# Assignment 8 - SGD using linear Regression

In [1]:
```
## Lets import Packages from the required libraries
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import SGDRegressor
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
```

In [2]:
```
## Loading our dataset
boston=load_boston()
```

In [3]:
```
# Shape of the dataset
boston.data.shape
```

Out[3]: (506, 13)

In [4]:
```
print(boston.feature_names)
```

```
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
 'B' 'LSTAT']
```

In [5]:

```
## Description of each feature
print(boston.DESCR)
```

```
.. _boston_dataset:

Boston house prices dataset
---------------------------

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Value (attribute 14) is usually the target.

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/ (https://archive.ics.uci.edu/ml/machine-learning-databases/housing/)


This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics
...', Wiley, 1980.   N.B. Various transformations are used in the table on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that address regression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of Collinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Tenth International Conference of Machine Learning, 236-243, University of Mass
achusetts, Amherst. Morgan Kaufmann.
```

In [6]:

```
boston_data=pd.DataFrame(data=boston.data,columns=boston.feature_names)
```

In [7]:
```
boston_data['price']=boston.target
boston_data.head(2)
```

Out[7]:

| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO | B | LSTAT | price |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.00632 | 18.0 | 2.31 | 0.0 | 0.538 | 6.575 | 65.2 | 4.0900 | 1.0 | 296.0 | 15.3 | 396.9 | 4.98 | 24.0 |
| **1** | 0.02731 | 0.0 | 7.07 | 0.0 | 0.469 | 6.421 | 78.9 | 4.9671 | 2.0 | 242.0 | 17.8 | 396.9 | 9.14 | 21.6 |

## Train Test Split

In [8]:
```
x=boston.data
y=boston.target
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.33)
```

## Standardize the Train set

In [9]:
```
## using standard scalar starndardize the train data and tranform the test later to prevent data leakage
x=boston.data
y=boston.target

scaler=StandardScaler()

scaler.fit(x_train)
x_train=scaler.transform(x_train)
x_test=scaler.transform(x_test)
# Creating a new dataframe with tranformed data
transformed_train_set=pd.DataFrame(x_train)
transformed_train_set['price']=y_train

```

## Write custom SGD for linear Regression

In [10]:

```
class SGD_scratch :
    '''The class contains various functions to perform Schocastic Gradient Descent on Linear regression
       Class includes 2 functions primarily
       1.find_w_b function : To find best (W,b) that  fits the data giving minimum loss using SGD.
       Note: Some important parameters from sklearn.SGDRegressor have also been implemented
       2.Predict Function : that predicts y label given (w,b) '''
    def __init__(self,dataset,learning_rate,k,no_iterations,shuffle=False,divide=None):
        # initiating values
        self.data=dataset
        self.learning_rate=learning_rate
        self.k=k
        self.divide=divide
        self.no_iterations=no_iterations
        self.shuffle=shuffle

    def find_w_b(self):
        from sklearn.utils import shuffle
        k,no_iter=self.k,self.no_iterations
        w=np.random.randn(1,13)
        b=0
        lr_rate=self.learning_rate

        for i in range(no_iter):
            w_vector,b_vector=np.zeros(shape=(1,13)),0
            if shuffle:
                self.data=shuffle(self.data) #Shuffle the dataframe when shuffle=True for every iterations
            else:
                pass


            #take sample of size k and  store the data and target seprately
            sampled_data=self.data.sample(k,replace=False)
            x=np.array(sampled_data.drop('price',axis=1))
            y=np.array(sampled_data['price'])
            residual_sum_square=0
            for pt in range(k):
                actual_=y[pt] ## actual value y[i]
                predicts_ = np.dot(w,x[pt]) + b ## predicted value  w.T.x[pt] + intercept
                w_vector+=(-2) * x[pt] * (y[pt] - (np.dot(w,x[pt]) + b) )
                b_vector+=(-2) * (y[pt] - (np.dot(w,x[pt]) + b))
                error = (predicts_ - actual_) ** 2 # square( actual(y[i]) - predicted(y[i]) )
                residual_sum_square += error # sum of square of all the residue

            print("-"*20)
            print('Iteration {0} :  mean  squared error : {1} '.format(i,residual_sum_square/k))

            previous_state_b=b # storing the previous intercept value
            previous_state_w=w #storing the previous weights values

            w=w-lr_rate*(w_vector/k)
            b=b-lr_rate*(b_vector/k)

            diff_vec= previous_state_w - w #calculate the difference between the vectors
            if abs(previous_state_b -b) < 0.0008  and np.sqrt(np.sum(diff_vec**2)) < 0.0008 : # exit loop if w,b value is almost same,set random threshold 0.008
                print('*'*20)
                print('State wj :', previous_state_w)
                print('State wj+1 :', w)
                print('State bj :', previous_state_b)
                print('State bj+1 :', b)
                print('Iteration terminated at :',i)
```

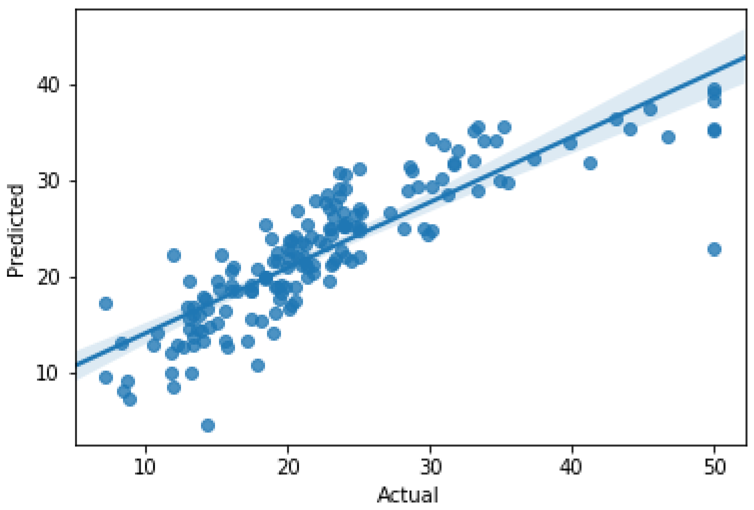
```
                print('Iteration terminated at :',i)
                break

            if self.divide:
                lr_rate = lr_rate / pow((i+1),0.05)
                print('lr_rate: ',lr_rate)
            print("-"*10)

        return w,b

    def predict_func(self,w,b,test_set):
        predict_set=[]
        test_set=np.array(test_set)
        for pt in range(len(test_set)):
            predictions=(np.dot(w,test_set[pt])+b)
            predict_set.append(predictions[0])
        return predict_set
```

## Sklearn SGD Regressor

In [46]:

```
# SkLearn SGD classifier
n_iter=500
sklearn_model = SGDRegressor(max_iter=n_iter,alpha=0.05,shuffle=True)
sklearn_model.fit(x_train, y_train)
y_pred=sklearn_model.predict(x_test)
weights= pd.DataFrame(data=sklearn_model.coef_.T,columns=['Sklearn SGD'])
## Plotting functions
#Scatter plot
predict_data=pd.DataFrame(data= {'Actual' : y_test,'Predicted' :y_pred})
sns.regplot(x='Actual', y='Predicted', data=predict_data)

print('Mean Squared Error :',mean_squared_error(y_test, y_pred))
```

```
Mean Squared Error : 22.2971947293
```

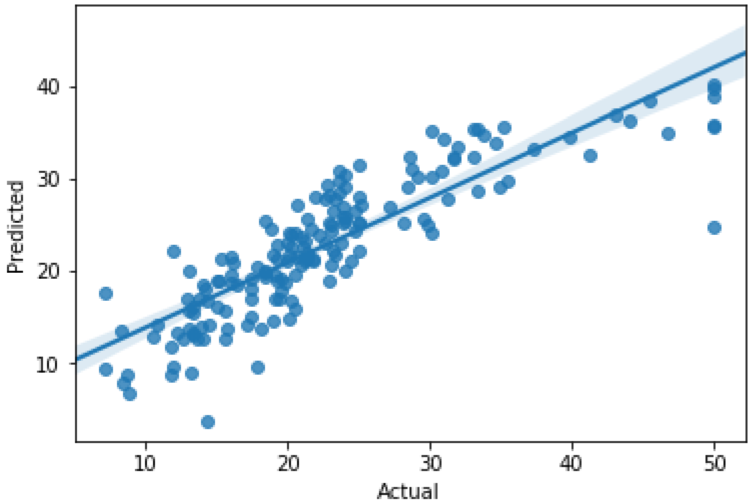



## Sklearn Linear Regression

In [47]:

```
# SkLearn SGD classifier
sklearn_lr_model = LinearRegression()
sklearn_lr_model.fit(x_train, y_train)
y_pred=sklearn_lr_model.predict(x_test)
weights['Sklearn Linear Regression '] = pd.DataFrame(sklearn_lr_model.coef_.T)
## Plotting functions
#Scatter plot
predict_data=pd.DataFrame(data= {'Actual' : y_test,'Predicted' :y_pred})
sns.regplot(x='Actual', y='Predicted', data=predict_data)

print('Mean Squared Error :',mean_squared_error(y_test, y_pred))
```

```
Mean Squared Error : 22.1950284502
```




## Custom SGD regressor : Trial 1

1. Initially implementing with learning rate =1 and iterations=100
2. For every iteration set learning rate = learning rate /2
3. Shuffle the dataset per iteration

In [13]:

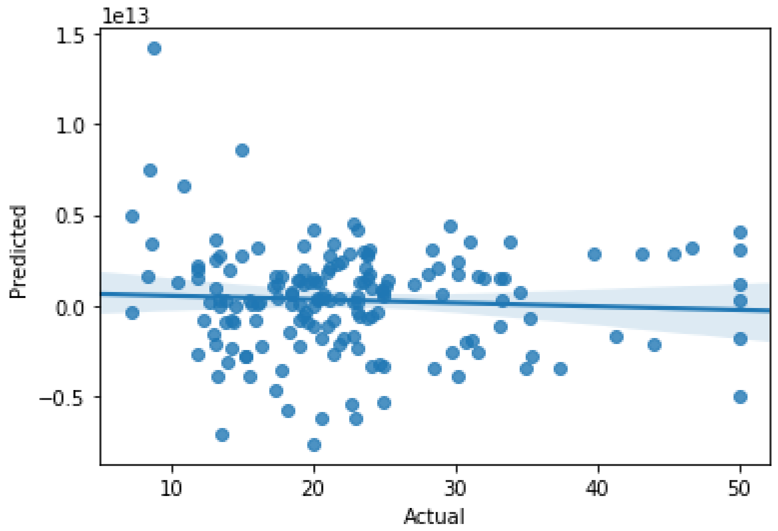
```
model=SGD_scratch(transformed_train_set,1,40,100,shuffle=True,divide=2)
w,b=model.find_w_b()
predictions=model.predict_func(w,b,x_test)
weights['Custom SGD (Trial 1)'] = pd.DataFrame(w.T)
```

```
lr_rate:  0.6529478154043212
----------
--------------------
Iteration 7 :  mean  squared error : [  8.57695860e+15]
lr_rate:  0.5884695206938755
----------
--------------------
Iteration 8 :  mean  squared error : [  2.98338967e+17]
lr_rate:  0.5272442454241162
----------
--------------------
Iteration 9 :  mean  squared error : [  1.02650061e+19]
lr_rate:  0.46990692835986236
----------
--------------------
Iteration 10 :  mean  squared error : [  4.48201807e+20]
lr_rate:  0.41681391978913523
----------
--------------------
Iteration 11 :  mean  squared error : [  9.33673670e+21]
```

In [14]:

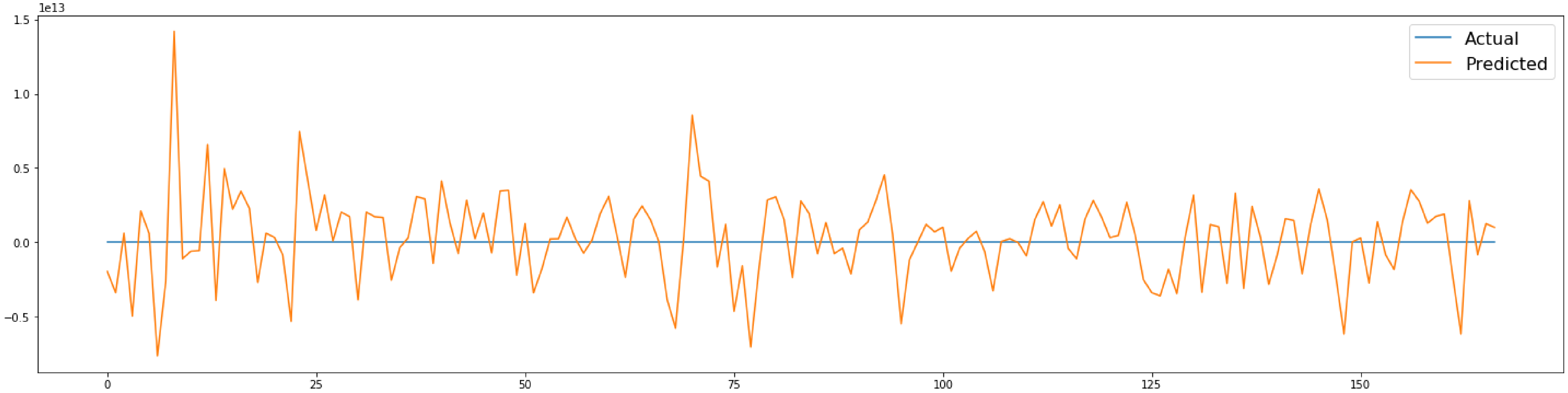
```
## Plotting functions
#Scatter plot
predict_data=pd.DataFrame(data= {'Actual' : y_test,'Predicted' :predictions})
sns.regplot(x='Actual', y='Predicted', data=predict_data)
print('Mean Squared Error :',mean_squared_error(y_test, predictions))
```

```
Mean Squared Error : 8.36279966769e+24
```



In [15]:

```
plt.figure(figsize=(25,6))
plt.plot(y_test, label='Actual')
plt.plot(predictions, label='Predicted')
plt.legend(prop={'size': 16})
plt.show()
print('Mean Squared Error :',mean_squared_error(y_test, predictions))
```




```
Mean Squared Error : 8.36279966769e+24
```

## Observations :

1. We see that the model does not perform well and gives a MSE of 8.36 e+24, with the above set parameters

## Custom SGD regressor :Trial 2

1. Implementing with learning rate =0.2 and iterations=500
2. Shuffle the dataset per iteration

In [55]:

```
model=SGD_scratch(transformed_train_set,0.2,40,500,shuffle=True,divide=1)
w,b=model.find_w_b()
predictions=model.predict_func(w,b,x_test)
weights['Custom SGD (Trial 2)'] = pd.DataFrame(w.T)
```

```
--------------------
Iteration 0 :  mean  squared error : [ 700.84819584]
lr_rate:  0.2
----------
--------------------
Iteration 1 :  mean  squared error : [ 316.38821338]
lr_rate:  0.19318726578496911
----------
--------------------
Iteration 2 :  mean  squared error : [ 441.89517938]
lr_rate:  0.18286156535236558
----------
--------------------
Iteration 3 :  mean  squared error : [ 629.41243007]
lr_rate:  0.17061587335782108
----------
--------------------
Iteration 4 :  mean  squared error : [ 1002.15283167]
lr_rate:  0.15742399642419647
----------
--------------------
Iteration 5 :  mean  squared error : [ 936.9914339]
lr_rate:  0.14393399205076884
----------
--------------------
Iteration 6 :  mean  squared error : [ 467.33421729]
lr_rate:  0.13058956308086425
----------
--------------------
Iteration 7 :  mean  squared error : [ 187.10058905]
lr_rate:  0.1176939041387751
----------
--------------------
Iteration 8 :  mean  squared error : [ 105.94217708]
lr_rate:  0.10544884908482326
----------
--------------------
Iteration 9 :  mean  squared error : [ 51.12823438]
lr_rate:  0.09398138567197249
----------
--------------------
Iteration 10 :  mean  squared error : [ 31.43511732]
lr_rate:  0.08336278395782706
----------
--------------------
Iteration 11 :  mean  squared error : [ 25.61902904]
lr_rate:  0.07362293851363488
----------
--------------------
Iteration 12 :  mean  squared error : [ 34.78255225]
lr_rate:  0.06476136289387929
----------
--------------------
Iteration 13 :  mean  squared error : [ 32.92859496]
lr_rate:  0.05675571223453064
```

```
----------
--------------------
Iteration 14 :  mean  squared error : [ 18.10594608]
lr_rate:  0.04956841305788922
----------
--------------------
Iteration 15 :  mean  squared error : [ 17.81742436]
lr_rate:  0.04315180990924041
----------
--------------------
Iteration 16 :  mean  squared error : [ 52.10837854]
lr_rate:  0.03745213411452443
----------
--------------------
Iteration 17 :  mean  squared error : [ 54.09439634]
lr_rate:  0.03241253096307968
----------
--------------------
Iteration 18 :  mean  squared error : [ 23.48152295]
lr_rate:  0.02797533298812906
----------
--------------------
Iteration 19 :  mean  squared error : [ 24.45075207]
lr_rate:  0.024083730836508242
----------
--------------------
Iteration 20 :  mean  squared error : [ 28.34336721]
lr_rate:  0.020682965145182638
----------
--------------------
Iteration 21 :  mean  squared error : [ 28.70609073]
lr_rate:  0.017721140385663577
----------
--------------------
Iteration 22 :  mean  squared error : [ 33.12278788]
lr_rate:  0.015149743303671968
----------
--------------------
Iteration 23 :  mean  squared error : [ 17.14495377]
lr_rate:  0.012923933424513973
----------
--------------------
Iteration 24 :  mean  squared error : [ 13.78995902]
lr_rate:  0.011002660480289505
----------
--------------------
Iteration 25 :  mean  squared error : [ 15.31844472]
lr_rate:  0.009348653091118545
----------
--------------------
Iteration 26 :  mean  squared error : [ 27.17433044]
lr_rate:  0.007928314257640278
----------
--------------------
Iteration 27 :  mean  squared error : [ 20.48470372]
lr_rate:  0.006711551925354233
----------
--------------------
Iteration 28 :  mean  squared error : [ 13.99175996]
lr_rate:  0.005671566840000688
----------
```

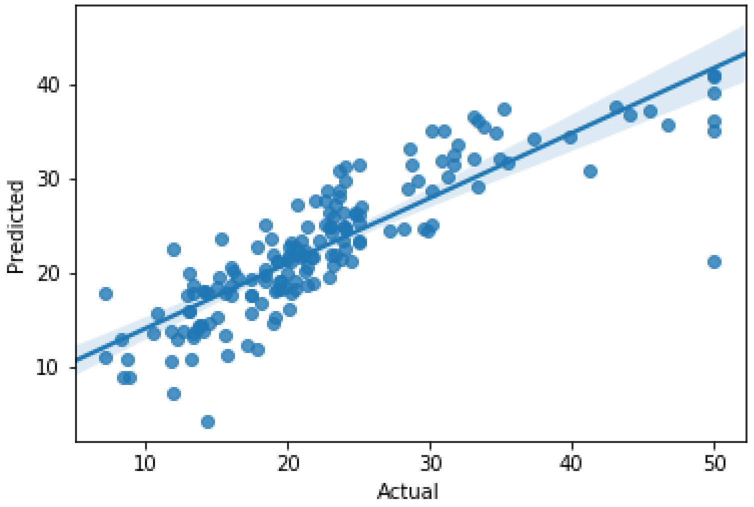
```
--------------------
Iteration 29 :  mean  squared error : [ 15.27902335]
lr_rate:  0.004784614938798966
----------
--------------------
Iteration 30 :  mean  squared error : [ 20.8187387]
lr_rate:  0.004029757453381585
----------
--------------------
Iteration 31 :  mean  squared error : [ 19.11246344]
lr_rate:  0.0033886085968905127
----------
--------------------
Iteration 32 :  mean  squared error : [ 13.30765294]
lr_rate:  0.002845088048680407
----------
--------------------
Iteration 33 :  mean  squared error : [ 30.13956293]
lr_rate:  0.002385183330304127
----------
--------------------
Iteration 34 :  mean  squared error : [ 22.7648264]
lr_rate:  0.001996725495132476
----------
--------------------
Iteration 35 :  mean  squared error : [ 33.75817345]
lr_rate:  0.00166918025298942
----------
--------------------
Iteration 36 :  mean  squared error : [ 19.68642721]
lr_rate:  0.001393455653778815
----------
--------------------
Iteration 37 :  mean  squared error : [ 40.08183626]
lr_rate:  0.001161726703474252
----------
--------------------
Iteration 38 :  mean  squared error : [ 22.5424867]
lr_rate:  0.0009672767341122829
----------
--------------------
Iteration 39 :  mean  squared error : [ 19.71794527]
lr_rate:  0.0008043549563845681
----------
--------------------
Iteration 40 :  mean  squared error : [ 19.30155866]
lr_rate:  0.0006680493556141013
----------
--------------------
Iteration 41 :  mean  squared error : [ 10.77947171]
lr_rate:  0.0005541739216873886
----------
--------------------
Iteration 42 :  mean  squared error : [ 41.22611897]
lr_rate:  0.0004591691082456614
----------
--------------------
Iteration 43 :  mean  squared error : [ 14.04456581]
lr_rate:  0.00038001437770191166
----------
--------------------
```

```
Iteration 44 :  mean  squared error : [ 27.34772574]
lr_rate:  0.00031415169161586796
----------
--------------------
Iteration 45 :  mean  squared error : [ 25.92401596]
lr_rate:  0.0002594188387963494
----------
--------------------
Iteration 46 :  mean  squared error : [ 24.32942131]
lr_rate:  0.00021399154687634468
----------
--------------------
Iteration 47 :  mean  squared error : [ 18.12495464]
lr_rate:  0.00017633338974736067
----------
--------------------
Iteration 48 :  mean  squared error : [ 33.92756921]
lr_rate:  0.0001451525775591791
----------
--------------------
Iteration 49 :  mean  squared error : [ 43.50212676]
lr_rate:  0.00011936479377123923
----------
--------------------
Iteration 50 :  mean  squared error : [ 13.43127502]
********************
State wj : [[-0.625174    1.08020076 -1.39007842  1.2845673  -0.40765815  2.93915697
  -0.50619766 -2.86436668  0.98714701 -0.70267729 -1.45645257  0.56144044
  -3.45457129]]
State wj+1 : [[-0.62514733  1.07999767 -1.3900024   1.28460618 -0.40767812  2.93881272
  -0.50603737 -2.86438383  0.98706158 -0.70279323 -1.45648774  0.56146142
  -3.45412598]]
State bj : [ 22.49808336]
State bj+1 : [ 22.49809334]
Iteration terminated at : 50
Iteration terminated at : 50
```

In [56]:

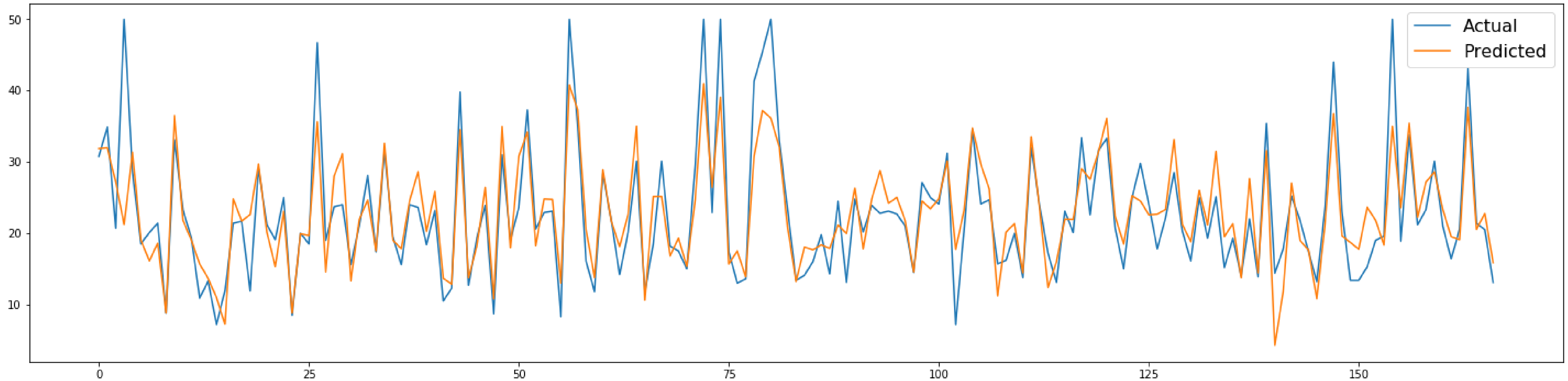
```
## Plotting functions
#Scatter plot
predict_data=pd.DataFrame(data= {'Actual' : y_test,'Predicted' :predictions})
sns.regplot(x='Actual', y='Predicted', data=predict_data)
print('Mean Squared Error :',mean_squared_error(y_test, predictions))
```

```
Mean Squared Error : 22.5748669474
```




In [57]:

```
plt.figure(figsize=(25,6))
plt.plot(y_test, label='Actual')
plt.plot(predictions, label='Predicted')
plt.legend(prop={'size': 16})
plt.show()
print('Mean Squared Error :',mean_squared_error(y_test, predictions))
```




```
Mean Squared Error : 22.5748669474
```

# Observations :

1. We see that the model performs well and gives a MSE of 29.3 as close as sklearn's SGDREgressor, with the above set parameters

## Effect of learning rate on model performance

1. Considering a set of learning rate and check the model performance of each with constant iteration 100

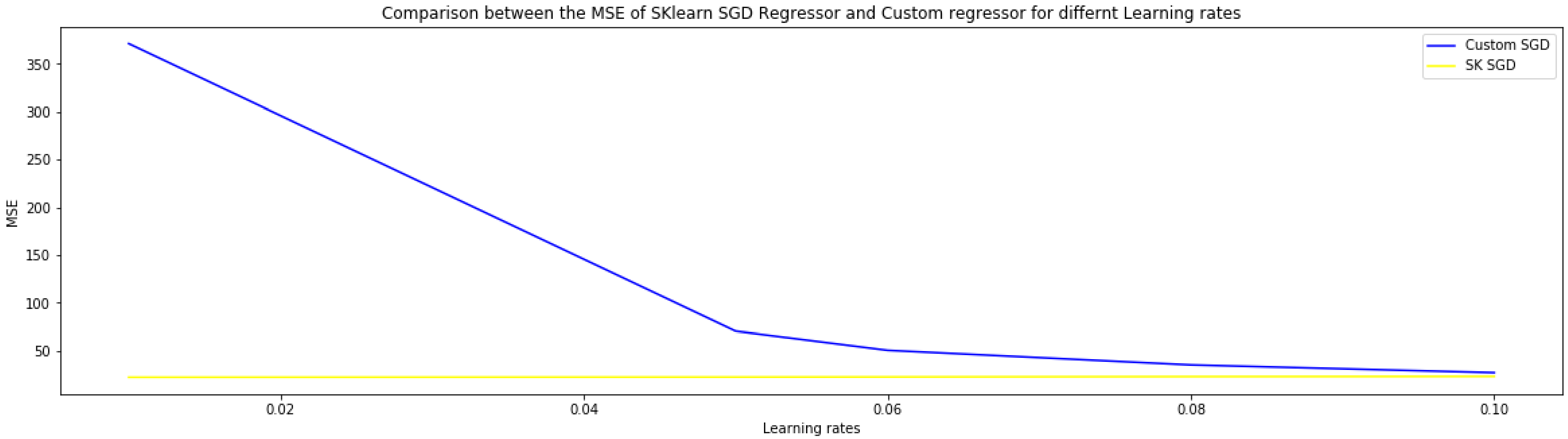
In [40]:

```python
alpha_set = [0.1,0.08,0.06,0.05,0.01]
custom_sgd_mse=[]
sk_sgd_mse=[]
for i in alpha_set:
    # train custom model
    model=SGD_scratch(transformed_train_set,i,40,500,shuffle=True,divide=1)
    w,b=model.find_w_b()
    predictions=model.predict_func(w,b,x_test)
    custom_sgd_mse.append(mean_squared_error(y_test, predictions))

    ##train SGDregressor
    sklearn_model = SGDRegressor(max_iter=500,alpha=i)
    sklearn_model.fit(x_train, y_train)
    y_pred=sklearn_model.predict(x_test)
    sk_sgd_mse.append(mean_squared_error(y_test, y_pred))
```

```
lr_rate:  0.0075748716518359844
----------
--------------------
Iteration 23 :  mean  squared error : [ 19.99525915]
lr_rate:  0.0064619667122569864
----------
--------------------
Iteration 24 :  mean  squared error : [ 29.64388956]
lr_rate:  0.005501330240144753
----------
--------------------
Iteration 25 :  mean  squared error : [ 45.95801426]
lr_rate:  0.004674326545559273
----------
--------------------
Iteration 26 :  mean  squared error : [ 9.25083364]
lr_rate:  0.003964157128820139
----------
--------------------
```

## Plotting MSE for custom SGD vs Sklearn SGD

In [41]:
```
plt.figure(figsize=(20,5))
plt.plot(alpha_set, custom_sgd_mse, color='blue', label='Custom SGD')
plt.plot(alpha_set, sk_sgd_mse, color='yellow', label='SK SGD')
plt.xlabel('Learning rates')
plt.ylabel('MSE')
plt.title("Comparison between the MSE of SKlearn SGD Regressor and Custom regressor for differnt Learning rates")
plt.legend()
plt.show()
```




## Observations

1.We can observe that our custom model has effect/fluctuations of learning rate to a greater extent than sklearn SGD.
2.With lower values of learning rate tends to decrease MSE (i.e learning rate < 0.06 )

## Effect of number of iterations on model performance

1. Considering a set of iterations rate and check the model performance of each with constant learning rate 0.05

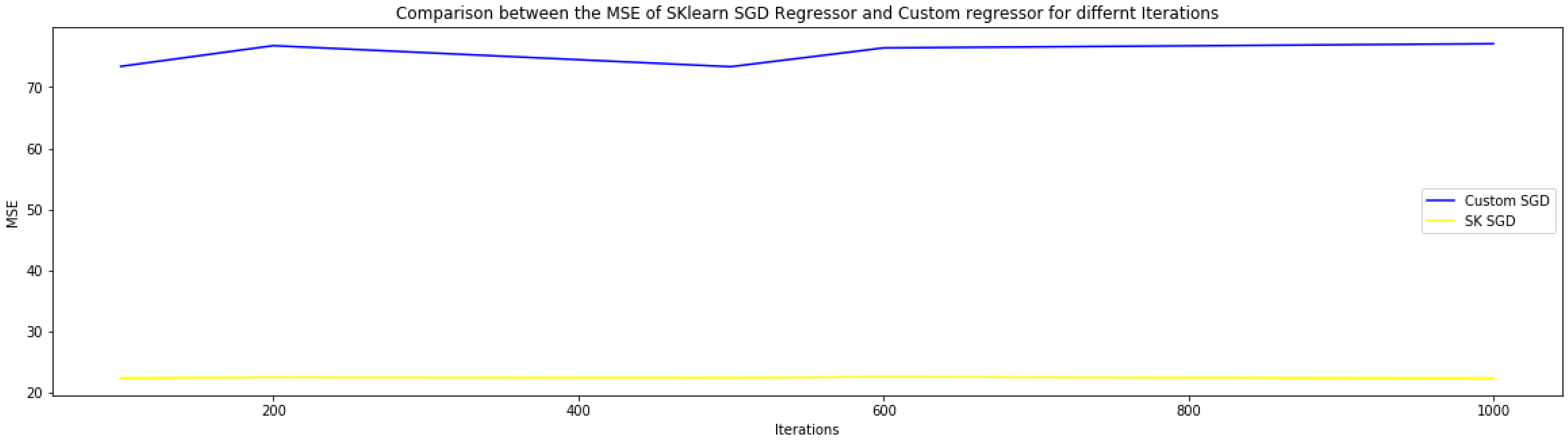
In [42]:
```
iter_set = [100,200,500,600,1000]
custom_sgd_mse=[]
sk_sgd_mse=[]
for i in iter_set:
    # train custom model
    model=SGD_scratch(transformed_train_set,0.05,40,i,shuffle=True,divide=1)
    w,b=model.find_w_b()
    predictions=model.predict_func(w,b,x_test)
    custom_sgd_mse.append(mean_squared_error(y_test, predictions))

    ##train SGDregressor
    sklearn_model = SGDRegressor(max_iter=i,alpha=0.05,shuffle=True)
    sklearn_model.fit(x_train, y_train)
    y_pred=sklearn_model.predict(x_test)
    sk_sgd_mse.append(mean_squared_error(y_test, y_pred))
```

```
Iteration 5 :  mean  squared error : [ 235.62347647]
lr_rate:  0.03598349801269221
----------
--------------------
Iteration 6 :  mean  squared error : [ 195.01186375]
lr_rate:  0.03264739077021606
----------
--------------------
Iteration 7 :  mean  squared error : [ 163.57443764]
lr_rate:  0.029423476034693776
----------
--------------------
Iteration 8 :  mean  squared error : [ 149.84551938]
lr_rate:  0.026362212271205814
----------
--------------------
Iteration 9 :  mean  squared error : [ 125.73590188]
lr_rate:  0.023495346417993123
----------
--------------------
```

## Plotting MSE for custom SGD vs Sklearn SGD

In [43]:

```
plt.figure(figsize=(20,5))
plt.plot(iter_set, custom_sgd_mse, color='blue', label='Custom SGD')
plt.plot(iter_set, sk_sgd_mse, color='yellow', label='SK SGD')
plt.xlabel('Iterations')
plt.ylabel('MSE')
plt.title("Comparison between the MSE of SKlearn SGD Regressor and Custom regressor for differnt Iterations")
plt.legend()
plt.show()
```




## Observations

1.We can observe that our custom model has effect/fluctuations of iterations to a greater extent than sklearn SGD.
2.As number of iterations increase it converges to optimumum better.

## Comparing Weights and MSE

In [44]:

```
weights
```

Out[44]:

| | Sklearn SGD | Sklearn Linear Regression | Custom SGD (Trial 1) | Custom SGD (Trial 2) |
|---|---|---|---|---|
| **0** | -0.712181 | -0.863947 | 1.652214e+12 | 0.088773 |
| **1** | 0.548530 | 0.743271 | -1.479200e+12 | 0.999658 |
| **2** | -0.709605 | -0.412015 | -1.618272e+12 | -0.749712 |
| **3** | 1.082881 | 1.000662 | -2.599226e+11 | 0.912983 |
| **4** | -1.573638 | -2.252131 | -9.759675e+11 | -0.076970 |
| **5** | 2.437577 | 2.217033 | -3.254227e+10 | 3.052044 |
| **6** | -0.056164 | 0.065871 | -9.869387e+11 | 0.302897 |
| **7** | -2.604154 | -3.255986 | -7.281176e+11 | -1.504199 |
| **8** | 1.513546 | 2.875749 | 6.985335e+11 | -0.798913 |
| **9** | -0.917969 | -1.996244 | -1.128136e+12 | -0.229589 |
| **10** | -1.927254 | -2.152966 | -1.426380e+12 | -2.003892 |
| **11** | 0.756422 | 0.751297 | -6.406850e+11 | 0.738550 |
| **12** | -3.610867 | -3.884441 | 1.531544e+12 | -2.944202 |

In [58]:

```
##http://zetcode.com/python/prettytable/

from prettytable import PrettyTable
x = PrettyTable()
x.field_names = [ "Model", "MSE"]
print('MSE comparisons ')
x.add_row(["Sklearn SGD",22.2])
x.add_row(["Sklearn Linear Regression",22.1])
x.add_row(["Custom SGD (Trial 1)",8.9e+24])
x.add_row(["Custom SGD (Trial 2)",22.5])
print(x)
```

```
MSE comparisons
+---------------------------+---------+
|           Model           |   MSE   |
+---------------------------+---------+
|        Sklearn SGD        |   22.2  |
| Sklearn Linear Regression |   22.1  |
|    Custom SGD (Trial 1)   | 8.9e+24 |
|    Custom SGD (Trial 2)   |   22.5  |
+---------------------------+---------+
```

## Conclusion:

1. After applying custom SGD to our dataset with learning rate 0.2 and iterations of 500 we see that MSE is closer to the SKlearn SGD MSE 22.2.

Ref :https://www.kaggle.com/premvardhan/stocasticgradientdescent-implementation-lr-python (https://www.kaggle.com/premvardhan/stocasticgradientdescent-implementation-lr-python), https://www.kaggle.com/tentotheminus9/linear-regression-from-scratch-gradient-descent (https://www.kaggle.com/tentotheminus9/linear-regression-from-scratch-gradient-descent)

In [ ]:

```

```