```
In [1]: from sklearn.datasets import load_boston
        boston = load_boston()
```

```
In [2]: boston.keys()
```

```
Out[2]: dict_keys(['data', 'target', 'feature_names', 'DESCR'])
```

```
In [3]: print(boston.DESCR)
```

```
Boston House Prices dataset
===========================

Notes
------
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.
http://archive.ics.uci.edu/ml/datasets/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.

**References**

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of Collinearity', Wil
ey, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Tenth Internationa
l Conference of Machine Learning, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.
   - many more! (see http://archive.ics.uci.edu/ml/datasets/Housing)
```

```
In [4]: import numpy as np
        from time import time as t
        import matplotlib.pyplot as plt
        from pandas import DataFrame
        from sklearn.linear_model import LinearRegression,SGDRegressor
        from sklearn.model_selection import train_test_split
        from sklearn.preprocessing import StandardScaler
        import matplotlib.pyplot as plt
        import matplotlib.patches as mpatches
        %matplotlib inline
        %config InlineBackend.figure_format = 'retina'#to improve plot resolution
```

```
In [5]: def SGD(x, y, n_iter = 1000, r=.001, batch_percent = 0.02):
            n_samples = x.shape[0]

            #randomly initializing weight vector with ndim=no.of features
            w = np.random.normal(0,1,x.shape[1])
            b = np.random.normal(0,1)

            for i in range(n_iter):
                temp_w = temp_b = 0
                #'k' i.e batch size
                k = int(n_samples * batch_percent)

                #picks a random set of k indices from train data
                batch = np.random.randint(n_samples,size=k)
                for j in batch:
                    #from python3.5 PEP-0465 onwards '@'
                    #is dot product/matrix multiplication operator
                    deriv = 2*(w @ x[j] + b - y[j])
                    temp_b += deriv#  dL/db = 2*(w.T*x[j] + b - y[j])
                    temp_w += deriv * x[j]#  dL/dW = dL/db * x[j]
                w -= r * (temp_w/k)
                b -= r * (temp_b/k)
            return (w,b)
```

```
In [6]: %load_ext cython
```

```
In [69]: %%cython
         #'%%cython' jupyter's magic function takes care of
         #compling and loading the shared object for cython
         import numpy as np
         cimport numpy as np

         cpdef cySGD(np.ndarray x,np.ndarray y,int n_iter = 1000,
                     float r=.001,float batch_percent = 0.02):
             cdef:
                 int i,j,n_samples = x.shape[0],
                 k = int(n_samples * batch_percent),feats = x.shape[1]
                 np.ndarray w = np.random.normal(0,1,feats) ,
                 batch , temp_w = np.zeros(feats)
                 double b = np.random.normal(0,1),temp_b,deriv

             for i in range(n_iter):
                 temp_w = np.zeros(feats)
                 temp_b = 0.0
                 #picks a random set of k indices from train data
                 batch = np.random.randint(n_samples,size=k)
                 for j in batch:
                     #from python3.5 PEP-0465 onwards
                     #'@' is dot product/matrix multiplication operator
                     deriv = 2*(w@x[j] + b - y[j])
                     temp_b += deriv#  dL/db = 2*(w.T*x[j] + b - y[j])
                     temp_w += deriv * x[j]#  dL/dW = dL/db * x[j]
                 w -= r * (temp_w/k)
                 b -= r * (temp_b/k)
             return (w,b)
```

```
In [8]: predict = lambda x,w,b:np.array([w @ x[i]+b for i in range(x.shape[0])])
```

Testing with a dummy dataset:

```
In [379]: x1=list(range(1,20))
          x2=[x1[i]+np.random.normal(0,1) for i in range(len(x1))]
          y=[x1[i]+np.random.normal(0,1) for i in range(len(x1))]
```

```
In [380]: df = DataFrame({'x1':x1,'x2':x2,'y':y})
          df
```

Out[380]:

	x1	x2	y
0	1	1.159256	1.952945
1	2	1.707344	2.622734
2	3	4.070746	1.413368
3	4	4.133119	4.088485
4	5	4.060577	5.165887
5	6	5.960416	4.371164
6	7	6.787406	5.865331
7	8	7.026555	8.357963
8	9	10.318042	9.244177
9	10	9.037655	9.964847
10	11	9.792747	10.696835
11	12	12.216509	10.692455
12	13	12.956094	15.144408
13	14	14.773174	14.669891
14	15	12.340545	16.152469
15	16	16.019576	15.764950
16	17	16.529581	17.016216
17	18	18.291263	18.480761
18	19	19.091650	19.292535

```
In [393]: x_train, x_test, y_train, y_test = train_test_split(
              df[['x1','x2']].values, df.y.values, test_size=0.2)
```

```
In [406]: w,b = SGD(x_train,y_train,1400)
```

```
In [389]: lnr = LinearRegression()
          lnr.fit(x_train,y_train)
```

```
Out[389]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

```
In [407]: DataFrame({'actual_y':y_test,
                     'predictd_y':predict(x_test,w,b),
                     'sklearn':lnr.predict(x_test)})
```

Out[407]:

	actual_y	predictd_y	sklearn
0	9.964847	10.187533	10.280230
1	14.669891	13.223899	14.098139
2	16.152469	15.665724	15.892196
3	19.292535	18.260099	19.489579

Testing on Boston House Pricing Dataset:

```
In [54]: x_train, x_test, y_train, y_test = train_test_split(
             boston.data,boston.target,test_size=0.1)
         sclr = StandardScaler()
         df_train = DataFrame(sclr.fit_transform(x_train))
         df_test = DataFrame(sclr.transform(x_test))
         df_train.columns = df_test.columns = boston.feature_names
```

```
In [55]: df_train.head()
```

Out[55]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	1.371556	-0.491007	1.003613	-0.275046	0.987686	0.378673	1.080973	-1.075744	1.640215	1.507176	0.815274	0.444103	1.183943
1	-0.315856	-0.491007	1.551781	-0.275046	0.585716	-0.921990	0.925828	-0.861316	-0.646587	0.156432	1.276770	0.444103	0.799950
2	-0.398058	-0.491007	2.397897	-0.275046	0.457427	-1.184128	0.855308	-0.935321	-0.646587	1.772606	0.769125	0.424262	0.760710
3	-0.407988	-0.491007	0.111213	-0.275046	0.149535	0.447431	0.023171	-0.629202	-0.989607	-0.810913	1.184471	0.390279	-0.415092
4	-0.405735	-0.491007	0.399038	-0.275046	-1.013615	-0.220090	-1.764514	0.778332	-0.646587	-0.716538	-1.123006	0.444103	-0.567848

```
In [56]: df_test.head()
```

Out[56]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	-0.413506	2.709683	-1.035744	-0.275046	-1.244534	-0.562445	-0.734920	1.624034	-0.760927	0.345182	1.230620	0.444103	0.303843
1	-0.400628	-0.491007	-0.376207	-0.275046	-0.303751	0.276969	1.010453	-0.650473	-0.532247	-0.156186	1.138321	0.425906	-0.050720
2	-0.404608	0.959972	-0.733457	-0.275046	-1.047825	0.307051	-1.764514	0.773212	-0.303567	-0.480600	-1.076856	0.298415	-0.555235
3	-0.404707	-0.491007	-1.028512	-0.275046	-0.389277	0.193887	0.552072	-0.551148	-0.532247	-0.675249	-0.846108	0.428756	-0.503382
4	-0.285416	-0.491007	-0.436954	-0.275046	-0.149805	-0.830313	0.936406	-0.020129	-0.646587	-0.610366	1.184471	0.026114	0.801351

Predicting Using My SGD:

```
In [80]: w,b = SGD(df_train.values,y_train,2000)
```

```
In [81]: w
```

```
Out[81]: array([-0.72876396,  0.68339651, -0.59093222,  0.9116442 , -0.47927753,
                 3.39888125, -0.56717158, -2.07065412, -0.22277168,  0.22433667,
                -1.31674386,  1.00424692, -3.09631339])
```

```
In [82]: b
```

```
Out[82]: 21.982250765111942
```

Predicting Using Sklearn's SGD Regression:

```
In [83]: sk_sgd = SGDRegressor(max_iter=2000)
         sk_sgd.fit(df_train.values,y_train)
```

```
Out[83]: SGDRegressor(alpha=0.0001, average=False, epsilon=0.1, eta0=0.01,
                fit_intercept=True, l1_ratio=0.15, learning_rate='invscaling',
                loss='squared_loss', max_iter=2000, n_iter=None, penalty='l2',
                power_t=0.25, random_state=None, shuffle=True, tol=None, verbose=0,
                warm_start=False)
```

```
In [84]: sk_sgd.coef_
```

```
Out[84]: array([-0.9792777 ,  1.18298859,  0.15644212,  0.77110917, -2.10743977,
                 2.51933478,  0.18563474, -3.07748361,  2.72189864, -2.0947233 ,
                -1.91722223,  0.93284778, -3.88900744])
```

```
In [85]: sk_sgd.intercept_
```

```
Out[85]: array([22.4754217])
```

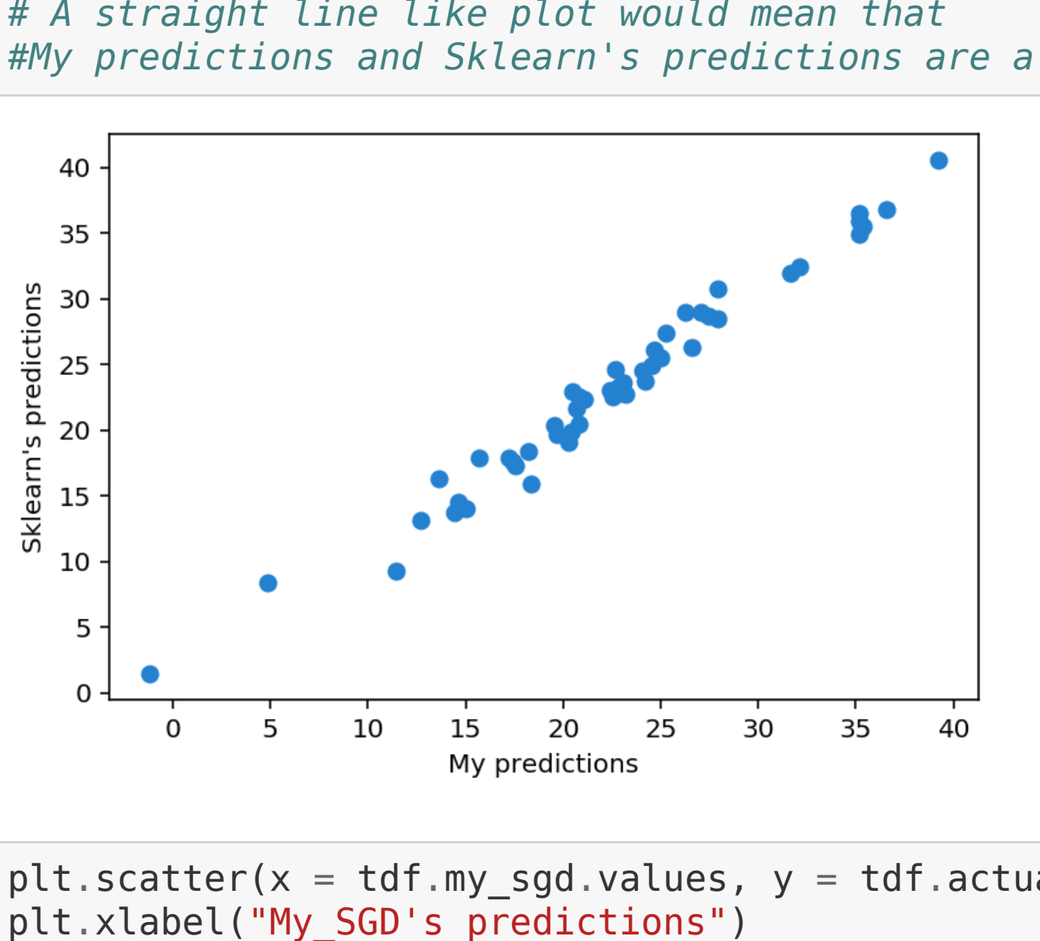
Comparision:

```
In [92]: tdf = DataFrame({'actual_y':y_test,
                          'my_sgd':predict(df_test.values,w,b),
                          'sk_sgd':sk_sgd.predict(df_test.values)}
                        )
         tdf.head(10)
```

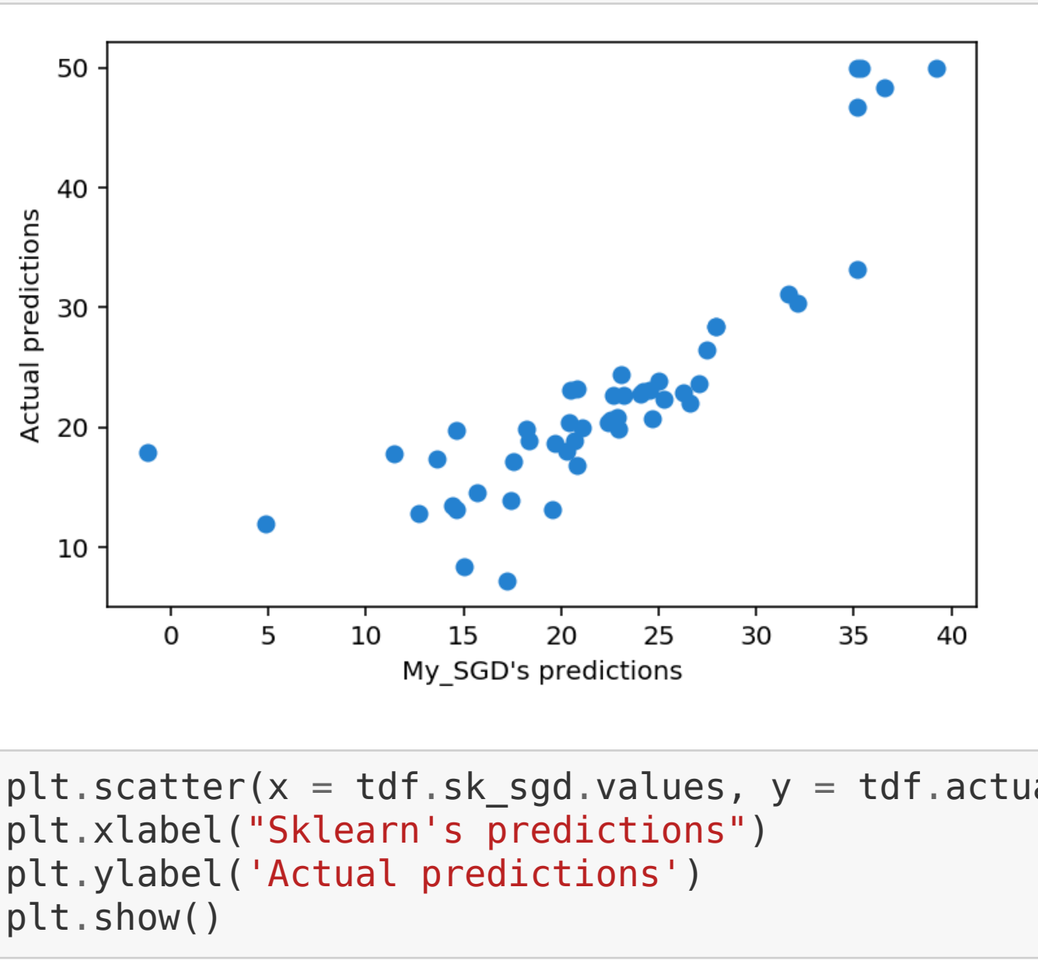
Out[92]:

	actual_y	my_sgd	sk_sgd
0	18.9	18.367237	15.862219
1	19.8	22.940415	22.833842
2	26.4	27.458010	28.637847
3	23.6	27.042762	28.970909
4	13.1	14.614863	14.508872
5	17.9	-1.230859	1.474956
6	16.8	20.819429	20.479773
7	50.0	35.383679	35.510999
8	22.7	23.214078	22.723922
9	12.8	12.725972	13.150482

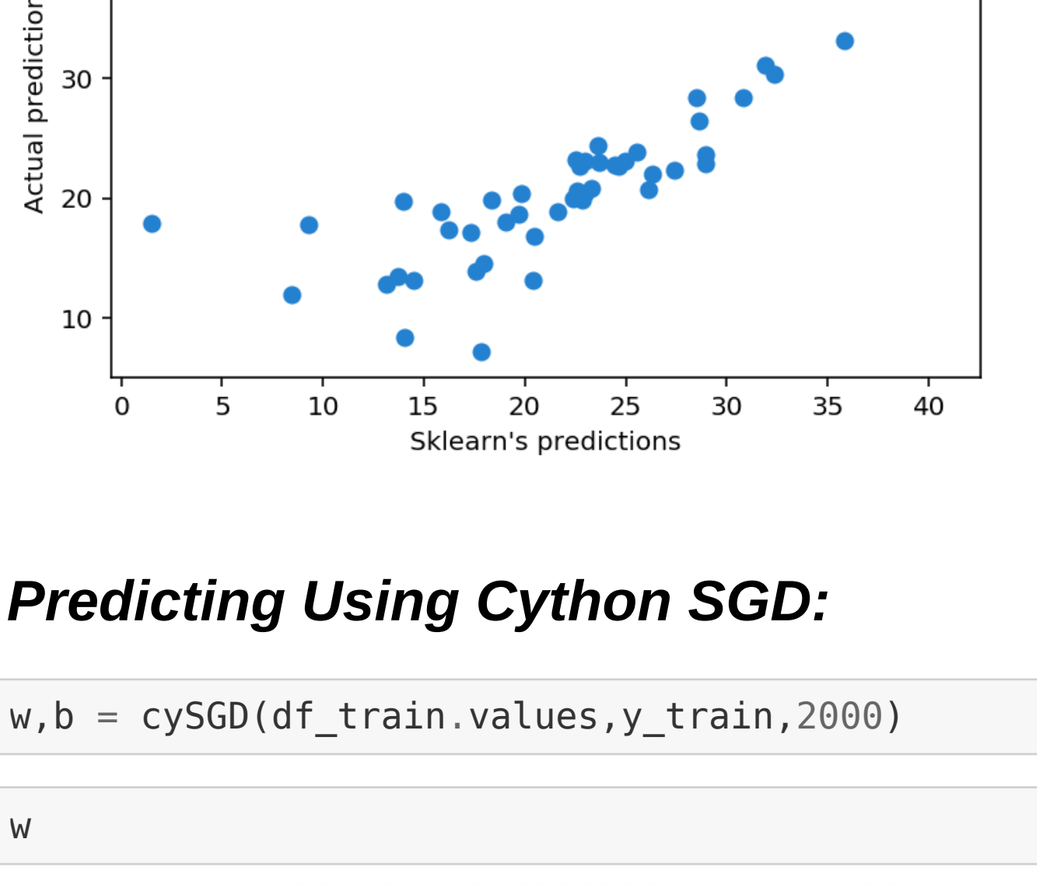
```
In [135]: plt.scatter(x = tdf.my_sgd.values, y = tdf.sk_sgd.values,color='#2481d0')
          plt.xlabel('My predictions')
          plt.ylabel('Sklearn\'s predictions')
          plt.show()
          # A straight line like plot would mean that
          #My predictions and Sklearn's predictions are almost near
```




```
In [134]: plt.scatter(x = tdf.my_sgd.values, y = tdf.actual_y.values,color='#2481d0')
          plt.xlabel("My_SGD's predictions")
          plt.ylabel('Actual predictions')
          plt.show()
```




```
In [133]: plt.scatter(x = tdf.sk_sgd.values, y = tdf.actual_y.values,color='#2481d0')
          plt.xlabel("Sklearn's predictions")
          plt.ylabel('Actual predictions')
          plt.show()
```



Predicting Using Cython SGD:

```
In [123]: w,b = cySGD(df_train.values,y_train,2000)
```

```
In [124]: w
```

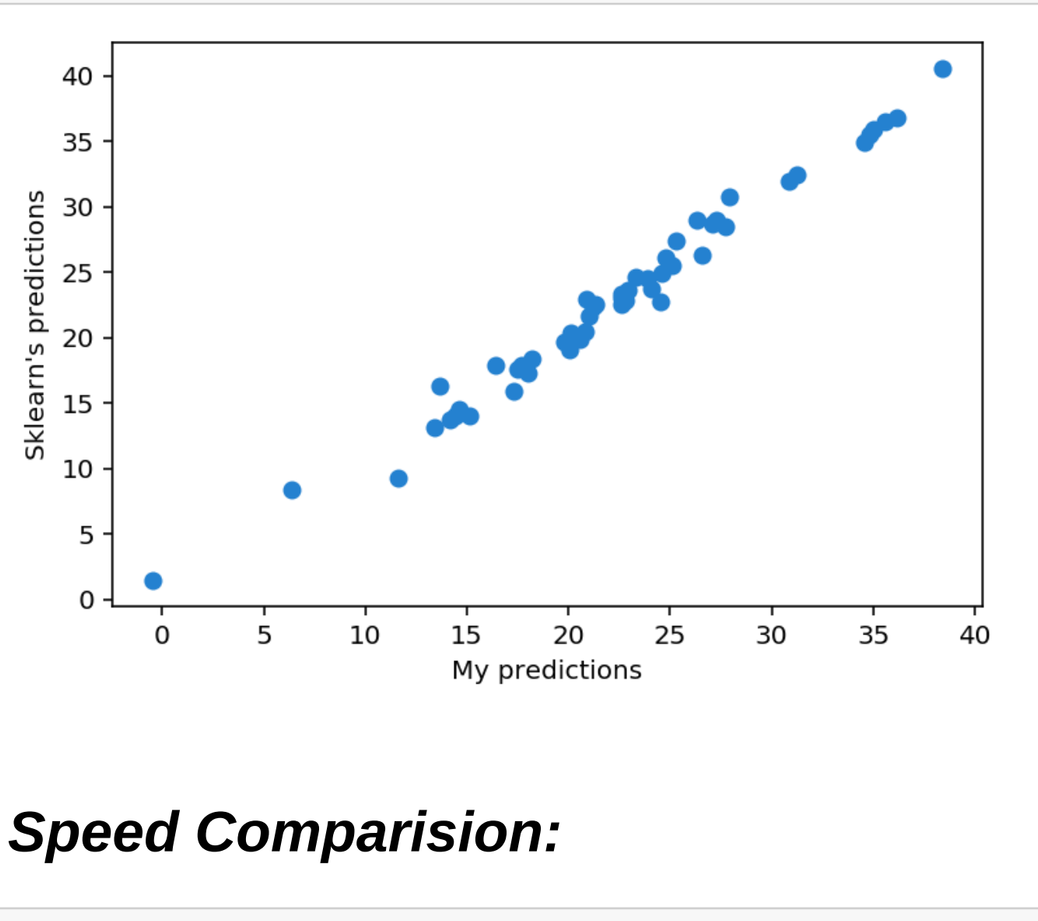
```
Out[124]: array([-0.77492273,  0.63682928, -0.45039006,  0.97198264, -0.42850322,
                  3.00481625, -0.27329114, -2.04924782,  0.43232938, -0.14049432,
                 -1.4249611 ,  0.93809801, -3.55437296])
```

```
In [125]: b
```

```
Out[125]: 22.118173263449666
```

Comparing Cython SGD with Sklearn Linear Regression:

```
In [136]: plt.scatter(x = predict(df_test.values,w,b), y = tdf.sk_sgd.values,color='#2481d0')
          plt.xlabel('My predictions')
          plt.ylabel('Sklearn\'s predictions')
          plt.show()
          # A straight line like plot would mean that
          #My predictions and Sklearn's predictions are almost near
```



Speed Comparision:

```
In [139]: %%timeit
          #pure Python
          w,b = SGD(df_train.values,y_train,2000)
```

```
87.5 ms ± 398 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
In [140]: %%timeit
          #Cython
          w,b = cySGD(df_train.values,y_train,2000)
```

```
65.4 ms ± 533 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Gist of the Assignment:

```
first of all I want to mention that I really enjoyed a lot implementing an algorithm like SGD!

for the data we used in assignments till now i.e text data there was features based on
frequency count so we didn't need to standard scalerize them
but the features in Boston House Pricing DataSet needs to be Standard Scalarized
else all weights and intercept increases exponentially.

I know that Sklearn implements their algorithms in Cython internally so I also tried it in Cython
there wasn't a remarkable speedup from pure python code to cython code ,
that maybe because in both cases I used numpy arrays
but a small ~20ms improvement can be observed that's because
python has to decide the type of a variable each time it encounters it at runtime but
on the other hand cython knows type of variables since I declared them beforehanded.
```