Song's Popularity - Social Media Management Project

Giuliano Lazzara

June 2021

Index

1 Introduction

The purpose of this project is to look for the existence about the possible correlation between the popularity of the tracks and their technical features.
The project is based on the database of the musical service of **Spotify**, which is one of the music streaming platforms with the greatest numbers of registered users (356 millions of active users in the first quarter of 2021 [1]) and with one of the more rich libraries (70 million songs on Spotify, with new songs added at a rate of 60,000/day [2]).
This project has been developed on Python 3.8, besides the "**`spotipy`**" library has been used to elaborate the data given by the API of Spotify. For the most impartial sampling possible, all playlists of the profile of Spotify have been taken, in addition, in order to be able to take a greater vast and varied quantity of possible music tracks, this algorithm cares to extrapolate them directly from the playlists that has been created and shared on the Spotify's profile on the black-green platform.

2 API Spotify

In order to work on the API of Spotify it need to create (or to have still) an account, so we can authenticate and then we can log in the page of "**Spotify for Developers**".
(`https://developer.spotify.com/dashboard/`)

After the login, it need to click on "**Create an App**" hence two codes can be obtained, able to access the API; they are the "**`client_ID`**" and the "**`client_secret`**". These codes are generated only for the application and not for the client's profile.
After this little regression about the possibility to work with the Spotify's provided data, it is going to describe the features chosen in order to develop this research [3]. These features are:
`-duration_ms` (called "**`length`**" within the code):
"The duration of the track in milliseconds." That in this program is converted in minutes and seconds for every track.
`-energy`:
"Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale. Perceptual features contributing to this attribute include dynamic range, perceived loudness, timbre, onset rate, and general entropy."
`-danceability`:
"Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable."

-acousticness:
"A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic."
it is important to report also the description of the **popularity**, fulcrum of this project:
"The value will be between 0 and 100, with 100 being the most popular. The popularity is calculated from the popularity of the album's individual tracks."
This feature has a little problem, sometimes it is equal to zero, irrelevant result for those not popular songs, but this result could be controversial for popular songs. To avoid any kind of controversial result, every track that has popularity equal to zero has not been considered for the study of this project.

3 Machine Learning's Algorithms and Performance Measures

About the research of any correlation between these features of the tracks and their popularity, two regressors have been used: the "**Multiple Linear Regressor**" and the "**Polynomial Regression**"
Generally, the Regression is the perfect function to search this correlation; let's start from the definition of the function of the straight line:

$$f(x) = y = q + mx$$

In the base case you only have one independent and the other dependent variable. Conventionally in this project the parameters q and m are indicated as θ_0 and θ_1:

$$y = \theta_0 + \theta_1 x$$

This one, that is the Simple Linear Regression, sets a correlation between x and y through the adjustment of the θ parameters.
Considering that this project is working with more features in input, the function of the regression is updated by adding one parameter θ for each x in input; therefore the applied function is :

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$$

where x_1 = **length**, x_2 = **energy**, x_3 = **danceability**, x_4 = **acousticness**.
The peculiarity of the Polynomial Regression consists of the features of being replicated **m** times up to **m-th** elevation. The advantage of using this regressor is that is possible to find, in the graphic 2D representation, a more complex geometric figure than the straight line that it is possible to obtain with a linear regressor. However it is important to set a degree that is not too high for the polynomial, because otherwise it could possible to bump into a case of **Overfitting** which would mean that the model fits the data very well of the **Training Set**, but not those of the **Test Set**. The implemented Performance Measures

have been the **MAE** (**Mean Absolute Error**) and the **RMSE** (**Root Mean Squared Error**).
MAE calculates the distance between the predicted and the actual value which is expressed with the below formula:

$$MAE(Y_{TE}, \hat{Y}_{TE}) = \frac{\sum_{i=1}^{|TE|} |y^{(i)} - \hat{y}^{(i)}|}{n}$$

The Root Mean Squared Error indicates the average discrepancy between the values of the observed data and the values of the estimated data. This one is calculated:

$$RSME(Y_{TE}, \hat{Y}_{TE}) = \sqrt{\frac{1}{|TE|} \sum_{i=1}^{|TE|} \parallel \hat{\mathbf{y}} - \mathbf{y} \parallel_2^2}$$

4 Implementation

Imported libraries:

```
import spotipy
from spotipy.oauth2 import SpotifyClientCredentials
import csv
import math
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error
from xgboost import XGBRegressor
```

-**`spotipy`** presents inside all the functions necessary to work better with the data provided by the API of Spotify.
-The library **`SpotifyClientCredentials`**, imported from **`spotipy.oauth2`**, is used to import the id_client and the secret_client.
-The **`csv`** library has been used to create a `.csv` file where it is possible to collect all the tracks and in the meantime elaborate their data.
-The **`panda`** and **`numpy`** libraries are very important to analyze, to manipulate and to view the data. Specifically in this project, these libraries have been used to read the file where all the tracks are present and above all to divide the dataset in two parts, Training Set and Test Set, selecting the relevant features for data processing (therefore excluding the column within the file that presents the names of all the tracks).

-**pyplot**, imported from **matplotlib**, has been used for the representation of the data in 2D graphics.

The **sklearn** library has been the most important library for this project, because from this one has been imported the libraries that allow:
-to split properly the dataset (**train_test split**);
-to adjust the θ parameters to calculate the polynomial regression (**PolynomialFeatures**);
-to calculate the linear and polynomial regression (**LinearRegression**, **Ridge**)
-to calculate the **MSE** of which the square root is made, thanks to the **math** library, so it is possible to obtain the **RMSE** (**mean_squared_error**);
Finally the **xgboost** library, to be able to implement the **XGBRegressor** that will be discussed later.

```
auth_manager = SpotifyClientCredentials(
    client_id='',     #Insert here your client_id
    client_secret='') #Insert here your client_secret
sp = spotipy.Spotify(auth_manager=auth_manager)
```

As anticipated in the second paragraph, these lines of code need to start the connection between the program and the Spotify's API, that using these codes, it allows the program to work with the API.

```
def getTrackIDs(playlist_id):
    ids = []
    play_lists = sp.playlist(playlist_id)
    for item in play_lists['tracks']['items']:
        track = item['track']
        ids.append(track['id'])
    return ids
```

With this function, as the name implies, it is possible to obtain the **URI** of all present songs in the playlists of duty.

Each track has an URI link that allows it to be recognised, through this link is possible to obtain all the features of the song, which are many (as it is described in the page "Spotify for Developers" [3]), but for the purpose of this project have been extrapolated only the interested features through this function:

```
def getTrackFeatures(id):
    meta = sp.track(id)
    features = sp.audio_features(id)
    name = meta['name']
    length = meta['duration_ms']
    popularity = meta['popularity']
    danceability = features[0]['danceability']
    energy = features[0]['energy']
    acousticness = features[0]['acousticness']

    track = [name, round(length / 60000, 2),  energy, danceability,
    acousticness, popularity]

    return track
```

Length is divided by 60000 ($60 \cdot 1000$) to view the time in minutes and not in milliseconds.

```
playlists = sp.user_playlists('spotify')
```

With this line of code it is possible to select the profile of who it is desired to analyze the playlists. For the purpose of this project has been chosen the profile of Spotify, where it is possible to find almost 1500 different playlists.

Now a crucial part of the program will be analyzed, the creation of the `.csv` file and the allocation of the tracks within it:

```
with open('Songs.csv', 'w', newline="") as file:
    writer = csv.writer(file)
    writer.writerow(['name','length', 'energy', 'danceability',
    'acousticness', 'popularity'])
```

Using **`writer.writerow`** the program is just writing, on the top of the file, the names of the different features.

```
while playlists:
    for i, playlist in enumerate(playlists['items']):
        try:
            ids = getTrackIDs(playlist['uri'])
            print("%4d %s " % (i + 1 + playlists['offset'], playlist['uri']))
            print(ids)
            for j in range(len(ids)):
                try:
                    track = getTrackFeatures(ids[j])
                    if track[5]!=0 and track[1]<8.0
                    and track[2]!=0 and track[3]!=0 and track[4]!=0:

                        writer.writerow([track[0], track[1],
                        track[2], track[3], track[4], track[5]])
                except:
                    pass
        except:
            pass
    if playlists['next']:
        playlists = sp.next(playlists)
    else:
        playlists = None
```

It works with **`playlists`** that contain the list of the URI links of all playlists of the profile of which it is given input in the line of code

```
playlists = sp.user_playlists('spotify')
```

In the first **for** cycle all the present playlist in the array **`playlists`** are scanned and then the **`GetTrackIDs`** function starts to work, so it is possible to collect all URI of the present songs in the playlist and these one are allocated in the **`ids`** array, which is reset to zero at each cycle.
Once this array is achieved, a new **for** cycle starts that goes from j to the length of the array to extrapolate the tracks from the IDs of the array. Taken an ID of a track, it is checked that all the considered features are non-zero, so it is possible to save the track and all its features inside the `.csv` file thanks to this line of the code:

```
writer.writerow([track[0], track[1], track[2], track[3], track[4], track[5]])
```

At this point the database created collects more than 68.000 songs. However it has been observed that a large number of songs are featured in multiple playlist even countless time. Therefore, a function has been applied to make the database more heterogeneous. This function is able to eliminate all duplicates from a `.csv` file.

```
play_lists = pd.read_csv('Songs.csv')
play_lists.drop_duplicates(subset=None, inplace=True)
play_lists.to_csv('Songs.csv', index=False)
```

When the database is ready it is possible to apply the suitable function to split the data set in training set and test set:

```
playlists_train, playlists_test = train_test_split(play_lists,
test_size=0.10, random_state=9)
```

Once the split is set it is possible to define explicitly the training set and test set by selecting the interested features:

```
    X_train = pd.DataFrame(np.c_[playlists_train['length'],
                                 playlists_train['energy'],
                                 playlists_train['danceability'],
                                 playlists_train['acousticness']],
            columns=['length', 'energy', 'danceability', 'acousticness'])

X_test = pd.DataFrame(np.c_[playlists_test['length'],
                            playlists_test['energy'],
                            playlists_test['danceability'],
                            playlists_test['acousticness']],
            columns=['length', 'energy', 'danceability', 'acousticness'])
```

The same thing for the labels:

```
y_train = playlists_train['popularity']
y_test = playlists_test['popularity']
```

No particular function was called to calculate the MAE, but it has been defined internally of the program:

```
def MAE(y_true, y_pred):
    return (y_true - y_pred).abs().mean()
```

The **`.fit()`** function is used to apply the Linear Regression, giving it in input the features of the training set and the labels of the training set.

```
lr = LinearRegression()
lr.fit(X_train, y_train)
```

While the function **`.predict()`** is being used to predict the labels.

```
y_train_preds = lr.predict(X_train)
y_test_preds = lr.predict(X_test)
```

The advantage of working with the regressors of the **`sklearn`** library is that there is no need to manually model the parameters used by the regressors. These parameters obtained by the regressors of this library are optimized, which means that the result of the regression is the best obtainable.

The application of the Polynomial Regression is rather similar to the application of the Linear Regression. However, in the Linear Regression is needed to adapt the features for the polynomial model.

```
pf = PolynomialFeatures(degree=4)
```

With this function it is possible to select the degree of the polynomial to obtain a good representation model avoiding the **Over-fitting**. Thereafter the features are updated and the polynomial regressor is defined:

```
X_train_poly = pf.fit_transform(X_train)
X_test_poly = pf.fit_transform(X_test)
pr = Ridge()
pr.fit(X_train_poly, y_train)
```

For this project has been applied another regressor imported from another library, **`XGBoost`**. This library is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the **Gradient Boosting** framework.[4]
As showed on the official web page of the documentation of this library [5] it is possible to set manually the parameters of the function **`XGBRegressor()`**. However for the purpose of this project the function has been left in default, anyway giving back the best results over all the regressors applied.

5 Results of the regressors

Necessary introduction, on the representation graphic of the obtained results, on the y-axis are showed the values of **Predicted Popularity**, while on x-axis are showed the actual values of **Popularity**.
An ideal representation of the relation between predicted labels and real labels would be a straight line passing from the origin with function $y = x$, which would mean that every predicted label is equal to a corresponding effective label.

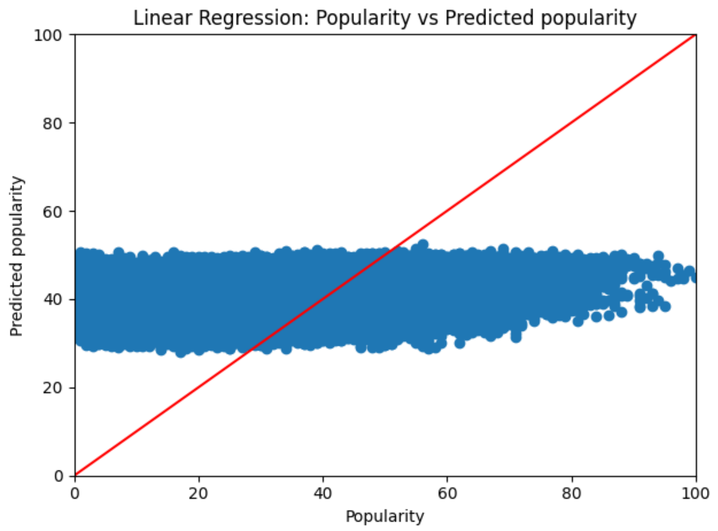

Figure 1

Observing the obtained graphic it is possible to notice that on the popularity of the 50.000 analyzed songs, just a minority of them fall on the function mentioned above (the red straight line in the **Figure 1**). From this, it can be concluded that the Linear Regression model is not optimal for the representation of this correlation.

The Polynomial Regression changes slightly the situation; it has been applied a fourth-degree polynomial that fits the data marginally better than the Linear Regression.

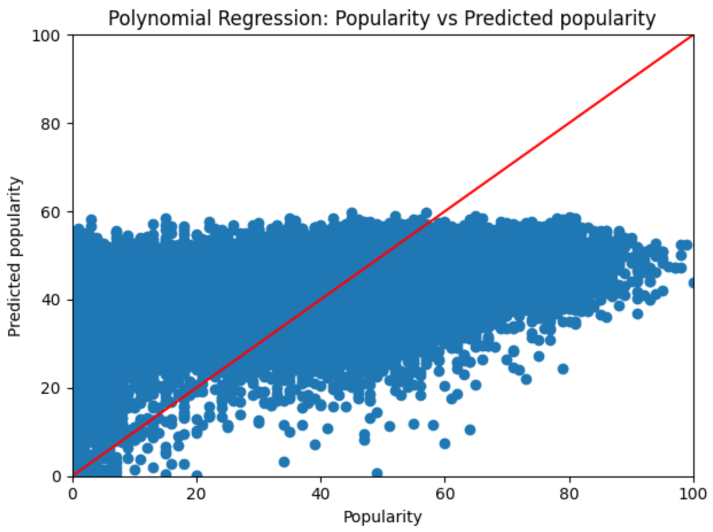

Figure 2

But even with this regressor it is still present a case of **Under-fitting**. With the application of the **XGBooster** the situation changes marginally again, in fact it is possible to notice that the distribution of the obtained values follows better the representing function of the ideal correlation.

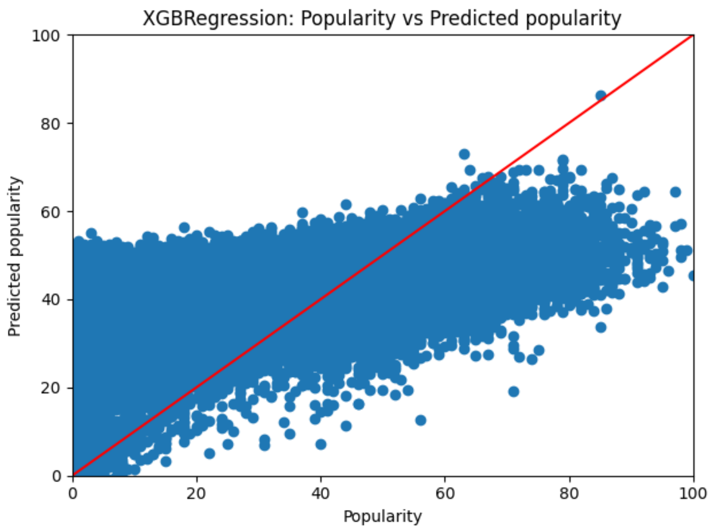

Figure 3

Consequently, among the applied models, the best performing has been the **`XGBRegressor()`**.

6 Analysis of the results and Conclusions

Even though it has been obtained representations very different among them, it is important to analyze also the results of the Performance Measures:

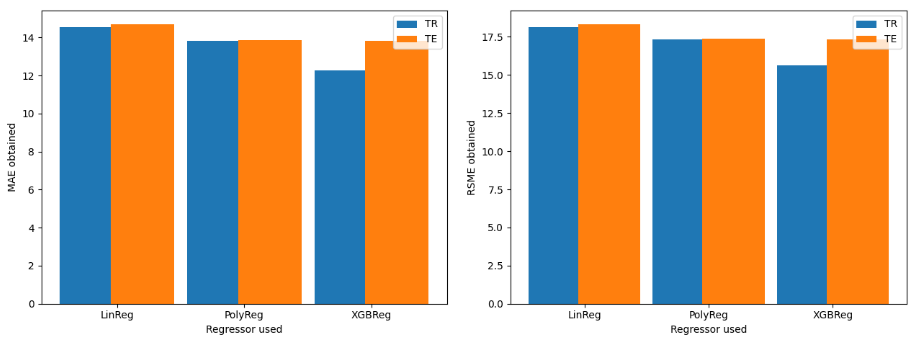

Figure 4

The MAE and RSME, in spite of decreasing before in the Polynomial Regression and subtly with the XGB regressor then, remain quite high; the best MAE is obtained with the **`XGBRegressor()`** on the **Training Set** (12,2673), while the best RMSE is registered still with this regressor and still on the **Training Set** (15,6370).
It is also interesting to notice the distribution of the popularity comparing individually with each feature considered.

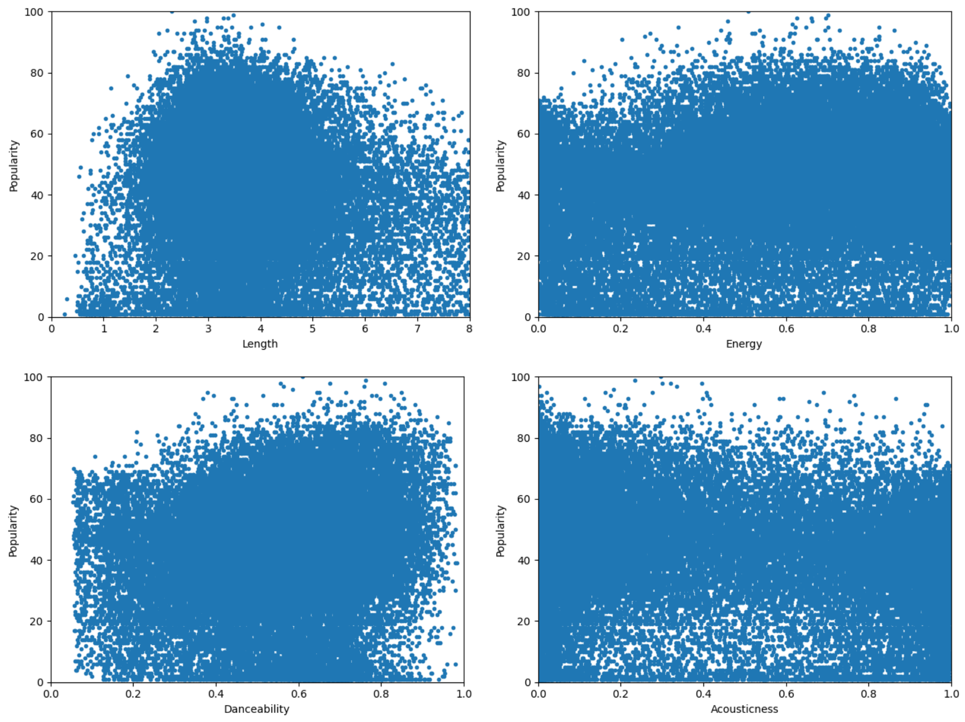

Figure 5

From this graphs looks factual that there is not a real correlation between the features and the popularity of the song, indeed, considering the second graph of the **Figure 5**, it is clear that for the same value of **`energy`** corresponds tracks with different values of popularity.

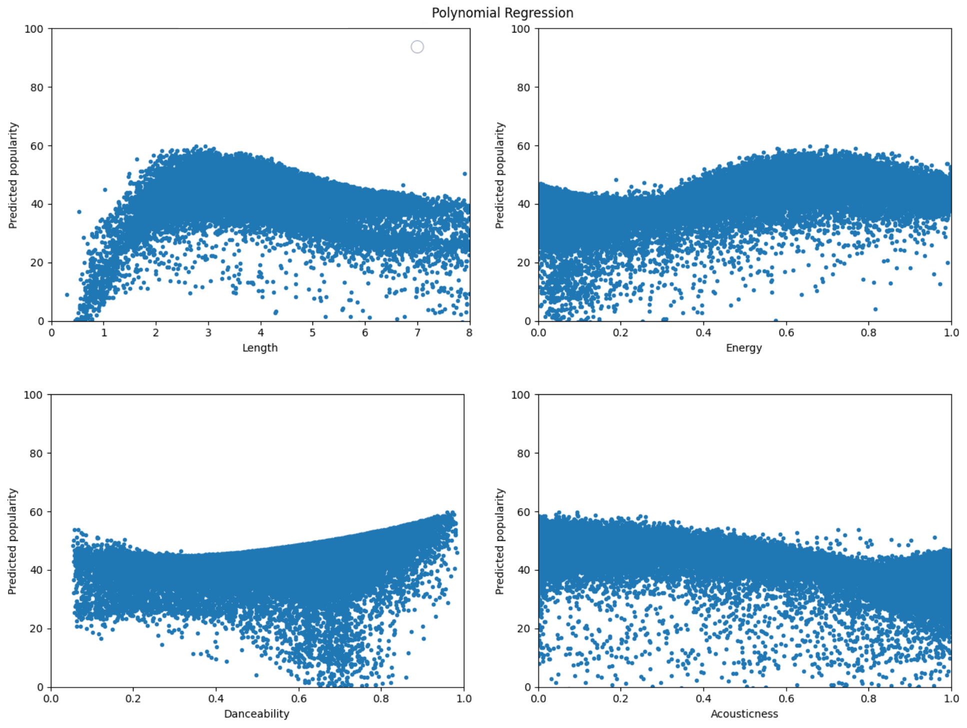

Figure 6

Plotting the features with the predicted popularity it is possible to see how the features remain equally distributed, meanwhile the popularity seems bounded in a certain defined neighborhood of values.(in **Figure 6** are plotted the features with the predicted labels obtained with the Polynomial Regression)
This can be explained by the fact that, according with all the elaborated data that have been used, the regressors try to find a direct correlation between the features length, energy, danceability, acousticness and the popularity of a track that, however, according with the results obtained, does not exist, disproving in this way the thesis of this project.

7 References

[1] - Spotify–Financials, "Shareholder-Letter-Q1-2021_FINAL.pdf",
URLs: https://investors.spotify.com/financials/default.aspx.
https://s22.q4cdn.com/540910603/files/doc_financials/2021/q1/Shareholder-Letter-Q1-2021_FINAL.pdf

[2] - Business of Apps, "Spotify Revenue and Usage Statistics (2021)",
URL: https://www.businessofapps.com/data/spotify-statistics/.

[3] - Business of Apps, "Spotify Revenue and Usage Statistics (2021)",
URL: https://www.businessofapps.com/data/spotify-statistics/.

[4] - XGBoost, "XGBoost Documentation",
URL: https://xgboost.readthedocs.io/en/latest/index.html.

[5] - XGBoost, "Python API Reference",
URL: https://xgboost.readthedocs.io/en/latest/python/python_api.html?highlight=xgbregressor#xgboost.XGBRegressor.