Sayak Paul
August 15th, 2018

PYTHON +1

Hyperparameter Optimization in Machine Learning Models

This tutorial covers what a parameter and a hyperparameter are in a machine learning model along with why it is vital in order to enhance your model's performance.

Machine learning involves predicting and classifying data and to do so, you employ various machine learning models according to the dataset. Machine learning models are parameterized so that their behavior can be tuned for a given problem. These models can have many parameters and finding the best combination of parameters can be treated as a search problem. But this very term called parameter may appear unfamiliar to you if you are new to applied machine learning. But don't worry! You will get to know about it in the very first place of this blog, and you will also discover what the difference between a parameter and a hyperparameter of a machine learning model is. This blog consists of following sections:

- What are a parameter and a hyperparameter in a machine learning model?
- Why hyperparameter optimization/tuning is vital in order to enhance your model's performance?
- Two simple strategies to optimize/tune the hyperparameters
- A simple case study in Python with the two strategies

A model parameter is a configuration variable that is internal to the model and whose value can be estimated from the given data.

- They are required by the model when making predictions.
- Their values define the skill of the model on your problem.
- They are estimated or learned from data.
- They are often not set manually by the practitioner.
- They are often saved as part of the learned model.

So your main take away from the above points should be parameters are crucial to machine learning algorithms. Also, they are the part of the model that is learned from historical training data. Let's dig it a bit deeper. Think of the function parameters that you use while programming in general. You may pass a parameter to a function. In this case, a parameter is a function argument that could have one of a range of values. In machine learning, the specific model you are using is the function and requires parameters in order to make a prediction on new data. Whether a model has a fixed or variable number of parameters determines whether it may be referred to as *"parametric"* or *"nonparametric"*.

Some examples of model parameters include:

- The weights in an artificial neural network.
- The support vectors in a support vector machine.
- The coefficients in a linear regression or logistic regression.

What is a hyperparameter in a machine learning learning model?

A model hyperparameter is a configuration that is external to the model and whose value cannot be estimated from data.

- They are often used in processes to help estimate model parameters.
- They are often specified by the practitioner.
- They can often be set using heuristics.
- They are often tuned for a given predictive modeling problem.

Want to leave a comment?

and error. When a machine learning algorithm is tuned for a specific problem then essentially you are tuning the hyperparameters of the model to discover the parameters of the model that result in the most skillful predictions.

According to a very popular book called "Applied Predictive Modelling" - "*Many models have important parameters which cannot be directly estimated from the data. For example, in the K-nearest neighbor classification model ... This type of model parameter is referred to as a tuning parameter because there is no analytical formula available to calculate an appropriate value.*"

Model hyperparameters are often referred to as model parameters which can make things confusing. A good rule of thumb to overcome this confusion is as follows: "*If you have to specify a model parameter manually, then it is probably a model hyperparameter.* " Some examples of model hyperparameters include:

- The learning rate for training a neural network.
- The C and sigma hyperparameters for support vector machines.
- The k in k-nearest neighbors.

In the next section, you will discover the importance of the right set of hyperparameter values in a machine learning model.

Importance of the right set of hyperparameter values in a machine learning model:

The best way to think about hyperparameters is like the settings of an algorithm that can be adjusted to optimize performance, just as you might turn the knobs of an AM radio to get a clear signal. When creating a machine learning model, you'll be presented with design choices as to how to define your model architecture. Often, you don't immediately know what the optimal model architecture should be for a given model, and thus you'd like to be able to explore a range of possibilities. In a true machine learning fashion, you'll ideally ask the machine to perform this exploration and select the optimal model architecture automatically.

Two simple strategies to optimize/tune the hyperparameters:

Models can have many hyperparameters and finding the best combination of parameters can be treated as a search problem.

Although there are many hyperparameter optimization/tuning algorithms now, this post discusses two simple strategies: 1. grid search and 2. Random Search.

Grid searching of hyperparameters:

Grid search is an approach to hyperparameter tuning that will methodically build and evaluate a model for each combination of algorithm parameters specified in a grid.

Let's consider the following example:

Suppose, a machine learning model X takes hyperparameters a_1, a_2 and a_3. In *grid searching*, you first define the range of values for each of the hyperparameters a_1, a_2 and a_3. You can think of this as an array of values for each of the hyperparameters. Now the *grid search* technique will construct many versions of X with all the possible combinations of hyperparameter (a_1, a_2 and a_3) values that you defined in the first place. This range of hyperparameter values is referred to as the ***grid***.

Suppose, you defined the grid as:
a_1 = [0,1,2,3,4,5]

a_2 = [10,20,30,40,5,60]

a_3 = [105,105,110,115,120,125]

Note that, the array of values of that you are defining for the hyperparameters has to be legitimate in a sense that you cannot supply *Floating* type values to the array if the hyperparameter only takes *Integer* values.

Now, *grid search* will begin its process of constructing several versions of X with the grid

computationally very expensive.

Let's take a look at the other search technique Random search:

Random searching of hyperparameters:

The idea of random searching of hyperparameters was proposed by James Bergstra & Yoshua Bengio. You can check the original paper here.

Random search differs from a grid search. In that you longer provide a discrete set of values to explore for each hyperparameter; rather, you provide a statistical distribution for each hyperparameter from which values may be randomly sampled.

Before going any further, let's understand what distribution and sampling mean:

In Statistics, by distribution, it is essentially meant an arrangement of values of a variable showing their observed or theoretical frequency of occurrence.

On the other hand, Sampling is a term used in statistics. It is the process of choosing a representative sample from a target population and collecting data from that sample in order to understand something about the population as a whole.

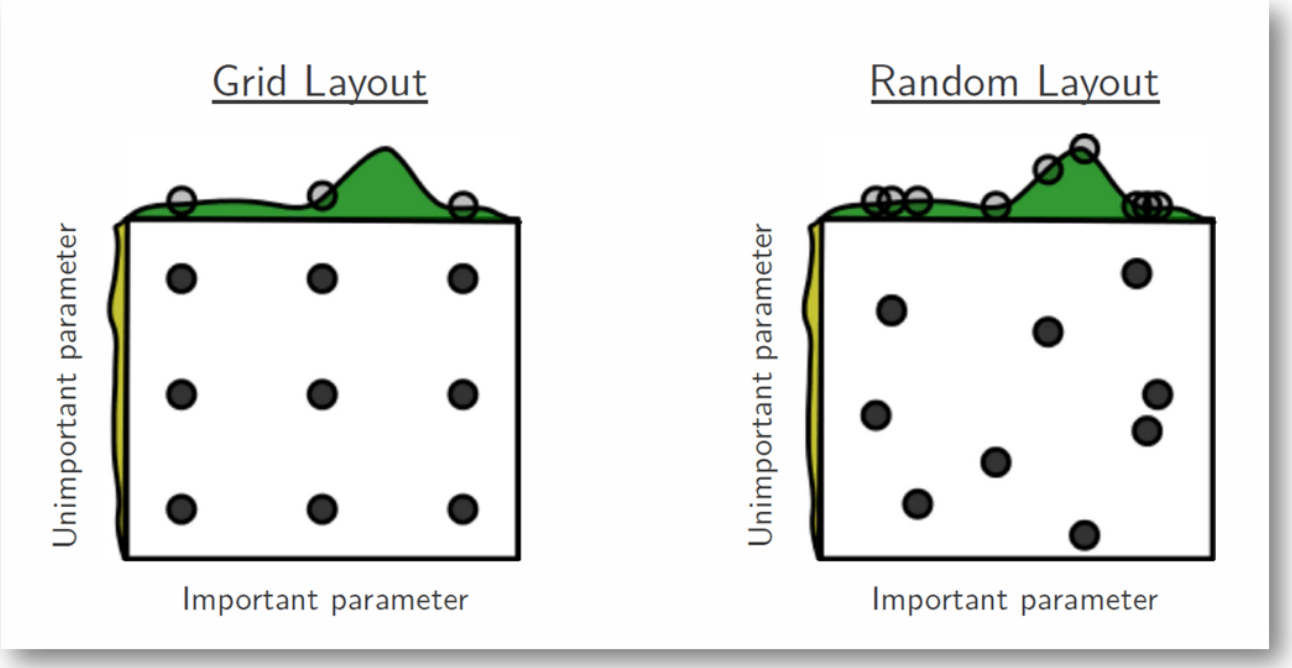
Now let's again get back to the concept of random *search*.

You'll define a sampling distribution for each hyperparameter. You can also define how many iterations you'd like to build when searching for the optimal model. For each iteration, the hyperparameter values of the model will be set by sampling the defined distributions. One of the primary theoretical backings to motivate the use of a random search in place of grid search is the fact that for most cases, hyperparameters are not equally important. According to the original paper:

"*....for most datasets only a few of the hyper-parameters really matter, but that different hyper-parameters are important on different datasets. This phenomenon makes grid search a poor choice for configuring algorithms for new datasets*."

Want to leave a comment?

model and spends redundant time exploring the unimportant parameter. During this grid search, we isolated each hyperparameter and searched for the best possible value while holding all other hyperparameters constant. For cases where the hyperparameter being studied has little effect on the resulting model score, this results in wasted effort. Conversely, the random search has much improved exploratory power and can focus on finding the optimal value for the critical hyperparameter.



Source: Random Search for Hyper-Parameter Optimization

In the following sections, you will see *grid search* and *random search* in action with Python. You will also be able to decide which is better regarding the effectiveness and efficiency.

Case study in Python:

Hyperparameter tuning is a final step in the process of applied machine learning before presenting results.

You will use the Pima Indian diabetes dataset. The dataset corresponds to a *classification* problem on which you need to make predictions on the basis of whether a person is to suffer diabetes given the 8 features in the dataset. You can find the complete description

Want to leave a comment?

need.

```
# Dependencies

import pandas as pd
import numpy as np
from sklearn.linear_model import LogisticRegression
```

Now that the dependencies are imported let's load Pima Indians dataset into a Dataframe object with the famous Pandas library.

```
data = pd.read_csv("diabetes.csv") # Make sure the .csv file and the notebook are re
```

The dataset is successfully loaded into the Dataframe object *data*. Now, let's take a look at the data.

```
data.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

So you can 8 different features labeled into the outcomes of 1 and 0 where 1 stands for the observation has diabetes, and 0 denotes the observation does not have diabetes. The dataset is known to have missing values. Specifically, there are missing observations for some columns that are marked as a zero value. We can corroborate this by the definition of those columns, and the domain knowledge that a zero value is invalid for those measures, e.g., zero for body mass index or blood pressure is invalid.

(Missing value creates a lot of problems when you try to build a machine learning model.

Want to leave a comment?

Let's get some statistics about the data with Pandas' *describe()* utility.

```
data.describe()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

This is useful.

We can see that there are columns that have a minimum value of zero (0). On some columns, a value of zero does not make sense and indicates an invalid or missing value.

Specifically, the following columns have an invalid zero minimum value:

- Plasma glucose concentration
- Diastolic blood pressure
- Triceps skinfold thickness
- 2-Hour serum insulin
- Body mass index

Now you need to identify and mark values as missing. Let's confirm this by looking at the raw data, the example prints the first 20 rows of data.

```
data.head(20)
```

Want to leave a comment?

1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
5	5	116	74	0	0	25.6	0.201	30	0
6	3	78	50	32	88	31.0	0.248	26	1
7	10	115	0	0	0	35.3	0.134	29	0
8	2	197	70	45	543	30.5	0.158	53	1
9	8	125	96	0	0	0.0	0.232	54	1
10	4	110	92	0	0	37.6	0.191	30	0
11	10	168	74	0	0	38.0	0.537	34	1
12	10	139	80	0	0	27.1	1.441	57	0
13	1	189	60	23	846	30.1	0.398	59	1
14	5	166	72	19	175	25.8	0.587	51	1
15	7	100	0	0	0	30.0	0.484	32	1
16	0	118	84	47	230	45.8	0.551	31	1
17	7	107	74	0	0	29.6	0.254	31	1
18	1	103	30	38	83	43.3	0.183	33	0
19	1	115	70	30	96	34.6	0.529	32	1

You can see 0 in several columns, right?

You can get a count of the number of missing values in each of these columns. You can do this by marking all of the values in the subset of the DataFrame you are interested in that have zero values as True. You can then count the number of true values in each column. For this, you will have to reimport the data without the column names.

```
data = pd.read_csv("https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima
print((data[[1,2,3,4,5]] == 0).sum())
```

```
1      5
2     35
3    227
4    374
5     11
```

Want to leave a comment?

that is natural. Column 8 denotes the target variable so, '0's in it is natural.

This highlights that different “missing value” strategies may be needed for different columns, e.g., to ensure that there are still a sufficient number of records left to train a predictive model.

In Python, specifically Pandas, NumPy and Scikit-Learn, you mark missing values as NaN.

Values with a NaN value are ignored from operations like sum, count, etc.

You can mark values as NaN easily with the Pandas DataFrame by using the replace() function on a subset of the columns you are interested in.

After you have marked the missing values, you can use the isnull() function to mark all of the NaN values in the dataset as True and get a count of the missing values for each column.

```
# Mark zero values as missing or NaN
data[[1,2,3,4,5]] = data[[1,2,3,4,5]].replace(0, np.NaN)
# Count the number of NaN values in each column
print(data.isnull().sum())
```

```
0      0
1      5
2     35
3    227
4    374
5     11
6      0
7      0
8      0
dtype: int64
```

You can see that the columns 1:5 have the same number of missing values as zero values

Want to leave a comment?

Below is the same example, except you print the first 5 rows of data.

```
data.head()
```

	0	1	2	3	4	5	6	7	8
0	6	148.0	72.0	35.0	NaN	33.6	0.627	50	1
1	1	85.0	66.0	29.0	NaN	26.6	0.351	31	0
2	8	183.0	64.0	NaN	NaN	23.3	0.672	32	1
3	1	89.0	66.0	23.0	94.0	28.1	0.167	21	0
4	0	137.0	40.0	35.0	168.0	43.1	2.288	33	1

It is clear from the raw data that marking the missing values had the intended effect. Now, you will impute the missing values. Imputing refers to using a model to replace missing values. Although there are several solutions for imputing missing values, you will use mean imputation which means replacing the missing values in a column with the mean of that particular column. Let's do this with Pandas' fillna() utility.

```
# Fill missing values with mean column values
data.fillna(data.mean(), inplace=True)
# Count the number of NaN values in each column
print(data.isnull().sum())
```

```
0    0
1    0
2    0
3    0
4    0
5    0
```

Want to leave a comment?

Cheers! You have now handled the missing value problem. Now let's use this data to build a Logistic Regression model using scikit-learn.

First, you will see the model with some random hyperparameter values. Then you will build two other Logistic Regression models with two different strategies - Grid search and Random search.

```
# Split dataset into inputs and outputs
values = data.values
X = values[:,0:8]
y = values[:,8]
```

```
# Initiate the LR model with random hyperparameters
lr = LogisticRegression(penalty='l1',dual=False,max_iter=110)
```

You have created the Logistic Regression model with some random hyperparameter values. The hyperparameters that you used are:

- penalty : Used to specify the norm used in the penalization (regularization).
- dual : Dual or primal formulation. The dual formulation is only implemented for l2 penalty with liblinear solver. Prefer dual=False when n_samples > n_features.
- max_iter : Maximum number of iterations taken to converge.

Later in the case study, you will optimize/tune these hyperparameters so see the change in the results.

```
# Pass data to the LR model
lr.fit(X,y)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
```

Want to leave a comment?

It's time to check the accuracy score.

```
lr.score(X,y)
```

```
0.7747395833333334
```

In the above step, you applied your LR model to the same data and evaluated its score. But there is always a need to validate the stability of your machine learning model. You just can't fit the model to your training data and hope it would accurately work for the real data it has never seen before. You need some kind of assurance that your model has got most of the patterns from the data correct.

Well, Cross-validation is there for rescue. I will not go into the details of it as it is out of the scope of this blog. But this post does a very fine job.

```
# You will need the following dependencies for applying Cross-validation and evaluat

from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score


# Build the k-fold cross-validator

kfold = KFold(n_splits=3, random_state=7)
```

You supplied n_splits as 3, which essentially makes it a 3-fold cross-validation. You also supplied random_state as 7. This is just to reproduce the results. You could have supplied any integer value as well. Now, let's apply this.

```
result = cross_val_score(lr, X, y, cv=kfold, scoring='accuracy')
print(result.mean())
```

```
0.765625
```

Want to leave a comment?

Let's build another LR model, but this time its hyperparameter will be tuned. You will first do this grid search.

Let's first import the dependencies you will need. Scikit-learn provides a utility called GridSearchCV for this.

```
from sklearn.model_selection import GridSearchCV
```

Let's define the grid values of the hyperparameters that you used above.

```
dual=[True,False]
max_iter=[100,110,120,130,140]
param_grid = dict(dual=dual,max_iter=max_iter)
```

You have defined the grid. Let's run the grid search over them and see the results with execution time.

```
import time

lr = LogisticRegression(penalty='l2')
grid = GridSearchCV(estimator=lr, param_grid=param_grid, cv = 3, n_jobs=-1)

start_time = time.time()
grid_result = grid.fit(X, y)
# Summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
print("Execution time: " + str((time.time() - start_time)) + ' ms')


Best: 0.752604 using {'dual': False, 'max_iter': 100}
Execution time: 0.3954019546508789 ms
```

You can define a larger grid of hyperparameter as well and apply grid search

Want to leave a comment?

```
max_iter=[100,110,120,130,140]
C = [1.0,1.5,2.0,2.5]
param_grid = dict(dual=dual,max_iter=max_iter,C=C)


lr = LogisticRegression(penalty='l2')
grid = GridSearchCV(estimator=lr, param_grid=param_grid, cv = 3, n_jobs=-1)

start_time = time.time()
grid_result = grid.fit(X, y)
# Summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
print("Execution time: " + str((time.time() - start_time)) + ' ms')


Best: 0.763021 using {'C': 2.0, 'dual': False, 'max_iter': 100}
Execution time: 0.793781042098999 ms
```

You can see an increase in the accuracy score, but there is a sufficient amount of growth in the execution time as well. The larger the grid, the more execution time.

Let's rerun everything but this time with the random search. Scikit-learn provides RandomSearchCV to do that. As usual, you will have to import the necessary dependencies for that.

```
from sklearn.model_selection import RandomizedSearchCV


random = RandomizedSearchCV(estimator=lr, param_distributions=param_grid, cv = 3, n_

start_time = time.time()
random_result = random.fit(X, y)
# Summarize results
print("Best: %f using %s" % (random_result.best_score_, random_result.best_params_))
print("Execution time: " + str((time.time() - start_time)) + ' ms')
```

Want to leave a comment?

That is all for the case study part. Now, let's wrap things up!

Conclusion and further reading:

In this tutorial, you learned about parameters and hyperparameters of a machine learning model and their differences as well. You also got to know about what role hyperparameter optimization plays in building efficient machine learning models. You built a simple Logistic Regression classifier in Python with the help of scikit-learn.

You tuned the hyperparameters with grid search and random search and saw which one performs better.

Besides, you saw small data preprocessing steps (like handling missing values) that are required before you feed your data into the machine learning model. You covered Cross-validation as well.

That is a lot to take in, and all of them are equally important in your data science journey. I will leave you with some further readings that you can do.

Further readings:

- Problems in hyperparameter optimization
- Hyperparameter optimization with soft computing techniques
- Random Search for Hyper-Parameter Optimization

For the ones who are a bit more advanced, I would highly recommend reading this paper for effectively optimizing the hyperparameters of neural networks. link

If you would like to learn more about Machine Learning, take the following courses from DataCamp.

- Introduction to Machine Learning
- Machine Learning Toolbox

Want to leave a comment?

COMMENTS

Sayak Paul
16/08/2018 08:26 AM
The unit of the execution time has been mistakenly given as "ms", it will be in "seconds". Apologies for the mistake.

▲ 3 REPLY

Ton Amachree
08/11/2018 06:31 PM
Really nice, I have just had my first introduction to machine learning and this article was a good read and helpful definitions

▲ 1 REPLY

Sayak Paul
08/11/2018 07:10 PM
Thanks Ton for the kind words.

▲ 1 REPLY

Jesus Obispo
01/12/2018 08:45 AM
I have a problem, I am analyzing another dataset, I find that the output is of continuous values and the output in your dataset is discrete, how do I solve it?

▲ 1 REPLY

Sayak Paul
01/12/2018 08:50 AM
I am not clear on your problem Jesus. You will have to elaborate more.

▲ 1 REPLY

Anthony Agudelo
28/12/2018 08:59 PM
Very well written and informative. I was however wondering how do you go about choosing

Want to leave a comment?

▲ 1 REPLY

Sayak Paul
28/12/2018 09:28 PM

Thanks Anthony for the kind words. Random Forests and Logistic Regression, Neural Networks are different types of ML algorithms. While Random Forest Models are non-parametric in nature Logistic Regression, Neural Networks these are parametric.

So naturally, their hyperparameters are way more different than each other. But when it comes to interpretation I find RF models a bit more classier than LR and NN models. However, there is a systematic way to define the hyperparameters of a Neural Network. This paper outlines that out in very elegant manner.

For Random Forests, the choice of hyperparameters largely depends on the problem (in fact choice of the hyperparameter values is problem dependent). I would suggest to check the tutorials of fastai on Random Forests.

▲ 1 REPLY

Shibu Mathew
07/01/2019 07:16 PM

Nice article. Correct me if i am wrong but we could avoid the snippet where we reimport the dataset excluding headers and instead, directly move to the replace step using: data[['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin','BMI']]=data[['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin','BMI']].replace(0,np.NaN)

Also if anyone is wondering why, in the Gridsearch method, we have specifically used only penalty = l2, its because penalty = l1 would fail in one of the search combinations. I was curious to include l1 and l2 in the grid and ended up with a failure. Which is what the author said at the start , that is, ensure you are using valid combinations. Thanks , very clear style of teaching

▲ 1 REPLY

Sayak Paul
08/01/2019 08:42 PM

Hi Shibu. Glad you liked the article. And you are correct regarding the direct replacement step.

▲ 1 REPLY

Want to leave a comment?

Aritra Roy Gosthipaty
19/03/2019 11:39 PM
Coming from an absolute beginner, the article is well written. The analogies and examples have enriched the experience.

▲ 3 REPLY

Sayak Paul
20/03/2019 08:15 AM
Thanks :)

▲ 2 REPLY

Amartya Sen
10/04/2019 07:09 PM
Nice Theme btw.. how do i install your theme?

▲ 2 REPLY

Sayak Paul
10/04/2019 08:19 PM
What theme?

▲ 1 REPLY

Amartya Sen
10/04/2019 11:05 PM
You Jupyter notebook's theme -.-' Its dark blue instead of that boring white and orange

▲ 1

Sayak Paul
10/04/2019 11:27 PM
That is not my theme. It is rendered from the backend when a blog is posted here.

▲ 1

Want to leave a comment?