we learned to evaluate how well a regression line estimated our actual data. In this lab, we'll turn these formulas into code. In doing so, we'll build lots of useful functions for both calculating and displaying our errors for a given regression line and dataset.

> In moving through this lab, we'll access to the functions that we previously built out to plot our data, available in the [graph (https://github.com/learn-co-curriculum/evaluating-regression-lines-lab/blob/master/graph.py)](https://github.com/learn-co-curriculum/evaluating-regression-lines-lab/blob/master/graph.py) here.

## Determining Quality

In the file, `movie_data.py` you will find movie data written as a python list of dictionaries, with each dictionary representing a movie. The movies are derived from the first 30 entries from the dataset containing 538 movies [provided here (https://raw.githubusercontent.com/fivethirtyeight/data/master/bechdel/movies.csv)](https://raw.githubusercontent.com/fivethirtyeight/data/master/bechdel/movies.csv).

In [1]:
```
from movie_data import movies
print(len(movies))
print(movies)
```


```
30
[{'budget': 13000000, 'domgross': 25682380.0, 'title': '21 &amp; Over'},
{'budget': 45658735, 'domgross': 13414714.0, 'title': 'Dredd 3D'}, {'budg
et': 20000000, 'domgross': 53107035.0, 'title': '12 Years a Slave'}, {'bu
dget': 61000000, 'domgross': 75612460.0, 'title': '2 Guns'}, {'budget': 4
0000000, 'domgross': 95020213.0, 'title': '42'}, {'budget': 225000000, 'd
omgross': 38362475.0, 'title': '47 Ronin'}, {'budget': 92000000, 'domgros
s': 67349198.0, 'title': 'A Good Day to Die Hard'}, {'budget': 12000000,
'domgross': 15323921.0, 'title': 'About Time'}, {'budget': 130000000, 'do
mgross': 60522097.0, 'title': 'After Earth'}, {'budget': 25000000, 'domgr
oss': 37304874.0, 'title': 'August: Osage County'}, {'budget': 50000000,
'domgross': 19452138.0, 'title': 'Beautiful Creatures'}, {'budget': 18000
000, 'domgross': 33345833.0, 'title': 'Blue Jasmine'}, {'budget': 5500000
0, 'domgross': 107136417.0, 'title': 'Captain Phillips'}, {'budget': 3000
0000, 'domgross': 35266619.0, 'title': 'Carrie'}, {'budget': 78000000, 'd
omgross': 119640264.0, 'title': 'Cloudy with a Chance of Meatballs 2'},
{'budget': 76000000, 'domgross': 368065385.0, 'title': 'Despicable Me
2'}, {'budget': 5500000, 'domgross': 24477704.0, 'title': 'Don Jon'}, {'b
udget': 120000000, 'domgross': 93050117.0, 'title': 'Elysium'}, {'budge
t': 110000000, 'domgross': 61737191.0, 'title': 'Ender&#39;s Game'}, {'bu
dget': 100000000, 'domgross': 107518682.0, 'title': 'Epic'}, {'budget': 7
0000000, 'domgross': 25213103.0, 'title': 'Escape Plan'}, {'budget': 1700
0000, 'domgross': 54239856.0, 'title': 'Evil Dead'}, {'budget': 16000000
0, 'domgross': 238679850.0, 'title': 'Fast and Furious 6'}, {'budget': 15
0000000, 'domgross': 393050114.0, 'title': 'Frozen'}, {'budget': 14000000
0, 'domgross': 122523060.0, 'title': 'G.I. Joe: Retaliation'}, {'budget':
60000000, 'domgross': 46000903.0, 'title': 'Gangster Squad'}, {'budget':
80000000, 'domgross': 133668525.0, 'title': 'Grown Ups'}, {'budget': 2300
0000, 'domgross': 25000178.0, 'title': 'Her'}, {'budget': 35000000, 'domg
ross': 134506920.0, 'title': 'Identity Thief'}, {'budget': 200000000, 'do
mgross': 408992272.0, 'title': 'Iron Man 3'}]
```


> Press shift + enter

In [2]:
```
movies[0]
```

Out[2]:
```
{'budget': 13000000, 'domgross': 25682380.0, 'title': '21 &amp; Over'}
```

In [3]:
```
movies[0]['budget']/1000000
```

Out[3]:
```
13.0
```

The numbers are in millions, so we will simplify things by dividing everything by a million

In [4]:
```
scaled_movies = list(map(lambda movie: {'title': movie['title'], 'budget':
print(scaled_movies[0])
print(scaled_movies)
```

```
{'title': '21 &amp; Over', 'budget': 13.0, 'domgross': 26.0}
[{'title': '21 &amp; Over', 'budget': 13.0, 'domgross': 26.0}, {'title':
'Dredd 3D', 'budget': 46.0, 'domgross': 13.0}, {'title': '12 Years a Slav
e', 'budget': 20.0, 'domgross': 53.0}, {'title': '2 Guns', 'budget': 61.
0, 'domgross': 76.0}, {'title': '42', 'budget': 40.0, 'domgross': 95.0},
{'title': '47 Ronin', 'budget': 225.0, 'domgross': 38.0}, {'title': 'A Go
od Day to Die Hard', 'budget': 92.0, 'domgross': 67.0}, {'title': 'About
Time', 'budget': 12.0, 'domgross': 15.0}, {'title': 'After Earth', 'budge
t': 130.0, 'domgross': 61.0}, {'title': 'August: Osage County', 'budget':
25.0, 'domgross': 37.0}, {'title': 'Beautiful Creatures', 'budget': 50.0,
'domgross': 19.0}, {'title': 'Blue Jasmine', 'budget': 18.0, 'domgross':
33.0}, {'title': 'Captain Phillips', 'budget': 55.0, 'domgross': 107.0},
{'title': 'Carrie', 'budget': 30.0, 'domgross': 35.0}, {'title': 'Cloudy
with a Chance of Meatballs 2', 'budget': 78.0, 'domgross': 120.0}, {'titl
e': 'Despicable Me 2', 'budget': 76.0, 'domgross': 368.0}, {'title': 'Don
Jon', 'budget': 6.0, 'domgross': 24.0}, {'title': 'Elysium', 'budget': 12
0.0, 'domgross': 93.0}, {'title': 'Ender&#39;s Game', 'budget': 110.0, 'd
omgross': 62.0}, {'title': 'Epic', 'budget': 100.0, 'domgross': 108.0},
{'title': 'Escape Plan', 'budget': 70.0, 'domgross': 25.0}, {'title': 'Ev
il Dead', 'budget': 17.0, 'domgross': 54.0}, {'title': 'Fast and Furious
6', 'budget': 160.0, 'domgross': 239.0}, {'title': 'Frozen', 'budget': 15
0.0, 'domgross': 393.0}, {'title': 'G.I. Joe: Retaliation', 'budget': 14
0.0, 'domgross': 123.0}, {'title': 'Gangster Squad', 'budget': 60.0, 'dom
gross': 46.0}, {'title': 'Grown Ups', 'budget': 80.0, 'domgross': 134.0},
{'title': 'Her', 'budget': 23.0, 'domgross': 25.0}, {'title': 'Identity T
hief', 'budget': 35.0, 'domgross': 135.0}, {'title': 'Iron Man 3', 'budge
t': 200.0, 'domgross': 409.0}]
```

Note that, like in previous lessons, the budget is our explanatory value and the revenue is our dependent variable. Here revenue is represented as the key `domgross` .

### Plotting our data

Let's write the code to plot this data set.

As a first task, convert the budget values of our `scaled_movies` to `x_values` , and convert the domgross values of the `scaled_movies` to `y_values` .

In [5]:
```
x_values = []
for i in range(0,len(scaled_movies)):
    x_values.append(scaled_movies[i]['budget'])
print("x values:"+str(x_values))
y_values = []
for movie in scaled_movies:
    y_values.append(movie['domgross'])
print('y values:'+str(y_values))
```

```
x values:[13.0, 46.0, 20.0, 61.0, 40.0, 225.0, 92.0, 12.0, 130.0, 25.0, 5
0.0, 18.0, 55.0, 30.0, 78.0, 76.0, 6.0, 120.0, 110.0, 100.0, 70.0, 17.0,
160.0, 150.0, 140.0, 60.0, 80.0, 23.0, 35.0, 200.0]
y values:[26.0, 13.0, 53.0, 76.0, 95.0, 38.0, 67.0, 15.0, 61.0, 37.0, 19.
0, 33.0, 107.0, 35.0, 120.0, 368.0, 24.0, 93.0, 62.0, 108.0, 25.0, 54.0,
239.0, 393.0, 123.0, 46.0, 134.0, 25.0, 135.0, 409.0]
```

In [6]:
```
x_values and x_values[0] # 13.0
```

Out[6]:
```
13.0
```

In [7]:
```
y_values and y_values[0] # 26.0
```

Out[7]:
```
26.0
```

Assign a variable called `titles` equal to the titles of the movies.

In [8]:
```
titles = []
for movie in scaled_movies:
    titles.append(movie['title'])
print("titles:"+str(titles))
```

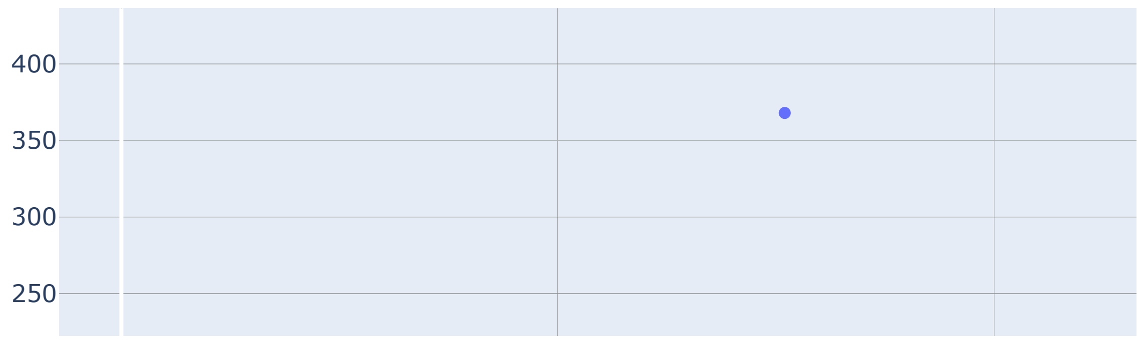
```
titles:['21 &amp; Over', 'Dredd 3D', '12 Years a Slave', '2 Guns', '42',
'47 Ronin', 'A Good Day to Die Hard', 'About Time', 'After Earth', 'Augus
t: Osage County', 'Beautiful Creatures', 'Blue Jasmine', 'Captain Phillip
s', 'Carrie', 'Cloudy with a Chance of Meatballs 2', 'Despicable Me 2',
'Don Jon', 'Elysium', 'Ender&#39;s Game', 'Epic', 'Escape Plan', 'Evil De
ad', 'Fast and Furious 6', 'Frozen', 'G.I. Joe: Retaliation', 'Gangster S
quad', 'Grown Ups', 'Her', 'Identity Thief', 'Iron Man 3']
```

In [9]:
```
titles and titles[0]
```

Out[9]:
```
'21 &amp; Over'
```

Great! Now we have the data necessary to make a trace of our data.

In [10]:
```
from plotly.offline import iplot, init_notebook_mode
init_notebook_mode(connected=True)
from graph import trace_values, plot

movies_trace = trace_values(x_values, y_values, text=titles, name='movie da

plot([movies_trace])
```




#### Plotting a regression line

Now let's add a regression line to make a prediction of output (revenue) based on an input (the budget). We'll use the following regression formula:

- $\hat{y} = mx + b$, with $m = 1.7$, and $b = 10$.
- $\hat{y} = 1.7x + 10$

Write a function called `regression_formula` that calculates our $\hat{y}$ for any provided value of $x$.

In [11]:
```
def regression_formula(x):
    return 1.7*x+10
    pass
```

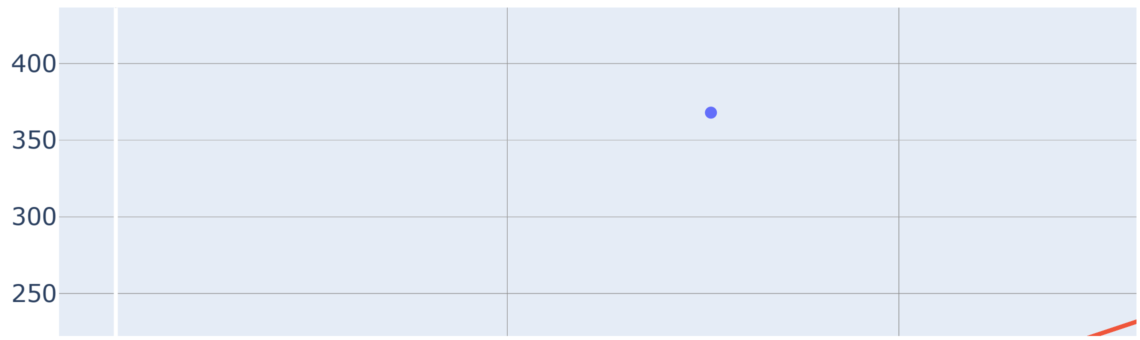
Check to see that the regression formula generates the correct outputs.

In [12]:
```
print(regression_formula(100)) # 180.0
print(regression_formula(250)) # 435.0
```

```
180.0
435.0
```

Let's plot the data as well as the regression line to get a sense of what we are looking at.

In [13]:
```
from plotly.offline import iplot, init_notebook_mode
init_notebook_mode(connected=True)
from graph import trace_values, m_b_trace, plot

if x_values and y_values:
    movies_trace = trace_values(x_values, y_values, text=titles, name='movi
    regression_trace = m_b_trace(1.7, 10, x_values, name='estimated revenue
    plot([movies_trace, regression_trace])
```




## Calculating errors of a regression Line

Now that we have our regression formula, we can move towards calculating the error. We provide a function called `y_actual` that given a data set of `x_values` and `y_values` , finds the actual y value, provided a value of `x` .

In [14]:
```
def y_actual(x, x_values, y_values):
    combined_values = list(zip(x_values, y_values))
    print("combined:"+str(combined_values))
    point_at = list(filter(lambda point: point[0] == x,combined_values))
    print("point(1):"+str(point_at))
    print("point(2):"+str(point_at[0]))
    print("point(3):"+str(point_at[0][0]))
    print("point(4):"+str(point_at[0][1]))
    point_at_x = list(filter(lambda point: point[0] == x,combined_values))[
    return point_at_x[1]
```

In [15]:
```
x_values and y_values and y_actual(13, x_values, y_values) # 26.0
```

```
combined:[(13.0, 26.0), (46.0, 13.0), (20.0, 53.0), (61.0, 76.0), (40.0,
95.0), (225.0, 38.0), (92.0, 67.0), (12.0, 15.0), (130.0, 61.0), (25.0, 3
7.0), (50.0, 19.0), (18.0, 33.0), (55.0, 107.0), (30.0, 35.0), (78.0, 12
0.0), (76.0, 368.0), (6.0, 24.0), (120.0, 93.0), (110.0, 62.0), (100.0, 1
08.0), (70.0, 25.0), (17.0, 54.0), (160.0, 239.0), (150.0, 393.0), (140.
0, 123.0), (60.0, 46.0), (80.0, 134.0), (23.0, 25.0), (35.0, 135.0), (20
0.0, 409.0)]
point(1):[(13.0, 26.0)]
point(2):(13.0, 26.0)
point(3):13.0
point(4):26.0
```

Out[15]:
```
26.0
```

Write a function called `error` , that given a list of `x_values` , and a list of `y_values` , the values `m` and `b` of a regression line, and a value of `x` , returns the error at that x value. Remember $\varepsilon_i = y_i - \hat{y}_i$.

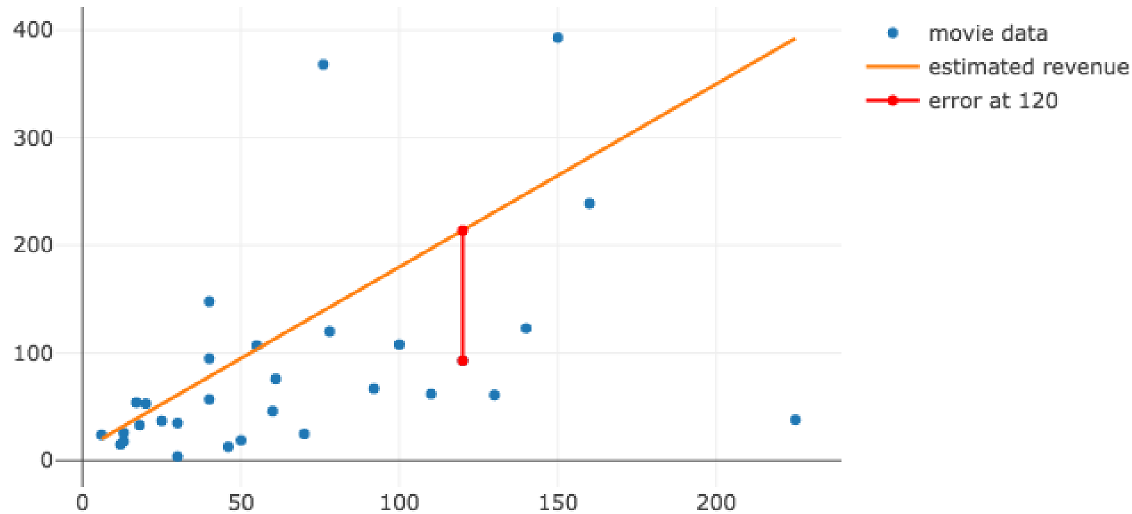
In [16]:
```
def error(x_values, y_values, m, b, x):
    y_fitted = m*x+b
    return y_actual(x, x_values, y_values) - y_fitted
    pass
```

In [17]:
```
error(x_values, y_values, 1.7, 10, 13) # -6.099999999999994
```

```
combined:[(13.0, 26.0), (46.0, 13.0), (20.0, 53.0), (61.0, 76.0), (40.0,
95.0), (225.0, 38.0), (92.0, 67.0), (12.0, 15.0), (130.0, 61.0), (25.0, 3
7.0), (50.0, 19.0), (18.0, 33.0), (55.0, 107.0), (30.0, 35.0), (78.0, 12
0.0), (76.0, 368.0), (6.0, 24.0), (120.0, 93.0), (110.0, 62.0), (100.0, 1
08.0), (70.0, 25.0), (17.0, 54.0), (160.0, 239.0), (150.0, 393.0), (140.
0, 123.0), (60.0, 46.0), (80.0, 134.0), (23.0, 25.0), (35.0, 135.0), (20
0.0, 409.0)]
point(1):[(13.0, 26.0)]
point(2):(13.0, 26.0)
point(3):13.0
point(4):26.0
```

Out[17]:
```
-6.099999999999994
```

Now that we have a formula to calculate our errors, write a function called `error_line_trace` that returns a trace of an error at a given point. So for a given movie budget, it will display the difference between the regression line and the actual movie revenue.




Ok, so the function `error_line_trace` takes our dataset of `x_values` as the first argument and `y_values` as the second argument. It also takes in values of $m$ and $b$ as the next two arguments to represent the regression line we will calculate errors from. Finally, the last argument is the value $x$ it is drawing an error for.

The return value is a dictionary that represents a trace, and looks like the following:

```
{'marker': {'color': 'red'},
 'mode': 'lines',
 'name': 'error at 120',
 'x': [120, 120],
 'y': [93.0, 214.0]}
```

The trace represents the error line above. The data in `x` and `y` represent the starting point and ending point of the error line. Note that the x value is the same for the starting and ending point, just as it is for each vertical line. It's just the y values that differ - representing the actual value and the expected value. The mode of the trace equals `'lines'`.

In [18]:
```
def error_line_trace(x_values, y_values, m, b, x):
    combined_values = list(zip(x_values, y_values))
    print("combine data:"+str(combined_values))
    error_line_dict = dict()
    x_values   = []
    y_values   = []
    point      = list(filter(lambda point: point[0] == x,combined_values))
    point_at   = list(filter(lambda point: point[0] == x,combined_values))[
    point_at_x = list(filter(lambda point: point[0] == x,combined_values))[
    point_at_y = list(filter(lambda point: point[0] == x,combined_values))[
    print("point:"+str(point))
    print("point_at:"+str(point_at))
    print("point_at_x:"+str(point_at_x))
    print("point_at_y:"+str(point_at_y))
    for i in range(0,2):
        x_values.append(point_at_x)
        if (i==0):
            y_values.append(point_at_y) # y_actual
        else:
            y_values.append(m*x+b) # y_fitted
    print("x:"+str(x_values))
    print("y:"+str(y_values))
    error_line_dict.update({'marker':{'color':'red'},'mode':'lines','name':
    return error_line_dict
    pass
```

In [19]:
```
error_at_120m = error_line_trace(x_values, y_values, 1.7, 10, 120)

# {'marker': {'color': 'red'},
#  'mode': 'lines',
#  'name': 'error at 120',
#  'x': [120, 120],
#  'y': [93.0, 214.0]}
error_at_120m
```

```
combine data:[(13.0, 26.0), (46.0, 13.0), (20.0, 53.0), (61.0, 76.0), (4
0.0, 95.0), (225.0, 38.0), (92.0, 67.0), (12.0, 15.0), (130.0, 61.0), (2
5.0, 37.0), (50.0, 19.0), (18.0, 33.0), (55.0, 107.0), (30.0, 35.0), (78.
0, 120.0), (76.0, 368.0), (6.0, 24.0), (120.0, 93.0), (110.0, 62.0), (10
0.0, 108.0), (70.0, 25.0), (17.0, 54.0), (160.0, 239.0), (150.0, 393.0),
(140.0, 123.0), (60.0, 46.0), (80.0, 134.0), (23.0, 25.0), (35.0, 135.0),
(200.0, 409.0)]
point:[(120.0, 93.0)]
point_at:(120.0, 93.0)
point_at_x:120.0
point_at_y:93.0
x:[120.0, 120.0]
y:[93.0, 214.0]
```

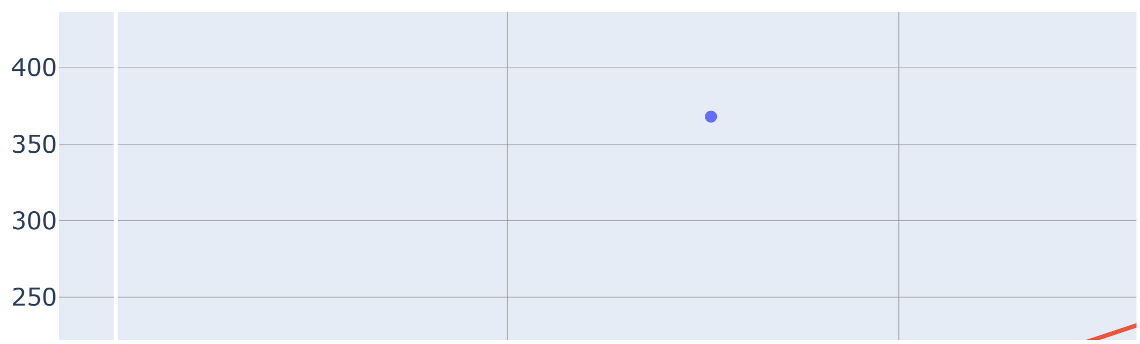
Out[19]:
```
{'marker': {'color': 'red'},
 'mode': 'lines',
 'name': 'error at 120',
 'x': [120.0, 120.0],
 'y': [93.0, 214.0]}
```

We just ran the our function to draw a trace of the error for the movie Elysium. Let's see how it looks.

In [20]:
```
scaled_movies[17]
```

Out[20]:
```
{'title': 'Elysium', 'budget': 120.0, 'domgross': 93.0}
```

In [21]:
```
from plotly.offline import iplot, init_notebook_mode
init_notebook_mode(connected=True)
from graph import trace_values, m_b_trace, plot
if x_values and y_values:
    movies_trace = trace_values(x_values, y_values, text=titles, name='movi
    regression_trace = m_b_trace(1.7, 10, x_values, name='estimated revenue
    plot([movies_trace, regression_trace, error_at_120m])
```




From there, we can write a function called `error_line_traces` , that takes in a list of `x_values` as an argument, `y_values` as an argument, and returns a list of traces for every x value provided.

In [22]:
```
def error_line_traces(x_values, y_values, m, b):
    error_line_list = []
    for i in range(0,len(x_values)):
        print("x:"+str(x_values[i]))
        error_line_list.append(error_line_trace(x_values, y_values, m, b, x
    print("error_line_list:"+str(error_line_list))
    return error_line_list
    pass
```

In [23]:
```
errors_for_regression = error_line_traces(x_values, y_values, 1.7, 10)
```

```
(140.0, 123.0), (60.0, 46.0), (80.0, 134.0), (23.0, 25.0), (35.0, 135.0),
(200.0, 409.0)]
point:[(13.0, 26.0)]
point_at:(13.0, 26.0)
point_at_x:13.0
point_at_y:26.0
x:[13.0, 13.0]
y:[26.0, 32.099999999999994]
x:46.0
combine data:[(13.0, 26.0), (46.0, 13.0), (20.0, 53.0), (61.0, 76.0), (4
0.0, 95.0), (225.0, 38.0), (92.0, 67.0), (12.0, 15.0), (130.0, 61.0), (2
5.0, 37.0), (50.0, 19.0), (18.0, 33.0), (55.0, 107.0), (30.0, 35.0), (78.
0, 120.0), (76.0, 368.0), (6.0, 24.0), (120.0, 93.0), (110.0, 62.0), (10
0.0, 108.0), (70.0, 25.0), (17.0, 54.0), (160.0, 239.0), (150.0, 393.0),
(140.0, 123.0), (60.0, 46.0), (80.0, 134.0), (23.0, 25.0), (35.0, 135.0),
(200.0, 409.0)]
point:[(46.0, 13.0)]
point_at:(46.0, 13.0)
point_at_x:46.0
```

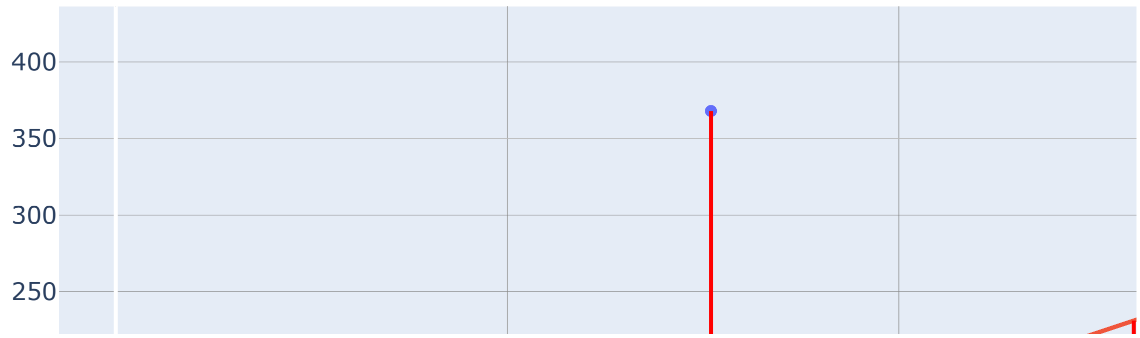
In [24]:
```
errors_for_regression and len(errors_for_regression) # 30
```

Out[24]:
```
30
```

In [25]:
```
errors_for_regression and errors_for_regression[-1]

# {'x': [200.0, 200.0],
#  'y': [409.0, 350.0],
#  'mode': 'lines',
#  'marker': {'color': 'red'},
#  'name': 'error at 200.0'}
```

Out[25]:
```
{'marker': {'color': 'red'},
 'mode': 'lines',
 'name': 'error at 120',
 'x': [200.0, 200.0],
 'y': [409.0, 350.0]}
```

In [26]:
```
from plotly.offline import iplot, init_notebook_mode
init_notebook_mode(connected=True)

from graph import trace_values, m_b_trace, plot

if x_values and y_values:
    movies_trace = trace_values(x_values, y_values, text=titles, name='movi
    regression_trace = m_b_trace(1.7, 10, x_values, name='estimated revenue
    plot([movies_trace, regression_trace, *errors_for_regression])
```




> Don't worry about some of the points that don't have associated error lines. It is a complication with Plotly and not our functions.

## Calculating RSS

Now write a function called `squared_error` , that given a value of x, returns the squared error at that x value.

$\varepsilon_i{}^2 = (y_i - \hat{y}_i)^2$

In [27]:
```
def squared_error(x_values, y_values, m, b, x):
    combo_data = list(zip(x_values,y_values))
    pt       = list(filter(lambda p:p[0]==x,combo_data))
    pt_at    = pt[0]
    pt_at_x  = pt[0][0]
    pt_at_y  = pt[0][1]
    y_fit    = m*x+b
    y_act    = pt_at_y
    return pow(y_act-y_fit,2)
    pass
```

In [28]:
```
x_values and y_values and squared_error(x_values, y_values, 1.7, 10, x_valu
```

Out[28]:
```
37.20999999999993
```

Now write a function that will iterate through the x and y values to create a list of squared errors at each point, $(x_i, y_i)$ of the dataset.

In [29]:
```
import math
def squared_errors(x_values, y_values, m, b):
    sqrt_err_list = []
    for i in range(0,len(x_values)):
        sqrt_err_list.append(squared_error(x_values, y_values, m, b, x_valu
    print(sqrt_err_list)
    return sqrt_err_list
    pass
```

In [30]:
```
x_values and y_values and squared_errors(x_values, y_values, 1.7, 10)
```

Out[30]:
```
[37.20999999999993,
 5655.040000000001,
 81.0,
 1421.2900000000002,
 289.0,
```

Next, write a function called `residual_sum_squares` that, provided a list of x_values, y_values, and the m and b values of a regression line, returns the sum of the squared error for the movies in our dataset.

In [31]:
```
def residual_sum_squares(x_values, y_values, m, b):
    return sum(squared_errors(x_values, y_values, m, b))
    pass
```

In [32]:
```
residual_sum_squares(x_values, y_values, 1.7, 10) # 327612.2800000001
```

```
[37.20999999999993, 5655.040000000001, 81.0, 1421.2900000000002, 289.0, 1
25670.25, 9880.36, 237.15999999999997, 28900.0, 240.25, 5776.0, 57.759999
99999991, 12.25, 676.0, 510.75999999999976, 52349.44, 14.440000000000005,
14641.0, 18225.0, 5184.0, 10816.0, 228.01000000000005, 1849.0, 16384.0, 1
5625.0, 4356.0, 144.0, 580.8100000000001, 4290.25, 3481.0]
```

Out[32]:
```
327612.2800000001
```

Finally, write a function called `root_mean_squared_error` that calculates the RMSE for the movies in the dataset, provided the same parameters as RSS. Remember that `root_mean_squared_error` is a way for us to measure the approximate error per data point.

In [33]:
```
import math
def root_mean_squared_error(x_values, y_values, m, b):
    return math.sqrt(residual_sum_squares(x_values, y_values, m, b)/len(x_v
```

In [34]:
```
root_mean_squared_error(x_values, y_values, 1.7, 10) # 104.50076235766578
```

```
[37.20999999999993, 5655.040000000001, 81.0, 1421.2900000000002, 289.0, 1
25670.25, 9880.36, 237.15999999999997, 28900.0, 240.25, 5776.0, 57.759999
99999991, 12.25, 676.0, 510.75999999999976, 52349.44, 14.440000000000005,
14641.0, 18225.0, 5184.0, 10816.0, 228.01000000000005, 1849.0, 16384.0, 1
5625.0, 4356.0, 144.0, 580.8100000000001, 4290.25, 3481.0]
```

Out[34]:
```
104.50076235766578
```

**Some functions for your understanding**

Now we'll provide a couple functions for you. Note that we can represent multiple regression lines by a list of m and b values:

In [35]:
```
regression_lines = [(1.7, 10), (1.9, 20)]
```

Then we can return a list of the regression lines along with the associated RMSE.

In [36]:
```
def root_mean_squared_errors(x_values, y_values, regression_lines):
    errors = []
    for regression_line in regression_lines:
        error = root_mean_squared_error(x_values, y_values, regression_line
        errors.append([regression_line[0], regression_line[1], round(error,
    return errors
```

Now let's generate the RMSE values for each of these lines.

In [37]:
```
x_values and y_values and root_mean_squared_errors(x_values, y_values, regr
```

```
[37.20999999999993, 5655.040000000001, 81.0, 1421.2900000000002, 289.0, 1
25670.25, 9880.36, 237.15999999999997, 28900.0, 240.25, 5776.0, 57.759999
99999991, 12.25, 676.0, 510.75999999999976, 52349.44, 14.440000000000005,
14641.0, 18225.0, 5184.0, 10816.0, 228.01000000000005, 1849.0, 16384.0, 1
5625.0, 4356.0, 144.0, 580.8100000000001, 4290.25, 3481.0]
[349.6900000000001, 8911.359999999999, 25.0, 3588.0099999999975, 1.0, 167
690.25, 16332.839999999997, 772.8399999999998, 42436.0, 930.25, 9216.0, 4
49.4399999999998, 306.25, 1764.0, 2323.239999999999, 41452.96, 54.7599999
9999998, 24025.0, 27889.0, 10404.0, 16384.0, 2.8900000000000095, 7225.0,
7744.0, 26569.0, 7744.0, 1444.0, 1497.6899999999996, 2352.25, 81.0]
```

Out[37]:
```
[[1.7, 10, 105.0], [1.9, 20, 120.0]]
```

Now we'll provide a couple functions for you:

- a function called `trace_rmse`, that builds a bar chart displaying the value of the RMSE. The return value is a dictionary with keys of `x` and `y`, both which point to lists. The $x$ key points to a list with one element, a string containing each regression line's m and b value. The $y$ key points to a list of the RMSE values for each corresponding regression line.

In [38]:
```
import plotly.graph_objs as go

def trace_rmse(x_values, y_values, regression_lines):
    errors = root_mean_squared_errors(x_values, y_values, regression_lines)
    x_values_bar = list(map(lambda error: 'm: ' + str(error[0]) + ' b: ' +
    y_values_bar = list(map(lambda error: error[-1], errors))
    return dict(
        x=x_values_bar,
        y=y_values_bar,
        type='bar'
    )


x_values and y_values and trace_rmse(x_values, y_values, regression_lines)
```

```
[37.20999999999993, 5655.040000000001, 81.0, 1421.2900000000002, 289.0, 1
25670.25, 9880.36, 237.15999999999997, 28900.0, 240.25, 5776.0, 57.759999
99999991, 12.25, 676.0, 510.75999999999976, 52349.44, 14.440000000000005,
14641.0, 18225.0, 5184.0, 10816.0, 228.01000000000005, 1849.0, 16384.0, 1
5625.0, 4356.0, 144.0, 580.8100000000001, 4290.25, 3481.0]
[349.6900000000001, 8911.359999999999, 25.0, 3588.0099999999975, 1.0, 167
690.25, 16332.839999999997, 772.8399999999998, 42436.0, 930.25, 9216.0, 4
49.4399999999998, 306.25, 1764.0, 2323.239999999999, 41452.96, 54.7599999
9999998, 24025.0, 27889.0, 10404.0, 16384.0, 2.8900000000000095, 7225.0,
7744.0, 26569.0, 7744.0, 1444.0, 1497.6899999999996, 2352.25, 81.0]
```

Out[38]:
```
{'x': ['m: 1.7 b: 10', 'm: 1.9 b: 20'], 'y': [105.0, 120.0], 'type': 'ba
r'}
```

Once this is built, we can create a subplot showing the two regression lines, as well as the related RMSE for each line.

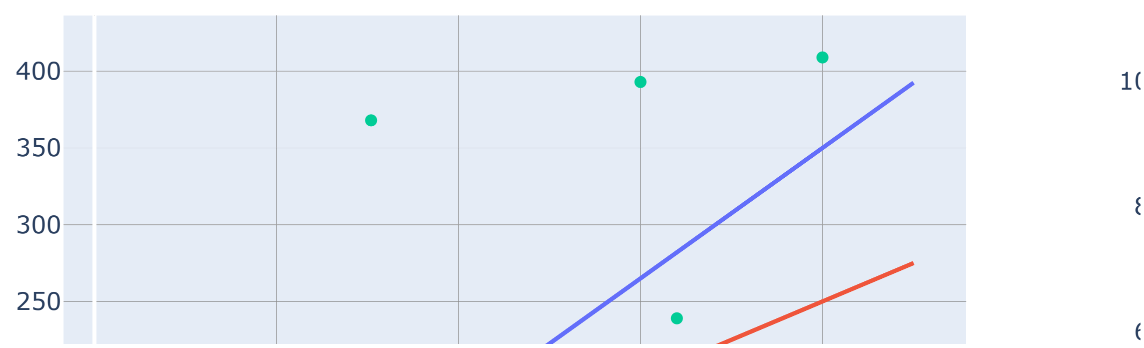
In [39]:

```
import plotly
from plotly.offline import iplot
from plotly import tools
import plotly.graph_objs as go

def regression_and_rss(scatter_trace, regression_traces, rss_calc_trace):
    fig = tools.make_subplots(rows=1, cols=2)
    for reg_trace in regression_traces:
        fig.append_trace(reg_trace, 1, 1)
    fig.append_trace(scatter_trace, 1, 1)
    fig.append_trace(rss_calc_trace, 1, 2)
    iplot(fig)
```

In [40]:
```
# add more regression lines here, by adding new elements to the list
gression_lines = [(1.7, 10), (1, 50)]

 x_values and y_values:
  regression_traces = list(map(lambda line: m_b_trace(line[0], line[1], x_va

  scatter_trace = trace_values(x_values, y_values, text=titles, name='movie
  rmse_calc_trace = trace_rmse(x_values, y_values, regression_lines)

  regression_and_rss(scatter_trace, regression_traces, rmse_calc_trace)
```

```
[37.20999999999993, 5655.040000000001, 81.0, 1421.2900000000002, 289.0, 1
25670.25, 9880.36, 237.15999999999997, 28900.0, 240.25, 5776.0, 57.759999
99999991, 12.25, 676.0, 510.75999999999976, 52349.44, 14.440000000000005,
14641.0, 18225.0, 5184.0, 10816.0, 228.01000000000005, 1849.0, 16384.0, 1
5625.0, 4356.0, 144.0, 580.8100000000001, 4290.25, 3481.0]
[1369.0, 6889.0, 289.0, 1225.0, 25.0, 56169.0, 5625.0, 2209.0, 14161.0, 1
444.0, 6561.0, 1225.0, 4.0, 2025.0, 64.0, 58564.0, 1024.0, 5929.0, 9604.
0, 1764.0, 9025.0, 169.0, 841.0, 37249.0, 4489.0, 4096.0, 16.0, 2304.0, 2
500.0, 25281.0]
```

```
/opt/conda/envs/learn-env/lib/python3.6/site-packages/plotly/tools.py:46
5: DeprecationWarning:

plotly.tools.make_subplots is deprecated, please use plotly.subplots.make
_subplots instead
```



As we can see above, the second line (m: 1.0, b: 50) has the lower RMSE. We thus can conclude that the second line "fits" our set of movie data better than the first line. Ultimately, our goal will be to choose the regression line with the lowest RSME or RSS. We will learn how to accomplish this goal in the following lessons and labs.