Introduction to Machine Learning (ML) with Sklearn

by Mats Bauer

Machine Learning is an exciting area of programming. Basically ML describes the ability for computers to understand data and generate output for unknown content, based on the data. The best known ML algorithm is probably the spam filter. What it does is learns what spam looks like, based on people marking e-mails as spam, and filtering incoming e-mails based on the learned data.

Sklearn - learning by doing.

Let's start with a little project. First you need all the right plugins. I myself am working with Cloud9, but you can also work with a local Python application like Spyder or even Terminal. For this example you will need to install: - matlibplot - pandas - numpy - sklearn

Next you will need the data file we will use in this example. This is downloaded by typing or getting it directly via Kaggle:

```
$ git clone https://github.com/esabunor/MLWorkspace/blob/master/melb_data.csv
```

Now you should create your Python script by typing

```
$ sudo nano predict.py
```

You can name your Python script anyway you like, mine is called predict. This line opens a new nano editor in Terminal, if you are using Spyder just create a new file in the same folder as your data CSV file from above.

When it comes to ML, understanding your data is really important. In order to understand the CSV file, we will first open a live Python editor by typing Python into the Terminal or opening a new console in Spyder. You should now see something like:

```
>>>
```

We start by importing the data package pandas and loading our CSV file.

```
>>> import pandas as pd
>>> df = pd.read_csv("melb_data.csv)
```

What we did is load our CSV file and save it into the variable df (datafile). Let's start looking at what we have here. First function is df.info(), giving us all column names and number of entries:

```
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 18396 entries, 0 to 18395
Data columns (total 22 columns):
Unnamed: 0       18396 non-null int64
Suburb           18396 non-null object
Address          18396 non-null object
Rooms            18396 non-null int64
Type             18396 non-null object
Price            18396 non-null float64
Method           18396 non-null object
SellerG          18396 non-null object
Date             18396 non-null object
Distance         18395 non-null float64
Postcode         18395 non-null float64
Bedroom2         14927 non-null float64
Bathroom         14925 non-null float64
Car              14820 non-null float64
Landsize         13603 non-null float64
BuildingArea     7762 non-null float64
YearBuilt        8958 non-null float64
CouncilArea      12233 non-null object
Lattitude        15064 non-null float64
Longtitude       15064 non-null float64
Regionname       18395 non-null object
Propertycount    18395 non-null float64
dtypes: float64(12), int64(2), object(8)
memory usage: 3.1+ MB
```

We see now, that we have a file with 18396 entries and 22 columns, with all the names and data types. Second thing we see is that for 18396 houses, only for 7762 houses the BuildingArea is known and only for 8958 houses the YearBuilt. This can mean, that these rows of data can be hard to use in full context. In this introduction we want to estimate the house price based on size. As the BuildingArea is only available for half the houses and the Landsize for 90%, when it comes to learning from data, we will later use Landsize to predict House Prices. Next up, we want to find some basic mathematical values for the data set. This is done using the describe() function.

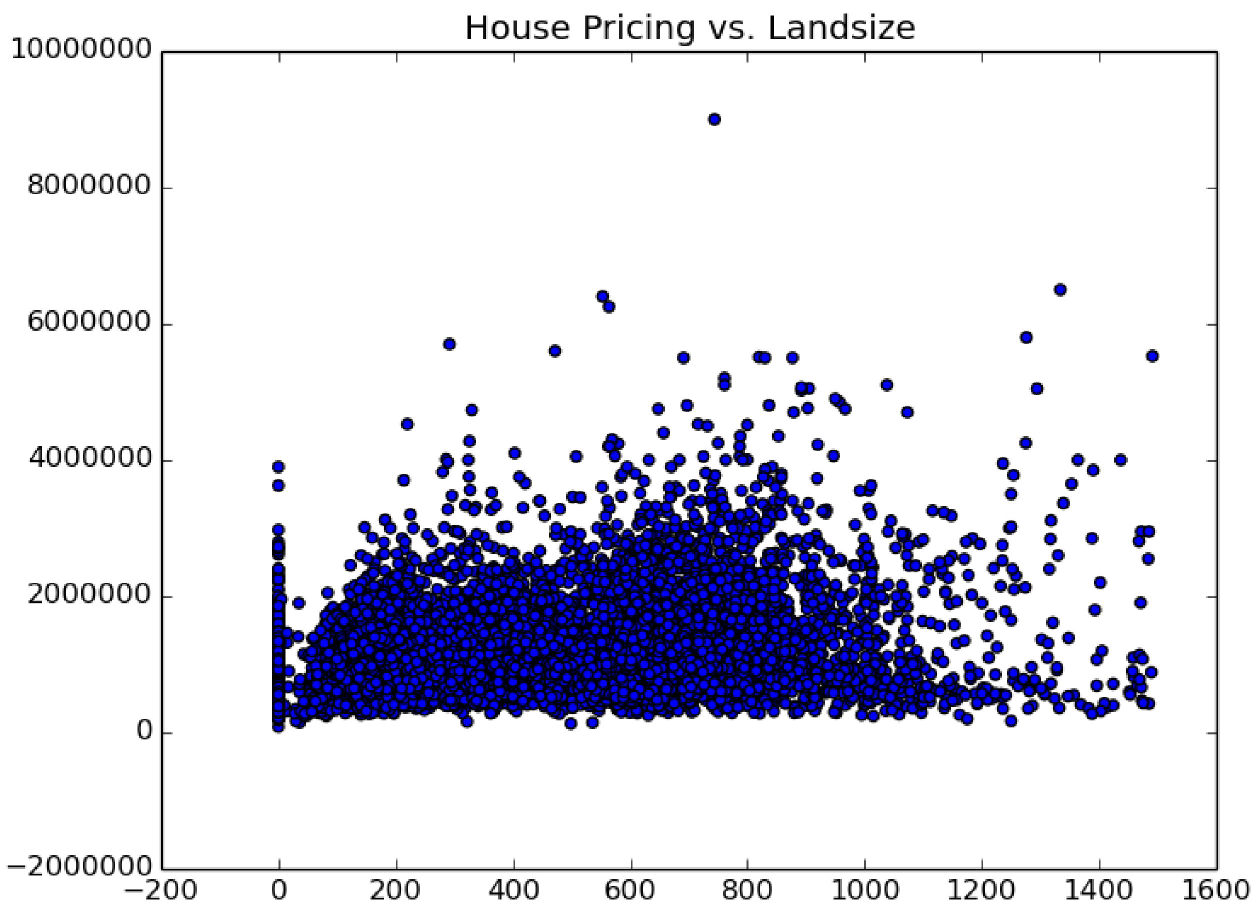
```
>>> df.describe().round()
       Unnamed: 0    Rooms      Price  Distance  Postcode  Bedroom2  Bathroom  \
count     18396.0  18396.0    18396.0   18395.0   18395.0   14927.0   14925.0
mean      11827.0      3.0  1056697.0      10.0    3107.0       3.0       2.0
std        6801.0      1.0   641922.0       6.0      95.0       1.0       1.0
min           1.0      1.0    85000.0       0.0    3000.0       0.0       0.0
25%        5937.0      2.0   633000.0       6.0    3046.0       2.0       1.0
50%       11820.0      3.0   880000.0      10.0    3085.0       3.0       1.0
75%       17734.0      3.0  1302000.0      13.0    3149.0       3.0       2.0
max       23546.0     12.0  9000000.0      48.0    3978.0      20.0       8.0

           Car  Landsize  BuildingArea  YearBuilt  Lattitude  Longtitude  \
count  14820.0   13603.0        7762.0     8958.0    15064.0      15064.0
mean       2.0     558.0         151.0     1966.0      -38.0        145.0
std        1.0    3987.0         519.0       37.0        0.0          0.0
min        0.0       0.0           0.0     1196.0      -38.0        144.0
25%        1.0     176.0          93.0     1950.0      -38.0        145.0
50%        2.0     440.0         126.0     1970.0      -38.0        145.0
75%        2.0     651.0         174.0     2000.0      -38.0        145.0
max       10.0  433014.0       44515.0     2018.0      -37.0        146.0

       Propertycount
count        18395.0
mean          7518.0
std           4488.0
min            249.0
25%           4294.0
50%           6567.0
75%          10331.0
max          21650.0
```

This function gives us a beautiful overview of values like mean, std, min, max and quantiles. We can see that house prices start at $85,000 and end with $9 million. The smallest houses have 1 room, the largest 12, on average houses in this data table have 3 rooms and a landsize of 558m^2^.

Let's get started with our prediction. For a start, we will use a simple linear regression to predict house prices based on the landsize. We call values to predict based on learning data labels, and the values we base our predictions on predictors. This means that our label is the house price and predictor the landsize. To visualize this we will use matlibplot and a scatter plot. This following is now inserted into the predict.py file and executing.

```
from sklearn.linear_model import LinearRegression
import pandas as pd
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt #this enables us to save the graph as a file in
Cloud9

df = pd.read_csv("melb_data.csv") #load the CSV file, just as we did before
df = df.loc[df['Landsize'] < 1500].round() #filter the landsize to max. 1500m^2

X = df.Landsize     #declare X as the landsize
y = df.Price        #and y as the price

fig, ax = plt.subplots(1, 1)
ax.set_title('House Pricing vs. Landsize')
ax.ticklabel_format(style = 'plain') #basic formatting, to make it look good
ax.scatter(X, y)
fig.savefig('graph.png')
```

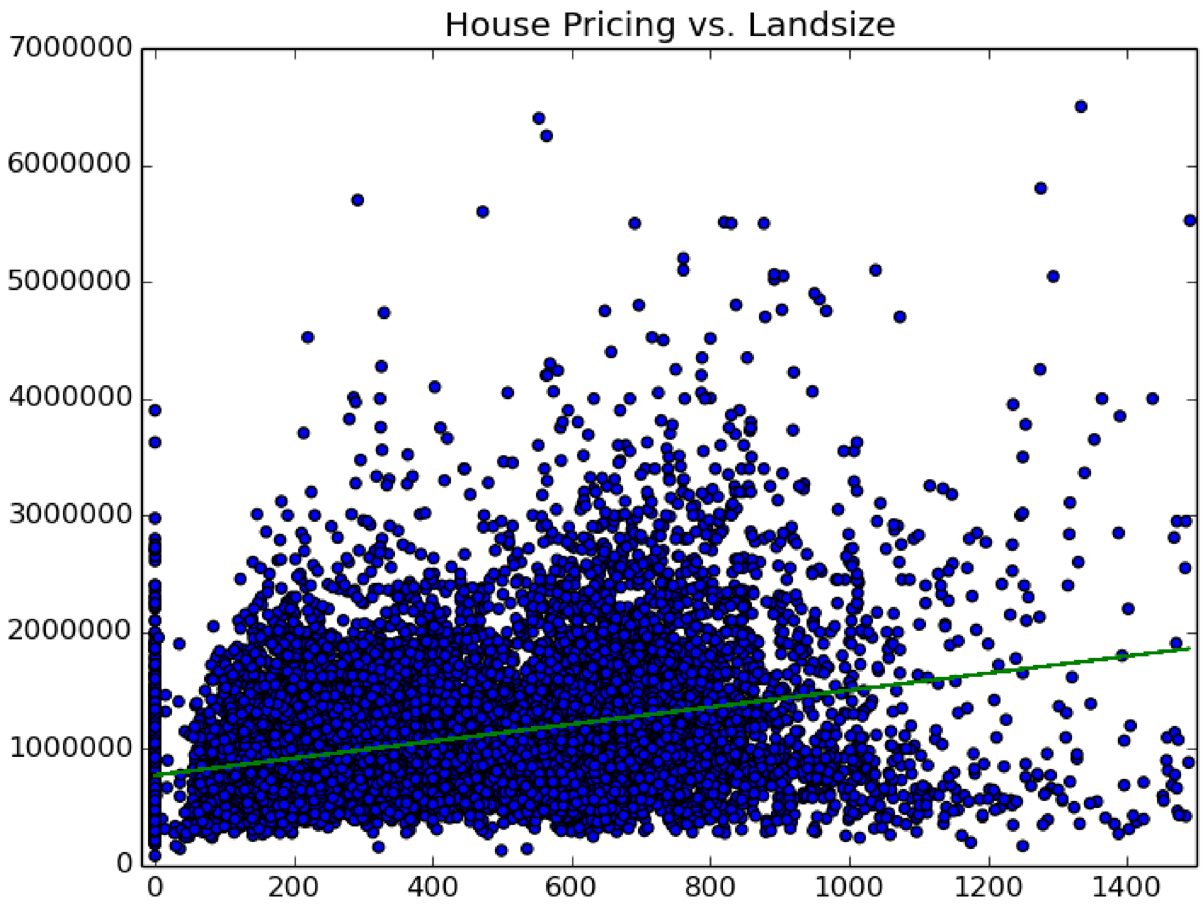
The result of this plotting is this scatter of data.



What we see from this graph is, that there are lots of houses in the range from very little to $2 million dollars in the range of 150 to 1000m^2^ landsize. What we need to understand now is, how this graph shows sums up as a regression. In our case: linear regression. To do that, we need to add a second graph to our plot. Our main block of code from above will increase to looking like this:

```
fig, ax = plt.subplots(1, 1)
ax.set_title('House Pricing vs. Landsize')
ax.ticklabel_format(style = 'plain')
ax.set_xlim(-20, 1500) #remove the white area with negative landsize
ax.set_ylim(-20, 7000000) #remove negative house prices
ax.scatter(X, y)
m,b = np.polyfit(X, y, 1)
ax.plot(X, m*X + b, '-', color="green")
fig.savefig('display2.png')
```

What we use for creating a linear regression is the numpy function `polyfit`, to get our values for m and b (remember maths in 5th grade :-) -> y = mx + b In the second to last line we plot exactly these values in green, resulting in this graph:



Based on the linear regression, we can now estimate the price for a house with a landsize of 355.6m^2^. Just add the following line at the bottom of your script to get the exact result:

```
y = m*355.6 + b
print(y)
```

The result output for y, the price, is `1034505.28788`, meaning that, based on our simple ML function, the price for a house with a landsize of 355.6m^2^ should be just above $1 million. Pretty cool, right? This was simple, let's start adding a little bit of ML-extra with `sklearn` to the mix. What we will do now is essentially the same thing as above, except that we will use a more adaptable and powerful function. So instead of the excerpt above, we will insert following lines of code - and I will explain after:

```
X = X.values.reshape(-1, 1)

model = LinearRegression()
model.fit(X, y)

X_new = [[355.6]]
print(model.predict(X_new))
```

The first line of code is needed only, because we are considering only a single predictor, the landsize, you don't need to worry about that for now. Usually with sklearn you make predictions based on multiple inputs like also the number of rooms, bathrooms and area code. For now, we will stay with only the landsize. Next we declare or model, meaning the type of regression to use, in our case (as we did above) a linear regression. This then get's fitted with our two data lists, X and y. All this basically does is generating a linear regression based on X and y, same as we did above - manually. Second to last, we have to create a list with the value, in our case the landsize, we want to predict the price to. For every value in `X_new` we have to create another list inside the outer list (see further example). Printing the result of the predict function gives us basically the values y from above:

```
[ 1034505.28787783]
```

This result matches our result for y with the manual linear regression. Advantage, we can automate and expand this very easily. Let's replace the `X_new` with following:

```
X_new = [[355.6],[521.4],[721.9]]
```

This generates a multivalue-input and -output function, predicting housing prices for all three input landsizes:

```
[ 1034505.28787783  1155213.33627154  1301184.16680796]
```

Multiple predictors

In the next step in creating our first ML application, we want to increase the input predictors from only landsize to landsize, number of rooms and bathrooms. This will increase our prediction accuracy simply based on the principle of "more input, better output".

We start with introducing a new regression scheme, the DecisionTreeRegressor. This is a function in sklearn that enables you to use multiple input predictors to predict the outcome. The inputs now are landsize, number of rooms and number of bathrooms, and we are still predicting the price.

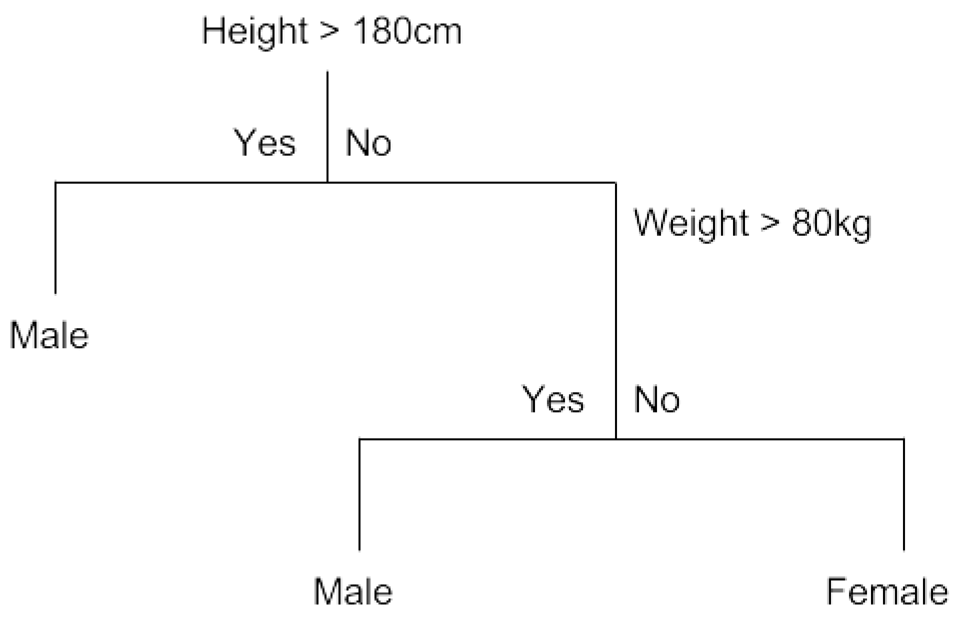
Practical example

I am currently looking at buying a house and want a landsize of 355m^2^. I have $750,000 and want to know whether I can buy a house with three rooms or only two rooms, with one bathroom. Let's first look at the new script we use for the DecisionTreeRegressor (DTR):

```
from sklearn.tree import DecisionTreeRegressor
import pandas as pd

df = pd.read_csv("melb_data.csv")
df = df.fillna(df.mean())
```

```
predictors = ["Landsize", "Rooms", "Bathroom"]
X = df[predictors]
y = df.Price

model = DecisionTreeRegressor()
model.fit(X, y)

X_new = [[355, 2, 1],[355, 3, 1]]
print(model.predict(X_new))
```

The script starts by importing both packages pandas and sklearn. For sklearn we will now use the DTR, meaning that it uses a flexible regression algorithm to predict the pricing. This Decission or Classification Tree can be understood as a successive process, in which all predictors are asked one after the other. Following the tree from top to bottom results in the output.



As seen in this example, there is offcourse room for error for women that are over 180cm, even though they weigh less than 80kg, because the DTR generates the output early.

In the second passage of the script we do the same as in the example above, reading the CSV file and saving its values to the variable df. With the function fillna, we are filling the empty fields with mean values. The third block defines the three predictors and saves the data for these to X and the price to Y.

Now we come to the bit that's changed, the definition of the regression. If you look back at the last example, we used `LinearRegression()` as our regression model, now we change this to the `DecisionTreeRegressor`. This changes the prediction algorithm to the tree structure above. Next we fit the model with out data, as before.

To generate our prediction we create the variable `X_new` and fill it with two datasets. Before each set included one value, the landsize, now we need to add three predictors, the landsize, number of rooms and number of bathrooms, to predict the house price. As planned, we want to find out if we can afford a house with 355m^2^ and 3 rooms or only 2 rooms with our $750,000.

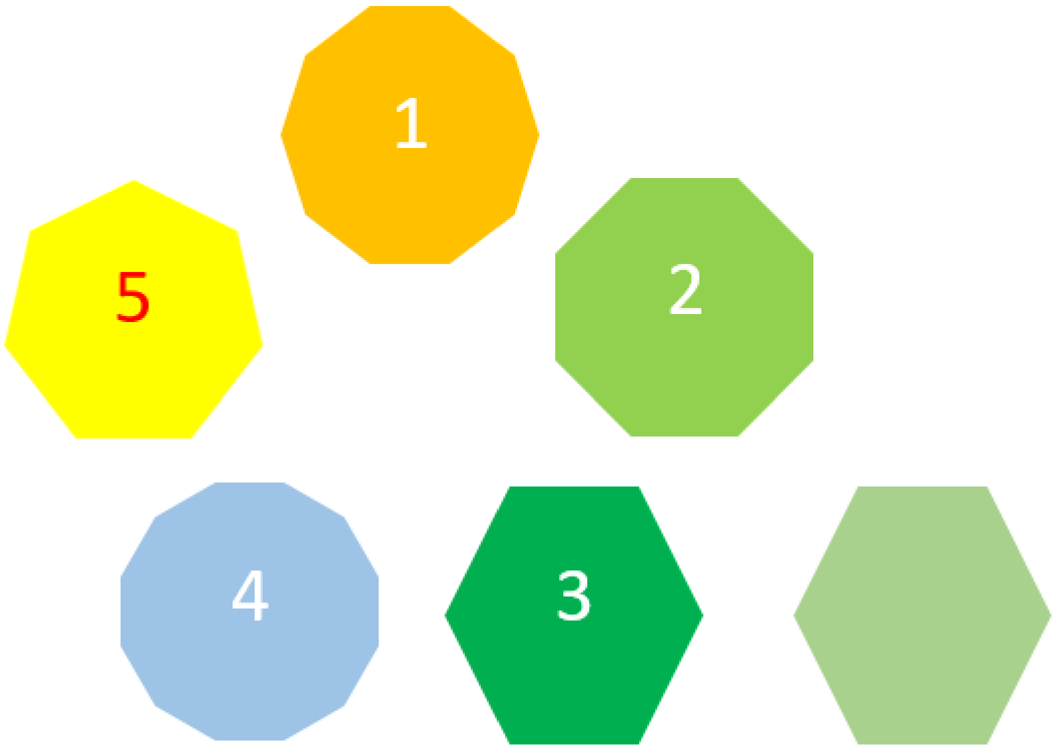
Pressing execute generates the prediction outcome:

```
[ 745000.  756000.]
```

It seems like we won't be able to buy that house with 3 rooms, as a second room is predicted to cost $11,000 more than the same house with 2 rooms.

Neighbor Classification

Let's look at one more classification method, the KNeighborsRegressor. Below we have 5 object with different forms and colors. On the bottom right we have a object that we want to predict on basis of the others. What we see immediately is, that the form matches with number 3, so we can categorize the two forms as neighbors. Looking at the color we see a light green, that can be neighbored with number 2 and number 3. I hope you understand the principle of neighboring - we are finding the closest match to the input value in our existing data and grouping these. In the given example, we would group the input form with its neighbors 2 and 3, but find a better match with number 3 due to the match of form.



Let's apply this new regression method to our example of calculating housing costs for our 355m^2^ house with two or three rooms. The adaption is very straight forward and I will only give you the final script. The only change is the model and import for sklearn.

```
from sklearn.neighbors import KNeighborsRegressor
import pandas as pd

df = pd.read_csv("melb_data.csv")
df = df.fillna(df.mean())

predictors = ["Landsize", "Rooms", "Bathroom"]
X = df[predictors]
y = df.Price

model = KNeighborsRegressor(n_neighbors=2)
model.fit(X, y)

X_new = [[355, 2, 1],[355, 3, 1]]
print(model.predict(X_new))
```

In the model declaration we call the function `KNeighborsRegressor`, in which we pass the number of neighbors we want to consider. This is the clustering factor - in englisch: how many items are allowed in a group / cluster. If we have 4 similar green objects and set `n_neighbors` to 2, the algorithm will generate two clusters of 2 green objects, instead of one larger cluster with all 4 objects. This value should be chosen upon the amount of similar value available. This can be judged based on the standard deviation (std) given when running `df.describe()`. The larger the std, the smaller the clustering should be, to avoid over clustering. Example, we count apples that people have at home. If the most apples we found are 129 and the second largest is 12, we don't want to force these two completely different values to cluster.

Let's look at the results of this method:

```
[ 667500.  756000.]
```

Again, it looks like we can't afford the house with three rooms, as we get the same result of $756,000. In this prediction, the result for our two room house has dropped by over 10% to $667,500. This is where experience and good understanding is needed by the data engineer.

Optimization

In order of increasing the prediction quality, optimization can be done by increasing the number of attributes / predictors or improving the input data. Understanding your data and functions is crucial in creating accurate predictions. We can see that there is some deviance between the different methods, but researchers found, that the deviance between the methods decreases, the more data you use. Basically, if you insert 2,000,000,000,000 data points into a system, the difference between the regression methods goes against 0.

I hope you got a good first taste of what you can do with ML. If you want to continue and have a look at the available regression methods, have a look at this -> sklearn-models. See you next time!