Applying Gradient Descent Lab

Introduction

In this lab, we'll put our knowledge about data science to the test. We will have access to functions in the [error (https://github.com/learn-co-curriculum/applying-gradient-descent-lab/blob/master/error.py)](https://github.com/learn-co-curriculum/applying-gradient-descent-lab/blob/master/error.py), [graph (https://github.com/learn-co-curriculum/applying-gradient-descent-lab/blob/master/graph.py)](https://github.com/learn-co-curriculum/applying-gradient-descent-lab/blob/master/graph.py) and [linear equations (https://github.com/learn-co-curriculum/applying-gradient-descent-lab/blob/master/linear_equations.py)](https://github.com/learn-co-curriculum/applying-gradient-descent-lab/blob/master/linear_equations.py) libraries that we previously wrote.

This is our task: We are an employee for *Good Lion Studios*. For *Good Lion*, our job is first to gather, explore, and format our data so that we can build a regression line of this data. Then we will work through various attempts of building out these regression lines. By the end of this lab, we should have a working version that we can proudly show to our manager.

Learning Objectives

- Review how to use built-in functions, like filter and map, to clean data
- Evaluate the quality of regression lines using Residual Sum of Squares (RSS)
- Review how RSS changes with varying values of the slope and y-intercept of a regression line
- Implement gradient descent to find a "best fit" regression line

This lesson is an opportunity to review the concepts explained in our introduction to machine learning section and practice what we recently learned about gradient descent to find an optimal regression line.

> **Use the round method**: For many of the methods, we round down the return value to two decimal places. We can do so by using the **round** function, as in **round(12.1212, 2)** to round 12.1212 to 12.12. We did this to allow for slight differences between our results and expectations. So when we see our data differing from the expectation, check if using the **round** function helps.

Determining Quality

Retrieve the data

First, let's get some movies from the FiveThirtyEight dataset [provided here (https://raw.githubusercontent.com/fivethirtyeight/data/master/bechdel/movies.csv)](https://raw.githubusercontent.com/fivethirtyeight/data/master/bechdel/movies.csv). The code below parses this data from the csv file and saves it to the `movies` variable as a list of dictionaries.

In [1]:
```
import pandas as pd

def parse_file(fileName):
    movies_df = pd.read_csv(fileName)
    print(movies_df.keys())
    return movies_df.to_dict('records')

movies = parse_file('https://raw.githubusercontent.com/fivethirtyeight/data
print(type(movies)) # list
print(len(movies)) # 1794
```

```
Index(['year', 'imdb', 'title', 'test', 'clean_test', 'binary', 'budget',
       'domgross', 'intgross', 'code', 'budget_2013$', 'domgross_2013$',
       'intgross_2013$', 'period code', 'decade code'],
      dtype='object')
<class 'list'>
1794
```

In [2]:
```
movies_df_update = pd.read_csv('https://raw.githubusercontent.com/fivethirt
print(movies_df_update.shape)
print(movies_df_update.head())
movies_id = list(set(movies_df_update.imdb))
print(movies_id)
```

```
(1794, 15)
   year       imdb              title             test clean_test binary
\
0  2013  tt1711425     21 &amp; Over           notalk     notalk   FAIL
1  2012  tt1343727           Dredd 3D      ok-disagree         ok   PASS
2  2013  tt2024544  12 Years a Slave  notalk-disagree     notalk   FAIL
3  2013  tt1272878             2 Guns           notalk     notalk   FAIL
4  2013  tt0453562                 42              men        men   FAIL

     budget    domgross     intgross      code  budget_2013$  domgross_20
13$  \
0  13000000  25682380.0   42195766.0  2013FAIL      13000000      2568238
0.0
1  45000000  13414714.0   40868994.0  2012PASS      45658735      1361108
6.0
2  20000000  53107035.0  158607035.0  2013FAIL      20000000      5310703
5.0
3  61000000  75612460.0  132493015.0  2013FAIL      61000000      7561246
0.0
```

As you can see, this list is full of 1794 dictionaries, each one representing a movie.

Explore the data

Let's take a look at that first movie in the dataset.

In [3]:
```
print(movies[0])
print('\r\n')
print(movies_df_update.to_dict('records')[0])
```

```
{'year': 2013, 'imdb': 'tt1711425', 'title': '21 &amp; Over', 'test': 'no
talk', 'clean_test': 'notalk', 'binary': 'FAIL', 'budget': 13000000, 'dom
gross': 25682380.0, 'intgross': 42195766.0, 'code': '2013FAIL', 'budget_2
013$': 13000000, 'domgross_2013$': 25682380.0, 'intgross_2013$': 4219576
6.0, 'period code': 1.0, 'decade code': 1.0}


{'year': 2013, 'imdb': 'tt1711425', 'title': '21 &amp; Over', 'test': 'no
talk', 'clean_test': 'notalk', 'binary': 'FAIL', 'budget': 13000000, 'dom
gross': 25682380.0, 'intgross': 42195766.0, 'code': '2013FAIL', 'budget_2
013$': 13000000, 'domgross_2013$': 25682380.0, 'intgross_2013$': 4219576
6.0, 'period code': 1.0, 'decade code': 1.0}
```

Here we can see the data available for each movie. The information most relevant for our task is:

1. `budget_2013$` is the budget adjusted for inflation in 2013 dollars
2. `domgross_2013$` is the domestic revenue adjusted for inflation in 2013 dollars
3. `intgross_2013$` is the international revenue adjusted for inflation in 2013 dollars

Cleaning our data

1. Handle missing data

Now, let's look at the values associated with these attributes. The first movie looks good since it has nice data for each of these attributes. Unfortunately, the data for some other movies might not be so fun to play with. Let's remove the movies whose `domgross_2013` points to values of `nan`, which stands for "not a number". This data is missing. There are only a few pieces of missing data here, so we can safely remove these movies without causing too much damage.

In [4]:
```
import math
domgross_miss_list = list(filter(lambda movie: math.isnan(movie['domgross_2
print(domgross_miss_list)
```

[{'year': 2013, 'imdb': 'tt2005374', 'title': 'The Frozen Ground', 'tes
t': 'nowomen-disagree', 'clean_test': 'nowomen', 'binary': 'FAIL', 'budge
t': 19200000, 'domgross': nan, 'intgross': nan, 'code': '2013FAIL', 'budg
et_2013$': 19200000, 'domgross_2013$': nan, 'intgross_2013$': nan, 'perio
d code': 1.0, 'decade code': 1.0}, {'year': 2011, 'imdb': 'tt1422136', 't
itle': 'A Lonely Place to Die', 'test': 'ok', 'clean_test': 'ok', 'binar
y': 'PASS', 'budget': 4000000, 'domgross': nan, 'intgross': 442550.0, 'co
de': '2011PASS', 'budget_2013$': 4142763, 'domgross_2013$': nan, 'intgros
s_2013$': 458345.0, 'period code': 1.0, 'decade code': 1.0}, {'year': 201
1, 'imdb': 'tt1701990', 'title': 'Detention', 'test': 'ok', 'clean_test':
'ok', 'binary': 'PASS', 'budget': 10000000, 'domgross': nan, 'intgross':
nan, 'code': '2011PASS', 'budget_2013$': 10356908, 'domgross_2013$': nan,
'intgross_2013$': nan, 'period code': 1.0, 'decade code': 1.0}, {'year':
2010, 'imdb': 'tt1216520', 'title': 'Womb', 'test': 'ok', 'clean_test':
'ok', 'binary': 'PASS', 'budget': 13000000, 'domgross': nan, 'intgross':
nan, 'code': '2010PASS', 'budget_2013$': 13887014, 'domgross_2013$': nan,
'intgross_2013$': nan, 'period code': 1.0, 'decade code': 1.0}, {'year':
2009, 'imdb': 'tt1024744', 'title': 'I Come with the Rain', 'test': 'nowo
men', 'clean_test': 'nowomen', 'binary': 'FAIL', 'budget': 18000000, 'dom
gross': 0.0, 'intgross': 627422.0, 'code': '2009FAIL', 'budget_2013$': 19
543169, 'domgross_2013$': nan, 'intgross_2013$': 681212.0, 'period code':
2.0, 'decade code': 2.0}, {'year': 2009, 'imdb': 'tt1068678', 'title': 'V
eronika Decides to Die', 'test': 'ok', 'clean_test': 'ok', 'binary': 'PAS
S', 'budget': 9000000, 'domgross': nan, 'intgross': nan, 'code': '2009PAS
S', 'budget_2013$': 9771584, 'domgross_2013$': nan, 'intgross_2013$': na
n, 'period code': 2.0, 'decade code': 2.0}, {'year': 2009, 'imdb': 'tt044
8182', 'title': 'Yesterday Was a Lie', 'test': 'ok', 'clean_test': 'ok',
'binary': 'PASS', 'budget': 200000, 'domgross': nan, 'intgross': nan, 'co
de': '2009PASS', 'budget_2013$': 217146, 'domgross_2013$': nan, 'intgross
_2013$': nan, 'period code': 2.0, 'decade code': 2.0}, {'year': 2008, 'im
db': 'tt0489018', 'title': 'Day of the Dead', 'test': 'ok', 'clean_test':
'ok', 'binary': 'PASS', 'budget': 18000000, 'domgross': nan, 'intgross':
nan, 'code': '2008PASS', 'budget_2013$': 19480614, 'domgross_2013$': nan,
'intgross_2013$': nan, 'period code': 2.0, 'decade code': 2.0}, {'year':
2008, 'imdb': 'tt0942903', 'title': 'Stargate: The Ark of Truth', 'test':
'dubious-disagree', 'clean_test': 'dubious', 'binary': 'FAIL', 'budget':
7000000, 'domgross': nan, 'intgross': nan, 'code': '2008FAIL', 'budget_20
13$': 7575794, 'domgross_2013$': nan, 'intgross_2013$': nan, 'period cod
e': 2.0, 'decade code': 2.0}, {'year': 2008, 'imdb': 'tt0882978', 'titl
e': 'Three Kingdoms: Resurrection of the Dragon', 'test': 'notalk', 'clea
n_test': 'notalk', 'binary': 'FAIL', 'budget': 20000000, 'domgross': nan,
'intgross': 22139590.0, 'code': '2008FAIL', 'budget_2013$': 21645126, 'do
mgross_2013$': nan, 'intgross_2013$': 23960711.0, 'period code': 2.0, 'de
cade code': 2.0}, {'year': 2007, 'imdb': 'tt1038988', 'title': '[Rec]',
'test': 'ok', 'clean_test': 'ok', 'binary': 'PASS', 'budget': 2100000, 'd
omgross': nan, 'intgross': 27117954.0, 'code': '2007PASS', 'budget_2013
$': 2359441, 'domgross_2013$': nan, 'intgross_2013$': 30468201.0, 'period
code': 2.0, 'decade code': 2.0}, {'year': 2007, 'imdb': 'tt0756683', 'tit
le': 'The Man from Earth', 'test': 'notalk', 'clean_test': 'notalk', 'bin
ary': 'FAIL', 'budget': 200000, 'domgross': nan, 'intgross': nan, 'code':
'2007FAIL', 'budget_2013$': 224709, 'domgross_2013$': nan, 'intgross_2013
$': nan, 'period code': 2.0, 'decade code': 2.0}, {'year': 2007, 'imdb':
'tt0496436', 'title': 'White Noise 2: The Light', 'test': 'notalk', 'clea

n_test': 'notalk', 'binary': 'FAIL', 'budget': 10000000, 'domgross': nan, 'intgross': 8243567.0, 'code': '2007FAIL', 'budget_2013$': 11235435, 'domgross_2013$': nan, 'intgross_2013$': 9262006.0, 'period code': 2.0, 'decade code': 2.0}, {'year': 2006, 'imdb': 'tt0416496', 'title': 'Bandidas', 'test': 'ok', 'clean_test': 'ok', 'binary': 'PASS', 'budget': 35000000, 'domgross': nan, 'intgross': 18400000.0, 'code': '2006PASS', 'budget_2013$': 40452872, 'domgross_2013$': nan, 'intgross_2013$': 21266653.0, 'period code': 2.0, 'decade code': 2.0}, {'year': 2006, 'imdb': 'tt0490166', 'title': 'London To Brighton', 'test': 'ok', 'clean_test': 'ok', 'binary': 'PASS', 'budget': 825000, 'domgross': nan, 'intgross': 610776.0, 'code': '2006PASS', 'budget_2013$': 953532, 'domgross_2013$': nan, 'intgross_2013$': 705933.0, 'period code': 2.0, 'decade code': 2.0}, {'year': 1985, 'imdb': 'tt0088993', 'title': 'Day of the Dead', 'test': 'nowomen', 'clean_test': 'nowomen', 'binary': 'FAIL', 'budget': 18000000, 'domgross': nan, 'intgross': nan, 'code': '1985FAIL', 'budget_2013$': 38971004, 'domgross_2013$': nan, 'intgross_2013$': nan, 'period code': nan, 'decade code': nan}, {'year': 1985, 'imdb': 'tt0091578', 'title': 'My Beautiful Laundrette', 'test': 'men', 'clean_test': 'men', 'binary': 'FAIL', 'budget': 400000, 'domgross': nan, 'intgross': nan, 'code': '1985FAIL', 'budget_2013$': 866022, 'domgross_2013$': nan, 'intgross_2013$': nan, 'period code': nan, 'decade code': nan}, {'year': 1972, 'imdb': 'tt0068156', 'title': '1776', 'test': 'notalk', 'clean_test': 'notalk', 'binary': 'FAIL', 'budget': 4000000, 'domgross': nan, 'intgross': nan, 'code': '1972FAIL', 'budget_2013$': 22288557, 'domgross_2013$': nan, 'intgross_2013$': nan, 'period code': nan, 'decade code': nan}]

Write a function called `remove_movies_missing_data` that returns the subset of movies that don't have `nan`.

> To do so, you can import the math library and make use of the `math.isnan` method. More information about this method can be [found here (https://stackoverflow.com/questions/944700/how-can-i-check-for-nan-in-python)](https://stackoverflow.com/questions/944700/how-can-i-check-for-nan-in-python).

In [5]:
```
import math

def remove_movies_missing_data(movies):
    good_movies_list = list(filter(lambda m: not math.isnan(m['domgross_201
    return good_movies_list
    pass

def drop_allmovies_missing_data(movies):
    movies_df_frame = pd.DataFrame(movies) # list of dictionaries to DataFr
    print(movies_df_frame)
    print('\r\n')
    good_movies_df = movies_df_frame.dropna() # remove movies with missing
    print(good_movies_df)
    good_movies_list = good_movies_df.values.tolist() # turn DataFrame back
    print('\r\n')
    print(good_movies_list)
    return good_movies_list
    pass
```

In [6]:
```
parsed_movies = remove_movies_missing_data(movies) or []
droped_movies = drop_allmovies_missing_data(movies) or []
```

```
...    ...    ...                                        ...
1789  1971  tt0067741                                  Shaft
1790  1971  tt0067800                             Straw Dogs
1791  1971  tt0067116                  The French Connection
1792  1971  tt0067992  Willy Wonka &amp; the Chocolate Factory
1793  1970  tt0065466           Beyond the Valley of the Dolls

                  test clean_test binary    budget    domgross     intgros
s  \
0               notalk     notalk   FAIL  13000000  25682380.0   42195766.
0
1          ok-disagree         ok   PASS  45000000  13414714.0   40868994.
0
2      notalk-disagree     notalk   FAIL  20000000  53107035.0  158607035.
0
3               notalk     notalk   FAIL  61000000  75612460.0  132493015.
0
4                  men        men   FAIL  40000000  95020213.0   95020213.
0
...                ...        ...    ...       ...         ...
```

After writing the `remove_movies_missing_data` function, notice that we have reduced the number of movies down from 1794 to 1776 movies.

In [7]:
```
print(len(parsed_movies)) # 1776
print(len(droped_movies))
```

```
1776
1600
```

Also, we can see that no movies with a `domgross_2013` value of `nan` are included.

In [8]:
```
list(filter(lambda movie: math.isnan(movie['domgross_2013$']),parsed_movies
```

Out[8]:
```
[]
```

2. Changing the scale of our data

Currently, our data has some very large numbers:

In [9]:
```
movies[0]['budget']
```

Out[9]:
```
13000000
```

It takes some time to figure out if the number above is 13 million. It would be frustrating to count all of the zeros whenever we come across another set of movie budgets and revenues.

To make things simpler, let's divide both our budget and revenue numbers for each movie by 1 million. It will make some of our future calculations easier to interpret. The attributes that we can scale down are `budget`, `budget_2013$`, `domgross`, `domgross_2013$`, `intgross`, and `intgross_2013$`.

Write a function called `scale_down_movie` that can take an element from our movies list and return that same movie but with the `budget`, `budget_2013$`, `domgross`, `domgross_2013$`, `intgross`, and `intgross_2013$` numbers all divided by 1 million and rounded to two decimal places.

In [10]:
```
def scale_down_movie(movie):
    scale_value = 1000000
    movie_dict_update = dict()
    for k,v in movie.items():
        if (k == 'budget' or k == 'budget_2013$' or k == 'domgross' or k ==
            v = round(float(v)/scale_value,2)
        movie_dict_update.update({k:str(v)})
    print(movie_dict_update)
    return movie_dict_update
    pass
```

In [11]:
```
movies[0]
```

Out[11]:
```
{'year': 2013,
 'imdb': 'tt1711425',
 'title': '21 &amp; Over',
 'test': 'notalk',
 'clean_test': 'notalk',
 'binary': 'FAIL',
 'budget': 13000000,
 'domgross': 25682380.0,
 'intgross': 42195766.0,
 'code': '2013FAIL',
 'budget_2013$': 13000000,
 'domgross_2013$': 25682380.0,
 'intgross_2013$': 42195766.0,
 'period code': 1.0,
 'decade code': 1.0}
```

In [12]:
```
scale_down_movie(movies[0])
```

```
{'year': '2013', 'imdb': 'tt1711425', 'title': '21 &amp; Over', 'test':
'notalk', 'clean_test': 'notalk', 'binary': 'FAIL', 'budget': '13.0', 'do
mgross': '25.68', 'intgross': '42.2', 'code': '2013FAIL', 'budget_2013$':
'13.0', 'domgross_2013$': '25.68', 'intgross_2013$': '42.2', 'period cod
e': '1.0', 'decade code': '1.0'}
```

Out[12]:
```
{'year': '2013',
 'imdb': 'tt1711425',
 'title': '21 &amp; Over',
 'test': 'notalk',
 'clean_test': 'notalk',
 'binary': 'FAIL',
 'budget': '13.0',
 'domgross': '25.68',
 'intgross': '42.2',
 'code': '2013FAIL',
 'budget_2013$': '13.0',
 'domgross_2013$': '25.68',
 'intgross_2013$': '42.2',
 'period code': '1.0',
 'decade code': '1.0'}
```

Ok, now that we have a function to scale down our movies, lets `map` through all of our `parsed_movies` to return a list of `scaled_movies`.

In [13]:
```
def scale_down_movies(movies):
    scaled_movies = list(map(lambda m:scale_down_movie(m),movies))
    print(scaled_movies)
    return scaled_movies
    pass
```

In [14]:
```
first_ten_movies = parsed_movies[0:10]
first_ten_scaled = scale_down_movies(first_ten_movies) or []
first_ten_scaled[-2:]
#[{'year': 2013,
#  'imdb': 'tt1814621',
#  'title': 'Admission',
#  'test': 'ok',
#  'clean_test': 'ok',
#  'binary': 'PASS',
#  'budget': 13.0,
#  'domgross': 18.01,
#  'intgross': 18.01,
#  'code': '2013PASS',
#  'budget_2013$': 13.0,
#  'domgross_2013$': 18.01,
#  'intgross_2013$': 18.01,
#  'period code': 1.0,
#  'decade code': 1.0},
# {'year': 2013,
#  'imdb': 'tt1815862',
#  'title': 'After Earth',
#  'test': 'notalk',
#  'clean_test': 'notalk',
#  'binary': 'FAIL',
#  'budget': 130.0,
#  'domgross': 60.52,
#  'intgross': 244.37,
#  'code': '2013FAIL',
#  'budget_2013$': 130.0,
#  'domgross_2013$': 60.52,
#  'intgross_2013$': 244.37,
#  'period code': 1.0,
#  'decade code': 1.0}]
```

```
{'year': '2013', 'imdb': 'tt1711425', 'title': '21 &amp; Over', 'test':
'notalk', 'clean_test': 'notalk', 'binary': 'FAIL', 'budget': '13.0', 'do
mgross': '25.68', 'intgross': '42.2', 'code': '2013FAIL', 'budget_2013$':
'13.0', 'domgross_2013$': '25.68', 'intgross_2013$': '42.2', 'period cod
e': '1.0', 'decade code': '1.0'}
{'year': '2012', 'imdb': 'tt1343727', 'title': 'Dredd 3D', 'test': 'ok-di
sagree', 'clean_test': 'ok', 'binary': 'PASS', 'budget': '45.0', 'domgros
s': '13.41', 'intgross': '40.87', 'code': '2012PASS', 'budget_2013$': '4
5.66', 'domgross_2013$': '13.61', 'intgross_2013$': '41.47', 'period cod
e': '1.0', 'decade code': '1.0'}
{'year': '2013', 'imdb': 'tt2024544', 'title': '12 Years a Slave', 'tes
t': 'notalk-disagree', 'clean_test': 'notalk', 'binary': 'FAIL', 'budge
t': '20.0', 'domgross': '53.11', 'intgross': '158.61', 'code': '2013FAI
L', 'budget_2013$': '20.0', 'domgross_2013$': '53.11', 'intgross_2013$':
'158.61', 'period code': '1.0', 'decade code': '1.0'}
{'year': '2013', 'imdb': 'tt1272878', 'title': '2 Guns', 'test': 'notal
k', 'clean_test': 'notalk', 'binary': 'FAIL', 'budget': '61.0', 'domgros
s': '75.61', 'intgross': '132.49', 'code': '2013FAIL', 'budget_2013$': '6
1.0', 'domgross_2013$': '75.61', 'intgross_2013$': '132.49', 'period cod
e': '1.0', 'decade code': '1.0'}
{'year': '2013', 'imdb': 'tt0453562', 'title': '42', 'test': 'men', 'clea
n_test': 'men', 'binary': 'FAIL', 'budget': '40.0', 'domgross': '95.02',
```

```
'intgross': '95.02', 'code': '2013FAIL', 'budget_2013$': '40.0', 'domgros
s_2013$': '95.02', 'intgross_2013$': '95.02', 'period code': '1.0', 'deca
de code': '1.0'}
{'year': '2013', 'imdb': 'tt1335975', 'title': '47 Ronin', 'test': 'men',
'clean_test': 'men', 'binary': 'FAIL', 'budget': '225.0', 'domgross': '3
8.36', 'intgross': '145.8', 'code': '2013FAIL', 'budget_2013$': '225.0',
'domgross_2013$': '38.36', 'intgross_2013$': '145.8', 'period code': '1.
0', 'decade code': '1.0'}
{'year': '2013', 'imdb': 'tt1606378', 'title': 'A Good Day to Die Hard',
'test': 'notalk', 'clean_test': 'notalk', 'binary': 'FAIL', 'budget': '9
2.0', 'domgross': '67.35', 'intgross': '304.25', 'code': '2013FAIL', 'bud
get_2013$': '92.0', 'domgross_2013$': '67.35', 'intgross_2013$': '304.2
5', 'period code': '1.0', 'decade code': '1.0'}
{'year': '2013', 'imdb': 'tt2194499', 'title': 'About Time', 'test': 'ok-
disagree', 'clean_test': 'ok', 'binary': 'PASS', 'budget': '12.0', 'domgr
oss': '15.32', 'intgross': '87.32', 'code': '2013PASS', 'budget_2013$':
'12.0', 'domgross_2013$': '15.32', 'intgross_2013$': '87.32', 'period cod
e': '1.0', 'decade code': '1.0'}
{'year': '2013', 'imdb': 'tt1814621', 'title': 'Admission', 'test': 'ok',
'clean_test': 'ok', 'binary': 'PASS', 'budget': '13.0', 'domgross': '18.0
1', 'intgross': '18.01', 'code': '2013PASS', 'budget_2013$': '13.0', 'dom
gross_2013$': '18.01', 'intgross_2013$': '18.01', 'period code': '1.0',
'decade code': '1.0'}
{'year': '2013', 'imdb': 'tt1815862', 'title': 'After Earth', 'test': 'no
talk', 'clean_test': 'notalk', 'binary': 'FAIL', 'budget': '130.0', 'domg
ross': '60.52', 'intgross': '244.37', 'code': '2013FAIL', 'budget_2013$':
'130.0', 'domgross_2013$': '60.52', 'intgross_2013$': '244.37', 'period c
ode': '1.0', 'decade code': '1.0'}
[{'year': '2013', 'imdb': 'tt1711425', 'title': '21 &amp; Over', 'test':
'notalk', 'clean_test': 'notalk', 'binary': 'FAIL', 'budget': '13.0', 'do
mgross': '25.68', 'intgross': '42.2', 'code': '2013FAIL', 'budget_2013$':
'13.0', 'domgross_2013$': '25.68', 'intgross_2013$': '42.2', 'period cod
e': '1.0', 'decade code': '1.0'}, {'year': '2012', 'imdb': 'tt1343727',
'title': 'Dredd 3D', 'test': 'ok-disagree', 'clean_test': 'ok', 'binary':
'PASS', 'budget': '45.0', 'domgross': '13.41', 'intgross': '40.87', 'cod
e': '2012PASS', 'budget_2013$': '45.66', 'domgross_2013$': '13.61', 'intg
ross_2013$': '41.47', 'period code': '1.0', 'decade code': '1.0'}, {'yea
r': '2013', 'imdb': 'tt2024544', 'title': '12 Years a Slave', 'test': 'no
talk-disagree', 'clean_test': 'notalk', 'binary': 'FAIL', 'budget': '20.
0', 'domgross': '53.11', 'intgross': '158.61', 'code': '2013FAIL', 'budge
t_2013$': '20.0', 'domgross_2013$': '53.11', 'intgross_2013$': '158.61',
'period code': '1.0', 'decade code': '1.0'}, {'year': '2013', 'imdb': 'tt
1272878', 'title': '2 Guns', 'test': 'notalk', 'clean_test': 'notalk', 'b
inary': 'FAIL', 'budget': '61.0', 'domgross': '75.61', 'intgross': '132.4
9', 'code': '2013FAIL', 'budget_2013$': '61.0', 'domgross_2013$': '75.6
1', 'intgross_2013$': '132.49', 'period code': '1.0', 'decade code': '1.
0'}, {'year': '2013', 'imdb': 'tt0453562', 'title': '42', 'test': 'men',
'clean_test': 'men', 'binary': 'FAIL', 'budget': '40.0', 'domgross': '95.
02', 'intgross': '95.02', 'code': '2013FAIL', 'budget_2013$': '40.0', 'do
mgross_2013$': '95.02', 'intgross_2013$': '95.02', 'period code': '1.0',
'decade code': '1.0'}, {'year': '2013', 'imdb': 'tt1335975', 'title': '47
Ronin', 'test': 'men', 'clean_test': 'men', 'binary': 'FAIL', 'budget':
'225.0', 'domgross': '38.36', 'intgross': '145.8', 'code': '2013FAIL', 'b
udget_2013$': '225.0', 'domgross_2013$': '38.36', 'intgross_2013$': '145.
8', 'period code': '1.0', 'decade code': '1.0'}, {'year': '2013', 'imdb':
'tt1606378', 'title': 'A Good Day to Die Hard', 'test': 'notalk', 'clean_
test': 'notalk', 'binary': 'FAIL', 'budget': '92.0', 'domgross': '67.35',
```

```
'intgross': '304.25', 'code': '2013FAIL', 'budget_2013$': '92.0', 'domgro
ss_2013$': '67.35', 'intgross_2013$': '304.25', 'period code': '1.0', 'de
cade code': '1.0'}, {'year': '2013', 'imdb': 'tt2194499', 'title': 'About
Time', 'test': 'ok-disagree', 'clean_test': 'ok', 'binary': 'PASS', 'budg
et': '12.0', 'domgross': '15.32', 'intgross': '87.32', 'code': '2013PAS
S', 'budget_2013$': '12.0', 'domgross_2013$': '15.32', 'intgross_2013$':
'87.32', 'period code': '1.0', 'decade code': '1.0'}, {'year': '2013', 'i
mdb': 'tt1814621', 'title': 'Admission', 'test': 'ok', 'clean_test': 'o
k', 'binary': 'PASS', 'budget': '13.0', 'domgross': '18.01', 'intgross':
'18.01', 'code': '2013PASS', 'budget_2013$': '13.0', 'domgross_2013$': '1
8.01', 'intgross_2013$': '18.01', 'period code': '1.0', 'decade code':
'1.0'}, {'year': '2013', 'imdb': 'tt1815862', 'title': 'After Earth', 'te
st': 'notalk', 'clean_test': 'notalk', 'binary': 'FAIL', 'budget': '130.
0', 'domgross': '60.52', 'intgross': '244.37', 'code': '2013FAIL', 'budge
t_2013$': '130.0', 'domgross_2013$': '60.52', 'intgross_2013$': '244.37',
'period code': '1.0', 'decade code': '1.0'}]
```

Out[14]:
```
[{'year': '2013',
  'imdb': 'tt1814621',
  'title': 'Admission',
  'test': 'ok',
  'clean_test': 'ok',
  'binary': 'PASS',
  'budget': '13.0',
  'domgross': '18.01',
  'intgross': '18.01',
  'code': '2013PASS',
  'budget_2013$': '13.0',
  'domgross_2013$': '18.01',
  'intgross_2013$': '18.01',
  'period code': '1.0',
  'decade code': '1.0'},
 {'year': '2013',
  'imdb': 'tt1815862',
  'title': 'After Earth',
  'test': 'notalk',
  'clean_test': 'notalk',
  'binary': 'FAIL',
  'budget': '130.0',
  'domgross': '60.52',
  'intgross': '244.37',
  'code': '2013FAIL',
  'budget_2013$': '130.0',
  'domgross_2013$': '60.52',
  'intgross_2013$': '244.37',
  'period code': '1.0',
  'decade code': '1.0'}]
```

In [15]:
```
scaled_movies = scale_down_movies(parsed_movies) or []
```

```
{'year': '2013', 'imdb': 'tt1711425', 'title': '21 &amp; Over', 'test':
'notalk', 'clean_test': 'notalk', 'binary': 'FAIL', 'budget': '13.0', 'do
mgross': '25.68', 'intgross': '42.2', 'code': '2013FAIL', 'budget_2013$':
'13.0', 'domgross_2013$': '25.68', 'intgross_2013$': '42.2', 'period cod
e': '1.0', 'decade code': '1.0'}
{'year': '2012', 'imdb': 'tt1343727', 'title': 'Dredd 3D', 'test': 'ok-di
sagree', 'clean_test': 'ok', 'binary': 'PASS', 'budget': '45.0', 'domgros
s': '13.41', 'intgross': '40.87', 'code': '2012PASS', 'budget_2013$': '4
5.66', 'domgross_2013$': '13.61', 'intgross_2013$': '41.47', 'period cod
e': '1.0', 'decade code': '1.0'}
{'year': '2013', 'imdb': 'tt2024544', 'title': '12 Years a Slave', 'tes
t': 'notalk-disagree', 'clean_test': 'notalk', 'binary': 'FAIL', 'budge
t': '20.0', 'domgross': '53.11', 'intgross': '158.61', 'code': '2013FAI
L', 'budget_2013$': '20.0', 'domgross_2013$': '53.11', 'intgross_2013$':
'158.61', 'period code': '1.0', 'decade code': '1.0'}
{'year': '2013', 'imdb': 'tt1272878', 'title': '2 Guns', 'test': 'notal
k', 'clean_test': 'notalk', 'binary': 'FAIL', 'budget': '61.0', 'domgros
s': '75.61', 'intgross': '132.49', 'code': '2013FAIL', 'budget_2013$': '6
1.0', 'domgross_2013$': '75.61', 'intgross_2013$': '132.49', 'period cod
e': '1.0', 'decade code': '1.0'}
```

3. Continue exploring the data

Now let's plot our dataset using Plotly to see how much money a movie made domestically in 2013 dollars given a budget in 2013 dollars. Create a trace called `revenues_per_budgets_trace` that plots this data.

> To do so, set `budget_2013$` as the x values, and the `domgross_2013$` as the y values. Set the text of the trace equal to a list of the movie titles, so that we can see which movie is associated with each point. All of the data should be coming from our `scaled_movies` variable.

In [16]:
```
budgets = list(map(lambda movie: float(movie['budget_2013$']), scaled_movie
domestic_revenues = list(map(lambda movie: float(movie['domgross_2013$']),
titles = list(map(lambda movie: movie['title'], scaled_movies))
```

We'll check the first ten values of the `budgets` , `domestic_revenues` , and `titles` lists, but your trace should have an element for each of the `scaled_movies` in the dataset.

In [17]:
```
budgets[0:10] # [13.0, 45.66, 20.0, 61.0, 40.0, 225.0, 92.0, 12.0, 13.0, 13
```

Out[17]:
```
[13.0, 45.66, 20.0, 61.0, 40.0, 225.0, 92.0, 12.0, 13.0, 130.0]
```

In [18]:
```
domestic_revenues[0:10] # [25.68, 13.61, 53.11, 75.61, 95.02, 38.36, 67.35,
```

Out[18]:
```
[25.68, 13.61, 53.11, 75.61, 95.02, 38.36, 67.35, 15.32, 18.01, 60.52]
```

In [19]:
```
titles[0:10]
# ['21 &amp; Over',  'Dredd 3D', '12 Years a Slave', '2 Guns', '42', '47 Ro
# 'About Time',  'Admission',  'After Earth']
```

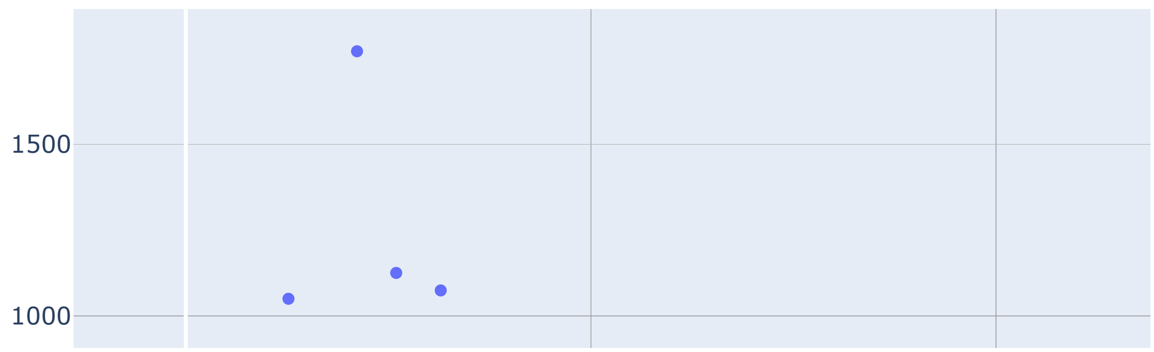
Out[19]:
```
['21 &amp; Over',
 'Dredd 3D',
 '12 Years a Slave',
 '2 Guns',
 '42',
 '47 Ronin',
 'A Good Day to Die Hard',
 'About Time',
 'Admission',
 'After Earth']
```

Once we have lists of these values, we are ready to create a trace. The following code creates a trace with the x values set as the `budgets` , the y values set as the `domestic_revenues` , and the text set as each of the movie `titles` .

In [20]:
```
from graph import trace_values
revenues_per_budgets_trace = trace_values(budgets, domestic_revenues, text
```

> Once we have written the above code, we'll be ready to plot this data. Press shift + enter on the code below and you should see all movies in a graph.

In [21]:
```
from graph import plot
from plotly.offline import iplot, init_notebook_mode
init_notebook_mode(connected=True)

plot([revenues_per_budgets_trace])
```



Look at that one datapoint that earned well over 1.5 billion dollars. What movie is that?

Write a function called `highest_domestic_gross` that finds the highest grossing movie given a list of movies.

In [22]:
```
def highest_domestic_gross(movies):
    # method 1
    #return max(movies, key=lambda m:float(m['domgross_2013$']))
    # method 2
    #return sorted(movies, key=lambda m:float(m['domgross_2013$']),reverse=
    # method 3
    domgross_2013_list = [float(movie['domgross_2013$']) for movie in movie
    domgross_list = [float(movie['domgross']) for movie in movies]
    print(domgross_2013_list == domgross_list)
    highest_domgross_2013 = max(domgross_2013_list)
    highest_domgross = max(domgross_list)
    print(max(highest_domgross_2013,highest_domgross))
    SelectedMovie = list(filter(lambda m:float(m['domgross_2013$'])==highes
    print(SelectedMovie)
    print(SelectedMovie['domgross_2013$'])
    #return list(filter(lambda m:float(m['domgross_2013$'])==highest_domgrc
    return SelectedMovie
    pass
```

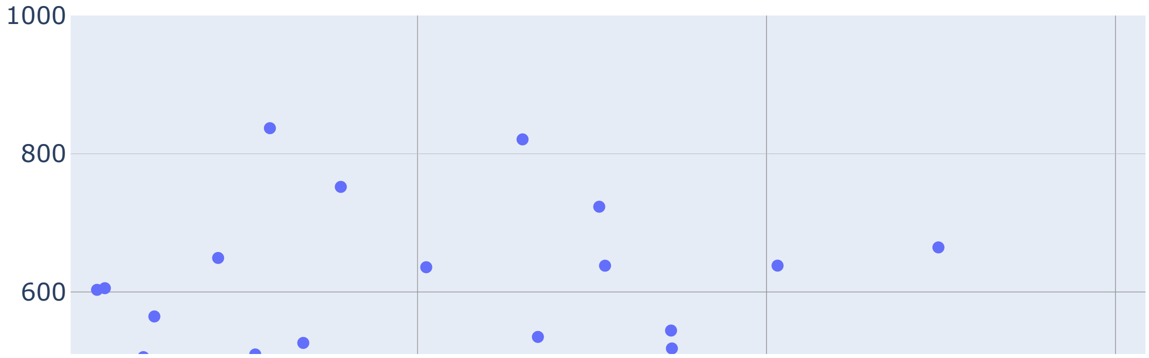
In [23]:
```
max_movie = highest_domestic_gross(scaled_movies) or {'title': 'some non mo
print(max_movie)
max_movie['title'] # 'Star Wars'
```

```
False
1771.68
{'year': '1977', 'imdb': 'tt0076759', 'title': 'Star Wars', 'test': 'nota
lk', 'clean_test': 'notalk', 'binary': 'FAIL', 'budget': '11.0', 'domgros
s': '461.0', 'intgross': '797.9', 'code': '1977FAIL', 'budget_2013$': '4
2.27', 'domgross_2013$': '1771.68', 'intgross_2013$': '3066.45', 'period
code': 'nan', 'decade code': 'nan'}
1771.68
{'year': '1977', 'imdb': 'tt0076759', 'title': 'Star Wars', 'test': 'nota
lk', 'clean_test': 'notalk', 'binary': 'FAIL', 'budget': '11.0', 'domgros
s': '461.0', 'intgross': '797.9', 'code': '1977FAIL', 'budget_2013$': '4
2.27', 'domgross_2013$': '1771.68', 'intgross_2013$': '3066.45', 'period
code': 'nan', 'decade code': 'nan'}
```

Out[23]: 'Star Wars'

Huh, well we should've known. Now let's zoom in on our dataset so that our plot no longer expands for just a few of the outliers. We will set the x-axis of our plot to go from zero to 300 million dollars, and the y-axis of our plot to go from zero to one billion dollars.

In [24]:
```
from graph import build_layout
revenues_per_budgets_trace = trace_values(budgets, domestic_revenues, text
revenues_layout = build_layout(x_range = [0, 300], y_range = [0, 1000])
plot([revenues_per_budgets_trace], revenues_layout)
```



Ok, well at least we have a closer look at our data. We still see Titanic up in the top right corner.

Building our models

Ok, now that we have collected and explored this data, our company hired an outside consultant to create a model that predicts revenue for us. The consultant provided us with the following:

$$R(x) = 1.5 * budget + 10$$

- where x is a movie's budget in 2013 dollars, and $R(x)$ is the expected revenue in 2013 dollars.

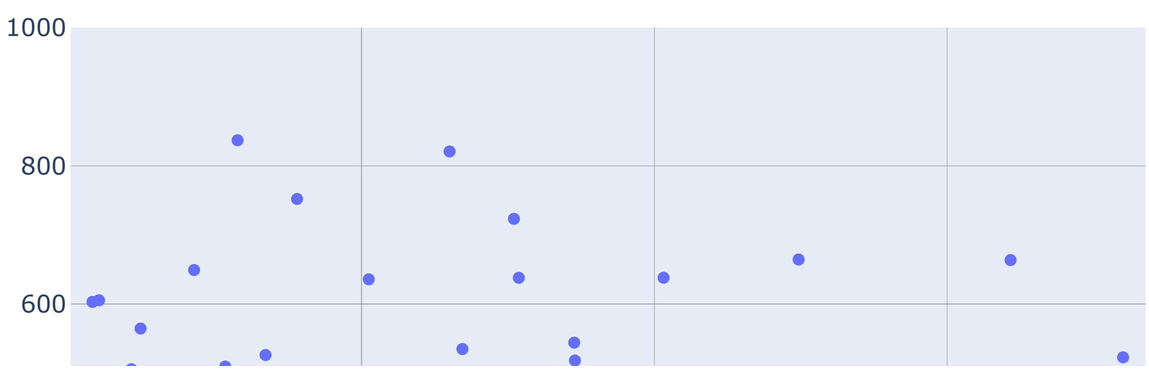
Write a function called `outside_consultant_predicted_revenue` that, provided a budget, returns the expected revenue according to the outside consultant's formula.

In [25]:
```
def outside_consultant_predicted_revenue(budget):
    return 1.5*float(budget)+10
    pass
```

Let's plot the consultant estimated revenue to see visually if his estimates line up. We will call this trace `external consultant estimate`.

In [26]:
```
budgets = list(map(lambda movie: float(movie['budget_2013$']), scaled_movie
domestic_revenues = list(map(lambda movie: float(movie['domgross_2013$']),
titles = list(map(lambda movie: movie['title'], scaled_movies))

consultant_estimated_revenues = list(map(lambda budget: outside_consultant_
consultant_estimated_revenues_trace = trace_values(budgets, consultant_esti
```

In [27]:
```
from plotly.offline import iplot, init_notebook_mode
init_notebook_mode(connected=True)

from graph import trace_values, m_b_trace, plot

plot([revenues_per_budgets_trace, consultant_estimated_revenues_trace], rev
```



Overall, the model doesn't look too bad. However, we can calculate the RSS to quantify how accurate his model really is.

Let's write a method called `error_for_consultant_model` which takes in a budget of a movie in our dataset, and returns the difference between the movie's gross domestic revenue in 2013 dollars, and the prediction from the consultant's model.

In [28]:
```
def error_for_consultant_model(movie):
    return float(movie['domgross_2013$']) - outside_consultant_predicted_re
    pass
```

In [29]:
```
american_hustle = {'binary': 'PASS', 'budget': 40.0, 'budget_2013$': 40.0,
         'code': '2013PASS', 'decade code': 1.0, 'domgross': 148.43, 'domgr
         'intgross': 249.48, 'intgross_2013$': 249.48, 'period code': 1.0,
         'title': 'American Hustle', 'year': 2013}
error_for_consultant_model(american_hustle) # 78.43
```

Out[29]:
```
78.43
```

Once we have written a formula that calculates the error for the consultant's model provided a budget, we can write a method that calculates the RSS for the consultant's model. When we move on to compare our consultant's model with others, we'll then have a metric for comparison.

In [30]:
```
def rss_consultant(movies):
    return round(sum(list(map(lambda m:error_for_consultant_model(m)**2,mov
    pass
```

In [31]:
```
rss_consultant(scaled_movies) # 23234357.68
```

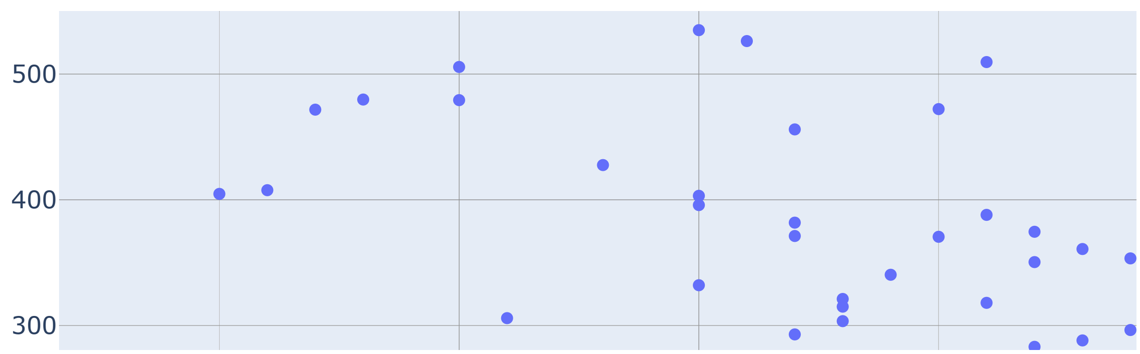
Out[31]:
```
23234357.68
```

Ok, we'll find out if this number is any good later, but for right now let's just say that our RSS is good enough. Use the derivative to write a function that provided a budget, returns the $\frac{\Delta R}{\Delta x}$ according to the consultant's model. Remember that our consultant's model is $R(x) = 1.5x + 10$ where x is a budget, and $R(x)$ is an expected revenue.

A new model

Now imagine a data scientist in your company wants to take a crack at his own model for predicting a movie's revenue. The data scientist notices, that in general, movies tend to make more money per year.

In [32]:
```
from graph import build_layout
years = list(map(lambda movie: movie['year'],movies))
years_and_revenues = trace_values(years, domestic_revenues, text = titles)
years_layout = build_layout(y_range = [0, 550])
plot([years_and_revenues], years_layout)
```



So the data scientist comes up with a new model, to indicate a movie's expected revenue is 1.5 million for every year after 1965 plus 1.1 times the movie's budget. Write a function called `revenue_with_year` that takes as arguments `budget` and `year` and returns expected revenue.

In [33]:
```
def revenue_with_year(budget, year):
    return 1.5*(float(year)-1965)+1.1*float(budget)
    pass
```

In [34]:
```
print(revenue_with_year(25, 1997)) # 75.5
print(revenue_with_year(40, 1983)) # 71.0
```

```
75.5
71.0
```

Notice that this model has **two variables**, the budget and year, and therefore is not a line function.

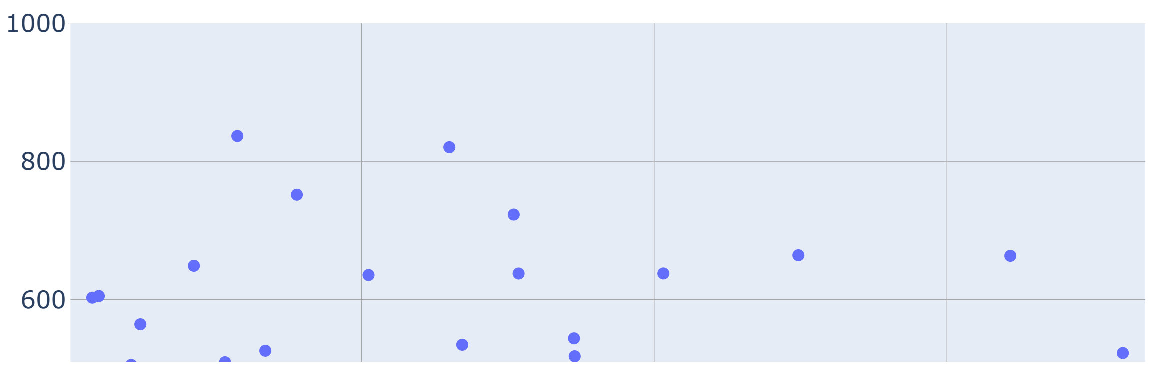
Let's compare these models by plotting the actual revenues and budgets, the prior `external consultant estimate` line trace, and the `internal consultant estimate` based upon this model's estimates. Since this model doesn't produce a line, we will set the mode for `internal_consultant_estimated_trace` to 'markers'.

In [35]:
```
budgets = list(map(lambda movie: float(movie['budget_2013$']), scaled_movie
domestic_revenues = list(map(lambda movie: float(movie['domgross_2013$']),
titles = list(map(lambda movie: movie['title'], scaled_movies))

internal_consultant_estimated_revenues = list(map(lambda movie: revenue_wit
internal_consultant_estimated_trace = trace_values(budgets, internal_consul
```

In [36]:
```
print(internal_consultant_estimated_trace['x'][0:10]) # [13.0, 45.66, 20.0,
print(internal_consultant_estimated_trace['y'][0:10]) # [86.3, 120.726, 94.
print(internal_consultant_estimated_trace['mode']) # 'markers'
```

```
[13.0, 45.66, 20.0, 61.0, 40.0, 225.0, 92.0, 12.0, 13.0, 130.0]
[86.3, 120.726, 94.0, 139.10000000000002, 116.0, 319.5, 173.2, 85.2, 86.
3, 215.0]
markers
```

In [37]:
```
from plotly.offline import iplot, init_notebook_mode
init_notebook_mode(connected=True)

from graph import trace_values, m_b_trace, plot
plot([revenues_per_budgets_trace, consultant_estimated_revenues_trace, inte
```



Although the `internal consultant model` isn't a line, it still seems to match our data fairly well. Let's find out how well. Even though it is not a line, we can still calculate the RSS for this model. Write a function called `rss_revenue_with_year` that returns the Residual Sum of Squares associated with the `revenue_with_year` model for the `scaled_movies` dataset. The `squared_error_revenue_with_year` function can be used to return the squared error of the model associated with just a single movie.

In [38]:
```
def squared_error_revenue_with_year(movie):
    return pow(float(movie['domgross_2013$']) - revenue_with_year(movie['bu
    pass

def rss_revenue_with_year(movies):
    return round(sum(list(map(lambda m:squared_error_revenue_with_year(m),m
    pass
```

In [39]:
```
rss_revenue_with_year(scaled_movies) # 25364329.23
```

Out[39]: 25364329.23

The RSS here is $25,364,329.23$ as opposed to the RSS of $23,234,357.68$ from the external consultant's model. According to RSS, this model isn't as accurate as the previous model. Still, it isn't bad enough to ignore completely.

Our initial regression line, and improving upon it

Now that we have evaluated the models of an outside consultant and an internal consultant, it's time to see if we can do any better. Let's go.

We have our dataset. Let's begin with an initial regression line that sets $b = 0.5$ and $m = 1.79$.

In [44]:
```
from linear_equations import build_regression_line
budgets = list(map(lambda movie: float(movie['budget_2013$']), scaled_movie
domestic_revenues = list(map(lambda movie: float(movie['domgross_2013$']),
```

In [45]:
```
initial_regression_line = {'b': 0.5, 'm': 1.79}
```

Using values for m and b from our initial regression line, we can write `expected_revenue_per_budget` that returns the expected revenue provided a budget.

In [46]:
```
def expected_revenue_per_budget(budget):
    return initial_regression_line['m'] * float(budget) + initial_regressic
    pass
```

In [47]:
```
budget = american_hustle['budget_2013$'] # 40.0
print(budget)
expected_revenue_per_budget(budget) # 72.1
print(expected_revenue_per_budget(budget))
```

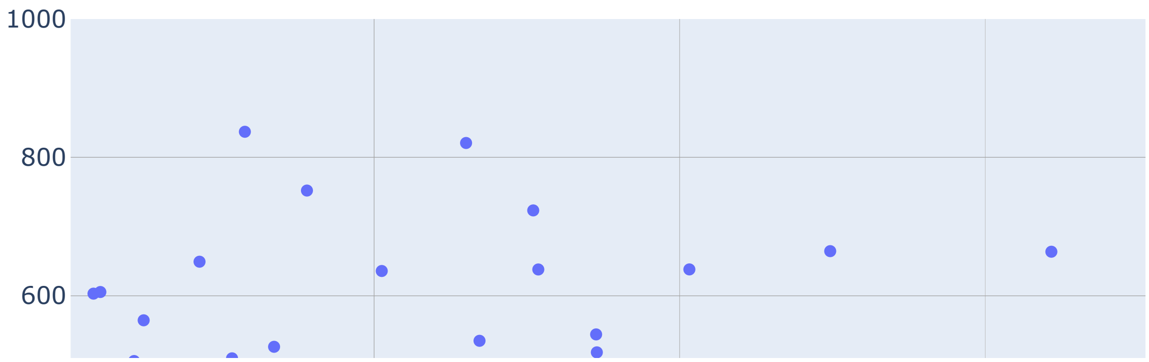
```
40.0
72.1
```

Now this initial regression line was not very sophisticated. We simply drew a line between the points with the lowest and highest x values.

Let's plot our initial regression line along our dataset to get a sense of the accuracy of this first line.

In [48]:
```
budgets = list(map(lambda movie: float(movie['budget_2013$']), scaled_movie
estimated_revenues = list(map(lambda budget: expected_revenue_per_budget(bu
len(estimated_revenues)
initial_regression_trace = trace_values(budgets, estimated_revenues, mode =
```

In [49]:
```
plot([revenues_per_budgets_trace, initial_regression_trace], revenues_layou
```



By now we should be able to guess the next step: quantify how well this line matches our data. We'll write a function called `regression_revenue_error` that, provided a movie and an `m` and `b` value of a regression line, returns the difference between our `initial_regression_lines` 's expected revenue and the actual revenue error.

In [50]:
```
def regression_revenue_error(m, b, movie):
    return round(float(movie['domgross_2013$']) - (m * float(movie['budget_
    pass
```

In [51]:
```
initial_regression_line
```

Out[51]:
```
{'b': 0.5, 'm': 1.79}
```

In [52]:
```
american_hustle = {'binary': 'PASS', 'budget': 40.0, 'budget_2013$': 40.0,
         'code': '2013PASS', 'decade code': 1.0, 'domgross': 148.43, 'domgr
         'intgross': 249.48, 'intgross_2013$': 249.48, 'period code': 1.0,
         'title': 'American Hustle', 'year': 2013}

regression_revenue_error(initial_regression_line['m'], initial_regression_l
# 76.33
```

Out[52]:
```
76.33
```

Ok, now plot the cost curve from changing values of m from 1.0 to 1.9.

> We don't ask you to write a function for calculating the RSS, as you already wrote one in the error library which is available to you, and you can see used here.

In [53]:
```
from error import residual_sum_squares
residual_sum_squares(budgets, domestic_revenues, initial_regression_line['m
```

Out[53]:
```
24179823.79
```

But we do ask you to plot a cost curve from 1.0, to 1.9 using that `residual_sum_squares` function. We start off with a list of values of m from 1.0 to 1.9, assigned to `m_range` below.

In [54]:
```
large_m_range = list(range(10, 20))
m_range = list(map(lambda m_value: m_value/10,large_m_range))
```

Now we need to calculate a list of RSS values associated with each value in the `m_range` .

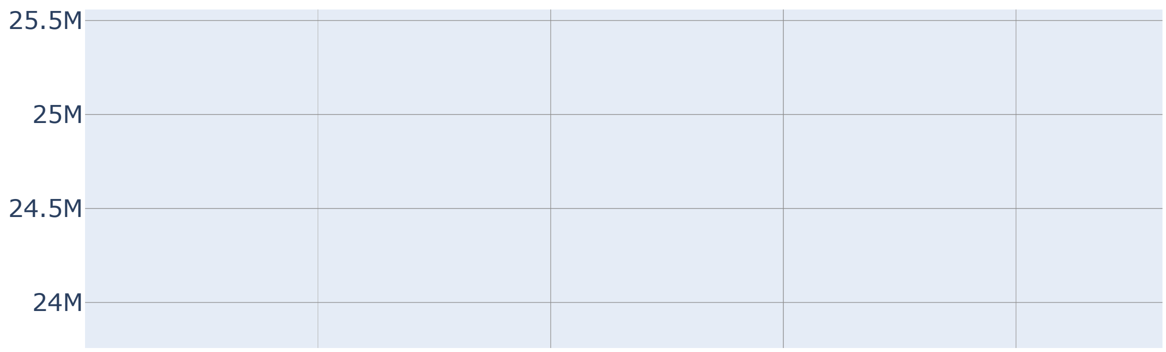
In [55]:
```
cost_values = list(map(lambda m_value: round(residual_sum_squares(budgets,
```

In [56]:
```
from graph import trace_values
rss_trace = trace_values(x_values=m_range, y_values=cost_values, mode = 'li
```

In [57]:
```
rss_trace
```

Out[57]:
```
{'x': [1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9],
 'y': [23360190.02,
  22710401.61,
  22279030.46,
  22066076.55,
  22071539.9,
  22295420.5,
  22737718.35,
  23398433.45,
  24277565.8,
  25375115.41],
 'mode': 'lines',
 'name': 'data',
 'text': []}
```

In [58]:
```
plot([rss_trace])
```



Ok, so based on this, it appears that with our $b = 0.5$, the slope of our regression line that produces the lowest error is between 1.3 and 1.4. In fact if we replace our initial line value of m with 1.3, we see that our RSS does in fact decline from our previous value of $24, 179, 824$.

In [59]:
```
residual_sum_squares(budgets, domestic_revenues, 1.3, initial_regression_li
```

Out[59]:
```
22066076.55
```

Changing multiple variables

Ok, now it's time to move beyond testing the accuracy of the line with changing only a single variable. We need to play with both variables to find the 'best fit regression line'. As we know, the technique for that is to use gradient descent.

Remember that we derived our gradient formulas by starting with our cost function, and saying the RSS is a function of our m and b variables:

$$J(m,b) = \sum_{i=1}^{n}(y_i - (mx_i + b))^2$$

From the above formula for our cost curve, we found the gradient descent of the cost function, as that is used to find the incremental changes to decrease RSS. We do this mathematically, by taking the partial derivative with respect to m and b.

$$\frac{\partial J}{\partial b}J(m,b) = -2\sum_{i=1}^{n}(y_i - (mx_i + b)) = -2\sum_{i=1}^{n}\epsilon_i$$
$$\frac{\partial J}{\partial m}J(m,b) = -2\sum_{i=1}^{n}x(y_i - (mx_i + b)) = -2\sum_{i=1}^{n}x_i * \epsilon_i$$

Looking at our top function $\frac{\partial J}{\partial m}$, we see that it equals negative 2, multiplied by the sum of the errors for a provided m and b values relative to our dataset. And luckily for us, we already have a function called `regression_revenue_error` that returns the error at a given point when provided our m and b values.

Recall, that we learned that the factor of 2 can be discarded since it is present in both formulas. Additionally, recall that the error needs to be scaled by the size of the dataset to prevent larger datasets from having larger errors.

$$\frac{\partial J}{\partial b}J(m,b) = -\frac{1}{n}\sum_{i=1}^{n}\epsilon_i$$
$$\frac{\partial J}{\partial m}J(m,b) = -\frac{1}{n}\sum_{i=1}^{n}x_i * \epsilon_i$$

Our task now is two write a function called `b_gradient` that takes in values of m, b and our (scaled) movies, and returns the `b` gradient.

In [63]:
```
def b_gradient(m, b, movies):
    return round((-1/len(movies)) * (sum(list(map(lambda movie:regression_r
    pass
```

In [64]:
```
b_gradient(1.79, 0.50, scaled_movies) # 5.37
```

Out[64]:
```
5.37
```

Next, write a function called `m_gradient` that returns the `m` gradient for values of m, b, and a list of movies.

In [65]:
```
def m_gradient(m, b, movies):
    return round((-1/len(movies)) * (sum(list(map(lambda movie:regression_r
    pass
```

In [66]:
```
m_gradient(1.79, 0.50, scaled_movies) # 2520.59
```

Out[66]: 2520.59

> Notice that the `m_gradient` is significantly larger than the `b_gradient`. This makes sense since the `m_gradient` formula is similar to the `b_gradient` formula, except that its output is also multiplied by the corresponding x value.

Ok, now we just wrote two functions that tell us how to update the corresponding values of m and b. Our next step is to write a function called `step_gradient` that will use these functions to take the step down along our cost curve.

Remember that with each step we want to move our `current_b` value in the negative direction of calculated `b_gradient`, and want to move our `current_m` value in the negative direction of the calculated `m_gradient`.

`current_m` = `old_m` $-\eta(-\frac{1}{n} * \sum_{i=1}^{n} x_i * \epsilon_i)$

`current_b` = `old_b` $-\eta(-\frac{1}{n} * \sum_{i=1}^{n} \epsilon_i)$

The `step_gradient` function would take as arguments the `b_current`, `m_current`, the list of scaled movies, and a learning rate, and returns a newly calculated `b_current` and `m_current` with a dictionary of keys `b` and `m` that point to the current values.

In [67]:
```
def step_gradient(b_current, m_current, movies, learning_rate):
    return {'b':b_current-learning_rate*b_gradient(m_current, b_current, mc
    pass
```

In [68]:
```
initial_regression_line # {'b': 0.5, 'm': 1.79}
```

Out[68]: {'b': 0.5, 'm': 1.79}

Then let's see how our formula changes over time using gradient descent.

In [69]:
```
step_gradient(initial_regression_line['b'], initial_regression_line['m'], s
```

Out[69]: {'b': 0.499463, 'm': 1.537941}

Now write a function that can operate given a set of 100 iterations and start from our `initial_regression_line`.

In [88]:
```
# set our initial step with m and b values, and the corresponding error.
def generate_steps(m, b, number_of_steps, movies, learning_rate):
    paras_update_dict = dict()
    paras_update_list = []
    b_update = 0
    m_update = 0
    for step in range(0,number_of_steps):
        paras_update_dict.update(step_gradient(b, m, movies, learning_rate)
        print(str(step)+":"+str(paras_update_dict))
        b = paras_update_dict['b']
        m = paras_update_dict['m']
        paras_update_list.append(paras_update_dict)
    print('\r\n')
    print(paras_update_list)
    return paras_update_list
    pass

#     iterations = []
#     for i in range(number_of_steps):
#         iteration = step_gradient(b, m, movies, learning_rate)
#         # {'b': value, 'm': value}
#         b = iteration['b']
#         m = iteration['m']
#         # update values of b and m
#         iterations.append(iteration)
#     return iterations
```

In [89]:
```
iterations = generate_steps(initial_regression_line['m'], initial_regressic
```

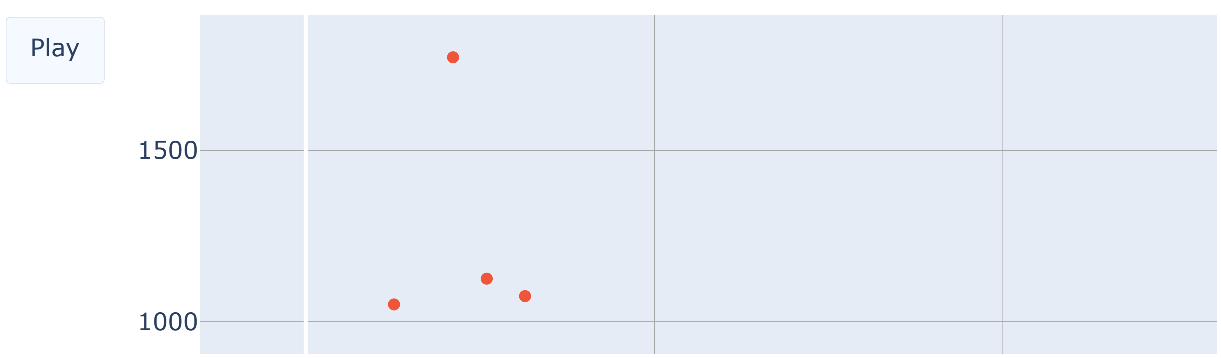
```
51:{'b': 0.5873779999999994, 'm': 1.3793229999999996}
52:{'b': 0.5891279999999994, 'm': 1.3793069999999996}
53:{'b': 0.5908779999999995, 'm': 1.3792909999999996}
54:{'b': 0.5926279999999995, 'm': 1.3792749999999996}
55:{'b': 0.5943769999999995, 'm': 1.3792609999999996}
56:{'b': 0.5961259999999995, 'm': 1.3792449999999996}
57:{'b': 0.5978749999999995, 'm': 1.3792289999999996}
58:{'b': 0.5996239999999995, 'm': 1.3792129999999996}
59:{'b': 0.6013729999999995, 'm': 1.3791969999999996}
60:{'b': 0.6031219999999995, 'm': 1.3791829999999996}
61:{'b': 0.6048709999999995, 'm': 1.3791659999999997}
62:{'b': 0.6066199999999995, 'm': 1.3791489999999997}
63:{'b': 0.6083689999999995, 'm': 1.3791329999999997}
64:{'b': 0.6101179999999995, 'm': 1.3791169999999997}
65:{'b': 0.6118669999999995, 'm': 1.3790999999999998}
66:{'b': 0.6136159999999995, 'm': 1.3790849999999997}
67:{'b': 0.6153639999999995, 'm': 1.3790699999999996}
68:{'b': 0.6171119999999994, 'm': 1.3790549999999995}
69:{'b': 0.6188599999999994, 'm': 1.3790389999999995}
70:{'b': 0.6206079999999994, 'm': 1.3790219999999995}
```

And we can see how this changes over time.

In [90]:
```
def to_line(m, b):
    initial_x = 0
    ending_x = 500
    initial_y = m*initial_x + b
    ending_y = m*ending_x + b
    return {'data': [{'x': [initial_x, ending_x], 'y': [initial_y, ending_y

frames = list(map(lambda iteration: to_line(iteration['m'], iteration['b'])
```

In [92]:
```
from plotly.offline import init_notebook_mode, iplot
from IPython.display import display, HTML

init_notebook_mode(connected=True)

budgets = list(map(lambda movie: float(movie['budget_2013$']), scaled_movie
domestic_revenues = list(map(lambda movie: float(movie['domgross_2013$']),

figure = {'data': [{'x': [0], 'y': [0]}, {'x': budgets, 'y': domestic_reven
          'layout': {'title': 'Regression Line',
                     'updatemenus': [{'type': 'buttons',
                                      'buttons': [{'label': 'Play',
                                                   'method': 'animate',
                                                   'args': [None]}]}]
                    },
          'frames': frames}
iplot(figure)
```

Regression Line



Finally, let's calculate the RSS associated with our formula as opposed to the other.

In [93]:
```
iterations[-1] # {'b': 0.5, 'm': 1.38}
```

Out[93]:
```
{'b': 0.6712680000000001, 'm': 1.378561999999999}
```

In [94]:
```
residual_sum_squares(budgets, domestic_revenues, iterations[-1]['m'], itera
```

Out[94]:
```
22043467.93
```

Using this last iteration, we have an RSS 22052973.85, better than all previous models - and we have the data, and knowledge to prove it:

In [95]:
```
external_consultant_model = rss_consultant(scaled_movies)
internal_consultant_model = rss_revenue_with_year(scaled_movies)
our_regression_model = residual_sum_squares(budgets, domestic_revenues, ite
```

In [96]:
```
print(external_consultant_model) # 23234357.68
print(internal_consultant_model) # 25364329.23
print(our_regression_model)      # 22052973.85
```

```
23234357.68
25364329.23
22043467.93
```

Nice work!