Hyperparameter Tuning in Python

Making better models, improving model performance

Black Raven (James Ng)

06 Mar 2021 · 21 min read

This is a memo to share what I have learnt in Hyperparameter Tuning (in Python), capturing the learning objectives as well as my personal notes. The course is taught by Alex Scriven from DataCamp, and it includes 4 chapters:

Chapter 1: Hyperparameters and Parameters
Chapter 2: Grid search
Chapter 3: Random Search
Chapter 4: Informed Search

Building powerful machine learning models depends heavily on the set of hyperparameters used. But with increasingly complex models with lots of options, how do you efficiently find the best settings for your particular problem?

In this course you will get practical experience in using some common methodologies for automated hyperparameter tuning in Python using Scikit Learn. These include Grid Search, Random Search & advanced optimization methodologies including Bayesian & Genetic algorithms . You will use a dataset predicting credit card defaults as you build skills to dramatically increase the efficiency and effectiveness of your machine learning model building.

Chapter 1. Hyperparameters and Parameters

In this introductory chapter you will learn the difference between hyperparameters and parameters. You will practice extracting and analyzing parameters, setting hyperparameter values for several popular machine learning algorithms. Along the way you will learn some best practice tips & tricks for choosing which hyperparameters to tune and what values to set & build learning curves to analyze your hyperparameter choices.

Introduction & 'Parameters'

Developing deeper understanding beyond the default settings is of vital importance to good model building.

What is a parameter

- Components of the model learned during the modeling process
- You do not set these manually (you are unable to, in fact)
- The algorithm will discover these for you

```
# Get the original variable names
original_variables = list(X_train.columns)
# Zip together the names and coefficients
zipped_together = list(zip(original_variables, log_reg_clf.coef_[0]))
coefs = [list(x) for x in zipped_together]
# Put into a DataFrame with column labels
coefs = pd.DataFrame(coefs, columns=["Variable", "Coefficient"])
```

```
log_reg_clf = LogisticRegression()
log_reg_clf.fit(X_train, y_train)
print(log_reg_clf.coef_)
```

```
array([[-2.88651273e-06, -8.23168511e-03,  7.50857018e-04,
         3.94375060e-04,  3.79423562e-04,  4.34612046e-04,
         4.37561467e-04,  4.12107102e-04, -6.41089138e-06,
        -4.39364494e-06,  cont... ]])
```

```
coefs.sort_values(by=["Coefficient"], axis=0, inplace=True, ascending=False)
print(coefs.head(3))
```

Variable	Coefficient
PAY_0	0.000751
PAY_5	0.000438
PAY_4	0.000435

In our data, the PAY variables relate to how many months people have previously delayed their payments. We can see that having a high number of months of delayed payments, makes someone more likely to default next month.

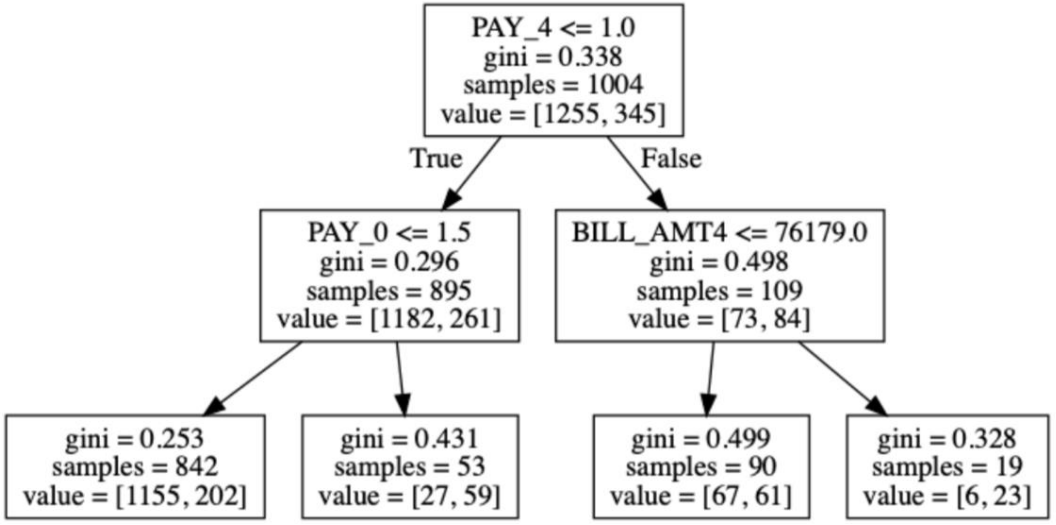
In the Scikit Learn documentation, parameters are under the ‘Attributes’ section (not ‘parameters’ section!)

Parameters in Random Forest are node decisions

```
# A simple random forest estimator
rf_clf = RandomForestClassifier(max_depth=2)
rf_clf.fit(X_train, y_train)
# Pull out one tree from the forest
chosen_tree = rf_clf.estimators_[7]
```

```
# Get the column it split on
split_column = chosen_tree.tree_.feature[1]
split_column_name = X_train.columns[split_column]
# Get the level it split on
split_value = chosen_tree.tree_.threshold[1]
print("This node split on feature {}, at a value of {}"
        .format(split_column_name, split_value))
```

> "This node split on feature PAY_0, at a value of 1.5"



Parameters in Logistic Regression

Now that you have had a chance to explore what a parameter is, let us apply this knowledge. It is important to be able to review any new algorithm and identify which elements are parameters and hyperparameters.

Which of the following is a parameter for the Scikit Learn logistic regression model? Here we mean *conceptually* based on the theory introduced in this course. **NOT** what the Scikit Learn documentation calls a parameter or attribute.

- ○ `n_jobs`
- ◉ `coef_`
- ○ `class_weight`
- ○ `LogisticRegression()`

Yes! coef_ contains the important information about coefficients on our variables in the model. We do not set this, it is learned by the algorithm through the modeling process.

Extracting a Logistic Regression parameter

You are now going to practice extracting an important parameter of the logistic regression model. The logistic regression has a few other parameters you will not explore here but you can review them in the **scikit-learn.org** documentation for the `LogisticRegression()` module under 'Attributes'.

This parameter is important for understanding the direction and magnitude of the effect the variables have on the target.

In this exercise we will extract the coefficient parameter (found in the `coef_` attribute), zip it up with the original column names, and see which variables had the largest positive effect on the target variable.

You will have available:

- A logistic regression model object named `log_reg_clf`
- The `X_train` DataFrame

`sklearn` and `pandas` have been imported for you.

```
# Create a list of original variable names from the training DataFrame
original_variables = X_train.columns

# Extract the coefficients of the logistic regression estimator
model_coefficients = log_reg_clf.coef_[0]

# Create a dataframe of the variables and coefficients & print it out
coefficient_df = pd.DataFrame({"Variable" : original_variables, "Coefficient": model_coefficients})
print(coefficient_df)

# Print out the top 3 positive variables
top_three_df = coefficient_df.sort_values(by="Coefficient", axis=0, ascending=False)[0:3]
print(top_three_df)
```

```
<script.py> output:
          Variable   Coefficient
    0     LIMIT_BAL -2.886513e-06
    1           AGE -8.231685e-03
    2         PAY_0  7.508570e-04
    3         PAY_2  3.943751e-04
    4         PAY_3  3.794236e-04
    5         PAY_4  4.346120e-04
    6         PAY_5  4.375615e-04
    7         PAY_6  4.121071e-04
    8     BILL_AMT1 -6.410891e-06
    9     BILL_AMT2 -4.393645e-06
    10    BILL_AMT3  5.147052e-06
    11    BILL_AMT4  1.476978e-05
    12    BILL_AMT5  2.644462e-06
    13    BILL_AMT6 -2.446051e-06
    14     PAY_AMT1 -5.448954e-05
    15     PAY_AMT2 -8.516338e-05
    16     PAY_AMT3 -4.732779e-05
    17     PAY_AMT4 -3.238528e-05
    18     PAY_AMT5 -3.141833e-05
    19     PAY_AMT6  2.447717e-06
    20        SEX_2 -2.240863e-04
    21  EDUCATION_1 -1.642599e-05
    22  EDUCATION_2 -1.777295e-04
    23  EDUCATION_3 -5.875596e-05
    24  EDUCATION_4 -3.681278e-06
    25  EDUCATION_5 -7.865964e-06
    26  EDUCATION_6 -9.450362e-07
    27   MARRIAGE_1 -5.036826e-05
    28   MARRIAGE_2 -2.254362e-04
    29   MARRIAGE_3  1.070545e-05
```

```
     Variable  Coefficient
2      PAY_0     0.000751
6      PAY_5     0.000438
5      PAY_4     0.000435
```

You have succesfully extracted and reviewed a very important parameter for the Logistic Regression Model. The coefficients of the model allow you to see which variables are having a larger or smaller impact on the outcome. Additionally the sign lets you know if it is a positive or negative relationship.

Extracting a Random Forest parameter

You will now translate the work previously undertaken on the logistic regression model to a random forest model. A parameter of this model is, for a given tree, how it decided to split at each level.

This analysis is not as useful as the coefficients of logistic regression as you will be unlikely to ever explore every split and every tree in a random forest model. However, it is a very useful exercise to peak under the hood at what the model is doing.

In this exercise we will extract a single tree from our random forest model, visualize it and programmatically extract one of the splits.

You have available:

- A random forest model object, `rf_clf`
- An image of the top of the chosen decision tree, `tree_viz_image`
- The `X_train` DataFrame & the `original_variables` list

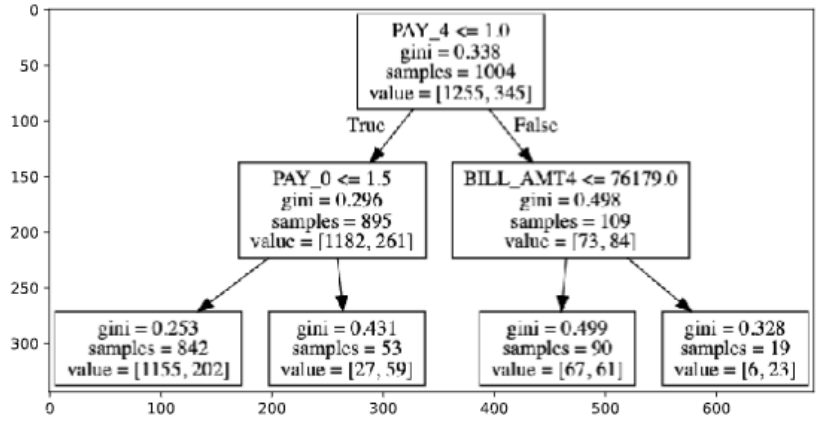
```
# Extract the 7th (index 6) tree from the random forest
chosen_tree = rf_clf.estimators_[6]

# Visualize the graph using the provided image
imgplot = plt.imshow(tree_viz_image)
plt.show()

# Extract the parameters and level of the top (index 0) node
split_column = chosen_tree.tree_.feature[0]
split_column_name = X_train.columns[split_column]
split_value = chosen_tree.tree_.threshold[0]

# Print out the feature and level
print("This node split on feature {}, at a value of {}".format(split_column_name, split_value))
```

```
<script.py> output:
    This node split on feature PAY_4, at a value of 1.0
```



You visualized and extracted some of the parameters of a random forest model.

Introducing Hyperparameters

- Something you set before the modeling process
- Like knobs on an old radio, you also ‘tune’ the hyperparameters
- The model algorithm does not learn hyperparameters

```
rf_clf = RandomForestClassifier()
print(rf_clf)
RandomForestClassifier(n_estimators='warn', criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=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_jobs=None,
            oob_score=False, random_state=None, verbose=0,bootstrap=True,
            class_weight=None, warm_start=False)
```

Refer to Scikit Learn documentation for their meanings – http://scikit-learn.org

For example `n_estimators`

Default n_estimators=10

Data Type & Default Value:

> n_estimators : integer, optional (default=10)

Definition:

> The number of trees in the forest.

```
rf_clf = RandomForestClassifier(n_estimators=100, criterion='entropy')
```

```
print(rf_clf)
RandomForestClassifier(n_estimators=100, criterion='entropy',
            max_depth=None, max_features='auto', max_leaf_nodes=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_jobs=None,
            oob_score=False, random_state=None, verbose=0,bootstrap=True,
            class_weight=None, warm_start=False)
```

Some important hyperparameters:

- `n_estimators` (high value)
- `max_features` (try different values)
- `max_depth` & `min_sample_leaf` (important for overfitting)
- (maybe) `criterion`

Some will **not** help model performance:

For the random forest classifier:

- `n_jobs`
- `random_state`
- `verbose`

Not all hyperparameters make sense to 'train'

```
log_reg_clf = LogisticRegression()
print(log_reg_clf)
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='warn',
          tol=0.0001, verbose=0, warm_start=False)
```

Hyperparameters in Random Forests

As you saw, there are many different hyperparameters available in a Random Forest model using Scikit Learn. Here you can remind yourself how to differentiate between a hyperparameter and a parameter, and easily check whether something is a hyperparameter.

You can create a random forest estimator yourself from the imported Scikit Learn package. Then print this estimator out to see the hyperparameters and their values.

Which of the following is a hyperparameter for the Scikit Learn random forest model?

- (●) `oob_score`
- ( ) `classes_`
- ( ) `trees`
- ( ) `random_level`

That's correct! `oob_score` set to `True` or `False` decides whether to use out-of-bag samples to estimate the generalization accuracy.

Exploring Random Forest Hyperparameters

Understanding what hyperparameters are available and the impact of different hyperparameters is a core skill for any data scientist. As models become more complex, there are many different settings you can set, but only some will have a large impact on your model.

You will now assess an existing random forest model (it has some bad choices for hyperparameters!) and then make better choices for a new random forest model and assess its performance.

You will have available:

- `X_train`, `X_test`, `y_train`, `y_test` DataFrames
- An existing pre-trained random forest estimator, `rf_clf_old`
- The predictions of the existing random forest estimator on the test set, `rf_old_predictions`

```
# Print out the old estimator, notice which hyperparameter is badly set
print(rf_clf_old)

# Get confusion matrix & accuracy for the old rf_model
print("Confusion Matrix: \n\n {} \n Accuracy Score: \n\n {}".format(
    confusion_matrix(y_test, rf_old_predictions),
    accuracy_score(y_test, rf_old_predictions)))
```

```
<script.py> output:
    RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                max_depth=None, max_features='auto', max_leaf_nodes=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=5, n_jobs=None,
                oob_score=False, random_state=42, verbose=0, warm_start=False)
    Confusion Matrix:

     [[276  37]
     [ 64  23]]
     Accuracy Score:

     0.7475
```

```
# Create a new random forest classifier with better hyperparamaters
rf_clf_new = RandomForestClassifier(n_estimators=500)

# Fit this to the data and obtain predictions
rf_new_predictions = rf_clf_new.fit(X_train, y_train).predict(X_test)

# Assess the new model (using new predictions!)
print("Confusion Matrix: \n\n", confusion_matrix(y_test, rf_new_predictions))
print("Accuracy Score: \n\n", accuracy_score(y_test, rf_new_predictions))
```

```
Confusion Matrix:

 [[300  13]
 [ 63  24]]
Accuracy Score:

 0.81
```

We got a nice 5% accuracy boost just from changing the `n_estimators`. You have had your first taste of hyperparameter tuning for a random forest model.

Hyperparameters of KNN

To apply the concepts learned in the prior exercise, it is good practice to try out learnings on a new algorithm. The k-nearest-neighbors algorithm is not as popular as it used to be but can still be an excellent choice for data that has groups of data that behave similarly. Could this be the case for our credit card users?

In this case you will try out several different values for one of the core hyperparameters for the knn algorithm and compare performance.

You will have available:

- `X_train`, `X_test`, `y_train`, `y_test` DataFrames

```
# Build a knn estimator for each value of n_neighbours
knn_5 = KNeighborsClassifier(n_neighbors=5)
knn_10 = KNeighborsClassifier(n_neighbors=10)
knn_20 = KNeighborsClassifier(n_neighbors=20)

# Fit each to the training data & produce predictions
knn_5_predictions = knn_5.fit(X_train, y_train).predict(X_test)
knn_10_predictions = knn_10.fit(X_train, y_train).predict(X_test)
knn_20_predictions = knn_20.fit(X_train, y_train).predict(X_test)

# Get an accuracy score for each of the models
knn_5_accuracy = accuracy_score(y_test, knn_5_predictions)
knn_10_accuracy = accuracy_score(y_test, knn_10_predictions)
knn_20_accuracy = accuracy_score(y_test, knn_20_predictions)
print("The accuracy of 5, 10, 20 neighbours was {}, {}, {}".format(knn_5_accuracy, knn_10_accuracy, knn_20_accuracy)
)
```

You succesfully tested 3 different options for 1 hyperparameter, but it was pretty exhausting. Next, we will try to find a way to make this easier.

Setting & Analyzing Hyperparameter Values

Beware of conflicting hyperparameter choices

- `LogisticRegression()` conflicting parameter options of `solver` & `penalty` that conflict.

```
The 'newton-cg', 'sag' and 'lbfgs' solvers support only l2 penalties.
```

Some aren't explicit but will just 'ignore' (from `ElasticNet` with the `normalize` hyperparameter):

```
This parameter is ignored when fit_intercept is set to False
```

Beware of setting 'silly' values for different algorithms

- Random forest with low number of trees
  - Would you consider it a 'forest' with only 2 trees?
- 1 Neighbour in KNN algorithm
  - Averaging the 'votes' of 1 person does not sound very robust!
- Increasing a hyperparameter by a very small amount
  - 1 more tree in a forest is not likely to have a large impact

```
neighbors_list = [3,5,10,20,50,75]
for test_number in neighbors_list:
    model = KNeighborsClassifier(n_neighbors=test_number)
    predictions = model.fit(X_train, y_train).predict(X_test)
    accuracy = accuracy_score(y_test, predictions)
    accuracy_list.append(accuracy)
results_df = pd.DataFrame({'neighbors':neighbors_list, 'accuracy':accuracy_list})
print(results_df)
```

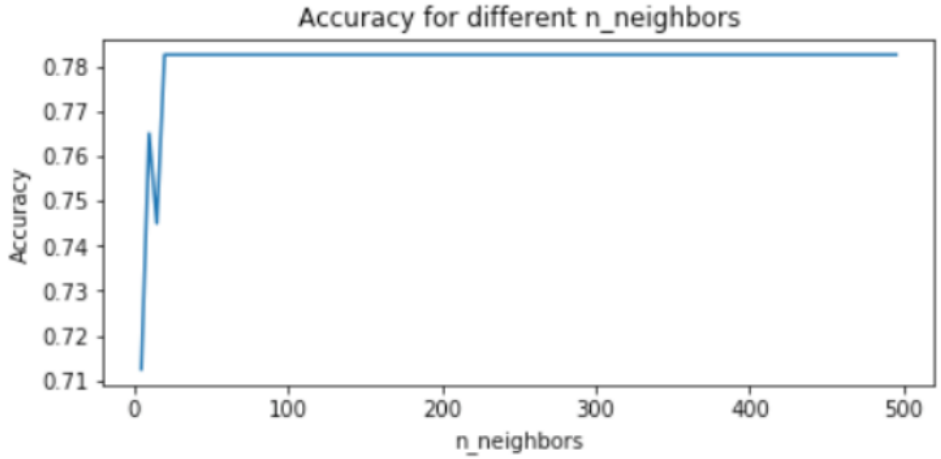
Neighbors	3	5	10	20	50	75
Accuracy	0.71	0.7125	0.765	0.7825	0.7825	0.7825

It seems that adding any more neighbours does not help beyond 20.

Learning Curves

```
neighbors_list = list(range(5,500, 5))
accuracy_list = []
for test_number in neighbors_list:
    model = KNeighborsClassifier(n_neighbors=test_number)
    predictions = model.fit(X_train, y_train).predict(X_test)
    accuracy = accuracy_score(y_test, predictions)
    accuracy_list.append(accuracy)
results_df = pd.DataFrame({'neighbors':neighbors_list, 'accuracy':accuracy_list})
```

```
plt.plot(results_df['neighbors'],
    results_df['accuracy'])
# Add the labels and title
plt.gca().set(xlabel='n_neighbors', ylabel='Accuracy',
    title='Accuracy for different n_neighbors')
plt.show()
```



Confirmed that accuracy does not improve beyond n_neighbours=20

Automating Hyperparameter Choice

Finding the best hyperparameter of interest without writing hundreds of lines of code for hundreds of models is an important efficiency gain that will greatly assist your future machine learning model building.

An important hyperparameter for the GBM algorithm is the learning rate. But which learning rate is best for this problem? By writing a loop to search through a number of possibilities, collating these and viewing them you can find the best one.

Possible learning rates to try include 0.001, 0.01, 0.05, 0.1, 0.2 and 0.5

You will have available `X_train`, `X_test`, `y_train` & `y_test` datasets, and `GradientBoostingClassifier` has been imported for you.

```
# Set the learning rates & results storage
learning_rates = [0.001, 0.01, 0.05, 0.1, 0.2, 0.5]
```

```
results_list = []

# Create the for loop to evaluate model predictions for each learning rate
for learning_rate in learning_rates:
    model = GradientBoostingClassifier(learning_rate=learning_rate)
    predictions = model.fit(X_train, y_train).predict(X_test)
    # Save the learning rate and accuracy score
    results_list.append([learning_rate, accuracy_score(y_test, predictions)])

# Gather everything into a DataFrame
results_df = pd.DataFrame(results_list, columns=['learning_rate', 'accuracy'])
print(results_df)
```

```
<script.py> output:
    learning_rate  accuracy
0           0.001    0.7825
1           0.010    0.8025
2           0.050    0.8100
3           0.100    0.7975
4           0.200    0.7900
5           0.500    0.7775
```

You efficiently tested a few different values for a single hyperparameter and can easily see which learning rate value was the best. Here, it seems that a learning rate of 0.05 yields the best accuracy.

Building Learning Curves

If we want to test many different values for a single hyperparameter it can be difficult to easily view that in the form of a DataFrame. Previously you learned about a nice trick to analyze this. A graph called a 'learning curve' can nicely demonstrate the effect of increasing or decreasing a particular hyperparameter on the final result.

Instead of testing only a few values for the learning rate, you will test many to easily see the effect of this hyperparameter across a large range of values. A useful function from NumPy is `np.linspace(start, end, num)` which allows you to create a number of values (`num`) evenly spread within an interval (`start`, `end`) that you specify.

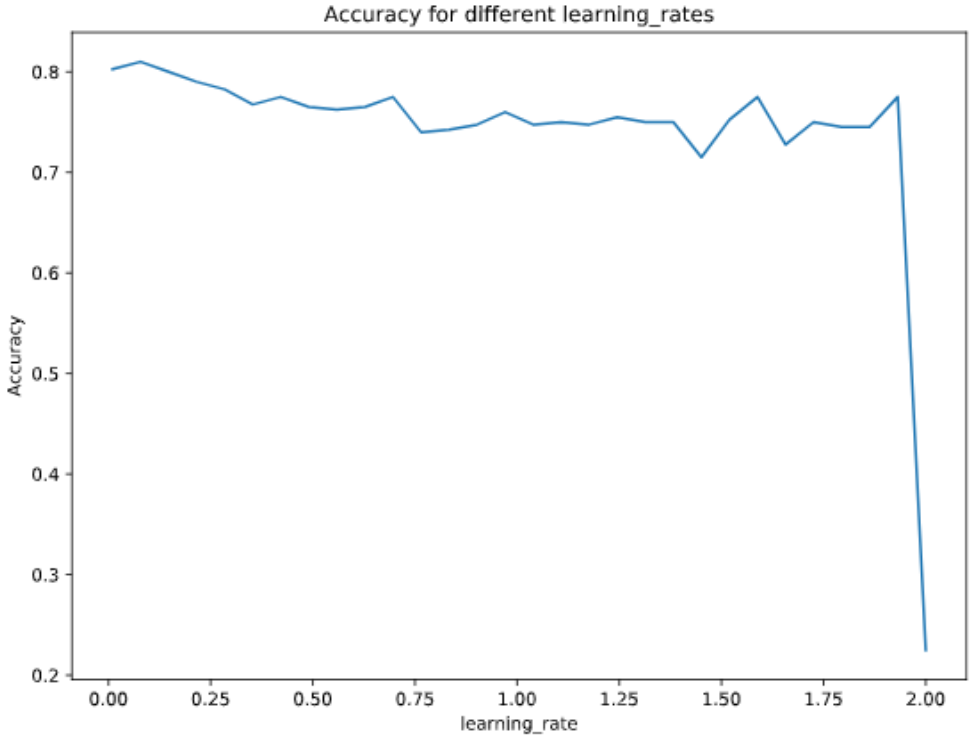
You will have available `X_train`, `X_test`, `y_train` & `y_test` datasets.

```
# Set the learning rates & accuracies list
learn_rates = np.linspace(0.01, 2, num=30)
accuracies = []

# Create the for loop
for learn_rate in learn_rates:
    # Create the model, predictions & save the accuracies as before
    model = GradientBoostingClassifier(learning_rate=learn_rate)
```

```
    predictions = model.fit(X_train, y_train).predict(X_test)
    accuracies.append(accuracy_score(y_test, predictions))

# Plot results
plt.plot(learn_rates, accuracies)
plt.gca().set(xlabel='learning_rate', ylabel='Accuracy', title='Accuracy for different learning_rates')
plt.show()
```



You can see that for low values, you get a pretty good accuracy. However once the learning rate pushes much above 1.5, the accuracy starts to drop. You have learned and practiced a useful skill for visualizing large amounts of results for a single hyperparameter.

Chapter 2. Grid search

This chapter introduces you to a popular automated hyperparameter tuning methodology called Grid Search. You will learn what it is, how it works and practice undertaking a Grid Search using Scikit Learn. You will then learn how to analyze the output of a Grid Search & gain practical experience doing this.

Introducing Grid Search

Chapter 3. Random Search

In this chapter you will be introduced to another popular automated hyperparameter tuning methodology called Random Search. You will learn what it is, how it works and importantly how it differs from grid search. You will learn some advantages and disadvantages of this method and when to choose this method compared to Grid Search. You will practice undertaking a Random Search with Scikit Learn as well as visualizing & interpreting the output.

Chapter 4. Informed Search

In this final chapter you will be given a taste of more advanced hyperparameter tuning methodologies known as "informed search". This includes a methodology known as Coarse To Fine as well as Bayesian & Genetic hyperparameter tuning algorithms. You will learn how informed search differs from uninformed search and gain practical skills with each of the mentioned methodologies, comparing and contrasting them as you go.

Course completed!

Recap topics covered:

- Feature

Next steps:

- Start with

Happy learning!