INTRODUCING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 11: REGRESSION

BY FRED ANNEXSTEIN, PHD

PYTHON PROGRAMMING

11. REGRESSION

BY FRED ANNEXSTEIN, PHD

CHAPTER 11 OVERVIEW

In this module we will be covering an important method for data science called regression. We will consider methods of simple linear regression with both singular and multiple variable data sets. We will practice methods using several new libraries and new datasets, one based on time-series of city weather data, and another using state-level housing data.

CHAPTER 11 OBJECTIVES

By the end of this chapter, you will be able to...

Understand regression problems and solutions using time-series and geographic data.

Understand forecasting with simple and multiple linear regression.

Practice processing datasets for regression analysis.

Practice using the Scipy module for linear regression and visualizing the regression line using the Seaborn module.

Using regression for making prediction for city temperature changes over time.

Applying multiple linear regression to the problem of geographic data of housing

prices.

Practice using scoring methods for evaluating the modeling power of linear regression.

REGRESSION PROBLEMS

Regression is another important example of a supervised learning problem, one which has as input a data table with an additional attribute that we want to be able to predict. With solutions to regression problems the desired output consists of a mathematical model (or formula) using one or more continuous variables to predict the value of the target. A prototypical example of a regression problem would be the prediction of economic statistics, such as the next months inflation rate or the unemployment rate. In this chapter we will be working with temperature data for cities and geographic data on housing prices. We will be trying to find a linear model to predict future average monthly temperatures and the future of housing prices.

TIME SERIES

A time-seres is any data set of observations ordered in time. In one of our

running example problems in this chapter, the data that we'll use is a time series in which the observations or samples are monthly average temperatures for cities ordered by year.

The times series we process here are 'univariate time series' which have one observation per time unit. We look at univariate time-series for the average of the October high temperatures in Cincinnati, and the January high temperatures in New York City. We will process these time-series over several consecutive years. We wish to use these time-series to predict future temperatures. Since temperature varies by the season, our time series will only consist of one month per year, restricting our predictions to that single month.

More data can be incorporated in a prediction model using multivariate time series, which have two or more observations per sample time unit. Multivariate models could account for more weather information such as humidity and barometric pressure readings. To begin we will analyze a simple univariate time series. Later in the chapter we will address using a multivariate data set.

There are two major tasks often performed with time series, they are analysis and forecasting. Time series analysis looks at

existing time series data for patterns, with the goal of helping data analysts understand the aggregate data. A common analysis task is to look for seasonality in the data. This seasonality is surely to be found in the temperature time-series data of historical weather from cities.

FORECASTING WITH SIMPLE LINEAR REGRESSION

Using a technique called simple linear regression, we can make predictions on future (or historical) values. Simple linear regression finds the best-fitting line on the graph (or plot) of the dependent (or observation) and independent (or prediction target) variables. For our running example, the dependent variable is the month/year, and the target variable is the average high temperature for that calendar date. Simple linear regression describes the relationship between these variables with a straight line, and line that best-fitting for the data set is known as the regression line.

COMPONENTS OF THE SIMPLE LINEAR REGRESSION

The points along any straight line in two

dimensions, can be computed with the unique x,y-equation for that line:

```
 y = mx + b

where
• m is the line's slope,
• b is the line's intercept with the y-axis (at x = 0),
• x is the independent variable (the date in our example), and
• y is the dependent variable (the temperature in our example).
```

In simple linear regression, y = mx + b is the predicted value for any given x. Simple linear regression determines the two parameters, the slope (m) and intercept (b) of a straight line that best fits the data expressed as points in two dimensions (x,y). Hence, when we call a library function to perform a linear regression, then object returned should have two attributes - a slope and an intercept.

GETTING WEATHER DATA FROM NOAA

Let us now apply the regression method to the weather data for our regression study. The National Oceanic and Atmospheric Administration (NOAA) offers public historical data including time series for average high temperatures in specific cities over various

time intervals.

We obtained the data from NOAA's "Climate at a Glance" website:

https://www.ncdc.noaa.gov/cag/

We downloaded the October average high temperatures for Cincinnati from 1948 through 2022, and we also downloaded the January average high temperatures for New York city from 1895 through 2022.

LOADING THE AVERAGE HIGH TEMPERATURES INTO A DATAFRAME

Let's load and display the data from two .csv files using the method read_csv() from the Pandas library. Once read, we can look at the DataFrame's head and tail to get a sense of the structure of the data sets.

```
import pandas as pd

cincy= pd.read_csv('/Users/fred/Desktop/
cincy_1948_2022.csv')

nyc = pd.read_csv('/Users/fred/Desktop/
ave_hi_nyc_jan_1895-2022.csv')

>>> cincy.shape
(75, 3)

>>> nyc.shape
(124, 3)
```

```
>>> cincy.head()

     Date  Value  Anomaly
0  194810   52.1     -3.3
1  194910   59.3      3.9
2  195010   59.2      3.8
3  195110   57.7      2.3
4  195210   49.7     -5.7

>>> cincy.tail()

      Date  Value  Anomaly
70  201810   57.4      2.0
71  201910   59.6      4.2
72  202010   56.1      0.7
73  202110   61.7      6.3
74  202210   55.1     -0.3

>>> nyc.head()

   Date Value Anomaly
0 189501 34.2 -3.2
1 189601 34.7 -2.7
2 189701 35.5 -1.9
3 189801 39.6 2.2
4 189901 36.4 -1.0

>>> nyc.tail()

    Date Value Anomaly
119 201401 35.5 -1.9
120 201501 36.1 -1.3
121 201601 40.8 3.4
122 201701 42.8 5.4
123 201801 38.7 1.3
```

We wish to simplify the date column to make it just the year of the sample. Since the values are integers we can do an integer

divide by 100 to truncate the last two digits. Recall that each column in a DataFrame is a Series. Calling the Series method floordiv performs this integer division on every element of the Series, as seen as follows:

```
cincy.columns = ['Date', 'Temperature',
‘Anomaly']
cincy.Date = cincy.Date.floordiv(100)

nyc.columns = ['Date', 'Temperature',
‘Anomaly']
nyc.Date = nyc.Date.floordiv(100)
```

Let us look at the basic mean, standard deviation, and quartile statistics for the Temperature column.

```
>>> cincy.Temperature.describe()

count    75.000000
mean     55.769333
std       3.150460
min      48.500000
25%      53.850000
50%      55.900000
75%      57.550000
max      62.000000
Name: Temperature, dtype: float64

>>> nyc.Temperature.describe()
count    124.000000
mean      37.595161
std        4.539848
min       26.100000
25%       34.575000
```

```
50%       37.600000
75%       40.600000
max       47.600000
Name: Temperature, dtype: float64
```

LINEAR REGRESSION USING SCIPY

The SciPy (Scientific Python) library is widely used for applications in engineering, science and math. SciPy has a stats module which provides a function to do linear regression called linregress(). Here is how to import and execute this function on the cincy and nyc DataFrames. The object returned by linregress() has two attributes - a slope and an intercept as expected.

```
from scipy import stats

cincy_regress=
stats.linregress(x=cincy.Date,
                 y=cincy.Temperature)

>>> cincy_regress.slope
0.1814285714285704

>>> cincy_regress.intercept
52.33428571428573

nyc_regress= stats.linregress(x=nyc.Date,
                              y=nyc.Temperature)

>>> nyc_regress.slope
0.014771361132966163

>>> nyc_regress.intercept
```

```
8.694993233674289
```

The line defined by the slope and intercept pair gives us a simple way make predictions or forecasts of future temperatures. We can make a prediction for the year 2023, by plugging in to slope-intercept formula as follows:

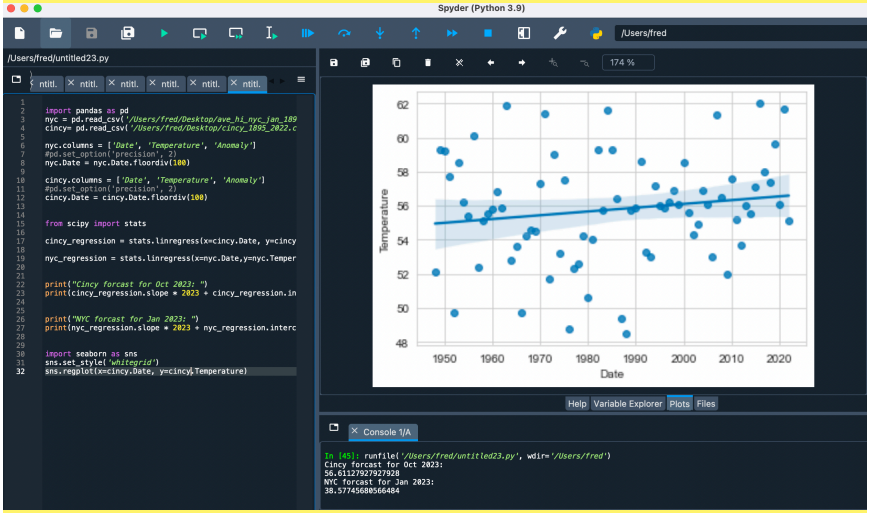
```
#Cincy forecast for avg high temp Oct 2023

>>> cincy_regress.slope * 2023 +
cincy_regress.intercept

56.6112792792792

# NYC forecast for avg high temp Jan 2023

>>> nyc_regress.slope * 2023 +
nyc_regression.intercept

38.57745680566484
```

DISPLAYING THE REGRESSION LINE USING SEABORN

We will use the Seaborn library to visualize the plot of the simple linear regression. To do this we use Seaborn's regplot function to plot each data point with the dates on the x-axis and the temperatures on the y-axis. The regplot function creates the scatter plot or scattergram in which the scattered blue dots

represent the Temperatures for the given Dates, and the straight line displayed through the points is the regression line.

```
import seaborn as sns
sns.set_style('whitegrid')
axes = sns.regplot(x=cincy.Date, y=cincy.Temperature)
```



The function regplot has x and y keyword arguments, which are expected to be one-dimensional arrays of the same length representing the x-y coordinate pairs to plot.

Recall that pandas automatically creates attributes for each column name if the name can be a valid Python identifier. Note the slope of the regression line in the plot below. It is clearly positive, showing that the trend is

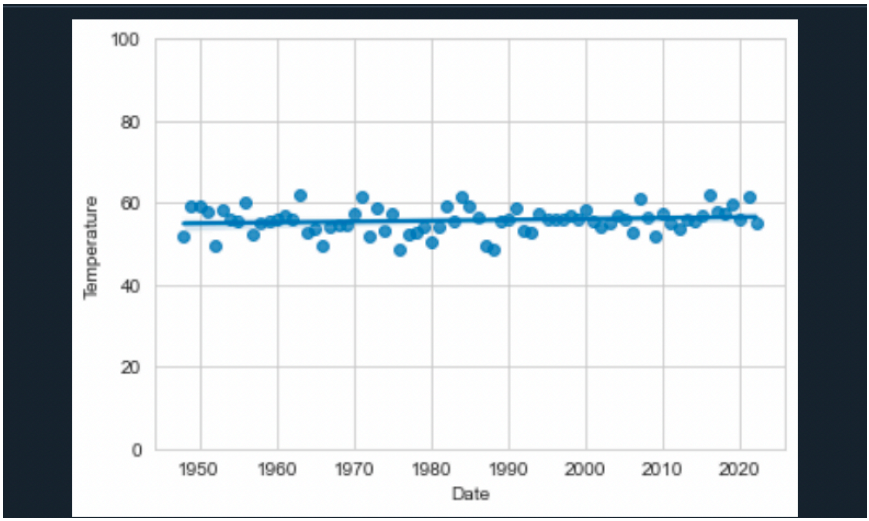
that temperature is rising by about 0.18 degrees per year for the cincy dataset.

SCALING FOR BETTER VISUALIZATION

The regression line's slope (lower at the left and higher at the right) indicates a warming trend over the last several years. In this graph, the y-axis represents a range of about 35-degrees. Thus the data appears to be spread significantly above and below the regression line, making it difficult to see the linear relationship between the x and the y variables. This is a common issue in data analytics visualizations, particularly when you have axes that reflect different kinds of data (i.e., dates and temperatures in this case).

In the preceding graph, this range is correlated to the height of the graph. The Seaborn and Matplotlib modules auto-scale the axes, based on the data's range of values. However, we can scale the y-axis range of values to emphasize the linear relationship. Here, we scaled the y-axis from a 35-degree range to a 100-degree range (from 0 to 100 degrees). An we show the resulting plot which has a more linear character.

```
axes = sns.regplot(x=cincy.Date,
y=cincy.Temperature)
axes.set_ylim(0, 100)
```



PREDICTING WHEN CERTAIN TEMPERATURE WILL BE REACHED

Assuming that the linear trend continues, we can ask the question what year do we expect the average October temperature in Cincinnati reach 70 degrees Fahrenheit. Of course we can use algebra to solve.

```
70 = m Year + b, so

Year = 70 – b / m
```

```
#Script for the Linear Regression Solution

import pandas as pd
cincy= pd.read_csv('/Users/fred/Desktop/
cincy_1948_2022.csv')
cincy.columns = ['Date', 'Temperature', '
Anomoly']
cincy.Date = cincy.Date.floordiv(100)

slope = cincy_regress.slope
intercept = cincy_regress.intercept

print("The Year for 70F for Oct in Cincy:" ,
(70 – intercept) / slope )

The Year for 70F for Oct in Cincy:
2627.2803458311932
```

An alternative is to run a while loop along the regression line starting from this year 2023.

```
thisyear = 2023

temp = slope * thisyear + intercept
while temp < 70.0:
    thisyear += 1
    temp = slope * thisyear + intercept

print(thisyear) # prints value approx 2628
```

REGRESSION WITH SKLEARN

The sklearn library gives us more powerful

tools to practice with linear regression, and can potentially be used to improve the model through more rigorous testing.

To carry out linear regression we will use the LinearRegression estimator from sklearn.linear_model module. By default, this estimator uses all the numeric features present in a dataset. Here, we perform simple linear regression using one feature as the independent variable. So, we'll need to select the Date feature from the dataset. When you select one column from a two-dimensional DataFrame, the result is a one dimensional Series.

However, scikit-learn estimators require their training and testing data to be two-dimensional arrays, or two-dimensional array-like data, such as lists of lists or Pandas DataFrames. To use one-dimensional data with an estimator, you must transform it from one dimension containing n elements, into two dimensions containing n rows and one column as you'll see below.

As we did in the previous case study, let's split the data into training and testing sets. Note the use of the keyword argument random_state, this is used for the purposes of reproducibility.

TRAINING THE MODEL

Sklearn does not have a separate class for simple linear regression. For sklearn we train a LinearRegression estimator as follows.

LinearRegression estimator computes the best fitting regression line for the data. The LinearRegression estimator iteratively adjusts to optimize the choice of slope and intercept values so as to minimize the sum of the squares of the distances of the data from the line.

The optimal slope is stored in the estimator's coeff_ attribute (m in the equation) and the optimal intercept is stored in the estimator's intercept_ attribute (b in the equation).

Note that the values obtained are somewhat different from the Scipy lingress function above, since we are using for the training dataset a subset of the original data.

```
from sklearn.linear_model import
LinearRegression

regress = LinearRegression()
regress.fit(X=X_train, y=y_train)

>>> regression.coef_
array([0.02116697])
```

```
>>> regress.intercept_
13.69890898941808
```

COMPARING SCIPY TO SKLEARN

Here is a script which shows the differences between using SciPy and Sklearn versions of Linear Regression. Sklearn is designed to create training and testing using random subsets of the data. Therefore the models will differ somewhat based on the random subset selected for training, which we show now.

```
import pandas as pd
cincy= pd.read_csv('/Users/fred/Desktop/
cincy_1948_2022.csv')
cincy.columns = ['Date', 'Temperature', '
Anomoly']
cincy.Date = cincy.Date.floordiv(100)


from scipy import stats
cincy_regress=
stats.linregress(x=cincy.Date,
                 y=cincy.Temperature)
slope = cincy_regress.slope
intercept = cincy_regress.intercept
print("SciPy Regression Solution:", slope,
intercept)



from sklearn.model_selection import
train_test_split
import random
```

```
X_train, X_test, y_train, y_test =
train_test_split(
    cincy.Date.values.reshape(-1, 1),
cincy.Temperature.values,
random_state=random.randint(1,10))


from sklearn.linear_model import
LinearRegression
skregress = LinearRegression()
skregress.fit(X=X_train, y=y_train)
print("Sklearn Regression Solution:",
skregress.coef_, skregress.intercept_ )

Output:

SciPy Regression Solution:
0.02215647226173543 11.788735893788505

Sklearn Regression Solution:
[0.02116697] 13.69890898941808
```

RESHAPING SERIES FOR TRAINING

Note that in the code above there several uses of the expression cincy.Date, which returns the Series object for the Date column. The Series values attribute returns a NumPy array containing that Series values.

To train we must transform this one dimensional array into two dimensions. To do this we call the array's reshape method. Normally, we pass two arguments which are the precise number of rows and columns. However, when the first argument is -1, this tells the reshape to infer the number of rows,

based on the number of columns and the number of elements in the original array. The transformed array will have only one column.

We can confirm the 75%–25% train-test split by checking the shapes of X_train and X_test, as follows:

```
>>> X_train.shape
(56, 1)
>>> X_test.shape
(19, 1)
```

TESTING THE MODEL

Let us now test the model using the data in X_test and check some of the predictions throughout the dataset by displaying the predicted and expected values for every fifth element, which we do by slicing the two arrays and zipping them together into a single object z:

```
predicted = regress.predict(X_test)
expected = y_test
z = zip(predicted[::5], expected[::5])

for p, e in z:
    print(f'predicted: {p:.2f}, expected:
{e:.2f}')

Output:
predicted: 55.26, expected: 54.20
predicted: 55.49, expected: 53.20
```

```
predicted: 56.09, expected: 53.30
predicted: 55.93, expected: 49.40
```

MULTIPLE LINEAR REGRESSION

With multiple linear regression we assume that we have more than one observed dependent variable. The model still assumes that the relationship between the dependent variable y and multiple dependent variables is linear. That is the target y is modeled as a linear sum of the dependent variables, each with their own computed coefficient.

To illustrate multiple linear regression we will process the California Housing dataset bundled with sklearn which has 20,640 samples, each with eight numerical features. We'll perform a multiple linear regression that uses all eight numerical features to make more sophisticated predictions of housing prices. We expect that the predictions will be better than if we were to use only a single feature or a subset of the features.

We will test this hypothesis in the lab exercises, where we will ask you to perform simple linear regressions with each individual features, and compare the results with the multiple linear regression computed here.

This housing dataset was derived from the 1990 U.S. census, using one row per census

group. The dataset has 20,640 samples—one per group—with eight features each:

```
Eight Features in the Housing Dataset:

• median income—in tens of thousands
• median house age – has a max of 52
• average number of rooms
• average number of bedrooms
• block population
• average house occupancy
• house block latitude
• house block longitude
```

Each sample also has as its target a corresponding median house value in hundreds of thousands of dollars, so a value of 3.55 would represent $355,000. In the dataset, the maximum value for this feature is 5, which represents a maximum house value of $500,000 - which is probably a serious problem of data accuracy.

Let's load the dataset and familiarize ourselves with it's structure. The fetch_california_housing function from the sklearn.datasets module returns a Bunch object containing the data and other metadata information about the dataset. For example, the string indices of the columns is stored in feature_names attribute, as seen as follows:

```
from sklearn.datasets import
fetch_california_housing
```

```
cali = fetch_california_housing()

>>> cali.feature_names
['MedInc',
'HouseAge',
'AveRooms',
'AveBedrms',
'Population',
'AveOccup',
'Latitude',
‘Longitude']
```

Next, let’s create a DataFrame from the Bunch’s data, target and feature_names arrays. The first snippet below creates the initial DataFrame using the data in cali.data and with the column names specified by cali.feature_names. The second statement adds a column for the median house values stored in cali.target:

```
cali_df = pd.DataFrame(cali.data,
     columns=cali.feature_names)

cali_df['MedHouseValue'] =
pd.Series(cali.target)
```

Once again we will use sklearn’s train_test_split to prepare the data for the purposes of training and testing the model.

```
from sklearn.model_selection import
train_test_split
```

```
X_train, X_test, y_train, y_test =
train_test_split( cali.data, cali.target,
random_state=11)

>>> X_train.shape
(15480, 8)
>>> X_test.shape
(5160, 8)
```

Now we call the estimator and attempt to fit the data with a linear regression:

```
from sklearn.linear_model import
LinearRegression

regress = LinearRegression()

regress.fit(X=X_train, y=y_train)
```

AN ITERABLE FOR THE REGRESSION SLOPE COEFFICIENTS

Multiple linear regression produces separate coefficients for the slope of each feature (stored in coeff_) and one intercept (stored in intercept_). Here we display these by using the built-in enumerate method used to create an iterable container for all the coefficient values:

```
>>> regress.intercept_
-36.88295065605547

e = enumerate(cali.feature_names)
for i, name in e:
   print(f'{name:>10}: {regress.coef_[i]}')
```

```
MedInc: 0.4377030215382206
HouseAge: 0.009216834565797713
AveRooms: -0.10732526637360985
AveBedrms: 0.611713307391811
Population: -5.756822009298454e-06
AveOccup: -0.0033845664657163703
Latitude: -0.419481860964907
Longitude: -0.4337713349874016
```

For positive coefficients, we can say that the median house value increases as the feature value increases. For example, the positive coefficient for number of bedrooms show a positive correlation with median home values. For negative coefficients, the median house value decreases as the feature value increases, for example as happens with the average number of bedrooms in a block group.

Note that the population coefficient has a negative exponent (e-06), so the coefficient's value is actually very close to zero, so a block group's population apparently has little effect the median house value. WIth multiple linear regression we use these coefficient values to make predictions with the following linear equation:

```
y = m1 x1 + m2 x2 + … mn xn + b
where
• m1, m2, …, mn are feature coefficients,
• b is the intercept,
```

```
• x1, x2, …, xn are the feature values (that
is, the values of the independent
variables),
and
• y is the predicted value (that is, the
dependent variable)
```

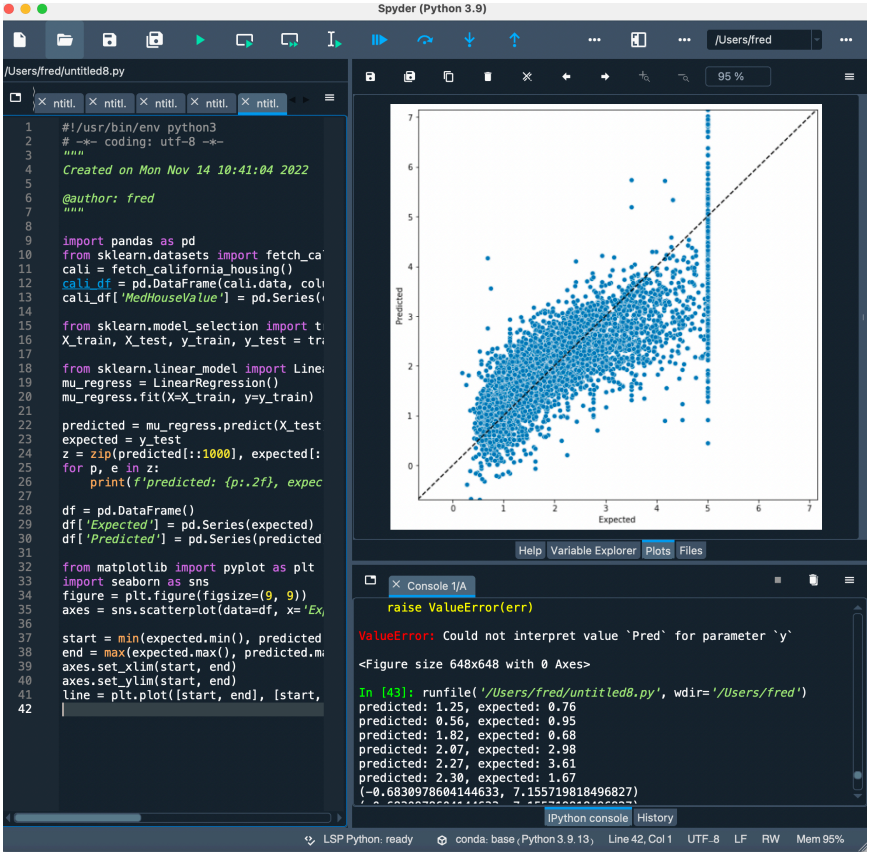
We use the same logic as above to list predictions by slicing the datasets.

```
predicted = regress.predict(X_test)
expected = y_test
z = zip(predicted[::1000], expected[::1000])

for p, e in z:
    print(f'predicted: {p:.2f}, expected:
{e:.2f}')

Output:
predicted: 1.25, expected: 0.76
predicted: 0.56, expected: 0.95
predicted: 1.82, expected: 0.68
predicted: 2.07, expected: 2.98
predicted: 2.27, expected: 3.61
predicted: 2.30, expected: 1.67
```

VISUALIZING THE EXPECTED VS. PREDICTED PRICES

Let us look at the expected vs. predicted median house values for the test data. First, let's create a DataFrame containing columns for the expected and predicted values. Next we plot the data as a scatter plot with the expected (target) prices along the x-axis and the predicted prices along the y-axis. Next we set the x- and y-axes' limits to use the same scale along both axes. Finally we plot a

line x=y that represents perfect predictions (note that this is not a regression line). The following displays a line between the points representing the lower-left corner of the graph (start, start) and the upper-right corner of the graph (end, end). The third argument ('k--') indicates the line's style. The letter k represents the color black,and the -- indicates that plot should draw a dashed line:



```
# Script to Visualize the the Expected vs.
# Predicted Prices using Multiple Linear
# Regression Housing Price Estimator

import pandas as pd
from sklearn.datasets import
fetch_california_housing
cali = fetch_california_housing()
cali_df = pd.DataFrame(cali.data,
columns=cali.feature_names)
cali_df['MedHouseValue'] =
pd.Series(cali.target)

from sklearn.model_selection import
train_test_split
X_train, X_test, y_train, y_test =
train_test_split( cali.data, cali.target,
random_state=11)

from sklearn.linear_model import
LinearRegression
mu_regress = LinearRegression()
mu_regress.fit(X=X_train, y=y_train)

predicted = mu_regress.predict(X_test)
expected = y_test
z = zip(predicted[::1000], expected[::1000])
for p, e in z:
    print(f'predicted: {p:.2f}, expected:
{e:.2f}')

df = pd.DataFrame()
df['Expected'] = pd.Series(expected)
df['Predicted'] = pd.Series(predicted)

from matplotlib import pyplot as plt
import seaborn as sns
figure = plt.figure(figsize=(9, 9))
axes = sns.scatterplot(data=df,
x='Expected', y='Predicted')

start = min(expected.min(), predicted.min())
```

```
end = max(expected.max(), predicted.max())
axes.set_xlim(start, end)
axes.set_ylim(start, end)
line = plt.plot([start, end], [start, end],
'k--')
```

REGRESSION MODEL METRICS

Sklearn provides many metrics functions for evaluating how well estimators predict results and for comparing estimators to choose the best one(s) for your particular study. These metrics vary by estimator type. Among the many metrics for regression estimators is the model's R2 score. To calculate an estimator's R2 score, call the sklearn.metrics module's r2_score function with the arrays representing the expected and predicted results:

R2 scores range from 0.0 to 1.0 with 1.0 being the best possible. Negative R2 scores are possible and indicate that the chosen model fits the data really poorly. An R2 score of 1.0 indicates that the estimator perfectly predicts the dependent variable's value, given the independent variable(s) value(s). An R2 score of 0.0 indicates the model cannot make predictions with any accuracy, based on the independent variables' values.

Another common metric for regression models is the mean squared error, which

calculates the difference between each expected and predicted value—this is called the standard mean squared error, which squares each difference and then calculates the average of the squared values. The mean_square_error is alway positive or zero, in the case there is no errors. Here are examples of the two metrics described:

```
>>> from sklearn import metrics

>>> metrics.r2_score(expected, predicted)
0.6008983115964333

>>> metrics.mean_squared_error(expected,
predicted)
0.5350149774449119
```

Both metrics give us a way of ranking or comparing estimators and their quality. It is best when the mean squared error value is closest to 0 and the R2 score closest to 1.0.

LAB FOR CHAPTER 11

For this lab we will practice with simple linear regression with the California Housing Dataset. In the previous sections we showed how to perform multiple linear regression for predicting housing prices using the California Housing dataset for both training and testing.

If a dataset has meaningful features for

predicting the target values, and you have the choice between running simple and multiple linear regression, then you will generally choose multiple linear regression, since using more data will generally result in better predictions. As we have seen when using sklearn's LinearRegression estimator, the method uses all the numerical features of the data by default thus performing multiple linear regression.

In this lab exercise you'll perform single linear regressions with each feature individually and compare the prediction results to the multiple linear regression we have carried out previously.

To do so, first extract a single feature or column of the dataset — recall how this was done in Chapter 10 on Pandas. Then split the dataset into training and testing sets, as we did with the DataFrame in this chapter's simple linear regression case study. Train the model using that one feature and make predictions as you we have done in the multiple linear regression case study. Using a for loop, do this for each of the eight features. Print out each of the simple linear regression's R2 score and mean squared error score.

Produce a table that can be used to determine which feature produced the best

results. Write a short explanation of how the simple linear regression estimators compare with the multiple linear regression. Upload your code and table for this assignment.

Here is an example output table which prints out for each feature in the data the R2 score and the MSE score.

```
Multiple Linear Regression using All features
R2 score : 0.6008983115964333
MSE score: 0.5350149774449118


Feature 0 has R2 score : 0.4630810035698606
          has MSE score 0.7197656965919478
Feature 1 has R2 score : 0.013185632224592903
          has MSE score 1.3228720450408296
Feature 2 has R2 score : 0.024105074271276283
          has MSE score 1.3082340086454287
Feature 3 has R2 score : -0.0011266270315772875
          has MSE score 1.3420583158224824
Feature 4 has R2 score : 8.471986797708997e-05
          has MSE score 1.3404344471369465
Feature 5 has R2 score : -0.00018326453581640756
          has MSE score 1.340793693098357
Feature 6 has R2 score : 0.020368890210145207
          has MSE score 1.3132425427841639
Feature 7 has R2 score : 0.0014837207852690382
          has MSE score 1.3385590192298276All
```