Linear Regression in python

Disclaimer: These notes and examples are an adaptation of the references listed at the end. They are compiled to fit the scope of this specific course.

Introduction

After learning the basics of `numpy` and `pandas`, we will explore building basic statistical models in python. Linear regression is the building block of many machine learning practices and prediction procedures.

Simple Linear Regression (SLR) is a statistical model that examines the linear relationship between two variables: dependent and independent. Roughly speaking, a Linear relationship means that when one (or more) independent variables changes, the dependent variable also changes. Usually, the dependent variable is shown with `y` and the independent variable is shown with `x`. The model is called Multiple Linear Regression (MLR) when there is more than one independent variable. In a regression model, we estimate model parameters from an existing data-set. Rather than the mathematics of regression, this handout will focus on implementing regression models in Python.

Linear regression libraries in python

The following are among the main libraries used for regression analysis in python:

- `scipy`
- `statsmodels`
- `pingouin`

On top of these libraries, any machine learning and/or artificial intelligence library such as `sklearn` or `tensorflow` would also offer regression modeling.

In the rest of this handout, we will use this minimal data set for illustration purposes:

> **Data-set**
>
> ```
> vector_x = [3, 7, 8, 8, 6, 7, 9, 2]
> vector_y = [6, 12, 12, 11, 9, 11, 14, 4]
> ```

`scipy`

About `scipy`

scientific **py**thon is used for scientific and technical computing. **Scipy** (https://www.scipy.org/) contains modules for:

- `statistics` :https://docs.scipy.org/doc/scipy/reference/tutorial/stats.html
- `linear algebra` : https://docs.scipy.org/doc/scipy/reference/tutorial/linalg.html
- `integration` : https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html
- `optimization` : https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html
- `interpolation` : https://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html

Among other modules for handling `special functions` , `image processing` , and `signal processing` . `scipy` is used mainly in science and engineering. `scipy` uses `numpy` arrays as its data structure.

Linear regression using `scipy`

`linregress` is available as a function under `stats` module of `scipy` library (https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.linregress.html). . The following snippet will build a simple linear regression model for the above given data-set.

> **SLR using `scipy`**
>
> ```
> import numpy as np
> from scipy.stats import linregress
> ```

```
vector_x = [3, 7, 8, 8, 6, 7, 9, 2]
vector_y = [6, 12, 12, 11, 9, 11, 14, 4]

# optional (but highly recommended) to turn to np array
arr_x = np.array(vector_x)
arr_y = np.array(vector_y)
result_scipy = linregress(x = arr_x, y = arr_y)

reg_slope = result_scipy.slope
reg_intercept = result_scipy.intercept
reg_r_value = result_scipy.rvalue
reg_r_value_sq = result_scipy.rvalue ** 2
reg_p_value = result_scipy.pvalue

print(f"{'slope' :<30s} {reg_slope:.5f}")
print(f"{'intercept':<30s} {reg_intercept:.5f}")
print(f"{'r-value':<30s} {reg_r_value:.5f}")
print(f"{'r-squared':<30s} {reg_r_value_sq:.5f}")
print(f"{'p-value for H_0: slope = 0':<30s} {reg_p_value:.5f}")
```

yields

```
slope                          1.31609
intercept                      1.64943
r-value                        0.97737
r-squared                      0.95526
p-value for H_0: slope = 0     0.00003
```

`statsmodels`

About `statsmodels`

`statsmodels` is a python module that provides capabilities for the estimation of many different statistical models, as well as for conducting statistical tests and statistical data exploration. The online documentation is available at https://www.statsmodels.org/stable/index.html.

Linear regression using `statsmodels`

After installing `statsmodels`, you will need to import it every time you want to use it using

```
import statsmodels.api as sm
```

`statsmodels` can easily work with `pandas` `DataFrames` and `numpy` arrays. In this example, we will use `numpy` arrays. When running `statsmodels`, it's important to note that `statsmodels` does not add a constant to the regression model by default and you need to specifically add it in your code using the command `X = sm.add_constant(C)` where `C` is the name of your `DataFrame` containing your input (independent) variables or, in our case, simply the `numpy` array containing values of the independent variable.

SLR Using `statsmodels`

```
import numpy as np
import statsmodels.api as sm

vector_x = [3, 7, 8, 8, 6, 7, 9, 2]
vector_y = [6, 12, 12, 11, 9, 11, 14, 4]

arr_x = np.array(vector_x)
arr_y = np.array(vector_y)

X = arr_x                   # independent var.
X = sm.add_constant(arr_x)  # add constant
y = arr_y                   # dependent var.

# Note the difference in argument order
model = sm.OLS(y, X).fit()

# Print out the results
model.summary()
```

yields

OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.955
Model:	OLS	**Adj. R-squared:**	0.948
Method:	Least Squares	**F-statistic:**	128.1
Date:	Sun, 25 Apr 2021	**Prob (F-statistic):**	2.85e-05
Time:	22:32:20	**Log-Likelihood:**	-8.0775
No. Observations:	8	**AIC:**	20.16
Df Residuals:	6	**BIC:**	20.31

Df Model:	1
Covariance Type:	nonrobust

	coef	std err	t	P>|t|	[0.025	0.975]
const	1.6494	0.776	2.126	0.078	-0.249	3.547
x1	1.3161	0.116	11.319	0.000	1.032	1.601

Omnibus:	0.116	**Durbin-Watson:**	1.387
Prob(Omnibus):	0.944	**Jarque-Bera (JB):**	0.112
Skew:	-0.066	**Prob(JB):**	0.946
Kurtosis:	2.436	**Cond. No.**	19.5

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Here in this output table, we pay immediate attention to `R-squared` for SLR and `Adj. R-squared` for MLR. Then we see the coefficient for the `intercept (const)` and the `x1 (slope)`; which are the same as those calculated previously using `scipy`. Another number that we care about is the provided `p-value`s. We can see that we also have the standard error for the coefficients, which we calculated only for the slope. If we want to plot a cool plot we can import this library.

Similar to the previous section, you can use `matplotlib` or `seaborn` to plot the regression results. One can use (rather sophisticated) graphing capabilities available via `statsmodels`, as well.

`pingouin`

About `pingouin`

`pingouin` is my go-to library for statistical analysis! `scipy` is easy to use but provides somewhat lacking output (e.g., the only test statistic and probability value). `statsmodels`, on the other hand, is very powerful. Still, its output is overkill, overwhelming, and challenging to comprehend for beginners. `pingouin` is something in between. `pingouin` is a relatively new library that provides simple yet exhaustive outputs for the most common statistical tests. Complete documentation of `pingouin` is available at https://pingouin-stats.org/index.html#.

pingouin is designed for users who want simple yet exhaustive statistical analysis capabilities!

`pingouin` integrates very well with `pandas` and `numpy`. Some of its main features are

- ANOVAs
- Pairwise tests
- Linear regression
- Logistic regression
- Multivariate tests
- Confidence intervals
- Chi-squared tests
- Plotting

`pingouin` can be installed by running

```
pip install pingouin
```

Its documentation suggests importing `pingouin` using

```
import pingouin as pg
```

Linear regression using `pingouin`

`linear_regression(x, y)` will perform a SLR $y = \text{intercept} + coef \times x$.

SLR using `pingouin`

```
import numpy as np
import pingouin as pg

vector_x = [3, 7, 8, 8, 6, 7, 9, 2]
vector_y = [6, 12, 12, 11, 9, 11, 14, 4]

arr_x = np.array(vector_x)
arr_y = np.array(vector_y)

pg.linear_regression(arr_x, arr_y)
```

yields

	names	coef	se	T	pval	r2	adj_r2	CI[2.5%]	CI[97.5%]
0	Intercept	1.64943	0.775656	2.12649	0.0775846	0.955262	0.947805	-0.248537	3.54739

	names	coef	se	T	pval	r2	adj_r2	CI[2.5%]	CI[97.5%]
1	x1	1.31609	0.116276	11.3187	2.84651e-05	0.955262	0.947805	1.03158	1.60061

`pingouin` is a low-code library. It is fully compatible with `numpy` and `pandas`; more importantly, its output is comprehensive enough for almost everyone.

Note: `pg.linear_regression()` has an optional alpha argument with a default value of `0.05`. This argument is used for building confidence intervals and is equal to `1 - confidence level`.

Visualizing SLR results

The easiest way to plot regression line and data-set is using `regplot` function of the `seaborn` library as in:

> **Using `seaborn`**
>
> ```
> import seaborn as sns
>
> sns.regplot(x = arr_x, y = arr_y, color="red")
> ```

Alternatively, you can use `matpotlib` for visualization of the regression line; as in:

> **Using `matplotlib`**
>
> ```
> import matplotlib.pyplot as plt
>
> plt.plot(arr_x, arr_y, "o", color="red", label="data-set")
> plt.plot(
>     arr_x,
>     reg_intercept + reg_slope * arr_x,
>     color="black",
>     label="regression",
> )
>
> plt.legend()
> plt.show()
> ```

Multiple Linear Regression (MLR) using python

Most practical problems will have more than one independent variable; a regression model with more than one independent variable is called Multiple Linear Regression (MLR). The following will provide guidelines for using `scipy`, `statsmodels`, and `pingouin` for MLR modeling. We will assume the following data-set:

x_1	x_2	y
3	3	13
10	13	27
8	7	22
5	9	16
6	7	18
7	12	21
12	15	31
2	1	10

MLR using `scipy` & `numpy`

MLR using `scipy` & `numpy` can be a bit confusing; you need to use `scipy.optimize.leastsq` or use `numpy.polyfit`. Here are the references for the interested students (looking for a fun challenge!)

- `scipy.optimize.leastsq` at https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.leastsq.html
- `numpy.polyfit` at https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html

MLR using `statsmodels`

> **Example**
>
> ```
> import numpy as np
> import pandas as pd
> import statsmodels.api as sm
> ```

```
arr_x_1 = np.array([3, 10, 8, 5, 6, 7, 12, 2])
arr_x_2 = np.array([3, 13, 7, 9, 7, 12, 15, 1])
arr_y = np.array([13, 27, 22, 16, 18, 21, 31, 10])

# build a matrix of vectors
arr_all = np.vstack([arr_x_1, arr_x_2, arr_y]).T

# DataFrame
df = pd.DataFrame(arr_all, columns=["x_1", "x_2", "y"])

X = df[["x_1", "x_2"]]
X = sm.add_constant(X)
y = df["y"]

model = sm.OLS(y, X).fit()
model.summary()
```

yields

OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.996
Model:	OLS	**Adj. R-squared:**	0.994
Method:	Least Squares	**F-statistic:**	593.4
Date:	Mon, 26 Apr 2021	**Prob (F-statistic):**	1.14e-06
Time:	01:25:55	**Log-Likelihood:**	-4.4954
No. Observations:	8	**AIC:**	14.99
Df Residuals:	5	**BIC:**	15.23
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	6.0640	0.442	13.730	0.000	4.929	7.199
x_1	1.9681	0.141	13.965	0.000	1.606	2.330
x_2	0.0773	0.098	0.791	0.465	-0.174	0.329

Omnibus:	0.352	**Durbin-Watson:**	0.859

Prob(Omnibus):	0.838	**Jarque-Bera (JB):**	0.337
Skew:	0.349	**Prob(JB):**	0.845
Kurtosis:	2.276	**Cond. No.**	28.2

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.'

MLR using `pingouin`

> **Example**
>
> ```
> import numpy as np
> import pingouin as pg
> import pandas as pd
>
> arr_x_1 = np.array([3, 10, 8, 5, 6, 7, 12, 2])
> arr_x_2 = np.array([3, 13, 7, 9, 7, 12, 15, 1])
> arr_y = np.array([13, 27, 22, 16, 18, 21, 31, 10])
>
> # build a matrix of vectors
> arr_all = np.vstack([arr_x_1, arr_x_2, arr_y]).T
>
> # DataFrame
> df = pd.DataFrame(arr_all, columns=["x_1", "x_2", "y"])
>
> # MLR
> pg.linear_regression(df[["x_1", "x_2"]], df["y"])
> ```
>
> yields >
>
> | | names | coef | se | T | pval | r2 | adj_r2 | CI[2.5%] | CI[97.5%] |
> |---|---|---|---|---|---|---|---|---|---|
> | 0 | Intercept | 6.06403 | 0.441653 | 13.7303 | 3.67733e-05 | 0.995805 | 0.994126 | 4.92872 | 7.19933 |
> | 1 | x_1 | 1.96807 | 0.140927 | 13.9651 | 3.38462e-05 | 0.995805 | 0.994126 | 1.6058 | 2.33033 |
> | 2 | x_2 | 0.0773185 | 0.0977933 | 0.790632 | 0.464989 | 0.995805 | 0.994126 | -0.174067 | 0.328704 |
>
> **Note:** Most of the above code is an effort to put the data in the proper format.

References

This document is an adaption of the following references:

1- *Python online documentation*; available at https://docs.python.org/3/

2- *Exploratory Data Analysis with Python*; Suresh Kumar Mukhiya and Usman Ahmed, Packt publications, 2020

3- *Introduction to Python for Computer Science and Data Science*, Paul J. Deitel, Harvey Deitel, Pearson, 1st edition, 2019

4- *Interactive Data Visualization with Python*; Abha Belorkar, Sharath Chandra Guntuka, Shubhangi Hora, and Anshu Kumar. Packt publications, 2020

5- https://www.learnpython.org/

6- https://numpy.org/

7- https://docs.scipy.org/doc/scipy/reference/index.html

8- https://www.statsmodels.org/stable/index.html

9- https://pingouin-stats.org/#