Hyperparameter Tuning in Python

Sunday, 23 August 2020 3:43 PM

Hyperparameters and Parameters

Introduction

Why study this course?

- New, complex algorithms with many hyperparameters
- Tuning can take a lot of time
- Develops deeper understanding beyond the default settings

You may be surprised what you find under the hood!

The dataset

The dataset relates to credit card defaults.

It contains variables related to the financial history of some consumers in Taiwan. It has 30,000 users and 24 attributes.

Our modeling target is whether they defaulted on their loan

It has already been preprocessed and at times we will take smaller samples to demonstrate a concept

Parameters in Logistic Regression

A simple logistic regression model:

```
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... ]])
```

Parameters in Logistic Regression

Tidy up the coefficients:

```
# 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"])
```

Parameters in Logistic Regression

Now sort and print the top three coefficients

```
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

Where to find Parameters

To find parameters we need:

1. To know a bit about the algorithm
2. Consult the Scikit Learn documentation

Parameters will be found under the 'Attributes' section, *not* the 'parameters' section!

Parameters in Random Forest

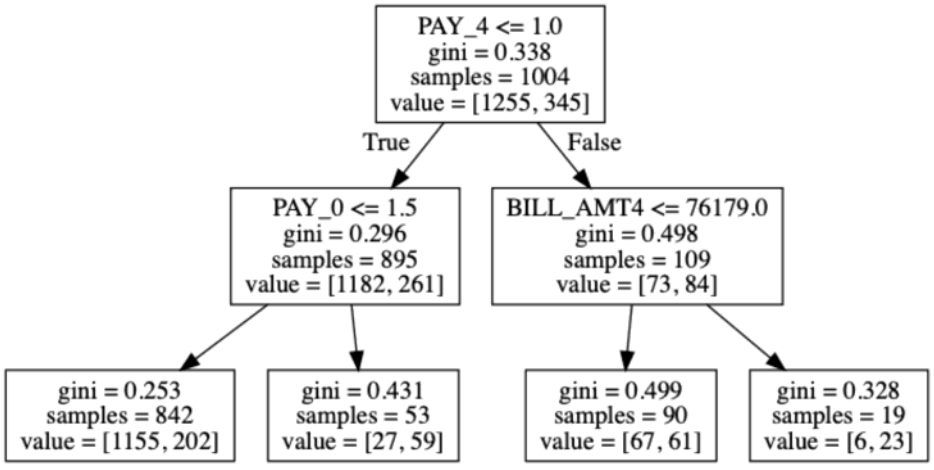
What about tree based algorithms?

Random forest has no coefficients, but node decisions (what feature and what value to split on).

```
# 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]
```

For simplicity we will show the final product (an image) of the decision tree. Feel free to explore the package used for this (graphviz & pydotplus) yourself.



Extracting Node Decisions

We can pull out details of the left, second-from-top node:

```
# 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"

What is a hyperparameter

Hyperparameters:

- Something **you** set before the modelling process (like knobs on an old radio)
  - You also 'tune' your hyperparameters!
- The algorithm does not learn these

This is the crucial differentiator between hyperparameters and parameters.

Hyperparameters in Random Forest

Create a simple random forest estimator and print it out:

```
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)
```

A single hyperparameter

Take the `n_estimators` parameter.

Data Type & Default Value:

> n_estimators : integer, optional (default=10)

Definition:

> The number of trees in the forest.

Setting hyperparameters

Set some hyperparameters at estimator creation:

```
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)
```

Hyperparameters in Logistic Regression

Find the hyperparameters of a Logistic Regression:

```
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)
```

There are less hyperparameters to tune with this algorithm!

Hyperparameter Importance

Some hyperparameters are more important than others.

Some will **not** help model performance:

For the random forest classifier:

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

Random Forest: Important Hyperparameters

Some important hyperparameters:

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

Remember: this is only a guide

The n_estimators (how many trees in the forest) should be set to a high value, 500 or 1000 or

even more is not uncommon (noting that there are computational costs to higher values)

max_features controls how many features to consider when splitting, which is vital to ensure tree diversity.

The next two control overfitting of individual trees.

The 'criterion' hyperparameter may have a small impact but it is not generally a primary hyperparameter to consider.

How to find hyperparameters that matter?

Some resources for learning this:

- Academic papers
- Blogs and tutorials from trusted sources (Like DataCamp!)
- The Scikit Learn module documentation
- Experience

Hyperparameter Values

Some hyperparameters are more important than others to begin tuning.

But which *values* to try for hyperparameters?

- Specific to each algorithm & hyperparameter
- Some best practice guidelines & tips do exist

Let's look at some top tips!

It is firstly important to know what values NOT to set as they may conflict.

Conflicting Hyperparameter Choices

Be aware 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`

Make sure to consult the Scikit Learn documentation!

Silly Hyperparameter Values

Be aware 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 Neighbor in KNN algorithm
  - Averaging the 'votes' of one person doesn't sound very robust!
- Increasing a hyperparameter by a very small amount

Spending time documenting sensible values for hyperparameters is a valuable activity.

Automating Hyperparameter Choice

In the previous exercise, we built models as:

```
knn_5 =  KNeighborsClassifier(n_neighbors=5)
knn_10 = KNeighborsClassifier(n_neighbors=10)
knn_20  = KNeighborsClassifier(n_neighbors=20)
```

This is quite inefficient. Can we do better?

Automating Hyperparameter Tuning

Try a for loop to iterate through options:

```
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)
```

Automating Hyperparameter Tuning

We can store the results in a DataFrame to view:

```
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

Learning Curves

Let's create a learning curve graph

We'll test many more values this time

```
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})
```

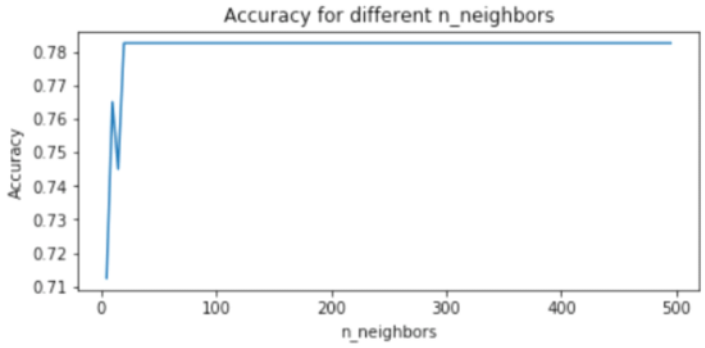
Learning Curves

We can plot the larger DataFrame:

```
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()
```

Learning Curves

Our graph:



A handy trick for generating values

Python's `range` function does not work for decimal steps.

A handy trick uses NumPy's `np.linspace(start, end, num)`

- Create a number of values (`num`) evenly spread within an interval (`start` , `end`) that you specify.

```
print(np.linspace(1,2,5))
```

```
[1. 1.25 1.5 1.75 2. ]
```

Grid search

Automating 2 Hyperparameters

What about testing values of 2 hyperparameters?

Using a GBM algorithm:

- `learn_rate` - [0.001, 0.01, 0.05]
- `max_depth` -[4,6,8,10]

We could use a (*nested*) for loop!

Automating 2 Hyperparameters

Firstly a model creation function:

```
def gbm_grid_search(learn_rate, max_depth):
    model = GradientBoostingClassifier(
            learning_rate=learn_rate,
            max_depth=max_depth)
    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:
        results_list.append(gbm_grid_search(learn_rate,max_depth))
```

Automating 2 Hyperparameters

We can put these results into a DataFrame as well and print out:

```
results_df = pd.DataFrame(results_list, columns=['learning_rate', 'max_depth', '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

How many models?

There were many more models built by adding more hyperparameters and values.

- The relationship is not linear, it is exponential
- One more value of a hyperparameter is not *just* one model
- 5 for Hyperparameter 1 and 10 for Hyperparameter 2 is 50 models!

What about cross-validation?

- 10-fold cross-validation would make 50x10 = 500 models!

From 2 to N hyperparameters

What about adding more hyperparameters?

We could nest our loop!

```
# Adjust the list of values to test
learn_rate_list = [0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5]
max_depth_list = [4,6,8, 10, 12, 15, 20, 25, 30]
subsample_list = [0.4,0.6, 0.7, 0.8, 0.9]
max_features_list = ['auto', 'sqrt']
```

From 2 to N hyperparameters

Adjust our function:

```
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)])
```

From 2 to N hyperparameters

Adjusting our for loop (nesting):

```
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)
```

From 2 to N hyperparameters

How many models now?

- 7x9x5x2 = 630 (6,300 if cross-validated!)

We can't keep nesting forever!

Plus, what if we wanted:

- Details on training times & scores
- Details on cross-validation scores

Introducing Grid Search

Let's create a grid:

- Down the left all values of max_depth
- Across the top all values of learning_rate

		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)

Introducing Grid Search

Working through each cell on the grid:

		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)

(4,0.001) is equivalent to making an estimator like so:

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

Grid Search Pros & Cons

Some advantages of this approach:

Advantages:

- You don't have to write thousands of lines of code
- Finds the best model within the grid (*special note here!)
- Easy to explain

Grid Search Pros & Cons

Some disadvantages of this approach:

- Computationally expensive! Remember how quickly we made 6,000+ models?
- It is 'uninformed'. Results of one model don't help creating the next model.

We will cover 'informed' methods later!

GridSearchCV Object

Introducing a `GridSearchCV` object:

```
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

Steps in a Grid Search:

1. An algorithm to tune the hyperparameters. (Sometimes called an 'estimator')
2. Defining which hyperparameters we will tune
3. Defining a range of values for each hyperparameter
4. Setting a cross-validation scheme; and
5. Define a score function so we can decide which square on our grid was 'the best'.
6. Include extra useful information or functions

GridSearchCV Object Inputs

The important inputs are:

- `estimator`
- `param_grid`
- `cv`
- `scoring`
- `refit`
- `n_jobs`
- `return_train_score`

GridSearchCV 'estimator'

The `estimator` input:

- Essentially our algorithm
- You have already worked with KNN, Random Forest, GBM, Logistic Regression

Remember:

- Only one estimator per GridSearchCV object

GridSearchCV 'param_grid'

The `param_grid` input:

- Setting which hyperparameters and values to test

Rather than a list:

```
max_depth_list = [2, 4, 6, 8]
min_samples_leaf_list = [1, 2, 4, 6]
```

This would be:

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

GridSearchCV 'param_grid'

The `param_grid` input:

Remember: The keys in your `param_grid` dictionary must be valid hyperparameters.

For example, for a Logistic regression estimator:

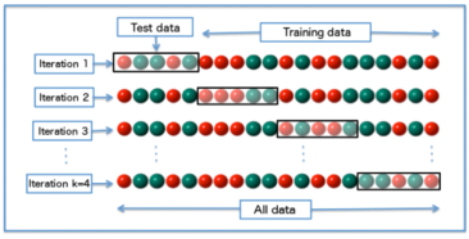
```
# Incorrect
param_grid = {'C': [0.1,0.2,0.5],
              'best_choice': [10,20,50]}
```

```
ValueError: Invalid parameter best_choice for estimator LogisticRegression
```

GridSearchCV 'cv'

The `cv` input:

- Choice of how to undertake cross-validation
- Using an integer undertakes k-fold cross validation where 5 or 10 is usually standard



GridSearchCV 'scoring'

The `scoring` input:

- Which score to use to choose the best grid square (model)
- Use your own or Scikit Learn's `metrics` module

You can check all the built in scoring functions this way:

```
from sklearn import metrics
sorted(metrics.SCORERS.keys())
```

GridSearchCV 'refit'

The `refit` input:

- Fits the best hyperparameters to the training data
- Allows the `GridSearchCV` object to be used as an estimator (for prediction)
- A very handy option!

GridSearchCV 'n_jobs'

The `n_jobs` input:

- Assists with parallel execution
- Allows multiple models to be created at the same time, rather than one after the other

Some handy code:

```
import os
print(os.cpu_count())
```

GridSearchCV 'return_train_score'

The `return_train_score` input:

- Logs statistics about the training runs that were undertaken
- Useful for analyzing bias-variance trade-off but adds computational expense.
- Does not assist in picking the best model, only for analysis purposes

Building a GridSearchCV object

Building our own GridSearchCV Object:

```
# 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')
```

Building a GridSearchCv Object

Putting the pieces together:

```
grid_rf_class = GridSearchCV(
    estimator = rf_class,
    param_grid = parameter_grid,
    scoring='accuracy',
    n_jobs=4,
    cv = 10,
    refit=True,
    return_train_score=True)
```

Using a GridSearchCV Object

Because we set `refit` to `True` we can directly use the object:

```
#Fit the object to our data
grid_rf_class.fit(X_train, y_train)

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

Analyzing the output

Let's analyze the GridSearchCV outputs.

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_`

Accessing object properties

Properties are accessed using the dot notation.

For example:

`grid_search_object.property`

Where `property` is the actual property you want to retrieve

The `.cv_results_` property

The `cv_results_` property:

Read this into a DataFrame to print and analyze:

```
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

The `.cv_results_` 'time' columns

The `time` columns refer to the time it took to fit (and score) the model.

Remember how we did a 5-fold cross-validation? This ran 5 times and stored the average and standard deviation of the times it took in seconds.

	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 .cv_results_ 'param_' columns

The `param_` columns store the parameters it tested on that row, one column per parameter

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 param_ columns contain information on the different parameters that were used in the model.

Remember, each row in this DataFrame is about one model.

The `.cv_results_` 'param' column

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 params column is a dictionary of all the parameters from the previous 'param' columns.

set_option here to ensure we don't truncate the results we are printing.

The `.cv_results_` 'test_score' columns

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 `.cv_results_` 'rank_test_score' column

The rank column, ordering the `mean_test_score` from best to worst:

rank_test_score
9
4
1
3
2

The rank column conveniently ranks the rows by the mean_test_score.

Extracting the best row

We can select the best grid square easily from `cv_results_` using the `rank_test_score` column

```
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

The .cv_results_ 'train_score' columns

The `test_score` columns are then repeated for the `training_scores` .

Some important notes to keep in mind:

- `return_train_score` must be `True` to include training scores columns.
- There is no ranking column for the training scores, as we only care about test set performance

The best grid square

Information on the best grid square is neatly summarized in the following three 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.

The `best_estimator_` property

The `best_estimator_` property is an estimator built using the best parameters from the grid search.

For us this is a Random Forest estimator:

```
type(grid_rf_class.best_estimator_)
```

`sklearn.ensemble.forest.RandomForestClassifier`

We could also directly use this object as an estimator if we want!

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)
```

Extra information

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.

Random Search

What you already know

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.

Why does this work?

Bengio & Bergstra (2012):

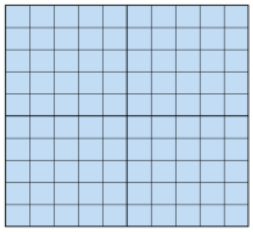
> This paper shows empirically and theoretically that randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid.

Two main reasons:

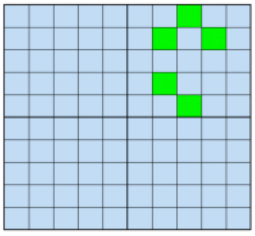
1. Not every hyperparameter is as important
2. A little trick of probability

A probability trick

A grid search:



Our best models:



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

A probability Trick

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.

A probability trick

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**

A probability trick

What does that all mean?

- You are unlikely to keep completely missing the 'good area' for a long time when randomly picking new spots
- A grid search may spend lots of time in a 'bad area' as it covers exhaustively.

Some important notes

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'

Creating a random sample of hyperparameters

We can create our own random sample of hyperparameter combinations:

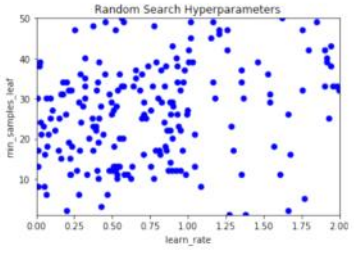
```
# Set some hyperparameter lists
learn_rate_list = np.linspace(0.001,2,150)
min_samples_leaf_list = list(range(1,51))
```

```
# 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_random)), 100,
```

Visualizing a Random Search

We can also visualize the random search coverage by plotting the hyperparameter choices on an X and Y axis.



Notice how this has a wide range of the scatter but not deep coverage?

Comparing to GridSearchCV

We don't need to reinvent the wheel. Let's recall the steps for a Grid Search:

1. Decide an algorithm/estimator
2. Defining which hyperparameters we will tune
3. Defining a range of values for each hyperparameter
4. Setting a cross-validation scheme; and
5. Define a score function
6. Include extra useful information or functions

Comparing to Grid Search

There is only one difference:

- Step 7 = Decide how many samples to take (then sample)

That's it! (mostly)

Comparing Scikit Learn Modules

The modules are similar too:

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')
```

Key differences

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.

Build a RandomizedSearchCV Object

Now we can build a random search object just like the grid search, but with our small change:

```
# 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
```

Build a RandomizedSearchCV Object

Now we can build the object

```
# 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)
```

Analyze the output

The output is exactly the same!

How do we see what hyperparameter values were chosen?

The `cv_results_` dictionary (in the relevant `param_` columns)!

Extract the lists:

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

Analyze the output

Build our visualization:

```
# 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()
```

Analyze the output

A similar graph to before:

What's the same?

Similarities between Random and Grid Search?

- Both are automated ways of tuning different hyperparameters
- For both you set the grid to sample from (which hyperparameters and values for each)

Remember to think carefully about your grid!

- For both you set a cross-validation scheme and scoring function

What's different?

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?

So which one should I use? What are my considerations?

- How much data do you have?
- How many hyperparameters and values do you want to tune?
- How much resources do you have? (Time, computing power)

- More data means random search may be better option.
- More of these means random search may be a better option.
- Less resources means random search may be a better option.

Informed Search

Informed vs Uninformed Search

So far everything we have done has been uninformed search:

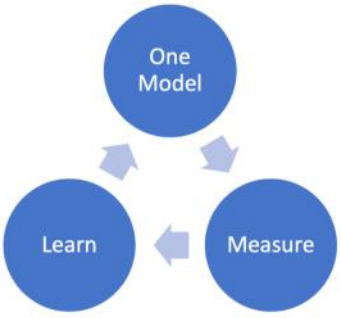
> Uninformed search: Where each iteration of hyperparameter tuning does **not** learn from the previous iterations.

This is what allows us to parallelize our work. Though this doesn't sound very efficient?

Informed vs Uninformed

The process so far:

An alternate way:



Coarse to Fine Tuning

A basic informed search methodology:

Start out with a rough, random approach and iteratively refine your search.

The process is:

1. Random search
2. Find promising areas
3. Grid search in the smaller area
4. Continue until optimal score obtained

You could substitute (3) with further random searches before the grid search

Why Coarse to Fine?

Coarse to fine tuning has some advantages:

- Utilizes the advantages of grid and random search.
  - Wide search to begin with
  - 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!

Note: This isn't informed on one model but batches

Undertaking Coarse to Fine

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

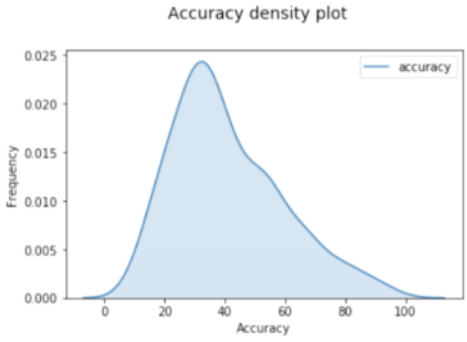
How many possible models do we have?

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

Visualizing Coarse to Fine

Let's do a random search on just 500 combinations.

Here we plot our accuracy scores:



Which models were the good ones?

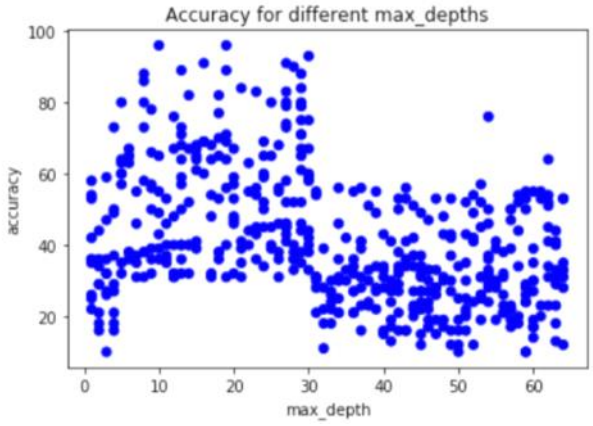
Visualizing Coarse to Fine

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

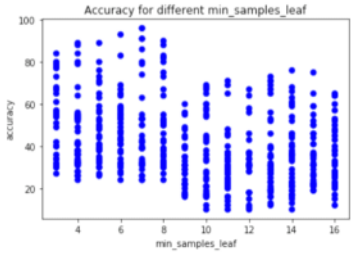
Visualizing Coarse to Fine

Let's visualize the `max_depth` values vs accuracy score:

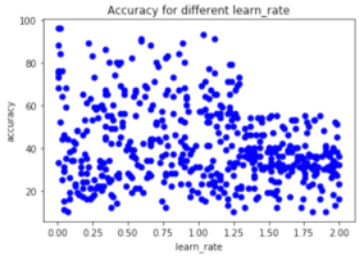


Visualizing coarse to Fine

`min_samples_leaf` better below 8



`learn_rate` worse above 1.3



The next steps

What we know from iteration one:

- `max_depth` between 8 and 30
- `learn_rate` less than 1.3
- `min_samples_leaf` perhaps less than 8

Where to next? Another random or grid search with what we know!

Bayes Introduction

Bayes Rule:

> A statistical method of using **new evidence** to iteratively update our *beliefs* about some *outcome*

- Intuitively fits with the idea of *informed* search. Getting better as we get more evidence.

Bayes Rule

Bayes Rule has the form:

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

- LHS = the probability of A, given B has occurred. B is some new evidence.
  - This is known as the 'posterior'
- RHS is how we calculate this.
- P(A) is the 'prior'. The initial hypothesis about the event. It is different to P(A|B), the P(A|B) is the probability *given* new evidence.

Bayes Rule

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

- 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.

This all may be quite confusing, but let's use a common example of a medical diagnosis to demonstrate.

Bayes in Medicine

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)

Bayes in Medicine

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?*

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

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

Bayes in Hyperparameter Tuning

We can apply this logic to hyperparameter tuning:

- Pick a hyperparameter combination
- Build a model
- Get new *evidence* (the score of the model)
- Update our beliefs and chose better hyperparameters next round

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

Bayesian Hyperparameter Tuning with Hyperopt

Introducing the `Hyperopt` package.

To undertake bayesian hyperparameter tuning we need to:

1. Set the Domain: Our Grid (with a bit of a twist)
2. Set the Optimization algorithm (use default TPE)
3. Objective function to minimize: we will use 1-Accuracy

Hyperopt: Set the Domain (grid)

Many options to set the grid:

- 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.

We will do a simple uniform distribution but there are many more if you check the documentation.

The Domain

Set up the grid:

```
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),
}
```

The objective function

The objective function runs the algorithm:

```
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

Run the algorithm:

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

A lesson on genetics

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

1. There are many creatures existing ('offspring')
2. The strongest creatures survive and pair off
3. There is some 'crossover' as they form offspring
4. There are random mutations to some of the offspring
    - These mutations sometimes help give some offspring an advantage
5. Go back to (1)!

Genetics in Machine Learning

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 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. Repeat until we are happy!

Why does this work well?

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*
- (The package we'll use) takes care of many tedious aspects of machine learning

Introducing TPOT

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!

TPOT components

The key arguments to a TPOT classifier are:

- `generations` – Iterations to run training for.
- `population_size` – The number of models to keep after each iteration.
- `offspring_size` – Number of models to produce in each iteration.
- `mutation_rate` – The proportion of pipelines to apply randomness to.
- `crossover_rate` – The proportion of pipelines to breed each iteration.
- `scoring` – The function to determine the best models
- `cv` – Cross-validation strategy to use.

A simple example

A simple example:

```
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.