Hyperparameter Tuning in Python

Making better models, improving model performance

Black Raven (James Ng)

06 Mar 2021 · 42 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.

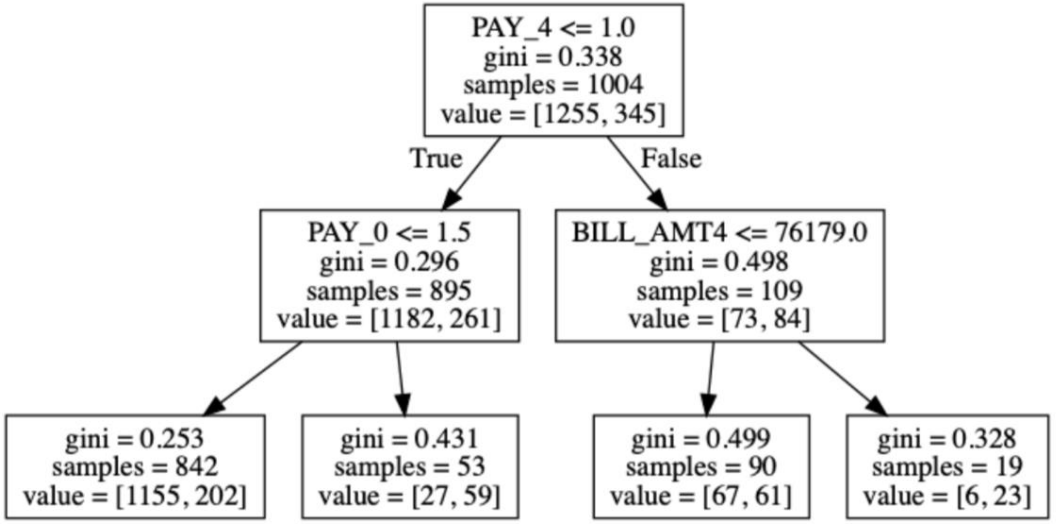
In the Scikit Learn documentation, parameters are under the ‘Attributes’ section (not ‘parameters’ section!)

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.

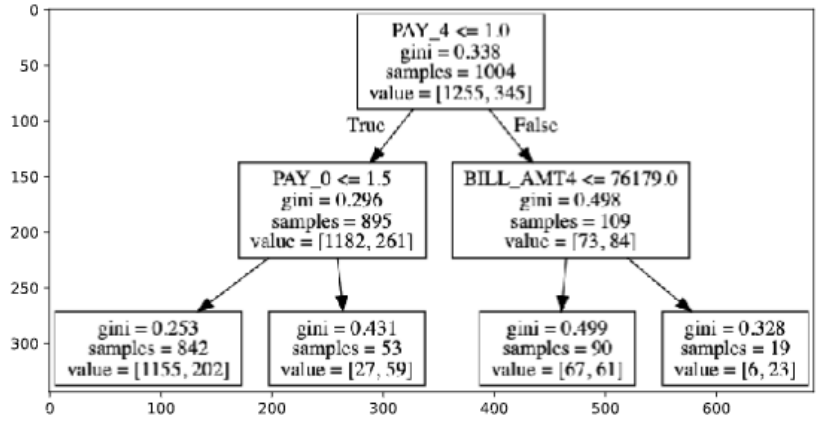
In this exercise we will extract a single tree from our random forest model, visualize it and programmatically extract one of the splits.

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)
```

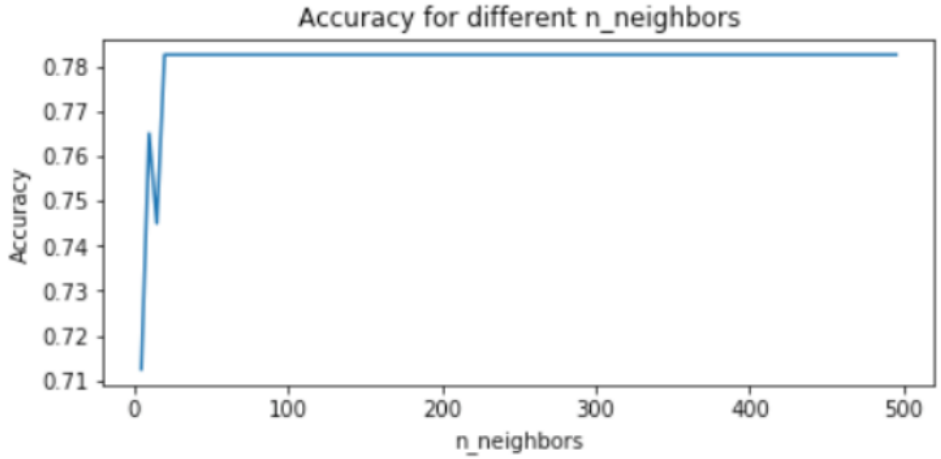
Neighbors	3	5	10	20	50	75
Accuracy	0.71	0.7125	0.765	0.7825	0.7825	0.7825

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

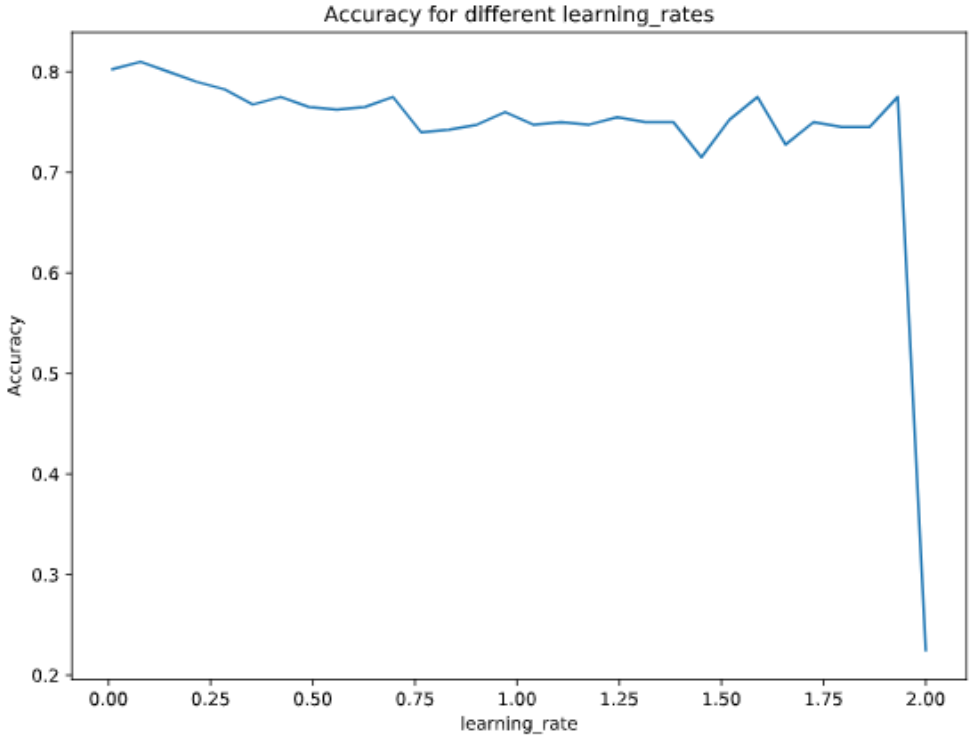
If we want to test many different values for a single hyperparameter it can be difficult to easily view that in the form of a DataFrame. Previously you learned about a nice trick to analyze this. A graph called a 'learning curve' can nicely demonstrate the effect of increasing or decreasing a particular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

Brute force approach, using for-loop

```
def gbm_grid_search(learn_rate, max_depth,subsample,max_features):
    model = GradientBoostingClassifier(
        learning_rate=learn_rate,
        max_depth=max_depth,
        subsample=subsample,
        max_features=max_features)
    predictions = model.fit(X_train, y_train).predict(X_test)
    return([learn_rate, max_depth, accuracy_score(y_test, predictions)])
results_list = []
for learn_rate in learn_rate_list:
    for max_depth in max_depth_list:
        for subsample in subsample_list:
            for max_features in max_features_list:
                results_list.append(gbm_grid_search(learn_rate,max_depth,
                                    subsample,max_features))
results_df = pd.DataFrame(results_list, columns=['learning_rate',
                          'max_depth', 'subsample', 'max_features','accuracy'])
print(results_df)
```

learning_rate	max_depth	accuracy
0.001	4	0.75
0.001	6	0.75
0.01	4	0.77
0.01	6	0.76

subsample | max_features

Grid Search approach

	learn_rate			
max_depth		0.001	0.01	0.05
	4	(4 , 0.001)	(4 , 0.01)	(4 , 0.05)
	6	(6 , 0.001)	(6 , 0.01)	(6 , 0.05)
	8	(8 , 0.001)	(8 , 0.01)	(8 , 0.05)

```
GradientBoostingClassifier(max_depth=4, learning_rate=0.001)
```

Advantages of Grid Search

- Programmatic, no need many lines of code
- Guaranteed to find the best model within the grid specified (without 'silly values)
- Easy methodology to explain

Disadvantages

- Computational expensive
- It is 'uninformed', ie, results of 1 model do not help creating the next model

Build Grid Search functions

In data science it is a great idea to try building algorithms, models and processes 'from scratch' so you can really understand what is happening at a deeper level. Of course there are great packages and libraries for this work (and we will get to that very soon!) but building from scratch will give you a great edge in your data science work.

In this exercise, you will create a function to take in 2 hyperparameters, build models and return results. You will use this function in a future exercise.

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

```
# Create the function
def gbm_grid_search(learn_rate, max_depth):

    # Create the model
```

```
    model = GradientBoostingClassifier(learning_rate=learn_rate, max_depth=max_depth)

    # Use the model to make predictions
    predictions = model.fit(X_train, y_train).predict(X_test)

    # Return the hyperparameters and score
    return([learn_rate, max_depth, accuracy_score(y_test, predictions)])
```

You now have a function you can call to test different combinations of two hyperparameters for the GBM algorithm. In the next exercise we will use it to test some values and analyze the results.

Iteratively tune multiple hyperparameters

In this exercise, you will build on the function you previously created to take in 2 hyperparameters, build a model and return the results. You will now use that to loop through some values and then extend this function and loop with another hyperparameter.

The function `gbm_grid_search(learn_rate, max_depth)` is available in this exercise. If you need to remind yourself of the function you can run the function `print_func()` that has been created for you.

```
# Create the relevant lists
results_list = []
learn_rate_list = [0.01, 0.1, 0.5]
max_depth_list = [2, 4, 6]

# Create the for loop
for learn_rate in learn_rate_list:
    for max_depth in max_depth_list:
        results_list.append(gbm_grid_search(learn_rate,max_depth))

# Print the results
print(results_list)
```

```
<script.py> output:
    [[0.01, 2, 0.78], [0.01, 4, 0.78], [0.01, 6, 0.76], [0.1, 2, 0.74], [0.1, 4, 0.76], [0.1, 6, 0.75], [
```

```
results_list = []
```

```
learn_rate_list = [0.01, 0.1, 0.5]
max_depth_list = [2,4,6]

# Extend the function input
def gbm_grid_search_extended(learn_rate, max_depth, subsample):

    # Extend the model creation section
    model = GradientBoostingClassifier(learning_rate=learn_rate, max_depth=max_depth, subsample=subsample)

    predictions = model.fit(X_train, y_train).predict(X_test)

    # Extend the return part
    return([learn_rate, max_depth, subsample, accuracy_score(y_test, predictions)])
```

```
results_list = []

# Create the new list to test
subsample_list = [0.4 , 0.6]

for learn_rate in learn_rate_list:
    for max_depth in max_depth_list:

        # Extend the for loop
        for subsample in subsample_list:

            # Extend the results to include the new hyperparameter
            results_list.append(gbm_grid_search_extended(learn_rate, max_depth, subsample))

# Print results
print(results_list)
```

```
<script.py> output:
    [[0.01, 2, 0.4, 0.73], [0.01, 2, 0.6, 0.74], [0.01, 4, 0.4, 0.73], [0.01, 4, 0.6, 0.75], [0.01, 6, 0.4, 0.72], [
```

You have effectively built your own grid search! You went from 2 to 3 hyperparameters and can see how you could extend that to even more values and hyperparameters. That was a lot of effort though. Be warned - we are now entering a world that can get very computationally expensive very fast!

How Many Models?

Adding more hyperparameters or values, you increase the amount of models created but the increases is not linear it is proportional to how many values and hyperparameters you already have.

How many models would be created when running a grid search over the following hyperparameters and values for a GBM algorithm?

- learning_rate = [0.001, 0.01, 0.05, 0.1, 0.2, 0.3, 0.5, 1, 2]
- max_depth = [4,6,8,10,12,14,16,18, 20]
- subsample = [0.4, 0.6, 0.7, 0.8, 0.9]
- max_features = ['auto', 'sqrt', 'log2']

These lists are in your console so you can utilize properties of them to help you!

- ○ 26
- ○ 9 of one model, 9 of another
- ○ 1 large model
- ◉ 1215

For every value of one hyperparameter, we test EVERY value of EVERY other hyperparameter. So you correctly multiplied the number of values (the lengths of the lists).

Grid Search with Scikit Learn

```
sklearn.model_selection.GridSearchCV(
    estimator,
    param_grid, scoring=None, fit_params=None,
    n_jobs=None, iid='warn', refit=True, cv='warn',
    verbose=0, pre_dispatch='2*n_jobs',
    error_score='raise-deprecating',
    return_train_score='warn')
```

Steps in a Grid Search:

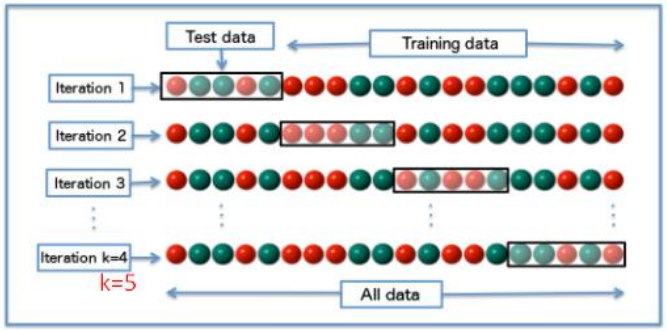
1. Select an algorithm/model (or 'estimator') to tune the hyperparameters
2. Define which hyperparameters we will tune
3. Define a range of values for each hyperparameter
4. Decide a cross validation scheme
5. Define a scoring function to determine which model is the best
6. Include extra useful information or functions

The important inputs are:

- estimator = an algorithm/model, eg. KNN, Random Forest, GBM, Logistic Regression
- param_grid = a dictionary to tell which hyperparameters (key) and list of values to test

```
param_grid = {'max_depth': [2, 4, 6, 8],
              'min_samples_leaf': [1, 2, 4, 6]}
```

- cv = how to undertake cross validation, eg k-fold of 5 or 10



- scoring = function used to evaluate model's performance, eg accuracy, roc_auc
- refit=True = fits the best hyperparameter combinations when fitting to the training data
- n_jobs = for parallel execution, allows multiple models to be created at the same time (not sequentially)
- return_train_score=True = logs statistics about the training runs that were undertaken, for plotting and understanding test vs training set performance (and hence bias-variance tradeoff)

```
# Create the grid
param_grid = {'max_depth': [2, 4, 6, 8], 'min_samples_leaf': [1, 2, 4, 6]}

#Get a base classifier with some set parameters.
rf_class = RandomForestClassifier(criterion='entropy', max_features='auto')
```

```
grid_rf_class = GridSearchCV(
    estimator = rf_class,
    param_grid = parameter_grid,
    scoring='accuracy',
    n_jobs=4,
    cv = 10,
    refit=True,
    return_train_score=True)
#Fit the object to our data
grid_rf_class.fit(X_train, y_train)

# Make predictions
grid_rf_class.predict(X_test)
```

GridSearchCV inputs

Let's test your knowledge of `GridSeachCV` inputs by answering the question below.

Three `GridSearchCV` objects are available in the console, named `model_1`, `model_2`, `model_3`. Note that there is no data available to fit these models. Instead, you must answer by looking at their construct.

Which of these `GridSearchCV` objects would not work when we try to fit it?

- `model_1` would not work when we try to fit it.
- `model_2` would not work when we try to fit it.
- (●) `model_3` would not work when we try to fit it.
- None - they will all work when we try to fit them.

```
Model #1:
 GridSearchCV(
    estimator = RandomForestClassifier(),
    param_grid = {'max_depth': [2, 4, 8, 15], 'max_features': ['auto', 'sqrt']},
    scoring='roc_auc',
    n_jobs=4,
    cv=5,
    refit=True, return_train_score=True)


Model #2:
 GridSearchCV(
    estimator = KNeighborsClassifier(),
    param_grid = {'n_neighbors': [5, 10, 20], 'algorithm': ['ball_tree', 'brute']},
    scoring='accuracy',
    n_jobs=8,
    cv=10,
    refit=False)


Model #3:
 GridSearchCV(
    estimator = GradientBoostingClassifier(),
    param_grid = {'number_attempts': [2, 4, 6], 'max_depth': [3, 6, 9, 12]},
    scoring='accuracy',
    n_jobs=2,
    cv=7,
    refit=True)
```

By looking at the Scikit Learn documentation (or your excellent memory!) you know that `number_attempts` is not a valid hyperparameter. This `GridSearchCV` will not fit to our data.

GridSearchCV with Scikit Learn

The `GridSearchCV` module from Scikit Learn provides many useful features to assist with efficiently undertaking a grid search. You will now put your learning into practice by creating a `GridSearchCV` object with certain parameters.

The desired options are:

- A Random Forest Estimator, with the split criterion as 'entropy'
- 5-fold cross validation
- The hyperparameters `max_depth` (2, 4, 8, 15) and `max_features` ('auto' vs 'sqrt')
- Use `roc_auc` to score the models
- Use 4 cores for processing in parallel
- Ensure you refit the best model and return training scores

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

```
# Create a Random Forest Classifier with specified criterion
rf_class = RandomForestClassifier(criterion='entropy')

# Create the parameter grid
param_grid = {'max_depth': [2, 4, 8, 15], 'max_features': ['auto', 'sqrt']}

# Create a GridSearchCV object
grid_rf_class=GridSearchCV(
    estimator=rf_class,
    param_grid=param_grid,
    scoring='roc_auc',
    n_jobs=4,
    cv=5,
    refit=True, return_train_score=True)
print(grid_rf_class)
```

You now understand all the inputs to a `GridSearchCV` object and can tune many different hyperparameters and many different values for each on a chosen algorithm!

Understanding a grid search output

Three different groups for the GridSearchCV properties:

- A results log
  - `cv_results_`
- The best results
  - `best_index_` , `best_params_` & `best_score_`
- 'Extra information'
  - `scorer_` , `n_splits_` & `refit_time_`

```
cv_results_df = pd.DataFrame(grid_rf_class.cv_results_)


print(cv_results_df.shape)
```

```
(12, 23)
```

- The 12 rows for the 12 squares in our grid or 12 models we ran

For the 23 columns:

The `time` columns refer to the time it took to fit (and score) the model during 5-fold cross validation.

	mean_fit_time	std_fit_time	mean_score_time	std_score_time
0	0.321069	0.007236	0.015008	0.000871
1	0.678216	0.066385	0.034155	0.003767
2	0.939865	0.009502	0.055868	0.004148
3	0.296547	0.006261	0.017990	0.002803
4	0.686065	0.016163	0.040048	0.001304
5	1.097201	0.006327	0.057136	0.004468
6	0.416973	0.085533	0.021157	0.003901
7	0.788864	0.021954	0.042638	0.004802
8	1.198466	0.054694	0.049674	0.006884
9	0.398824	0.027500	0.025307	0.009473
10	0.719588	0.019231	0.035629	0.005712
11	0.847477	0.036584	0.029104	0.005220

The `param_` columns store the parameters it tested on that row, one column per parameter. Each row in this DataFrame is about one model.

param_max_depth	param_min_samples_leaf	param_n_estimators
10	1	100
10	1	200
10	2	100
10	2	200
10	2	300

The `params` column contains dictionary of all the parameters.

```
pd.set_option("display.max_colwidth", -1)
print(cv_results_df.loc[:, "params"])
```

params
{'max_depth': 10, 'min_samples_leaf': 1, 'n_estimators': 100}
{'max_depth': 10, 'min_samples_leaf': 1, 'n_estimators': 200}
{'max_depth': 10, 'min_samples_leaf': 2, 'n_estimators': 100}
{'max_depth': 10, 'min_samples_leaf': 2, 'n_estimators': 200}
{'max_depth': 10, 'min_samples_leaf': 2, 'n_estimators': 300}

The test_score columns contain the scores on our test set for each of our cross-folds as well as some summary statistics.

split0_test_score	split1_test_score	...	mean_test_score	std_test_score
0.72820401	0.7859811	...	0.76010401	0.02995142
0.73539669	0.7963085	...	0.76590708	0.02721413
0.72929381	0.78686003	...	0.7718143	0.02775648
0.72820401	0.78554164	...	0.77044862	0.02794597
0.72885789	0.78795869	...	0.77122424	0.03288053

The `rank` column, ordering the `mean_test_score` from best to worst.
For example, the model in the third row had the best mean_test_score.

rank_test_score
9
4
1
3
2

```
best_row = cv_results_df[cv_results_df["rank_test_score"] == 1]
print(best_row)
```

mean_fit_time	...	params	...	mean_test_score	rank_test_score
0.97765441	...	{'max_depth': 10, 'min_samples_leaf': 2, 'n_estimators': 200}	...	0.7718143	1

This table is the row from the cv_results object that was the best model created.

Information on the best grid square is found in three different properties

- `best_params_` , the dictionary of parameters that gave the best score.
- `best_score_` , the actual best score.
- `best_index` , the row in our `cv_results_.rank_test_score` that was the best.

GridSearchCV stores an estimator built with the best hyperparameters in the `best_estimator` property.

```
print(grid_rf_class.best_estimator_)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
            max_depth=10, 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=300, n_jobs=None,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

Some extra information is available in the following properties:

- `scorer_`

What scorer function was used on the held out data. (we set it to AUC)

- `n_splits_`

How many cross-validation splits. (We set to 5)

- `refit_time_`

The number of seconds used for refitting the best model on the whole dataset.

Using the best outputs

Which of the following parameters must be set in order to be able to directly use the `best_estimator_` property for predictions?

- ( ) `return_train_score = True`
- (•) `refit = True`

  When we set this to true, the creation of the grid search object automatically refits the best parameters on the whole training set and creates the `best_estimator_` property.
- ( ) `refit = False`
- ( ) `verbose = 1`

Exploring the grid search results

You will now explore the `cv_results_` property of the GridSearchCV object defined in the video. This is a dictionary that we can read into a pandas DataFrame and contains a lot of useful information about the grid search we just undertook.

A reminder of the different column types in this property:

- `time_` columns
- `param_` columns (one for each hyperparameter) and **the** singular `params` column (with all hyperparameter settings)
- a `train_score` column for each cv fold including the `mean_train_score` and `std_train_score` columns
- a `test_score` column for each cv fold including the `mean_test_score` and `std_test_score` columns
- a `rank_test_score` column with a number from 1 to n (number of iterations) ranking the rows based on their `mean_test_score`

```
# Read the cv_results property into a dataframe & print it out
cv_results_df = pd.DataFrame(grid_rf_class.cv_results_)
print(cv_results_df)

# Extract and print the column with a dictionary of hyperparameters used
column = cv_results_df.loc[:, ["params"]]
print(column)

# Extract and print the row that had the best mean test score
best_row = cv_results_df[cv_results_df["rank_test_score"] == 1]
print(best_row)
```

```
<script.py> output:
    mean_fit_time  std_fit_time  mean_score_time  std_score_time param_max_depth param_min_samples_leaf param_n_estimators                                                 params  e  std_train_score
0        0.324582      0.004639         0.020115        0.003147              10                     1                100  {'max_depth': 10, 'min_samples_leaf': 1, 'n_es...  8     1.248173e-03
1        0.671539      0.018667         0.039659        0.006038              10                     1                200  {'max_depth': 10, 'min_samples_leaf': 1, 'n_es...  3     1.319340e-03
2        0.977654      0.011564         0.054303        0.003766              10                     1                300  {'max_depth': 10, 'min_samples_leaf': 1, 'n_es...  8     1.311001e-03
3        0.313844      0.008520         0.017680        0.002485              10                     2                100  {'max_depth': 10, 'min_samples_leaf': 2, 'n_es...  6     3.382634e-03
4        0.645001      0.003995         0.034633        0.004523              10                     2                200  {'max_depth': 10, 'min_samples_leaf': 2, 'n_es...  2     2.134008e-03
5        0.967691      0.010192         0.057124        0.003817              10                     2                300  {'max_depth': 10, 'min_samples_leaf': 2, 'n_es...  9     2.131063e-03
6        0.342355      0.002576         0.020001        0.002485              20                     1                100  {'max_depth': 20, 'min_samples_leaf': 1, 'n_es...  0     4.965068e-17
7        0.724901      0.003929         0.034221        0.002700              20                     1                200  {'max_depth': 20, 'min_samples_leaf': 1, 'n_es...  0     4.965068e-17
8        1.078176      0.013817         0.056005        0.002604              20                     1                300  {'max_depth': 20, 'min_samples_leaf': 1, 'n_es...  0     0.000000e+00
9        0.351497      0.010176         0.020151        0.004323              20                     2                100  {'max_depth': 20, 'min_samples_leaf': 2, 'n_es...  8     7.821910e-04
10       0.695131      0.018459         0.040476        0.001271              20                     2                200  {'max_depth': 20, 'min_samples_leaf': 2, 'n_es...  2     3.932301e-04
11       0.792999      0.021011         0.030246        0.003575              20                     2                300  {'max_depth': 20, 'min_samples_leaf': 2, 'n_es...  4     7.443455e-04

[12 rows x 23 columns]
```

```
                                               params
0   {'max_depth': 10, 'min_samples_leaf': 1, 'n_es...
1   {'max_depth': 10, 'min_samples_leaf': 1, 'n_es...
2   {'max_depth': 10, 'min_samples_leaf': 1, 'n_es...
3   {'max_depth': 10, 'min_samples_leaf': 2, 'n_es...
4   {'max_depth': 10, 'min_samples_leaf': 2, 'n_es...
5   {'max_depth': 10, 'min_samples_leaf': 2, 'n_es...
6   {'max_depth': 20, 'min_samples_leaf': 1, 'n_es...
7   {'max_depth': 20, 'min_samples_leaf': 1, 'n_es...
8   {'max_depth': 20, 'min_samples_leaf': 1, 'n_es...
9   {'max_depth': 20, 'min_samples_leaf': 2, 'n_es...
10  {'max_depth': 20, 'min_samples_leaf': 2, 'n_es...
11  {'max_depth': 20, 'min_samples_leaf': 2, 'n_es...
```

```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time param_max_depth param_min_samples_leaf param_n_estimators                                               params
2       0.977654      0.011564         0.054303        0.003766              10                      1                300  {'max_depth': 10, 'min_samples_leaf': 1, 'n_es...

[1 rows x 23 columns]
```

You have built invaluable skills in looking 'under the hood' at what your grid search is doing by extracting and analysing the `cv_results_` property.

Analyzing the best results

At the end of the day, we primarily care about the best performing 'square' in a grid search. Luckily Scikit Learn's `gridSearchCv` objects have a number of parameters that provide key information on just the best square (or row in `cv_results_`).

Three properties you will explore are:

- `best_score_` – The score (here ROC_AUC) from the best-performing square.
- `best_index_` – The index of the row in `cv_results_` containing information on the best-performing square.
- `best_params_` – A dictionary of the parameters that gave the best score, for example `'max_depth': 10`

The grid search object `grid_rf_class` is available.

A dataframe (`cv_results_df`) has been created from the `cv_results_` for you on line 6. This will help you index into the results.

```
# Print out the ROC_AUC score from the best-performing square
best_score = grid_rf_class.best_score_
print(best_score)
```

```
# Create a variable from the row related to the best-performing square
cv_results_df = pd.DataFrame(grid_rf_class.cv_results_)
best_row = cv_results_df.loc[[grid_rf_class.best_index_]]
print(best_row)

# Get the n_estimators parameter from the best-performing square and print
best_n_estimators = grid_rf_class.best_params_["n_estimators"]
print(best_n_estimators)
```

```
<script.py> output:
    0.7718143027468853
       mean_fit_time  std_fit_time  mean_score_time  std_score_time param_max_depth param_min_samples_leaf param_n_estimators
    2       0.977654      0.011564         0.054303        0.003766              10                     1               300  {'max_depth': 10, 'min_samples_leaf': 1,

    [1 rows x 23 columns]
    300
```

Being able to quickly find and prioritize the huge volume of information given back from machine learning modeling output is a great skill. Here you had great practice doing that with `cv_results_` by quickly isolating the key information on the best performing square. This will be very important when your grids grow from 12 squares to many more!

Using the best results

While it is interesting to analyze the results of our grid search, our final goal is practical in nature; we want to make predictions on our test set using our estimator object.

We can access this object through the `best_estimator_` property of our grid search object.

Let's take a look inside the `best_estimator_` property, make predictions, and generate evaluation scores. We will firstly use the default `predict` (giving class predictions), but then we will need to use `predict_proba` rather than `predict` to generate the roc-auc score as roc-auc needs probability scores for its calculation. We use a slice `[:,1]` to get probabilities of the positive class.

You have available the `X_test` and `y_test` datasets to use and the `grid_rf_class` object from previous exercises.

```
# See what type of object the best_estimator_ property is
print(type(grid_rf_class.best_estimator_))

# Create an array of predictions directly using the best_estimator_ property
```

```
predictions = grid_rf_class.best_estimator_.predict(X_test)

# Take a look to confirm it worked, this should be an array of 1's and 0's
print(predictions[0:5])

# Now create a confusion matrix
print("Confusion Matrix \n", confusion_matrix(y_test, predictions))

# Get the ROC-AUC score
predictions_proba = grid_rf_class.best_estimator_.predict_proba(X_test)[:,1]
print("ROC-AUC Score \n", roc_auc_score(y_test, predictions_proba))
```

```
<script.py> output:
    <class 'sklearn.ensemble.forest.RandomForestClassifier'>
    [0 0 0 0 1]
    Confusion Matrix
     [[140   8]
     [ 36  16]]
    ROC-AUC Score
     0.7436330561330562
```

The `.best_estimator_` property is a really powerful property to understand for streamlining your machine learning model building process. You now can run a grid search and seamlessly use the best model from that search to make predictions. Piece of cake!

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.

Introducing Random Search

Very similar to grid search:

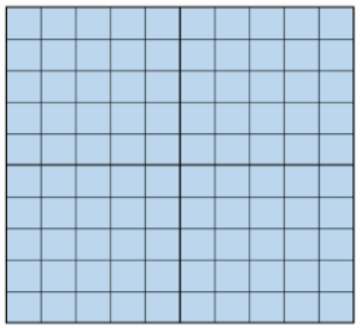
- define an estimator, which hyperparameters to tune and the range of values for each hyperparameter
- we still set a cross-validation scheme and scoring function

BUT we instead randomly select grid squares, ie, randomly sample N combinations.

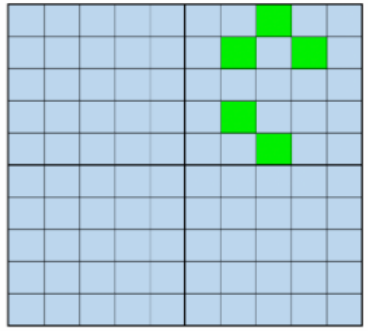
Why randomly chosen trials are more efficient for hyperparameter optimisation:

- Not every hyperparameter is as important
- A little trick of probability

A grid search:



Our best models:



How many models must we run to have a 95% chance of getting one of the green squares?

If we randomly select hyperparameter combinations uniformly, let's consider the chance of MISSING every single trial, to show how unlikely that is

- Trial 1 = 0.05 chance of success and (1 - 0.05) of missing
  - Trial 2 = (1-0.05) x (1-0.05) of missing the range
    - Trial 3 = (1-0.05) x (1-0.05) x (1-0.05) of missing again
- In fact, with n trials we have (1-0.05)^n chance that every single trial misses that desired spot.

So how many trials to have a high (95%) chance of getting **in** that region?

- We have (1-0.05)^n chance to miss everything.
- So we must have (1- miss everything) chance to get in there or (1-(1-0.05)^n)
- Solving 1-(1-0.05)^n >= 0.95 gives us **n >= 59**

Conclusion:
With relatively few trials we can get close to our maximum score with a relatively high probability. In essence, it is very unlikely that you will continue to miss everything for a long time.
A Grid Search may spend lots of time in a 'bad area' as it covers exhaustively.

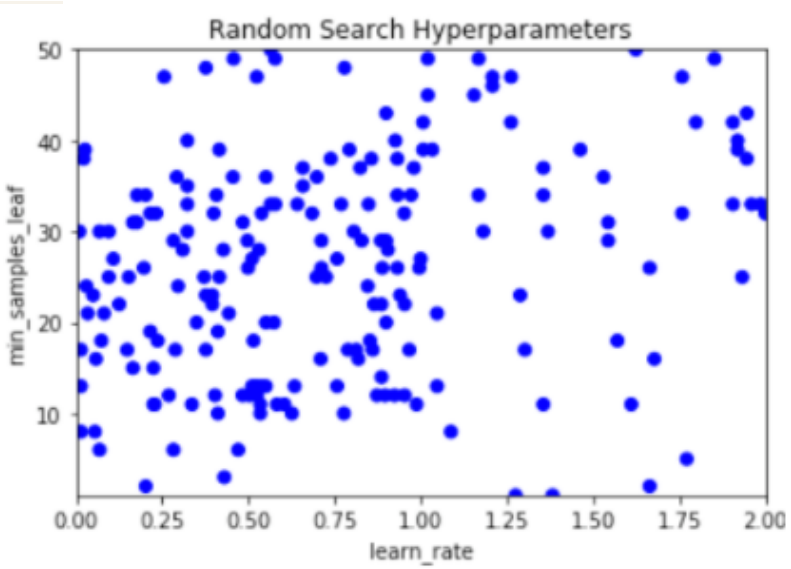
Remember:

1. The maximum is still only as good as the grid you set!
2. Remember to fairly compare this to grid search, you need to have the same modeling 'budget'

```
# Set some hyperparameter lists
learn_rate_list = np.linspace(0.001,2,150)
min_samples_leaf_list = list(range(1,51))
```

```
# Create list of combinations
from itertools import product
combinations_list = [list(x) for x in
                     product(learn_rate_list, min_samples_leaf_list)]
```

```
# Select 100 models from our larger set
random_combinations_index = np.random.choice(
                            range(0,len(combinations_list)), 100,
                            replace=False)
combinations_random_chosen = [combinations_list[x] for x in
                            random_combinations_index]
```



Randomly Sample Hyperparameters

To undertake a random search, we firstly need to undertake a random sampling of our hyperparameter space.

In this exercise, you will firstly create some lists of hyperparameters that can be zipped up to a list of lists. Then you will randomly sample hyperparameter combinations preparation for running a random search.

You will use just the hyperparameters `learning_rate` and `min_samples_leaf` of the GBM algorithm to keep the example illustrative and not overly complicated.

```
# Create a list of values for the learning_rate hyperparameter
learn_rate_list = list(np.linspace(0.01,1.5,200))

# Create a list of values for the min_samples_leaf hyperparameter
min_samples_list = list(range(10,41))

# Combination list
combinations_list = [list(x) for x in product(learn_rate_list, min_samples_list)]

# Sample hyperparameter combinations for a random search.
```

```
random_combinations_index = np.random.choice(range(0, len(combinations_list)), 250, replace=False)
combinations_random_chosen = [combinations_list[x] for x in random_combinations_index]

# Print the result
print(combinations_random_chosen)
```

```
<script.py> output:
    [[1.305326633165829, 14], [0.6015075376884422, 24], [0.6089949748743718, 35], [1.3128140703517588, 33], [0.3244723618090452, 23], [0.07738693467
```

You generated some hyperparameter combinations and randomly sampled in that space. The output was not too nice though, in the next lesson we will use a much more efficient method for this. In a future lesson we will also make this output look much nicer!

Randomly Search with Random Forest

To solidify your knowledge of random sampling, let's try a similar exercise but using different hyperparameters and a different algorithm.

As before, create some lists of hyperparameters that can be zipped up to a list of lists. You will use the hyperparameters `criterion`, `max_depth` and `max_features` of the random forest algorithm. Then you will randomly sample hyperparameter combinations in preparation for running a random search.

You will use a slightly different package for sampling in this task, `random.sample()`.

```
# Create lists for criterion and max_features
criterion_list = ["gini", "entropy"]
max_feature_list = ["auto", "sqrt", "log2", None]

# Create a list of values for the max_depth hyperparameter
max_depth_list = list(range(3,56))

# Combination list
combinations_list = [list(x) for x in product(criterion_list, max_feature_list, max_depth_list)]

# Sample hyperparameter combinations for a random search
combinations_random_chosen = random.sample(combinations_list, 150)

# Print the result
```

```
print(combinations_random_chosen)
```

```
<script.py> output:
    [['gini', 'log2', 10], ['entropy', 'auto', 30], ['gini', 'auto', 46], ['gini', 'auto', 32], ['gini', None, 55], ['gini', 'auto', 14], ['entropy',
```

This one was a bit harder but you managed to sample using text options and learned a new function to sample your lists.

Visualizing a Random Search

Visualizing the search space of random search allows you to easily see the coverage of this technique and therefore allows you to see the effect of your sampling on the search space.

In this exercise you will use several different samples of hyperparameter combinations and produce visualizations of the search space.

The function `sample_and_visualize_hyperparameters()` takes a single argument (number of combinations to sample) and then randomly samples hyperparameter combinations, just like you did in the last exercise! The function will then visualize the combinations.

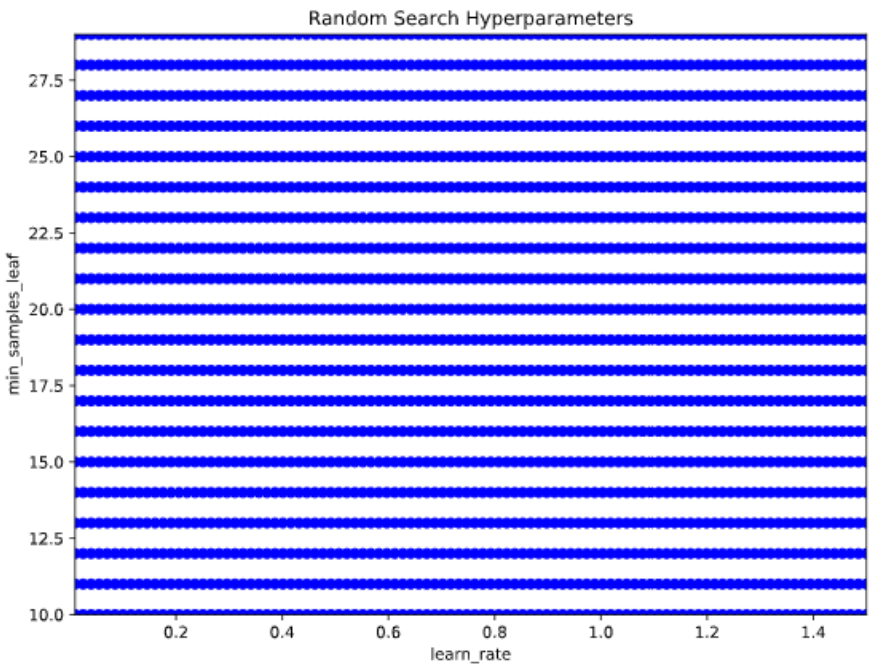
If you want to see the function definition, you can use Python's handy `inspect` library, like so:

`print(inspect.getsource(sample_and_visualize_hyperparameters))`

```
# Confirm how many hyperparameter combinations & print
number_combs = len(combinations_list)
print(number_combs)

# Sample and visualise specified combinations
for x in [50, 500, 1500]:
    sample_and_visualize_hyperparameters(x)

# Sample all the hyperparameter combinations & visualise
sample_and_visualize_hyperparameters(number_combs)
```

```
<script.py> output:
    2000
```

Notice how the bigger your sample space of a random search the more it looks like a grid search? In a later lesson we will look closer at comparing these two methods side by side.



Random Search in Scikit Learn

Steps in a Random Search (1-6 are the same as Grid Search):

1. Select an algorithm/model (or 'estimator') to tune the hyperparameters
2. Define which hyperparameters we will tune
3. Define a range of values for each hyperparameter
4. Decide a cross validation scheme
5. Define a scoring function to determine which model is the best
6. Include extra useful information or functions
7. Decide how many samples to take, ie, how many hyperparameter combinations we will randomly sample to build models and then undertake this sampling before we model

GridSearchCV:

```
sklearn.model_selection.GridSearchCV(estimator, param_grid,
        scoring=None, fit_params=None,
        n_jobs=None,
        iid='warn',
        refit=True, cv='warn', verbose=0,
        pre_dispatch='2*n_jobs',
        error_score='raise-deprecating',
        return_train_score='warn')
```

RandomizedSearchCV:

```
sklearn.model_selection.RandomizedSearchCV(estimator,
        param_distributions, n_iter=10,
        scoring=None, fit_params=None,
        n_jobs=None, iid='warn', refit=True,
        cv='warn', verbose=0,
        pre_dispatch='2*n_jobs',
        random_state=None,
        error_score='raise-deprecating',
        return_train_score='warn')
```

Two key differences:

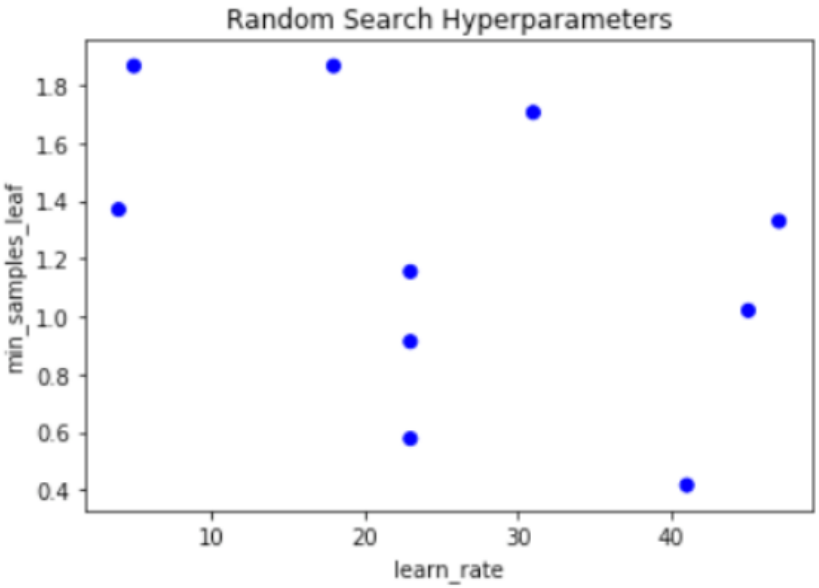
- `n_iter` which is the number of samples for the random search to take from your grid. In the previous example you did 300.
- `param_distributions` is slightly different from `param_grid`, allowing optional ability to set a distribution for sampling.
  - The default is all combinations have equal chance to be chosen.

```
# Set up the sample space
learn_rate_list = np.linspace(0.001,2,150)
min_samples_leaf_list = list(range(1,51))

# Create the grid
parameter_grid = {
    'learning_rate' : learn_rate_list,
    'min_samples_leaf' : min_samples_leaf_list}

# Define how many samples
number_models = 10
```

```
# Create a random search object
random_GBM_class = RandomizedSearchCV(
    estimator = GradientBoostingClassifier(),
    param_distributions = parameter_grid,
    n_iter = number_models,
    scoring='accuracy',
    n_jobs=4,
    cv = 10,
    refit=True,
    return_train_score = True)
# Fit the object to our data
random_GBM_class.fit(X_train, y_train)
```

```
rand_x = list(random_GBM_class.cv_results_['param_learning_rate'])
rand_y = list(random_GBM_class.cv_results_['param_min_samples_leaf'])
```

```
# Make sure we set the limits of Y and X appriately
x_lims = [np.min(learn_rate_list), np.max(learn_rate_list)]
y_lims = [np.min(min_samples_leaf_list), np.max(min_samples_leaf_list)]
# Plot grid results
plt.scatter(rand_y, rand_x, c=['blue']*10)
plt.gca().set(xlabel='learn_rate', ylabel='min_samples_leaf',
                title='Random Search Hyperparameters')
plt.show()
```



RandomSearchCV inputs

Let's test your knowledge of how `RandomizedSearchCV` differs from `GridSearchCV`. You can check the documentation on Scitkit Learn's website to compare these two functions. Which of these parameters is *only* for a `RandomizedSearchCV`?

- ( ) `param_grid`
- ( ) `n_jobs`
- ( ) `best_estimator_`
- (•) `n_iter`

RandomizedSearchCV asks you for how many models to sample from the grid you set.

The RandomizedSearchCV Object

Just like the `GridSearchCV` library from Scikit Learn, `RandomizedSearchCV` provides many useful features to assist with efficiently undertaking a random search. You're going to create a `RandomizedSearchCV` object, making the small adjustment needed from the `GridSearchCV` object.

The desired options are:

- A default Gradient Boosting Classifier Estimator
- 5-fold cross validation
- Use accuracy to score the models
- Use 4 cores for processing in parallel
- Ensure you refit the best model and return training scores
- Randomly sample 10 models

The hyperparameter grid should be for `learning_rate` (150 values between 0.1 and 2) and `min_samples_leaf` (all values between and including 20 and 64).

You will have available `X_train` & `y_train` datasets.

```
# Create the parameter grid
```

```
param_grid = {'learning_rate': np.linspace(0.1, 2, 150), 'min_samples_leaf': list(range(20, 65))}

# Create a random search object
random_GBM_class = RandomizedSearchCV(
    estimator = GradientBoostingClassifier(),
    param_distributions = param_grid,
    n_iter = 10,
    scoring='accuracy', n_jobs=4, cv = 5, refit=True, return_train_score = True)

# Fit to the training data
random_GBM_class.fit(X_train, y_train)

# Print the values used for both hyperparameters
print(random_GBM_class.cv_results_['param_learning_rate'])
print(random_GBM_class.cv_results_['param_min_samples_leaf'])
```

```
<script.py> output:
    [1.1073825503355705 1.0691275167785235 0.4697986577181208
     1.2476510067114095 1.5664429530201343 1.7577181208053692
     1.859731543624161 1.5791946308724834 0.5463087248322147
     1.7577181208053692]
    [47 54 61 30 63 32 60 43 38 27]
```

You have succesfully taken the knowledge gained from the grid search section and adjusted it to be able to run a random search. This is a very valuable tool to add to your Machine Learning Toolkit!

RandomSearchCV in Scikit Learn

Let's practice building a RandomizedSearchCV object using Scikit Learn.

The hyperparameter grid should be for `max_depth` (all values between and including 5 and 25) and `max_features` ('auto' and 'sqrt').

The desired options for the RandomizedSearchCV object are:

- A RandomForestClassifier Estimator with `n_estimators` of 80.
- 3-fold cross validation (`cv`)

- Use `roc_auc` to **score** the models
- Use 4 cores for processing in parallel (`n_jobs`)
- Ensure you refit the best model and return training scores
- Only sample 5 models for efficiency (`n_iter`)

`X_train` & `y_train` datasets are loaded for you.

Remember, to extract the chosen hyperparameters these are found in `cv_results_` with a column per hyperparameter. For example, the column for the hyperparameter `criterion` would be `param_criterion`.

```
# Create the parameter grid
param_grid = {'max_depth': list(range(5,26)), 'max_features': ['auto' , 'sqrt']}

# Create a random search object
random_rf_class = RandomizedSearchCV(
    estimator = RandomForestClassifier(n_estimators=80),
    param_distributions = param_grid, n_iter = 5,
    scoring='roc_auc', n_jobs=4, cv = 3, refit=True, return_train_score = True)

# Fit to the training data
random_rf_class.fit(X_train, y_train)

# Print the values used for both hyperparameters
print(random_rf_class.cv_results_['param_max_depth'])
print(random_rf_class.cv_results_['param_max_features'])
```

```
<script.py> output:
    [18 11 10 22 10]
    ['sqrt' 'auto' 'sqrt' 'sqrt' 'auto']
```

You adapted your knowledge to a new algorithm and set of hyperparameters and values. Being able to transpose your knowledge to new situations is an invaluable skill - excellent!

Comparing Grid and Random Search

Similarities

- Both are automated ways of tuning different hyperparameters
- For both you set the grid to sample from, ie, which hyperparameters and values for each grid)
- For both you set a cross validation scheme and scoring function

Differences

Grid Search:

- Exhaustively tries all combinations within the sample space
- No Sampling methodology
- More computationally expensive
- Guaranteed to find the best score in the sample space

Random Search:

- Randomly selects a subset of combinations within the sample space (that you must specify)
- Can select a sampling methodology (other than uniform which is default)
- Less computationally expensive
- Not guaranteed to find the best score in the sample space (but likely to find a *good* one *faster*)

Which should I use?

- How much data do you have?
  - More data means random search may be better option.
- How many hyperparameters and values do you want to tune?
  - More of these means random search may be a better option.
- How much resources do you have? (Time, computing power)
  - Less resources means random search may be a better option.

Comparing Random & Grid Search

In the video, you just studied some of the advantages and disadvantages of random search as compared to grid search. Which of the following is an advantage of random search?

- ○ It exhaustively searches all possible hyperparameter combinations, so is guaranteed to find the best model within the specified grid.
- ○ It doesn't matter what grid you sample from, it will still find the best model.
- ○ There are no advantages, it is worse than Grid Search.
- ◉ It is more computationally efficient than Grid Search.

As you saw in the slides, random search tests a larger space of values so is more likely to get close to the best score, given the same computational resources as Grid Search.

Grid and Random Search Side by Side

Visualizing the search space of random and grid search together allows you to easily see the coverage that each technique has and therefore brings to life their specific advantages and disadvantages.

In this exercise, you will sample hyperparameter combinations in a grid search way as well as a random search way, then plot these to see the difference.

You will have available:

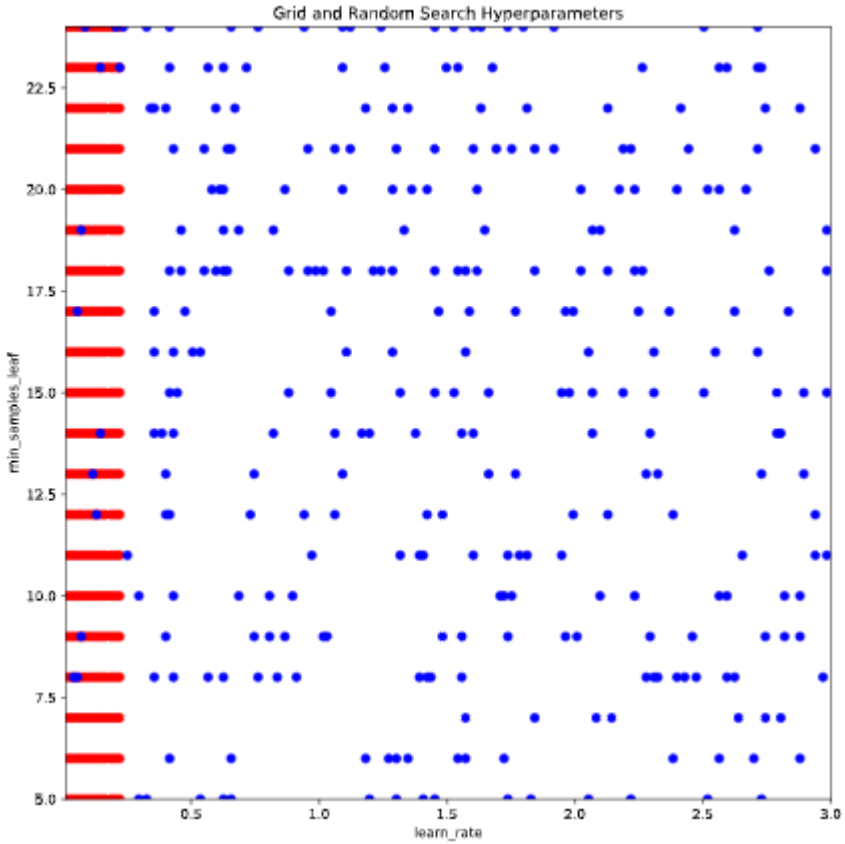
- `combinations_list` which is a list of combinations of `learn_rate` and `min_samples_leaf` for this algorithm
- The function `visualize_search()` which will make your hyperparameter combinations into X and Y coordinates and plot both grid and random search combinations on the same graph. It takes as input two lists of hyperparameter combinations.

```
# Sample grid coordinates
grid_combinations_chosen = combinations_list[0:300]

# Print result
print(grid_combinations_chosen)
```

```
<script.py> output:
    [[0.01, 5], [0.01, 6], [0.01, 7], [0.01, 8], [0.01, 9], [0.01, 10], [0.01, 11], [0.01, 12], [0.01, 13], [0.01, 14], [0.01, 15], [0.01, 16], [
```

```
# Create a list of sample indexes
```

```
sample_indexes = list(range(0,len(combinations_list)))

# Randomly sample 300 indexes
random_indexes = np.random.choice(sample_indexes, 300, replace=False)

# Use indexes to create random sample
random_combinations_chosen = [combinations_list[index] for index in random_indexes]

# Call the function to produce the visualization
visualize_search(grid_combinations_chosen, random_combinations_chosen)
```



You can really see how a grid search will cover a small area completely whilst random search will cover a much larger area but not completely.

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.

Informed Search: Coarse to Fine

When performing uninformed search, each iteration of hyperparameter tuning (each model) does not learn from the previous iterations.

Basic informed search methodology: Coarse to Fine
Start out with a rough random approach and iteratively refine your hyperparameter search.
Step 1. undertake a random search
Step 2. review results to see promising areas in your hyperparameter search space
Step 3. undertake a grid search in the smaller area, or further random searches before the grid search
Step 4. continue this process until an optimal score is obtained or the area becomes too small (meaning not many hyperparameter values) to properly search.

Advantages:

- Utilises the advantages of grid and random search – wide search to begin with, and deeper search once you know where a good spot is likely to be.
- Better spending of time and computational efforts mean you can iterate quicker, no need to waste time on search spaces that are not giving good results.

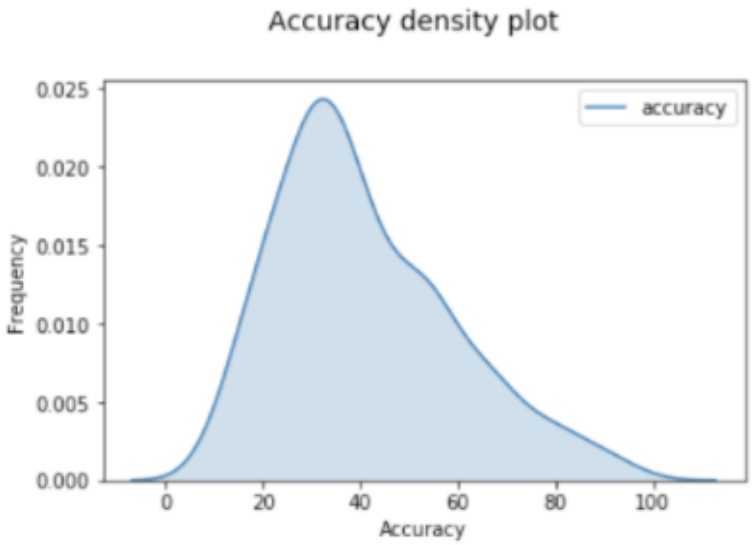
Let's take an example with the following hyperparameter ranges:

- `max_depth_list` between 1 and 65
- `min_sample_list` between 3 and 17
- `learn_rate_list` 150 values between 0.01 and 150

How many possible models do we have?

```
combinations_list = [list(x) for x in product(max_depth_list, min_sample_list, learn_rate_list)]
print(len(combinations_list))
134400
```

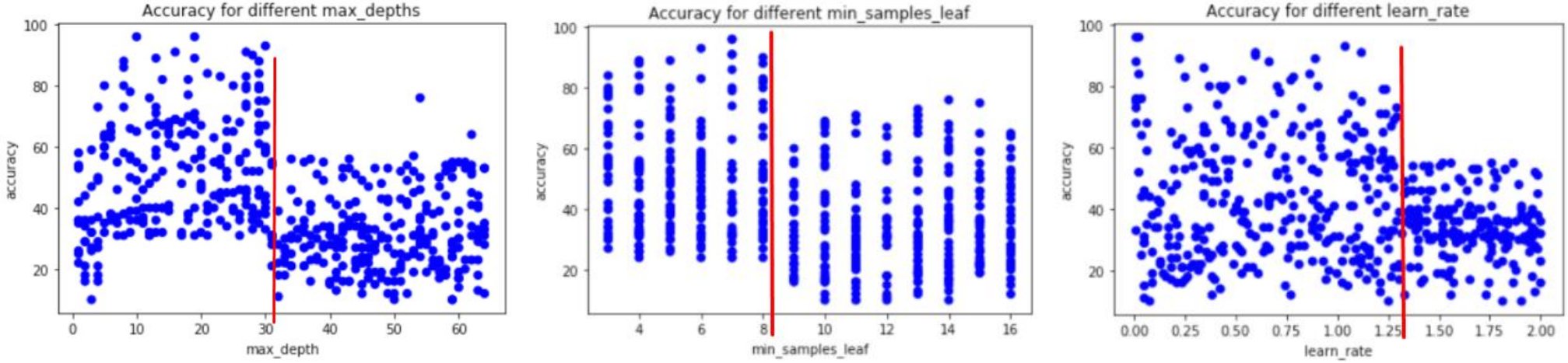
Rather than running all those models, we can do a random search on just 500 models. Let's first visualize just the accuracy column as a density plot. There are some good models in there, what hyperparameters did they use?



Top results:

max_depth	min_samples_leaf	learn_rate	accuracy
10	7	0.01	96
19	7	0.023355705	96
30	6	1.038389262	93
27	7	1.11852349	91
16	7	0.597651007	91

Let’s visualise the accuracy scores vs some parameters:



Information available from previous iteration, for higher accuracy:

- max_depth between 8 and 30
- min_samples_leaf <8
- learn_rate <1.3

Use these information to undertake a more refined grid search or even another random search around these values to improve the model.

Visualizing Coarse to Fine

You're going to undertake the first part of a Coarse to Fine search. This involves analyzing the results of an initial random search that took place over a large search space, then deciding what would be the next logical step to make your hyperparameter search finer.

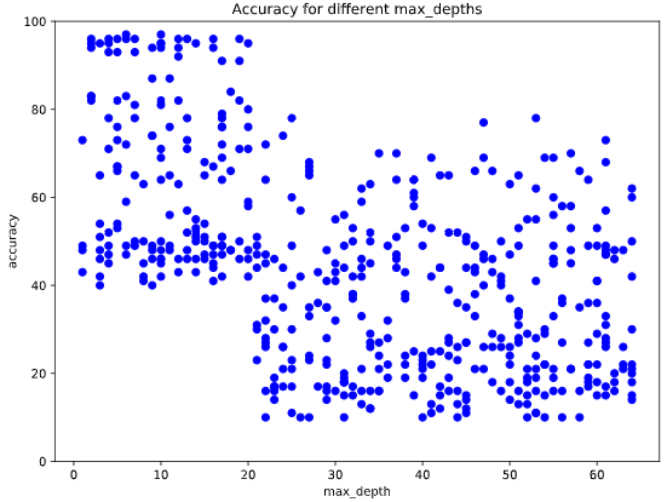
You have available:

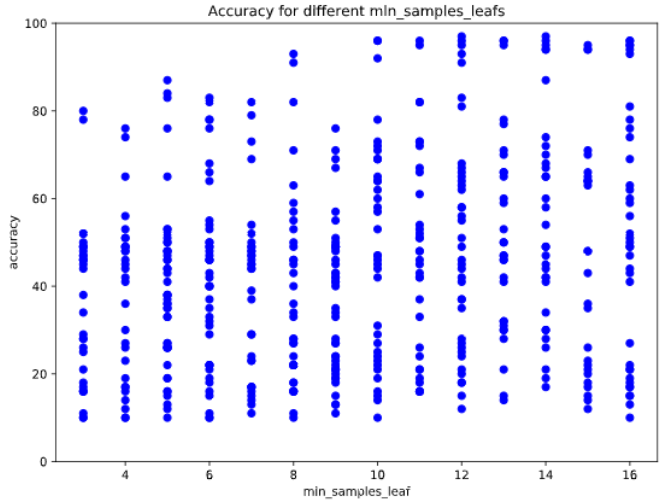
- `combinations_list` - a list of the possible hyperparameter combinations the random search was undertaken on.
- `results_df` - a DataFrame that has each hyperparameter combination and the resulting accuracy of all 500 trials. **Each hyperparameter is a column**, with the header the hyperparameter name.
- `visualize_hyperparameter()` - a function that takes in a column of the DataFrame (as a string) and produces a scatter plot of this column's values compared to the accuracy scores. An example call of the function would be `visualize_hyperparameter('accuracy')`

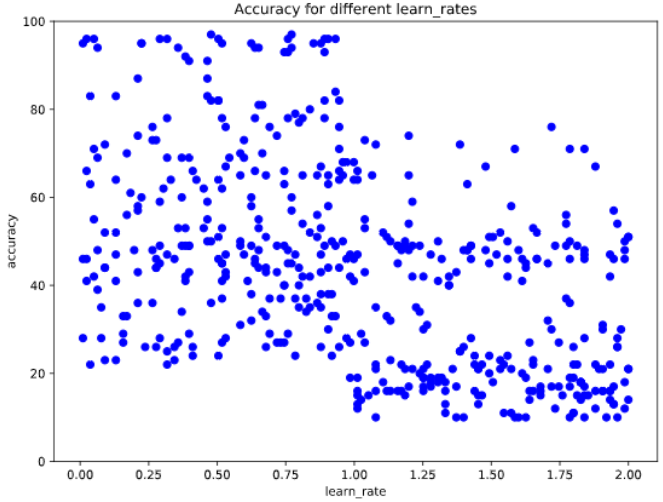
```
# Confirm the size of the combinations_list
```

```python
print(len(combinations_list))

# Sort the results_df by accuracy and print the top 10 rows
print(results_df.sort_values(by='accuracy', ascending=False).head(10))

# Confirm which hyperparameters were used in this search
print(results_df.columns)

# Call visualize_hyperparameter() with each hyperparameter in turn
visualize_hyperparameter('max_depth')
visualize_hyperparameter('min_samples_leaf')
visualize_hyperparameter('learn_rate')
```

```
<script.py> output:
    10000
        max_depth  min_samples_leaf  learn_rate  accuracy
    1          10                14    0.477450        97
    4           6                12    0.771275        97
    2           7                14    0.050067        96
    3           5                12    0.023356        96
    5          13                11    0.290470        96
    6           6                10    0.317181        96
    7          19                10    0.757919        96
    8           2                16    0.931544        96
    9          16                13    0.904832        96
    10         12                13    0.891477        96
    Index(['max_depth', 'min_samples_leaf', 'learn_rate', 'accuracy'], dtype='object')
```







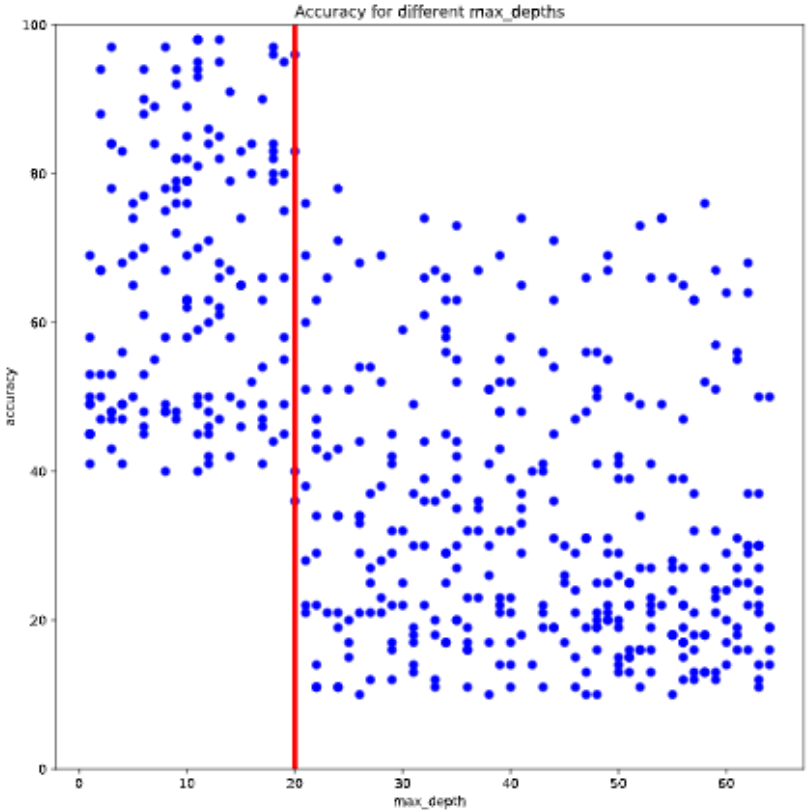
We have undertaken the first step of a Coarse to Fine search. Results clearly seem better when `max_depth` is below 20. `learn_rate`s smaller than 1 seem to perform well too. There is not a strong trend for `min_samples leaf` though. Let's use this in the next exercise!

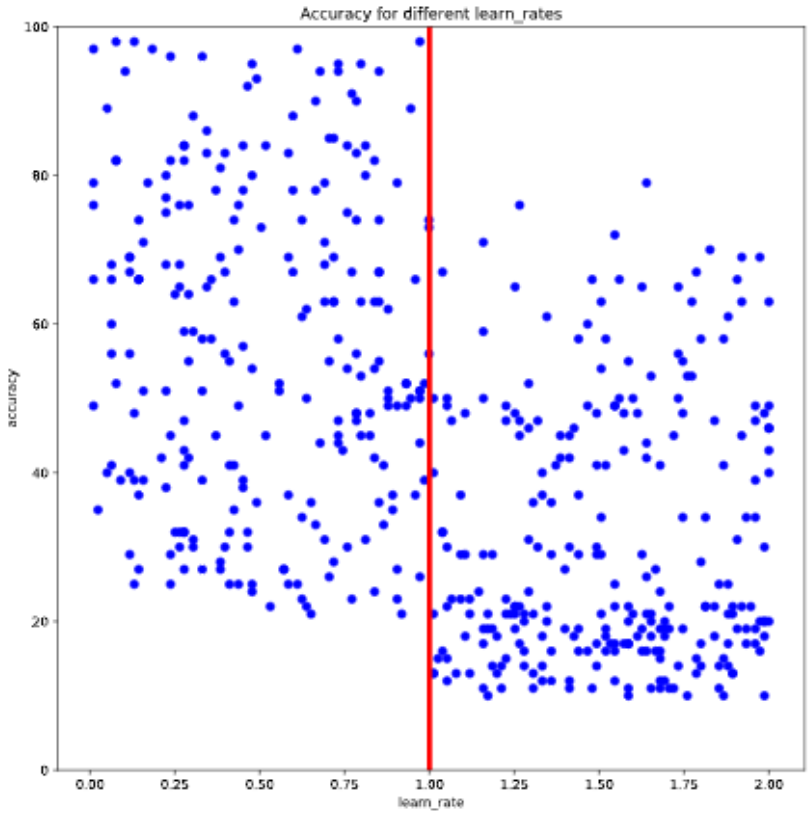
Coarse to Fine Iterations

You will now visualize the first random search undertaken, construct a tighter grid and check the results. You will have available:

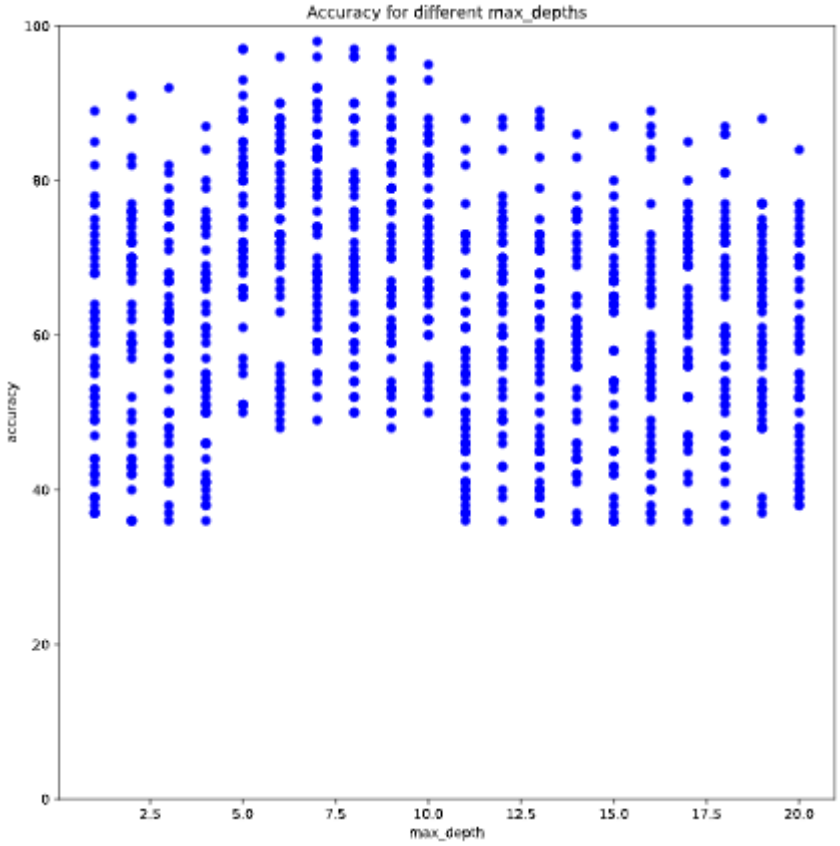
- `results_df` - a DataFrame that has the hyperparameter combination and the resulting accuracy of all 500 trials. Only the hyperparameters that had the strongest visualizations from the previous exercise are included (`max_depth` and `learn_rate`)
- `visualize_first()` - This function takes no arguments but will visualize each of your hyperparameters against accuracy for your first random search.

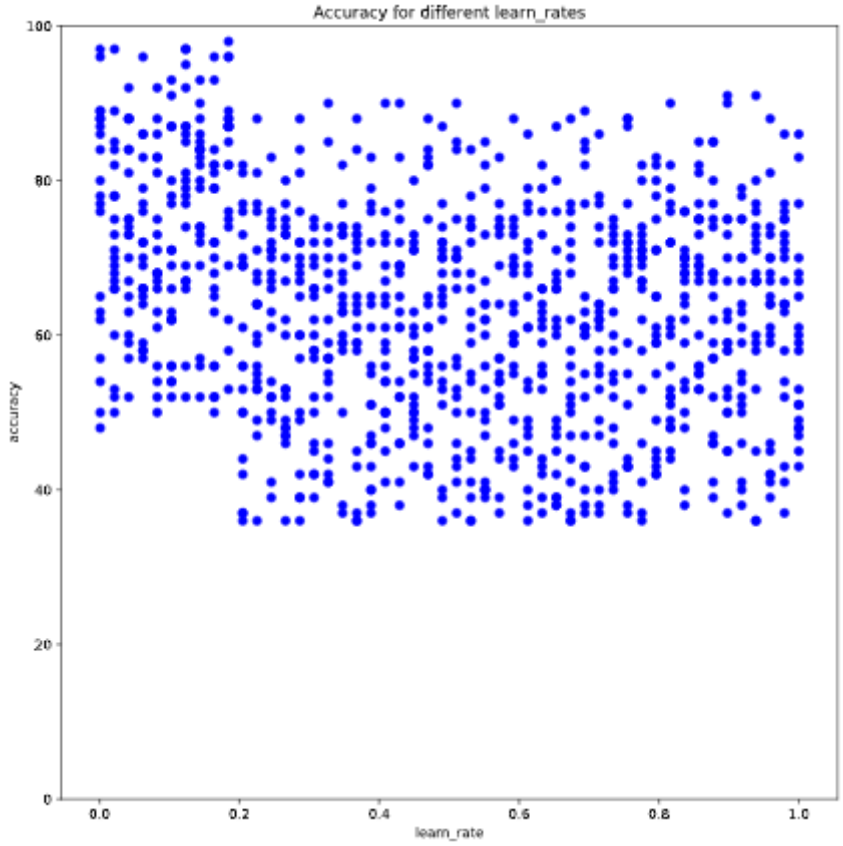
```
# Use the provided function to visualize the first results
visualize_first()
```





```
# Use the provided function to visualize the first results
# visualize_first()

# Create some combinations lists & combine:
max_depth_list = list(range(1,21))
learn_rate_list = np.linspace(0.001,1,50)

# Call the function to visualize the second results
visualize_second()
```





You can see in the second example our results are all generally higher. There also appears to be a bump around `max_depths` between 5 and 10 as well as `learn_rate` less than 0.2 so perhaps there is even more room for improvement!

Informed Search: Bayesian Statistics

Bayes Rule:

A statistical method of using new evidence to iteratively update our beliefs about an outcome.
Intuitively fits with the idea of informed seaarch, getting better as we get more evidence.

Bayes Rule has the form:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

- LHS is the probability of A (which we care about), given that B has occurred, where B is some new (relevant) evidence. This is known as the 'posterior'.
- RHS is how we calculate this.
- P(A) is the 'prior'. It is the initial hypothesis about the event we care about. It is different from P(A|B).
- P(A|B) is the probability given new evidence.
- P(B) is the 'marginal likelihood' and it is the probability of observing this new evidence.
- P(B|A) is the 'likelihood' which is the probability of observing the evidence, given the event we care about.

A medical example:

- 5% of people in the general population have a certain disease
  - P(D)
- 10% of people are predisposed
  - P(Pre)
- 20% of people with the disease are predisposed
  - P(Pre|D)

What is the probability that any person has the disease?

$$P(D) = 0.05$$

This is simply our prior as we have no evidence.

*What is the probability that a **predisposed** person has the disease?*

$$P(D \mid Pre) = \frac{P(Pre \mid D)\, P(D)}{P(pre)}$$

$$P(D \mid Pre) = \frac{0.2 \ * 0.05}{0.1} = 0.1$$

Hyperparameter tuning using the following process

1. Pick a hyperparameter combination.
2. Build a model.
3. Get new evidence (the score of the model).

4. Update our beliefs and chose better hyperparameters next round
5. Continue until we are happy with the result.

Bayesian hyperparameter tuning is very new but quite popular for larger and more complex hyperparameter tuning tasks, as they work well to find optimal hyperparameter combinations in these situations.

A useful package for Bayesian hyperparameter tuning is `Hyperopt` package:

1. set the domain, which is our Grid, with a bit of a twist.
2. set the optimization algorithm (use default TPE).
3. set the objective function to minimize: use 1-Accuracy because this package tries to minimize not maximize something.

To set the domain (Grid), there are many options: simple numbers, choose from a list, distribution of values. Hyperopt does not use point values on the grid but instead each point represents probabilities for each hyperparameter value. To keep it simple, we will use uniform distribution.

```
space = {
    'max_depth': hp.quniform('max_depth', 2, 10, 2),
    'min_samples_leaf': hp.quniform('min_samples_leaf', 2, 8, 2),
    'learning_rate': hp.uniform('learning_rate', 0.01, 1, 55),
}
```

```
def objective(params):
    params = {'max_depth': int(params['max_depth']),
        'min_samples_leaf': int(params['min_samples_leaf']),
        'learning_rate': params['learning_rate']}
    gbm_clf = GradientBoostingClassifier(n_estimators=500, **params)
    best_score = cross_val_score(gbm_clf, X_train, y_train,
                scoring='accuracy', cv=10, n_jobs=4).mean()
    loss = 1 - best_score
    write_results(best_score, params, iteration)
    return loss
```

Run the algorithm:

```
best_result = fmin(
            fn=objective,
            space=space,
            max_evals=500,
            rstate=np.random.RandomState(42),
            algo=tpe.suggest)
```

This function will only return the best hyperparameters which is why we logged out results at each iteration.

Bayes Rule in Python

In this exercise you will undertake a practical example of setting up Bayes formula, obtaining new evidence and updating your 'beliefs' in order to get a more accurate result. The example will relate to the likelihood that someone will close their account for your online software product.

These are the probabilities we know:

- 7% (0.07) of people are likely to close their account next month
- 15% (0.15) of people with accounts are unhappy with your product (you don't know who though!)
- 35% (0.35) of people who are likely to close their account are unhappy with your product

```
# Assign probabilities to variables
p_unhappy = 0.15
p_unhappy_close = 0.35

# Probabiliy someone will close
p_close = 0.07

# Probability unhappy person will close
p_close_unhappy = (p_close * p_unhappy_close) / p_unhappy
print(p_close_unhappy)
```

```
<script.py> output:
    0.16333333333333336
```

You correctly were able to frame this problem in a Bayesian way, and update your beliefs using new evidence. There's a 16.3% chance that a customer, given that they are unhappy, will close their account. Next we'll use a package which uses this methodology to automatically tune hyperparameters for us.

Bayesian Hyperparameter tuning with Hyperopt

In this example you will set up and run a Bayesian hyperparameter optimization process using the package Hyperopt (already imported as `hp` for you). You will set up the domain (which is similar to setting up the grid for a grid search), then set up the objective function. Finally, you will run the optimizer over 20 iterations.

You will need to set up the domain using values:

- `max_depth` using quniform distribution (between 2 and 10, increasing by 2)
- `learning_rate` using uniform distribution (0.001 to 0.9)

Note that for the purpose of this exercise, this process was reduced in data sample size and hyperopt & GBM iterations. If you are trying out this method by yourself on your own machine, try a larger search space, more trials, more cvs and a larger dataset size to really see this in action!

```
# Set up space dictionary with specified hyperparameters
space = {'max_depth': hp.quniform('max_depth', 2, 10, 2),'learning_rate': hp.uniform('learning_rate', 0.001, 0.9)}

# Set up objective function
def objective(params):
    params = {'max_depth': int(params['max_depth']),'learning_rate': params['learning_rate']}
    gbm_clf = GradientBoostingClassifier(n_estimators=100, **params)
    best_score = cross_val_score(gbm_clf, X_train, y_train, scoring='accuracy', cv=2, n_jobs=4).mean()
    loss = 1 - best_score
    return loss

# Run the algorithm
best = fmin(fn=objective,space=space, max_evals=20, rstate=np.random.RandomState(42), algo=tpe.suggest)
print(best)
```

```
<script.py> output:
      0%|          | 0/20 [00:00<?, ?it/s, best loss: ?]  5%|5         | 1/20 [00:00<00:03,  5.84it/s, best loss: 0.26759418985474637] 10%|#
    {'learning_rate': 0.11310589268581149, 'max_depth': 6.0}
```

You succesfully built your first Bayesian hyperparameter tuning algorithm. This will be a very powerful tool for your machine learning modeling in future. Bayesian hyperparameter tuning is a new and popular method so this first taster is a valuable thing to gain experience in. You are highly encouraged to extend this example on your own!

Informed Search: Genetic Algorithms

In genetic evolution in the real world, we have the following process.

1. There are many creatures existing called ‘offspring’.
2. The strongest creatures survive the tough environment and pair off.
3. There is some 'crossover' as they form offspring.
4. Random mutations occur with some of the offspring. These mutations sometimes help give some offspring an advantage.
5. Go back to (1) and the cycle continues.

This is how evolution works in nature.

We can apply the same idea to hyperparameter tuning.

1. We can create some models (that have hyperparameter settings).
2. We can pick the best (by our scoring function). These are the ones that 'survive'.
3. We can then create new models that are similar to the best ones.
4. We add in some randomness so we don't reach a local optimum.
5. We continue this until we are happy!

This is an informed search that has a number of advantages.

- It allows us to learn from previous iterations, just like Bayesian hyperparameter tuning.
- It has the additional advantage of some randomness. This randomness is important because it means we won't just be working on finding similar models and going down a singular path.
- It takes care of many tedious aspects of machine learning, like algorithm and hyperparameter choice.

A useful library for genetic hyperparameter tuning is TPOT:

> Consider TPOT your Data Science Assistant. TPOT is a Python Automated Machine Learning tool that optimizes machine learning **pipelines** using genetic programming.

Pipelines not only include the model (or multiple models) but also work on features and other aspects of the process. Plus it returns the Python code of the pipeline for you!

The key arguments to a TPOT classifier are:

- `generations`: Iterations to run training for, ie, number of cycles we undertake of creating offspring models, mutating and crossing over, picking the best and continuing.
- `population_size`: Number of models to keep after each iteration. The strongest 'offspring' (model).
- `offspring_size`: Number of models to produce in each iteration.
- `mutation_rate`: Proportion of pipelines to apply randomness to. This hyperparameter sets that proportion (between 0 and 1).
- `crossover_rate`: Proportion of pipelines to breed/crossover in each iteration to find similar ones. This sets the proportion of pipelines that we do this to.
- `scoring`: the objective function to determine the strongest models or offspring for example, accuracy.
- `cv`: the cross-validation strategy to use, ie, classic machine learning modeling.

```
from tpot import TPOTClassifier
tpot = TPOTClassifier(generations=3, population_size=5,
                      verbosity=2, offspring_size=10,
                      scoring='accuracy', cv=5)
tpot.fit(X_train, y_train)
print(tpot.score(X_test, y_test))
```

We will keep default values for `mutation_rate` and `crossover_rate` as they are best left to the default without deeper knowledge on genetic programming. The verbosity parameter will print out the process as it goes. Notice how we are not even selecting algorithms or hyperparameters? TPOT does it all!

Genetic Hyperparameter Tuning with TPOT

You're going to undertake a simple example of genetic hyperparameter tuning. `TPOT` is a very powerful library that has a lot of features. You're just scratching the surface in this lesson, but you are highly encouraged to explore in your own time.

This is a very small example. In real life, TPOT is designed to be run for many hours to find the best model. You would have a much larger population and offspring size as well as hundreds more generations to find a good model.

You will create the estimator, fit the estimator to the training data and then score this on the test data.

For this example we wish to use:

- 3 generations
- 4 in the population size
- 3 offspring in each generation
- accuracy for scoring

A `random_state` of 2 has been set for consistency of results.

```
# Assign the values outlined to the inputs
number_generations = 3
population_size = 4
offspring_size = 3
scoring_function = 'accuracy'

# Create the tpot classifier
tpot_clf = TPOTClassifier(generations=number_generations, population_size=population_size,
                          offspring_size=offspring_size, scoring=scoring_function,
                          verbosity=2, random_state=2, cv=2)

# Fit the classifier to the training data
tpot_clf.fit(X_train, y_train)

# Score on the test set
print(tpot_clf.score(X_test, y_test))
```

```
<script.py> output:
    Warning: xgboost.XGBClassifier is not available and will not be used by TPOT.
    Generation 1 - Current best internal CV score: 0.7575064376609415
    Generation 2 - Current best internal CV score: 0.7750693767344183
    Generation 3 - Current best internal CV score: 0.7750693767344183

    Best pipeline: BernoulliNB(input_matrix, alpha=0.1, fit_prior=True)
    0.76
```

You can see in the output the score produced by the chosen model (in this case a version of Naive Bayes) over each generation, and then the final accuracy score with the hyperparameters chosen for the final model. This is a great first example of using TPOT for automated hyperparameter tuning. You can now extend on this on your own and build great machine learning models!

Analysing TPOT's stability

You will now see the random nature of TPOT by constructing the classifier with different random states and seeing what model is found to be best by the algorithm. This assists to see that TPOT is quite unstable when not run for a reasonable amount of time.

```
# Create the tpot classifier
tpot_clf = TPOTClassifier(generations=2, population_size=4, offspring_size=3, scoring='accuracy', cv=2,
                          verbosity=2, random_state=42)

# Fit the classifier to the training data
tpot_clf.fit(X_train, y_train)

# Score on the test set
print(tpot_clf.score(X_test, y_test))
```

```
<script.py> output:
    Warning: xgboost.XGBClassifier is not available and will not be used by TPOT.
    Generation 1 - Current best internal CV score: 0.7549688742218555
    Generation 2 - Current best internal CV score: 0.7549688742218555

    Best pipeline: DecisionTreeClassifier(input_matrix, criterion=gini, max_depth=7, min_samples_leaf=11, min_samples_split=12)
    0.75
```

```
# Create the tpot classifier
tpot_clf = TPOTClassifier(generations=2, population_size=4, offspring_size=3, scoring='accuracy', cv=2,
                          verbosity=2, random_state=122)

# Fit the classifier to the training data
tpot_clf.fit(X_train, y_train)

# Score on the test set
print(tpot_clf.score(X_test, y_test))
```

```
<script.py> output:
    Warning: xgboost.XGBClassifier is not available and will not be used by TPOT.
    Generation 1 - Current best internal CV score: 0.7675066876671917
    Generation 2 - Current best internal CV score: 0.7675066876671917

    Best pipeline: KNeighborsClassifier(MaxAbsScaler(input_matrix), n_neighbors=57, p=1, weights=distance)
    0.75
```

```
# Create the tpot classifier
tpot_clf = TPOTClassifier(generations=2, population_size=4, offspring_size=3, scoring='accuracy', cv=2,
                          verbosity=2, random_state=99)

# Fit the classifier to the training data
tpot_clf.fit(X_train, y_train)

# Score on the test set
print(tpot_clf.score(X_test, y_test))
```

```
<script.py> output:
    Warning: xgboost.XGBClassifier is not available and will not be used by TPOT.
    Generation 1 - Current best internal CV score: 0.8075326883172079
    Generation 2 - Current best internal CV score: 0.8075326883172079

    Best pipeline: RandomForestClassifier(SelectFwe(input_matrix, alpha=0.033), bootstrap=False, criterion=gini, max_features=1.0, min_samples_leaf=19, min_samples_split=10, n_estimators=100)
    0.78
```

You can see that TPOT is quite unstable when only running with low generations, population size and offspring. The first model chosen was a Decision Tree, then a K-nearest Neighbor model and finally a Random Forest. Increasing the generations, population size and offspring and running this for a long time will assist to produce better models and more stable results. Don't hesitate to try it yourself on your own machine!

Course completed!

Recap topics covered:

- Difference between hyperparameters and parameters

  > Hyperparameters are components of the model that you set. They are not learned during the modeling process

  > Parameters are not set by you. The algorithm will discover these for you

- Best practice for setting hyperparameters and their values to search over.
  - Some hyperparameters are better to start with than others, for example with a Random Forest use case.
  - There are potentially silly values you can set for hyperparameters that will waste your effort.
  - You need to beware of conflicting hyperparameter values, especially when the error may not be obvious.
  - Best practice is specific to each algorithm and hyperparameter. So you have some work to do researching and learning this.

- Grid Search
  - Construct a matrix (‘grid’) of all the hyperparameters combinations and values we wish to test for.
  - Build models for all the different hyperparameter combinations
  - Finally pick the best model.
    You learned that this is a computationally expensive method but it is guaranteed to find the best model in the grid you specify. Remember the importance of setting good grid values!

- Random Search
  - Very similar to grid search
  - The main difference was instead of trying every square (or hyperparameter combination) on the grid, we randomly selected a certain number (n) of random combination.
    This method is more efficient at finding a reasonably good model faster but it is not guaranteed to find the best on your grid.

- Informed Search
  - This is where each iteration learns from the last, as opposed to grid and random where you do all your modeling at once and then pick the best.
  - 'Coarse to Fine': you iteratively build random searches to narrow your search space before a final grid search.
  - Bayesian hyperparameter tuning: you use the method of updating your prior beliefs when new evidence arrives about model performance.
  - Genetic algorithms: drawing from nature and how evolution selects the best species as you select your best models over the generations.

Happy learning!