In [1]:

```
import warnings
warnings.filterwarnings("ignore")
from sklearn.datasets import load_boston
from random import seed
from random import randrange
from csv import reader
from math import sqrt
from sklearn import preprocessing
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from prettytable import PrettyTable
from sklearn.linear_model import SGDRegressor
from sklearn import preprocessing
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import seaborn as sns
```

Understanding load_boston() function in sklearn.datasets

In [2]:

```
print(load_boston.__doc__)
```

```
Load and return the boston house-prices dataset (regression).

    ==============   ==============
    Samples total               506
    Dimensionality               13
    Features         real, positive
    Targets           real 5. - 50.
    ==============   ==============

    Read more in the :ref:`User Guide <boston_dataset>`.

    Parameters
    ----------
    return_X_y : boolean, default=False.
        If True, returns ``(data, target)`` instead of a Bunch
object.
        See below for more information about the `data` and `t
arget` object.

        .. versionadded:: 0.18

    Returns
    -------
    data : Bunch
        Dictionary-like object, the interesting attributes ar
e:
        'data', the data to learn, 'target', the regression ta
rgets,
        'DESCR', the full description of the dataset,
        and 'filename', the physical location of boston
        csv dataset (added in version `0.20`).

    (data, target) : tuple if ``return_X_y`` is True

        .. versionadded:: 0.18

    Notes
    -----
        .. versionchanged:: 0.20
            Fixed a wrong data point at [445, 0].

    Examples
    --------
    >>> from sklearn.datasets import load_boston
    >>> boston = load_boston()
    >>> print(boston.data.shape)
    (506, 13)
```

In [3]:

```
print(load_boston().DESCR)
```

```
.. _boston_dataset:

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

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictiv
e. 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 tra
ct bounds river; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 1
0 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built
prior to 1940
        - DIS      weighted distances to five Boston employ
ment centres
        - RAD      index of accessibility to radial highway
s
        - 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 propor
tion 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/h
ousing/ (https://archive.ics.uci.edu/ml/machine-learning-da
tabases/housing/)


This dataset was taken from the StatLib library which is ma
```

```
intained 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, 'Regr
ession 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 l
earning papers that address regression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identi
fying 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 C
onference of Machine Learning, 236-243, University of Massa
chusetts, Amherst. Morgan Kaufmann.
```

In [4]:

```
print(preprocessing.StandardScaler.__doc__)
```

```
Standardize features by removing the mean and scaling to un
it variance

    The standard score of a sample `x` is calculated as:

        z = (x - u) / s

    where `u` is the mean of the training samples or zero i
f `with_mean=False`,
    and `s` is the standard deviation of the training sampl
es or one if
    `with_std=False`.

    Centering and scaling happen independently on each feat
ure by computing
    the relevant statistics on the samples in the training
set. Mean and
    standard deviation are then stored to be used on later
data using the
```

Creating data frame

In [5]:

```
column_names = load_boston().feature_names
column_names
```

Out[5]:

```
array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DI
S', 'RAD',
       'TAX', 'PTRATIO', 'B', 'LSTAT'], dtype='<U7')
```

In [6]:

```
data = pd.DataFrame(load_boston().data, columns = column_names)
data.head(5)
```

Out[6]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTI
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	

In [7]:

```
data['PRICE'] = load_boston().target
```

In [8]:

```
data.head(5)
```

Out[8]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTI
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	

In [9]:

```
data.shape
```

Out[9]:

```
(506, 14)
```

In [10]:

```
data.describe(include='all')
```

Out[10]:

	CRIM	ZN	INDUS	CHAS	NOX	RI
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.00000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.28463
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.70261
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.56100
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.88550
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.20850
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.62350
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.78000

Splitting Train and Test data

In [11]:

```
X_train, X_test, Y_train, Y_test = train_test_split(data, data["PRICE"], test

X_train.shape, X_test.shape, Y_train.shape, Y_test.shape
```

Out[11]:

```
((404, 14), (102, 14), (404,), (102,))
```

In [12]:

```
X_train.head().isnull().sum()
```

Out[12]:

```
CRIM       0
ZN         0
INDUS      0
CHAS       0
NOX        0
RM         0
AGE        0
DIS        0
RAD        0
TAX        0
PTRATIO    0
B          0
LSTAT      0
PRICE      0
dtype: int64
```

In [13]:

```
Y_train.head().isnull().sum()
```

Out[13]:

```
0
```

Data Standardization

In [14]:

```
scaler=preprocessing.StandardScaler()

X_train_no_price = X_train.drop("PRICE", axis = 1, inplace = False)
X_test_no_price = X_test.drop("PRICE", axis = 1, inplace = False)

scaler.fit(X_train_no_price)
standardized_train= scaler.transform(X_train_no_price)
standardized_test= scaler.transform(X_test_no_price)
Y_train=np.array(Y_train)
Y_test=np.array(Y_test)

print(standardized_train.shape)
print(standardized_test.shape)
```

```
(404, 13)
(102, 13)
```

In [15]:

```
standardized_train_df = pd.DataFrame(standardized_train, columns=column_names
standardized_test_df = pd.DataFrame(standardized_test, columns=column_names)
```

In [16]:

```
standardized_train_df
standardized_test_df
```

Out[16]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AG
0	-0.335938	0.390839	-1.059455	-0.278089	0.749036	0.996149	0.54412
1	-0.416767	-0.475667	0.392482	3.595975	-0.074514	-0.459521	0.8740
2	-0.395188	-0.475667	1.563398	-0.278089	0.553761	-0.154145	0.86684
3	1.012680	-0.475667	1.008676	-0.278089	0.468859	-0.874448	0.65169
4	-0.263818	-0.475667	-0.449114	-0.278089	-0.176397	-0.460890	0.45090
...	...	...	...	...	...	...	
97	-0.315573	-0.475667	-0.449114	-0.278089	-0.176397	-0.328059	0.69472
98	0.352957	-0.475667	1.008676	3.595975	0.613193	0.998887	1.00668
99	-0.389329	0.477490	-0.782826	-0.278089	-1.084849	-0.244526	-1.23795
100	-0.396544	-0.475667	-0.222249	-0.278089	0.222643	-0.493756	-0.96186
101	-0.406310	-0.475667	-0.090521	-0.278089	-0.592417	-0.143190	-0.96903

102 rows × 13 columns

In [17]:

```
standardized_train_df['PRICE'] = Y_train
print(standardized_train_df.isnull().sum())
standardized_train_df
```

```
CRIM       0
ZN         0
INDUS      0
CHAS       0
NOX        0
RM         0
AGE        0
DIS        0
RAD        0
TAX        0
PTRATIO    0
B          0
LSTAT      0
PRICE      0
dtype: int64
```

Out[17]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AG
0	-0.416817	0.607466	-0.889672	-0.278089	-0.898064	0.232025	-0.05826
1	-0.324497	-0.475667	1.563398	-0.278089	0.553761	-0.889512	0.90628
2	-0.425545	-0.475667	-0.880890	-0.278089	-0.371671	0.032093	-1.10887
3	-0.420468	1.257346	-0.702325	-0.278089	-0.949005	0.267629	-1.33835
4	-0.420177	0.477490	-0.782826	-0.278089	-1.084849	0.918093	-2.24553
...	...	...	...	...	...	...	
399	-0.414692	-0.475667	-0.387641	-0.278089	-0.329220	0.256674	0.99233
400	-0.414536	-0.475667	0.396872	-0.278089	-1.033907	0.536031	-1.37421
401	-0.422746	-0.475667	0.105607	-0.278089	0.120761	0.944112	0.77361
402	-0.412014	-0.475667	2.113728	-0.278089	0.188682	-0.589614	0.98875
403	-0.366925	-0.475667	-0.733062	-0.278089	-0.465064	3.339188	0.48675

404 rows × 14 columns

In [18]:

```
standardized_test_df['PRICE']=Y_test
print(standardized_test_df.isnull().sum())
standardized_test_df
```

```
CRIM       0
ZN         0
INDUS      0
CHAS       0
NOX        0
RM         0
AGE        0
DIS        0
RAD        0
TAX        0
PTRATIO    0
B          0
LSTAT      0
PRICE      0
dtype: int64
```

Out[18]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AG
0	-0.335938	0.390839	-1.059455	-0.278089	0.749036	0.996149	0.54412
1	-0.416767	-0.475667	0.392482	3.595975	-0.074514	-0.459521	0.8740
2	-0.395188	-0.475667	1.563398	-0.278089	0.553761	-0.154145	0.86684
3	1.012680	-0.475667	1.008676	-0.278089	0.468859	-0.874448	0.65169
4	-0.263818	-0.475667	-0.449114	-0.278089	-0.176397	-0.460890	0.4509(
...	...	...	...	...	...	...	
97	-0.315573	-0.475667	-0.449114	-0.278089	-0.176397	-0.328059	0.69472
98	0.352957	-0.475667	1.008676	3.595975	0.613193	0.998887	1.00668
99	-0.389329	0.477490	-0.782826	-0.278089	-1.084849	-0.244526	-1.23795
100	-0.396544	-0.475667	-0.222249	-0.278089	0.222643	-0.493756	-0.96186
101	-0.406310	-0.475667	-0.090521	-0.278089	-0.592417	-0.143190	-0.96903

102 rows × 14 columns

Stochastic Gradient Descent Implementation

In [19]:

```
weight = np.random.randn(13)   # defining initial random weight from normal d
b = np.random.randn(1)   # generating initial random y-intercept from normal

learningRate = 0.1

for i in range(5000):    # running 5000 iterations
    Data_batch_10 = standardized_train_df.sample(n = 10)   # taking 10 stocha
    X = Data_batch_10.drop("PRICE", axis = 1, inplace = False)
    Y = Data_batch_10["PRICE"]
    PartialGradient = np.empty(13)
    sum2 = 0

    for j in range(13):     # as there are 13 dimensions in our dataset and d
        sum1 = 0
        for k in range(10):
            sum1 += -2 * X.iloc[k][j] * (Y.iloc[k] - np.dot(weight, X.iloc[k]
        PartialGradient[j] = sum1
    PartialGradient *= learningRate
#     print("Partial Gradient = "+str(PartialGradient))
#     print("Iteration number = "+str(i))

    for m in range(10):
        sum2 += -2 * (Y.iloc[m]- np.dot(weight, X.iloc[m]) - b)     # this is
    b = b - learningRate * sum2       #updating y-intercept 'b'


    for l in range(13):
        weight[l] -= PartialGradient[l]  # updating weights

    learningRate = 0.01 / pow(i+1, 0.25)     #learning rate at every iteratio


    weight = weight + 0.0001*np.dot(weight, weight)   #adding l2 regularizati
    b = b + 0.0001*np.dot(weight, weight) #adding L2 regularization

print("Weight = "+str(weight))
print("b = "+str(b))
```

```
Weight = [-1.11743617  1.33961154  0.82490262  0.8899498  -1.1
1578166  3.34938305
  0.04245584 -1.96149294  3.31820329 -2.69661236 -1.83149772
1.41364728
 -3.01396011]
b = [22.78205855]
```

In [20]:

```
#Results with different values of lr, weight, b and number of iterations
#14.793501965680582, 14.999,
#43.86016682060379, 34.623089278732245
#30.618230741632974 23.68008875764373
#35.38538690393456 31.734577285851948
#37.34140422232749 34.85719925686704 5k
#17.119463909490804 13.982028036046621 5k
#lr = 0.1 18.656299080261185 19.236328857973678
#35.10227157756609 31.836553250205604
#31.585165382559303 30.140203361795436
#30.351643740443986 31.01401009275466 3k
#18.329567718238483 18.822987303319824 5k
#lr 0.1, np.sqr, 5k 34.16413455371752 32.29560412336732
#lr 0.1, np.ones,5k 22.510381819122248 19.47328408419087
#25.060003188534818 23.28770502524364
#32.66000232875092 25.160381796731883
```

In [21]:

```
import math
test_data = standardized_test_df.drop("PRICE", axis = 1, inplace = False)
test_labels = standardized_test_df["PRICE"]
y_predicted = []

for i in range(102):
    test_i = 0
    test_i = np.dot(weight, test_data.iloc[i]) + b[0]  #making prediction by
    y_predicted.append(test_i)

y_true = []
for i in range(102):
    y_true.append(test_labels.iloc[i])
```

In [22]:

```
d1 = {'True Labels': Y_test, 'Predicted Labels': y_predicted}
df1 = pd.DataFrame(data = d1)
df1
```

Out[22]:

	True Labels	Predicted Labels
0	30.7	31.320988
1	21.5	26.114399
2	14.0	15.461704
3	20.8	18.186321
4	13.2	9.620679
...	...	...
97	14.8	15.122796
98	50.0	35.319184
99	24.3	21.021666
100	24.5	20.078456
101	25.0	24.036250

102 rows × 2 columns

In [23]:

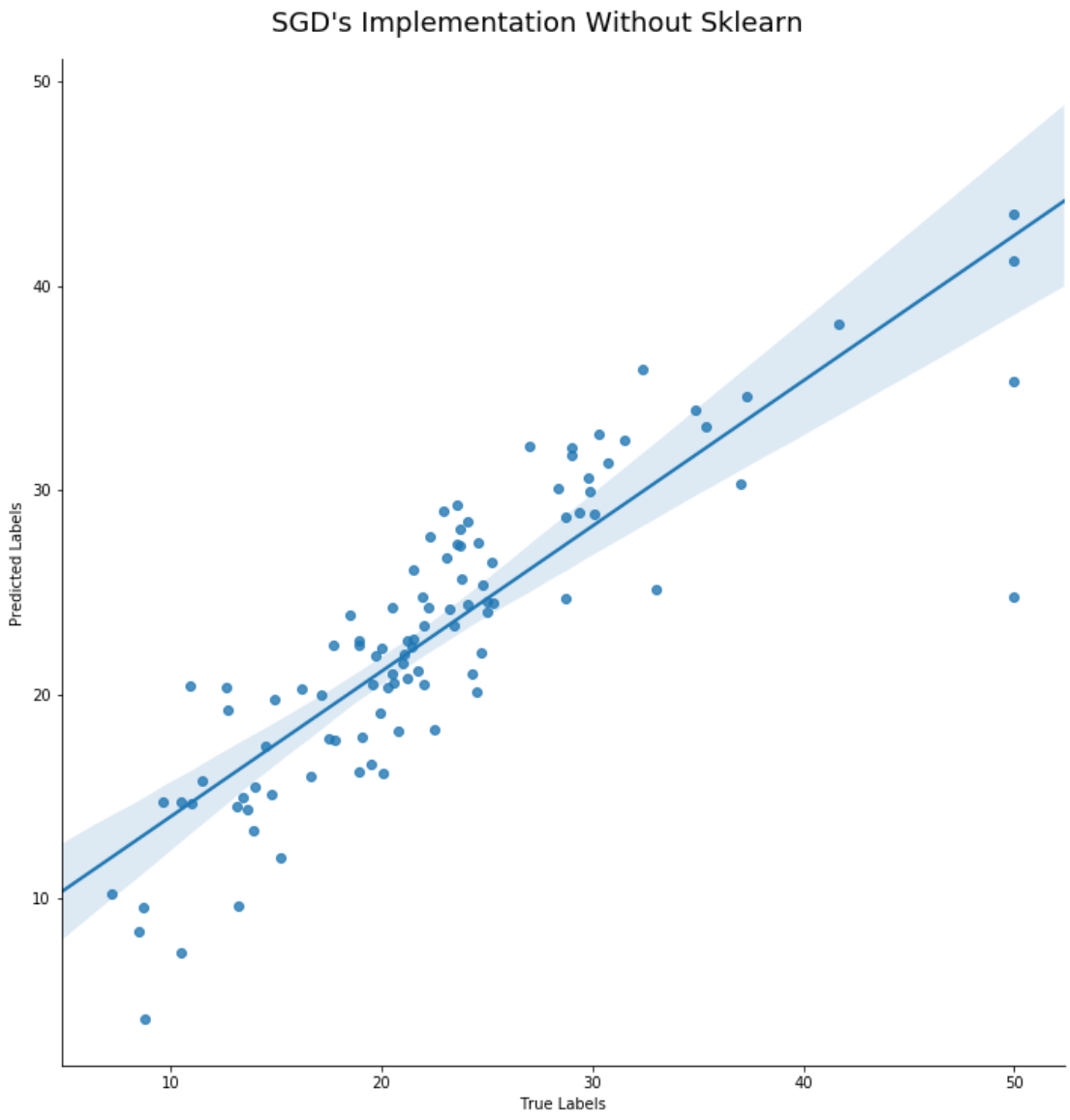
```
Mean_Sq_Error = mean_squared_error(y_true, y_predicted)
Mean_Sq_Error
```

Out[23]:

```
19.79642605519233
```

In [24]:

```
fig1 = sns.lmplot(x="True Labels", y="Predicted Labels", data = df1, size = 1
fig1.suptitle("SGD's Implementation Without Sklearn", fontsize=18, y=1.03)
fig1.show()
```



In [25]:

```
X = standardized_train_df.drop("PRICE", axis = 1, inplace = False)
Y = Y_train
X_te = standardized_test_df.drop("PRICE", axis = 1, inplace = False)
Y_te = Y_test
clf = SGDRegressor(shuffle = False, learning_rate= 'invscaling', max_iter = 5
clf.fit(X, Y)# fir train data
Y_pred = clf.predict(X_te)# predict test error
print("Weight = "+str(clf.coef_))
print("Y Intercept = "+str(clf.intercept_))
```

```
Weight = [-0.94554351  1.05190329 -0.12895158  0.63797891 -2.1
85374    2.60099438
 -0.14401095 -3.34889106  2.73055477 -1.98842653 -2.3150537
0.75699472
 -3.72076626]
Y Intercept = [22.57838697]
```

In [26]:

```
d2 = {'True Labels': Y_te, 'Predicted Labels': Y_pred}
df2 = pd.DataFrame(data = d2)
df2
```

Out[26]:

	True Labels	Predicted Labels
0	30.7	31.963188
1	21.5	24.946039
2	14.0	13.239594
3	20.8	18.680453
4	13.2	9.106476
...	...	...
97	14.8	14.639654
98	50.0	33.968334
99	24.3	19.971228
100	24.5	21.077130
101	25.0	24.695380

102 rows × 2 columns

In [27]:

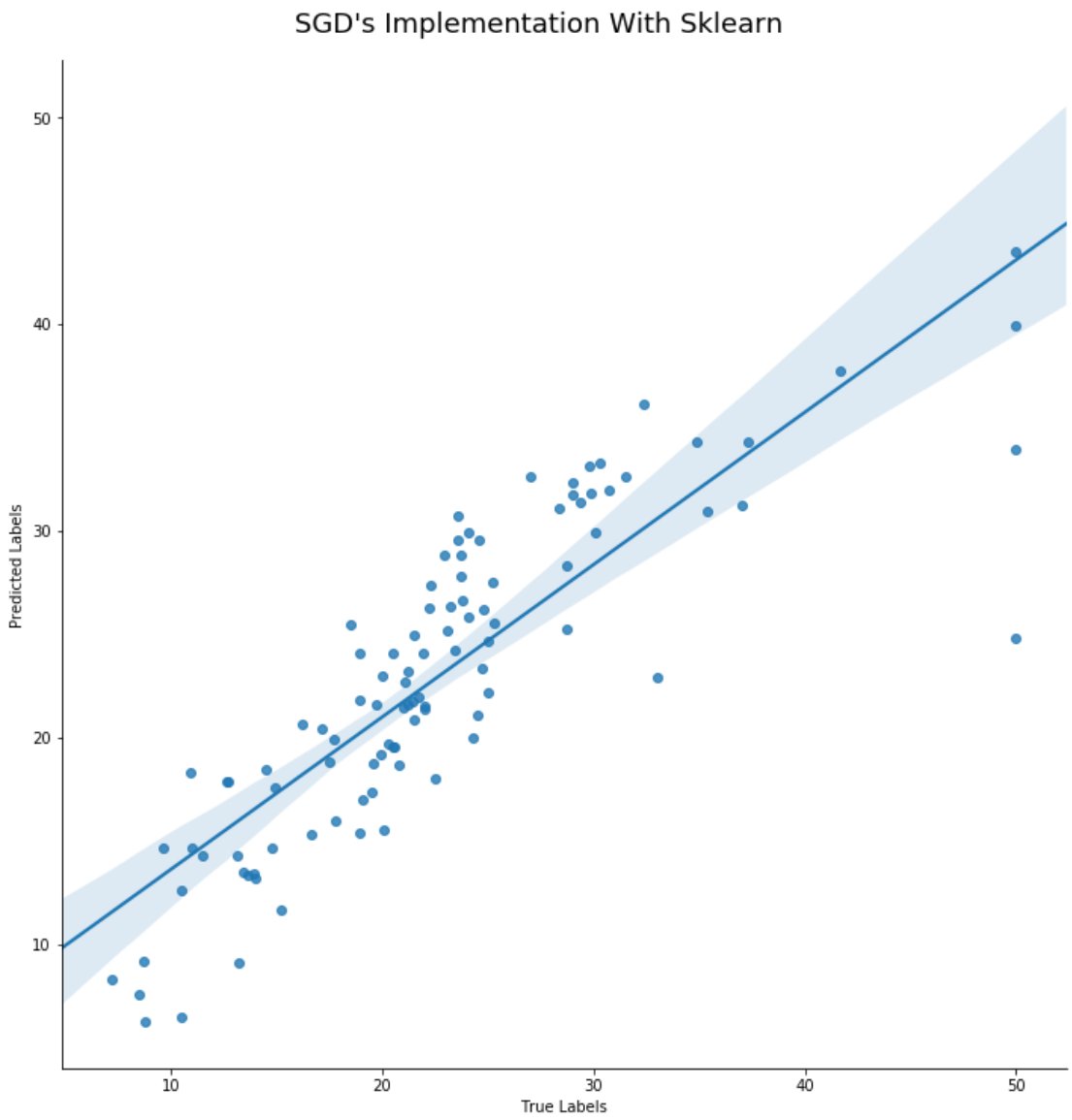
```
Mean_Sq_Error = mean_squared_error(Y_te, Y_pred)
Mean_Sq_Error
```

Out[27]:

```
21.15212674927223
```

In [28]:

```
fig2 = sns.lmplot(x="True Labels", y="Predicted Labels", data = df2, size = 1
fig2.suptitle("SGD's Implementation With Sklearn", fontsize=18, y=1.03)
fig2.show()
```



Summary

- Mean Squared Error obtained by my implementation of SGD is 19.79642605519233
- Mean Squared Error obtained by Sklearn's implementation of SGD is 21.15212674927223