Informed Search: Coarse to Fine

HYPERPARAMETER TUNING IN PYTHON

Alex Scriven
Data Scientist

datacamp

Informed vs Uninformed Search

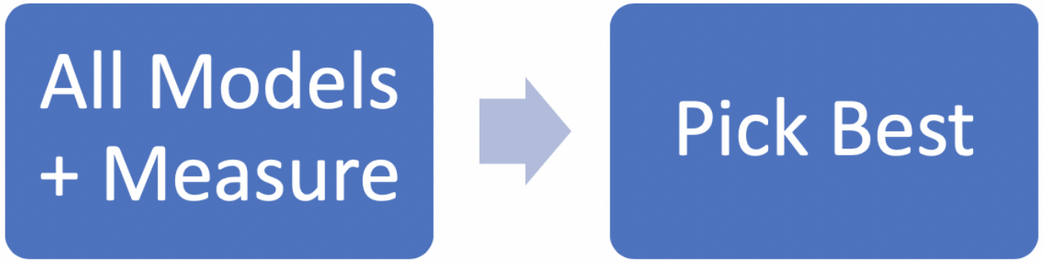
So far everything we have done has been uninformed search:

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

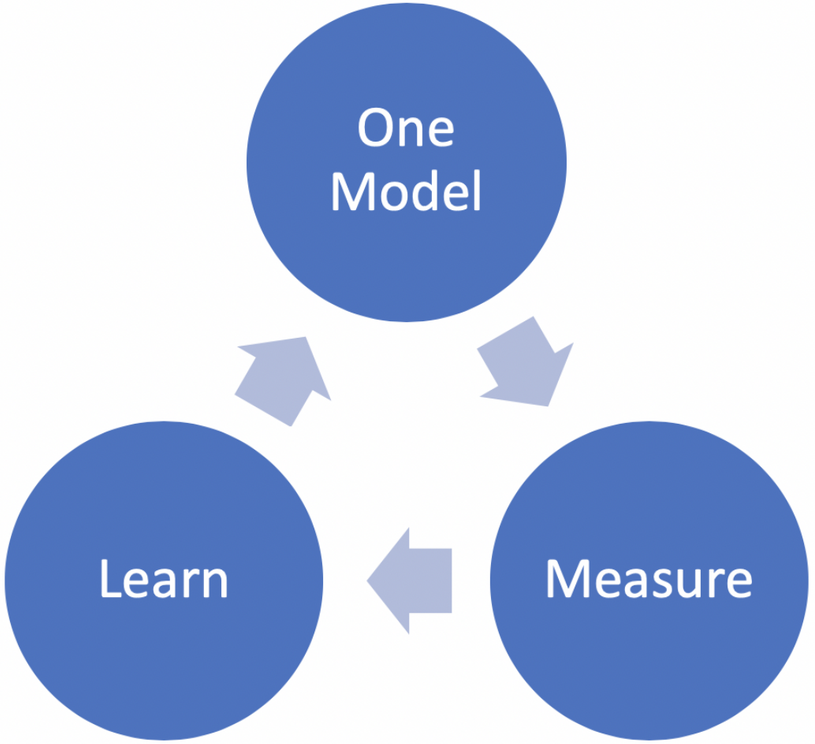
This is what allows us to parallelize our work. Though this doesn't sound very efficient?

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

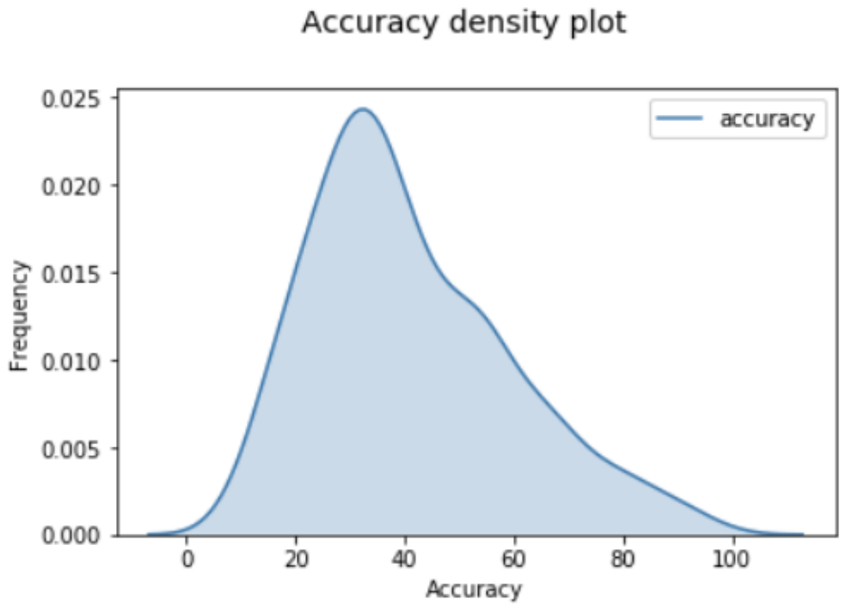
How many possible models do we have?

```
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?

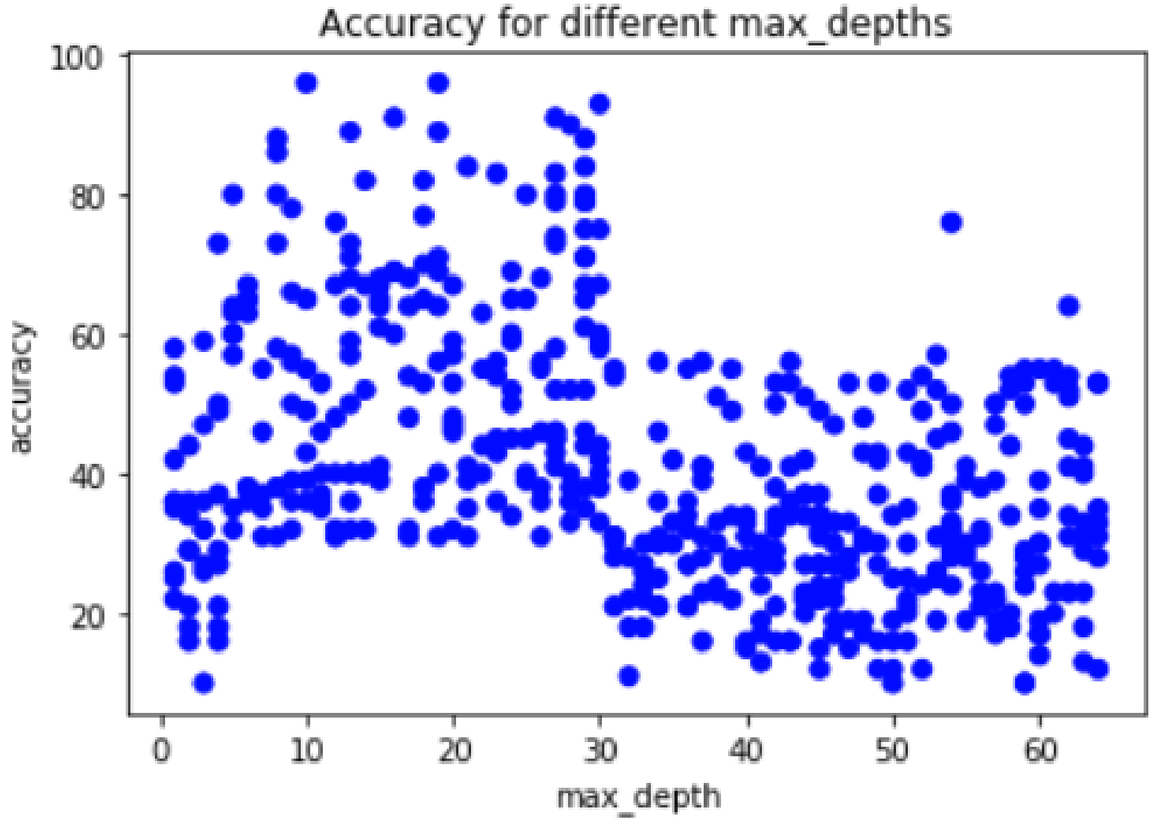
Visualizing Coarse to Fine

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

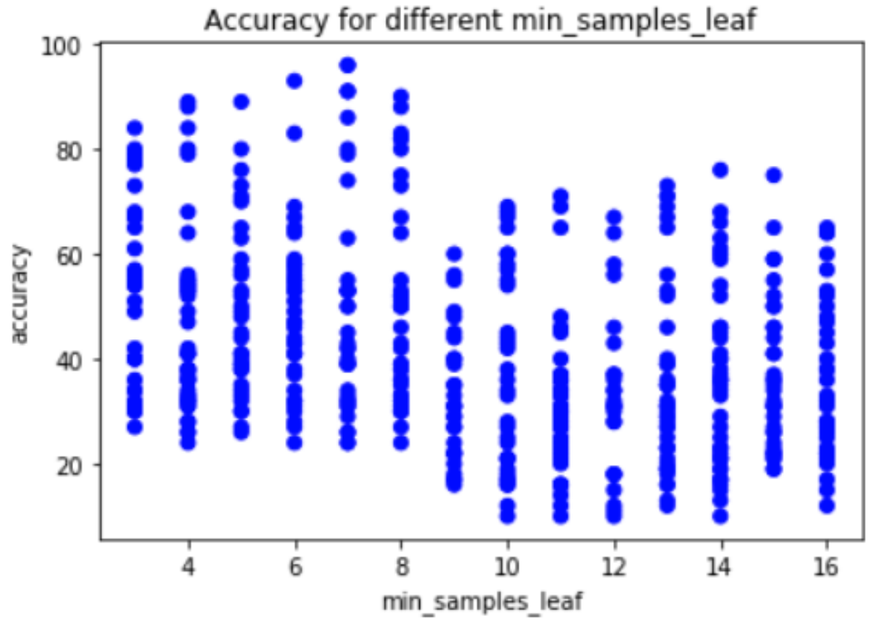
Visualizing Coarse to Fine

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

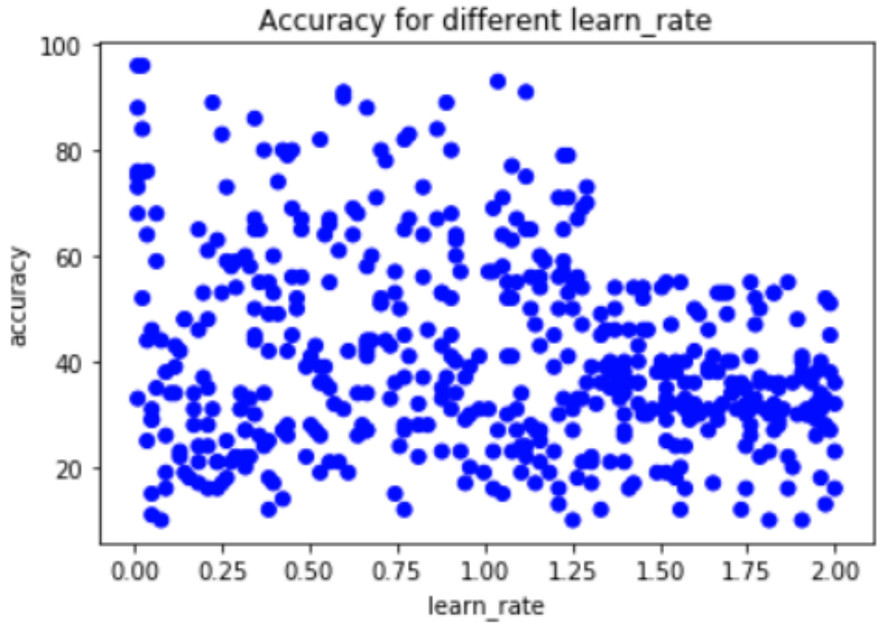


Visualizing coarse to Fine

`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!

Note: This was only *bivariate* analysis. You can explore looking at multiple hyperparameters (3, 4 or more!) on a single graph, but that's beyond the scope of this course.

Let's practice!

HYPERPARAMETER TUNING IN PYTHON

Informed Methods: Bayesian Statistics

HYPERPARAMETER TUNING IN PYTHON

Alex Scriven
Data Scientist

datacamp

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)
```

Let's practice!

HYPERPARAMETER TUNING IN PYTHON

Informed Methods: Genetic Algorithms

HYPERPARAMETER TUNING IN PYTHON

Alex Scriven
Data Scientist

datacamp

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.

Notice: No algorithm-specific hyperparamaters?

Let's practice!

HYPERPARAMETER TUNING IN PYTHON

Wrap up

HYPERPARAMETER TUNING IN PYTHON

Alex Scriven
Data Scientist

datacamp

Hyperparameters vs Parameters

Hyperparameters vs 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

Which hyperparameters & values?

You learned:

- Some hyperparameters are better to start with than others
- There are silly values you can set for hyperparameters
- You need to beware of conflicting hyperparameters
- Best practice is specific to algorithms and their hyperparameters

Remembering Grid Search

We introduced grid search:

- Construct a matrix (or 'grid') of hyperparameter combinations and values
- Build models for **all** the different hyperparameter combinations
- Then pick the winner

A computationally expensive option but is guaranteed to find the best in your grid. (Remember the importance of setting a good grid!)

Remembering Random Search

Random Search:

- Very similar to grid search
- Main difference is selecting (n) random combinations.

This method is faster at getting a *reasonable* model but will not get the *best* in your grid.

From uninformed to informed search

Looking at informed search:

> In informed search, each iteration learns from the last, whereas in Grid and Random, modeling is all done at once and then the best is picked.

Informed methods explored were:

- 'Coarse to Fine' (Iterative random then grid search)
- Bayesian hyperparameter tuning, updating beliefs using evidence on model performance
- Genetic algorithms, evolving your models over generations.

Thank you!

HYPERPARAMETER TUNING IN PYTHON