Linear Regression

Using the normal equation and scikit-learn

Brais Galvan Sotelo (19563)

CS550 - NPU

Table of Contents

Implementation - Linear Regression using Normal Equation
Step 1.1: General Set-up

```
[53] # Python ≥3.5 is required
     import sys
     assert sys.version_info >= (3, 5)

     # Scikit-Learn ≥0.20 is required
     import sklearn
     assert sklearn.__version__ >= "0.20"

     # Common imports
     import numpy as np
     import os

     # to make this notebook's output stable across runs
     np.random.seed(42)
```

First we are going to important some necessary modules to complete the project, and make sure than Python version and Scikit-learn version are appropriate.

We also set-up a seed to get consistent result over multiple runs of the code. (This would be important if we use random generated data)

Implementation - Linear Regression using Normal Equation
Step 1.2: Figure creation Set-up

```python
# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "training_linear_models"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

We define how we want our images/figures to look and then we define where we want to save the images.

Implementation - Linear Regression using Normal Equation
Step 1.3: Data Set-up

```
[55] import numpy as np
     import pandas as pd

     from google.colab import files
     uploaded  = files.upload()
     import io
     abalone = pd.read_csv(
         io.BytesIO(uploaded['abalone_train.csv']),
         names=["Length", "Diameter", "Height", "Whole weight", "Shucked weight",
                "Viscera weight", "Shell weight", "Age"])

     X1 = abalone["Length"]
     X2 = np.array(X1)
     X = X2.reshape(-1, 1)

     y1 = abalone["Height"]
     y2 = np.array(y1)
     y = y2.reshape(-1, 1)
```

```
Choose Files abalone_train.csv
• abalone_train.csv(application/vnd.ms-excel) - 149229 bytes, last modified: 6/7/2021 - 100% done
Saving abalone_train.csv to abalone_train (1).csv
```

We will upload a file containing the data for this problem. The data is contained in the file abalone_train.csv. The abalone_train data has 8 columns of information "Length", "Diameter", "Height", "Whole weight", "Shucked weight", "Viscera weight", "Shell weight" and"Age". (A sample of the data is shown in the next slide)

The we define X and Y to be Length and the Height

Abalone_train DATA Sample

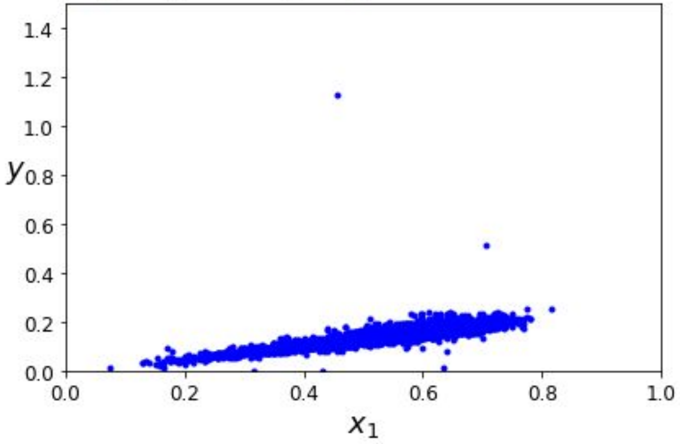
	A	B	C	D	E	F	G	H	I
1	0.435	0.335	0.11	0.334	0.1355	0.0775	0.0965	7	
2	0.585	0.45	0.125	0.874	0.3545	0.2075	0.225	6	
3	0.655	0.51	0.16	1.092	0.396	0.2825	0.37	14	
4	0.545	0.425	0.125	0.768	0.294	0.1495	0.26	16	
5	0.545	0.42	0.13	0.879	0.374	0.1695	0.23	13	
6	0.57	0.45	0.145	0.751	0.2825	0.2195	0.2215	10	
7	0.47	0.36	0.13	0.472	0.182	0.114	0.15	10	
8	0.61	0.45	0.19	1.0805	0.517	0.2495	0.2935	10	
9	0.52	0.425	0.125	0.79	0.372	0.205	0.19	8	
10	0.485	0.39	0.12	0.599	0.251	0.1345	0.169	8	
11	0.625	0.495	0.155	1.025	0.46	0.1945	0.34	9	
12	0.615	0.495	0.16	1.255	0.5815	0.3195	0.3225	12	
13	0.455	0.35	0.14	0.5185	0.221	0.1265	0.135	10	
14	0.475	0.355	0.115	0.5195	0.279	0.088	0.1325	7	
15	0.385	0.3	0.1	0.2895	0.1215	0.063	0.09	7	
16	0.67	0.525	0.165	1.6085	0.682	0.3145	0.4005	11	

The abalone_train data has 3320 rows of information with 8 parameter each.

Implementation - Linear Regression using Normal Equation
Step 2: Plotting X and Y values

```
[58] plt.plot(X, y, "b.")
     plt.xlabel("$x_1$", fontsize=18)
     plt.ylabel("$y$", rotation=0, fontsize=18)
     plt.axis([0, 1, 0, 1.5])
     save_fig("generated_data_plot")
     plt.show()
```

```
Saving figure generated_data_plot
```



Generate a figure by plotting the X and Y values. We define the axis in such as way that we accommodate the data properly, and then save the figure.

Implementation - Linear Regression using Normal Equation
Step 3: Finding the theta values

```
[60] X_b = np.c_[np.ones((3320, 1)), X]  # add x0 = 1 to each instance
     theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

```
[61] theta_best

     array([[-0.0108267 ],
            [ 0.28716253]])
```

```
[62] X_new = np.array([[0], [2]])
     X_new_b = np.c_[np.ones((2, 1)), X_new]  # add x0 = 1 to each instance
     y_predict = X_new_b.dot(theta_best)
     y_predict

     array([[-0.0108267 ],
            [ 0.56349837]])
```

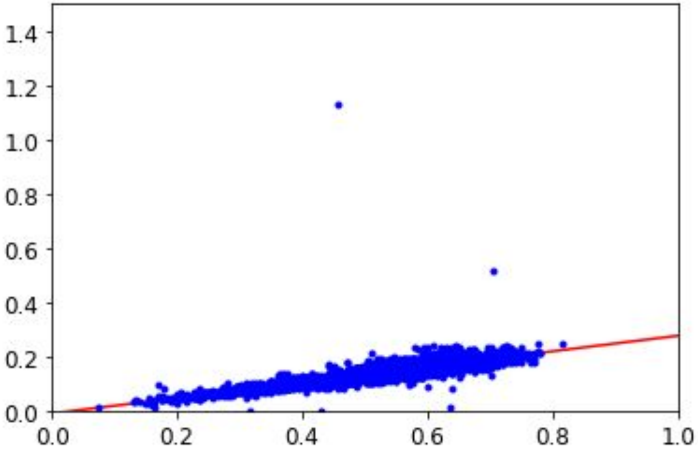
We apply the normal equation to get theta values. In this case we use np.linalg.inv() in order to calculate the multiplicative inverse of the matrix.

Theta = $(X^TX)^{-1}X^TY$

The we use theta to make predictions of Y values using new values of X.

Implementation - Linear Regression using Normal Equation
Step 4.1: Plotting X and Y and the new data

```
[64] plt.plot(X_new, y_predict, "r-")
     plt.plot(X, y, "b.")
     plt.axis([0, 1, 0, 1.5])
     plt.show()
```

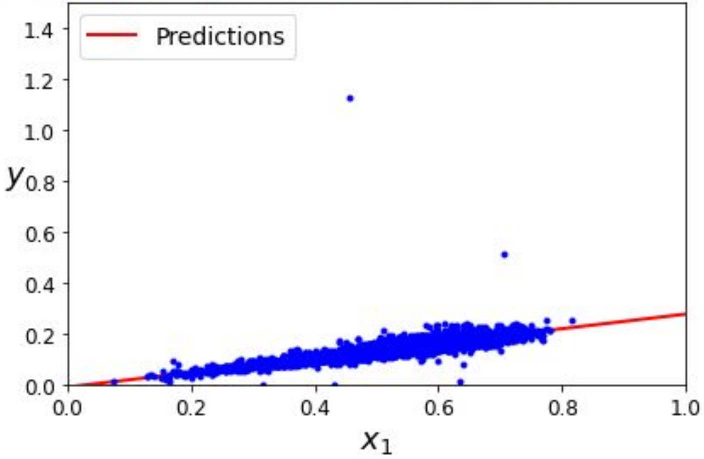


Now in red, we can see the linear regression line, defined by plotting X_new and y_pred.

Implementation - Linear Regression using Normal Equation
Step 4.2: Adding additional features to the figure

```
[66] plt.plot(X_new, y_predict, "r-", linewidth=2, label="Predictions")
     plt.plot(X, y, "b.")
     plt.xlabel("$x_1$", fontsize=18)
     plt.ylabel("$y$", rotation=0, fontsize=18)
     plt.legend(loc="upper left", fontsize=14)
     plt.axis([0, 1, 0, 1.5])
     save_fig("linear_model_predictions_plot")
     plt.show()
```

```
Saving figure linear_model_predictions_plot
```



Here we are simply going to add some additional features to our graph, including labels, for the axis and the legend for the regression prediction line (in red).

Implementation - Linear Regression using scikit-learn
Step 1: Creating a model and making a prediction

```
[68] from sklearn.linear_model import LinearRegression

     lin_reg = LinearRegression()
     lin_reg.fit(X, y)
     lin_reg.intercept_, lin_reg.coef_

     (array([-0.0108267]), array([[0.28716253]]))
```

```
[69] lin_reg.predict(X_new)

     array([[-0.0108267 ],
            [ 0.56349837]])
```

We utilize our previous definition of X and Y values together with the Linear Regression functionality from scikit-learn to get the intercept and coefficient of the regression equation, which we can then use to make a prediction to new values of X

Implementation - Linear Regression using scikit-learn
Step 2: Finding the theta values

The `LinearRegression` class is based on the `scipy.linalg.lstsq()` function (the name stands for "least squares"), which you could call directly:

```
[70] theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
     theta_best_svd

     array([[-0.0108267 ],
            [ 0.28716253]])
```

This function computes $\mathbf{X}^{+}\mathbf{y}$, where $\mathbf{X}^{+}$ is the *pseudoinverse* of $\mathbf{X}$ (specifically the Moore-Penrose inverse). You can use `np.linalg.pinv()` to compute the pseudoinverse directly:

```
np.linalg.pinv(X_b).dot(y)

array([[-0.0108267 ],
       [ 0.28716253]])
```

+ Code   + Text

We use predefined functions to directly find the theta values,

Conclusion

In this scenario, we implemented the Normal Equation and scikit-learn to get a Linear Regression algorithm and make a prediction. Both method yielded the same answer.

Scikit-learn, provides the tools to get implemented the Linear Regression faster and more easily, but it can be somewhat abstract and hard to understand.

On the other hand, using the Normal Equation was more tedious but I gave me a better understanding of the results.

Also, plotting the data was extremely useful to both understand the data and double-check the result of the regression model among other things.