Get the data

We use the “heart\_disease” dataset. You can download it by clicking . The first few rows of the dataset are:

Table

Description automatically generated

First 5 rows of the “heart\_disease” dataset (Image by author)

This dataset contains 303 samples and 13 features. The last column is the target column which contains 0s (not having a heart disease) and 1s (having a heart disease).

The method

We build a Random Forest Classifier on the above data. Then, we especially consider the **max\_depth** hyperparameter of that classifier. The **max\_depth** value determines the number of times that each tree in the ensemble splits and after that, it stops branching. By ranging the values from 1 to 10 of the**max\_depth** hyperparameter, we plot cross-validated training and test scores. Then, we decide the best value for the **max\_depth** hyperparameter.

Let’s code

After building the model, I use only 1 line of code (less code) to create the validation curve. I take the advantage of the **Yellowbrick** machine learning visualization library. However, it doesn't come with the Anaconda installer. You need to manually install it. Open your Anaconda prompt and just run the following command.

pip install yellowbrick

If that didn’t work for you, try the following with the ***user*** tag.

pip install yellowbrick --user

or try it with the **conda-forge** channel.

conda install -c conda-forge yellowbrick

or try it with the **DistrictDataLabs** channel.

conda install -c districtdatalabs yellowbrick

Now, see the following code.

|  |
| --- |
| # Importing libraries |
|  | import numpy as np |
|  | import pandas as pd |
|  | from sklearn.utils import shuffle |
|  | from sklearn.ensemble import RandomForestClassifier |
|  | from yellowbrick.model\_selection import validation\_curve |
|  |  |
|  | df = pd.read\_csv("heart\_disease.csv") # Loading the data |
|  | df = shuffle(df, random\_state=3) # Shuffling the data |
|  |  |
|  | X = df.iloc[:,:-1] # Feature matrix in pd.DataFrame format |
|  | y = df.iloc[:,-1] # Target vector in pd.Series format |
|  |  |
|  | # Making a Random Forest Classifier object |
|  | rf = RandomForestClassifier(n\_estimators=100, criterion='gini', |
|  | max\_depth=None, n\_jobs=-1, random\_state=42) |
|  |  |
|  | # Plot the validation curve |
|  | print(validation\_curve(rf, X, y, param\_name="max\_depth", n\_jobs=-1, |
|  | param\_range=np.arange(1, 11), cv=10, scoring="accuracy")) |

The output is:

Chart, line chart

Description automatically generated

Validation Curve on the max\_depth hyperparameter (Image by author)

Let’s explain

When we execute the **validation\_curve()** function, a lot of work happens behind the scenes. The first argument of this function should be a Scikit-learn estimator (here it is a Random Forest Classifier). The second and third ones should be**X** (feature matrix) and **y** (target vector). The **param\_name**contains the name of the hyperparameter that we want to measure the influence. **“n\_jobs=-1”** means that we use all the cores of the computer processor to do parallel computations when doing the cross-validation procedure. **“param\_range”** contains a 1-dimensional numpy array of possible parameter values. In our example, those values should be integers starting from 1. Zeros and negatives are not acceptable values for **max\_depth**. The **“cv”** defines the number of folds for the cross-validation. Standard values are 3, 5, and 10. The **scoring** argument contains the method of scoring of the model. In classification, **“accuracy”** and **“roc\_auc”** are most preferred. In regression, **“r2”** and **“neg\_mean\_squared\_error”** are commonly used. In addition to those, there are many evaluation metrics. You can find all of them by visiting.

When we execute the **validation\_curve()** function, the cross-validation procedure happens behind the scenes. Because of this, we just input**X** and**y**. We don’t need to split the dataset as **X\_train**, **y\_train**, **X\_test**, **y\_test**. In cross-validation, the splitting is done internally based on the number of folds we specified in **cv**. Using cross-validation here guarantees that the accuracy score of the model isn’t much affected by the random data splitting process. If you just use the**train\_test\_split()** function *withou*t cross-validation, the accuracy score will vary significantly based on the **random\_state** you provide inside the **train\_test\_split()** function. Here in cross-validation, the accuracy is calculated using the average of 10 (cv=10) such iterations!

In k-fold cross-validation, we make an assumption that all observations in the dataset are nicely distributed in a way that the data are not biased. That is why we first shuffle the dataset using the **shuffle** function.

**Note:** The same functionality of the **validation\_curve()** function can be achieved using the **ValidationCurve()**class. Here, you first create a visualizer (an object of the **ValidationCurve()**class), then use the common **.fit(X, y)**paradigm. Here is the code.

## Let’s interpret the validation curve

Now, we interpret the validation curve that we plotted previously. By looking at the curve, we can determine if the model is underfitting, overfitting or just-right for some range of hyperparameter values of**max\_depth**. Note that, in the graph, the accuracy score of the train set is marked as the “Training Score” and the accuracy score of the test set is marked as the “Cross-Validation Score”.

* **Underfitting:** Accuracy scores of both train and test sets are low. This indicates that the model is too simple or has been regularized too much. At the **max\_depth** values of 1 and 2, the random forests model is underfitting.
* **Overfitting:**The training accuracy score is very high and the accuracy score of the test set is low. The model fits very well for the training data, but it fails to generalize to new input data. For **max\_depth** values of 4, 5, …, 10, the model is highly overfitted.
* **Just-right:** No overfitting or underfitting. At the **max\_depth** value of 3, the model is just right. The model fits the training data very well and it is also generalizable to new input data. That’s what we want!

**Be careful:** When you use an evaluation metric such as MSE, the overfitting condition happens when the training MSE is very low (not high) and the MSE of the test set is high (not low). This is because here we consider an error (Mean Squared Error).

**Be careful:** Here, you got the optimal **max\_depth** hyperparameter value of 3. Keep in mind that this is what we got when we consider only the **max\_depth** hyperparameter. When we consider several hyperparameters at a time as in Grid Search or Randomized Search, the optimal **max\_depth** hyperparameter value will not be 3.

# **Summary**

Validation curve is a great tool that you should have in your machine learning toolkit. It can be used to plot the influence of a single hyperparameter. It should not be used to tune the model.

When creating the curve, the cross-validation method should be considered.