**Hyperparameter tuning for Machine learning models**

# ****Introduction****

This article covers the comparison and implementation of random search, grid search, and Bayesian optimization methods using Sci-kit learn and HyperOpt libraries for hyperparameter tuning of the machine learning model. Hyperparameters tuning is crucial as they control the overall behavior of a machine learning model. Every machine learning models will have different hyperparameters that can be set.

A hyperparameter is a parameter whose value is set before the learning process begins.

I will be using the [Titanic dataset](https://www.kaggle.com/c/titanic/data) from Kaggle for comparison. The purpose of this article to explore how the performance and the computational time of the random forest model are changing with various hyperparameter tuning methods. After all, machine learning is all about finding the right balance between computing time and the model’s performance.

## Baseline model with default parameters:

random\_forest = RandomForestClassifier(random\_state=1).fit(X\_train, y\_train)  
random\_forest.score(X\_test,y\_test)

The accuracy of this model, when used on the testing set, is **81.56**.

We can get the default parameters used for the model using the command. **randomforest.get\_params()**

**The default parameters are:**{'bootstrap': True, 'ccp\_alpha': 0.0, 'class\_weight': None, 'criterion': 'gini', 'max\_depth': None, 'max\_features': 'auto', 'max\_leaf\_nodes': None, 'max\_samples': None, 'min\_impurity\_decrease': 0.0, 'min\_impurity\_split': None, 'min\_samples\_leaf': 1, 'min\_samples\_split': 2, 'min\_weight\_fraction\_leaf': 0.0, 'n\_estimators': 100, 'n\_jobs': None, 'oob\_score': False, 'random\_state': 1, 'verbose': 0, 'warm\_start': False}

No need to worry if you don't know about these parameters and how they are used. Usually, information about all the parameters can be found in Scikit documentation of the models.

**Some important Parameters in Random Forest:**

1. **max\_depth: int, default=None**This is used to select how deep you want to make each tree in the forest. The deeper the tree, the more splits it has, and it captures more information about the data.
2. **criterion**:{“Gini,” “entropy”}, default=” Gini”: Measures the quality of each split. “Gini” uses the Gini impurity while “entropy” makes the split based on the information gain.
3. **max\_features: {“auto,” “sqrt,” “log2”}, int or float, default=” auto”:**This represents the number of features that are considered on a pre-split level when finding the best split. This improves the model's performance as each tree node is now considering a higher number of options.
4. **min\_samples\_leaf: int or float, default=1:**This parameter helps determine the minimum required number of observations at the end of each decision tree node in the random forest to split it.
5. **min\_samples\_split: int or float, default=2:**This specifies the minimum number of samples that must be present from your data for a split to occur.
6. **n\_estimators: int, default=100:**This is perhaps the most important parameter. This represents the number of trees you want to build within a random forest before calculating the predictions. Usually, the higher the number, the better, but this is more computationally expensive.

More info about other parameters can be found in the random forest classifier model [documentation](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html).

# Grid Search

One traditional and popular way to perform hyperparameter tuning is by using an Exhaustive Grid Search from Scikit learn. This method tries every possible combination of each set of hyper-parameters. Using this method, we can find the best set of values in the parameter search space. This usually uses more computational power and takes a long time to run since this method needs to try every combination in the grid size.

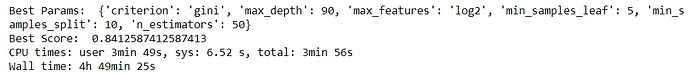
The parameter grid size will be a multiplication of all the parameters. i.e., for the following parameters in our model, the grid size will be 10\*2\*4\*5\*3\*5 = 12000

parameters ={'max\_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100],  
 'criterion' : ['gini', 'entropy'],  
 'max\_features': [0.3,0.5,0.7,0.9],  
 'min\_samples\_leaf': [3,5,7,10,15],  
 'min\_samples\_split': [2,5,10],  
 'n\_estimators': [50,100,200,400,600]}**from** sklearn.model\_selection **import** ParameterGrid  
param\_size = ParameterGrid(parameters)  
len(param\_size)**Output:  
12000**

Using sklearn’s GridSearchCV, we can search our grid over and then run the grid search.

%%time**from** sklearn.model\_selection **import** GridSearchCV  
grid\_search = RandomForestClassifier()grid\_search = GridSearchCV(  
 grid\_search,   
 parameters,   
 cv=5,  
 scoring='accuracy',n\_jobs=-1)  
  
grid\_result= grid\_search.fit(X\_train, y\_train)  
**print**('Best Params: ', grid\_result.best\_params\_)  
**print**('Best Score: ', grid\_result.best\_score\_)

## Output



Our cross-validation score is improved from 81.56% to 84.12% with the Grid search CV model compared with our baseline model. That is a 3.3% improvement. The computational time is almost 5hrs which is not feasible for a simple problem like this one.

More information about different ways of implementing a Gridsearch can be found [here](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html).

# Randomized Search

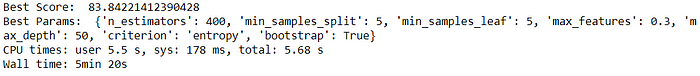
The main difference in the RandomizedSearch CV, when compared with GridCV, is that instead of trying every possible combination, this chooses the hyperparameter sample combinations randomly from grid space. Because of this reason, there is no guarantee that we will find the best result like Grid Search. But, this search can be extremely effective in practice as computational time is very less.

The computational time and model performs mainly depends on the n\_iter value. Because this value specifies how many times the model should search for parameters. If this value is high, there is a better chance of getting better accuracy, but also this comes with more computational power.

We can implement RandomizedSearchCV by using the sklearn’s library.

%%time  
**from** sklearn.model\_selection **import** RandomizedSearchCV  
random\_search=RandomizedSearchCV(estimator = RandomForestClassifier(), param\_distributions=parameters,verbose=1, n\_jobs=-1,  
 n\_iter=200)  
random\_result = random\_search.fit(X\_train, y\_train)  
print('Best Score: ', random\_result.best\_score\_\*100)  
print('Best Params: ', random\_result.best\_params\_)

## Output



Our cross-validation score is improved from 81.56% to 83.57% with the Randomized search CV model compared with our baseline model. That is a 2.5% improvement, which is 0.8% less than Grid CV. But the computational time is less than 5mins, which is almost 60 times faster. For most simple problems, this randomized search will be the most feasible option for hyperparameter tuning.

More information about different ways of implementing a randomized search can be found [here](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html).

# ****Bayesian model Optimization using HyperOpt****

To formulate the optimization problem in Hyperopt, we need an **Objective Function**thattakes in an input and returns a loss to minimize the model. And a **Domain space** for hyperparameterssimilar to grid search, we should create a parameter space with the range of input values to evaluate.

The function can be as simple as f(x) = sin(x), or it can be as complex as the error of a deep neural network. This model chooses hyperparameters based on previous steps.

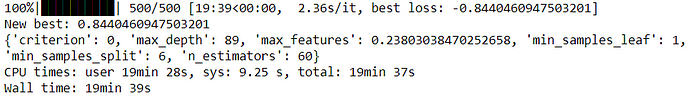
**The process of how this model works is simple:**

1. Create a surrogate probability model of the objective function.
2. Find the hyperparameters that perform best on the surrogate model.
3. Use these values on the true model to return the objective function and update the surrogate model.
4. Repeat steps 2 and 3 until maximum evaluations are reached.

To put it simply, if we want to find hyperparameters with the highest accuracy. The function to compute the accuracy is called the **objective function**.

%%time  
import **numpy** as **np**  
from **hyperopt** import **hp**, **tpe**, **fmin**,**STATUS\_OK**,**Trialsdef** accuracy\_model(params):  
 clf = RandomForestClassifier(\*\*params)  
 **return** cross\_val\_score(clf, X\_train, y\_train).mean()param\_space = {'max\_depth': hp.choice('max\_depth', range(10,100)),  
'max\_features': hp.uniform('max\_features', 0.1,1),  
'n\_estimators': hp.choice('n\_estimators', range(50,500)),  
'min\_samples\_leaf': hp.choice('min\_samples\_leaf',range(3,5)),  
'min\_samples\_split': hp.choice('min\_samples\_split',range(2,10)),  
'criterion': hp.choice('criterion', ["gini", "entropy"])}best = 0  
**def** f(params):  
 global best  
 acc = accuracy\_model(params)  
 if acc > best:  
 best = acc  
 **return** {'loss': -acc, 'status': STATUS\_OK}Trials = Trials()  
best\_params = fmin(f, param\_space , algo=tpe.suggest,max\_evals=500, trials= Trials)**print**('New best:', best, best\_params)  
**print**(best\_params)

## Output



Our cross-validation cross using Bayesian Optimization is 84.44%, which is better than Random Search and Grid Search. And the computational time is 20 min, which is reasonable considering this method performed best.

One more benefit of using the [Bayesian Optimization](https://en.wikipedia.org/wiki/Bayesian_optimization#:~:text=Bayesian%20optimization%20is%20a%20sequential,expensive%2Dto%2Devaluate%20functions.) model is unlike random search or grid search; we can track all past evaluation models used to form a probabilistic model mapping hyperparameters to a probability of a score on the objective function.

More information about installing and implementing the Hyperopt library can be found [here](https://hyperopt.github.io/hyperopt/).

## Conclusion

Hyperparameter tuning can be very advantageous to improve the accuracy of machine learning models. In our case, the random forest model is already good at predicting survival rate, so there was not much improvement in accuracy with hyperparameter tuning methods.

To conclude, using a grid search to choose optimal hyperparameters can be very time-consuming. And the random search is high-speed but not reliable. However, even these methods are inefficient than Bayesian optimization because they do not choose the next hyperparameters to evaluate based on previous results. Because of this reason, these methods consume more time evaluating useless parameters.

## Additional Resources

Apart from Random Search, Grid Search, and Bayesian Optimization, there are some advanced methods like Hyperband and BOHB which combine both HyperBand and Bayesian Optimization that are better for hyperparameter tuning. Detailed explanation about them is covered in an excellent blog by n**eptune.ai** which can be found [here](https://neptune.ai/blog/hyperband-and-bohb-understanding-state-of-the-art-hyperparameter-optimization-algorithms)

The complete data and code can be found in my [Github repository](https://github.com/JaswanthBadvelu/hypermater_tuning" \t "_blank).

I hope that’s useful! Thank you so much for reading it till here. If you have any questions regarding this article or want to connect and talk, feel free to [direct message me on LinkedIn](https://www.linkedin.com/in/jaswanth-badvelu/). I will be more than happy to chat with you and help in any way I can.