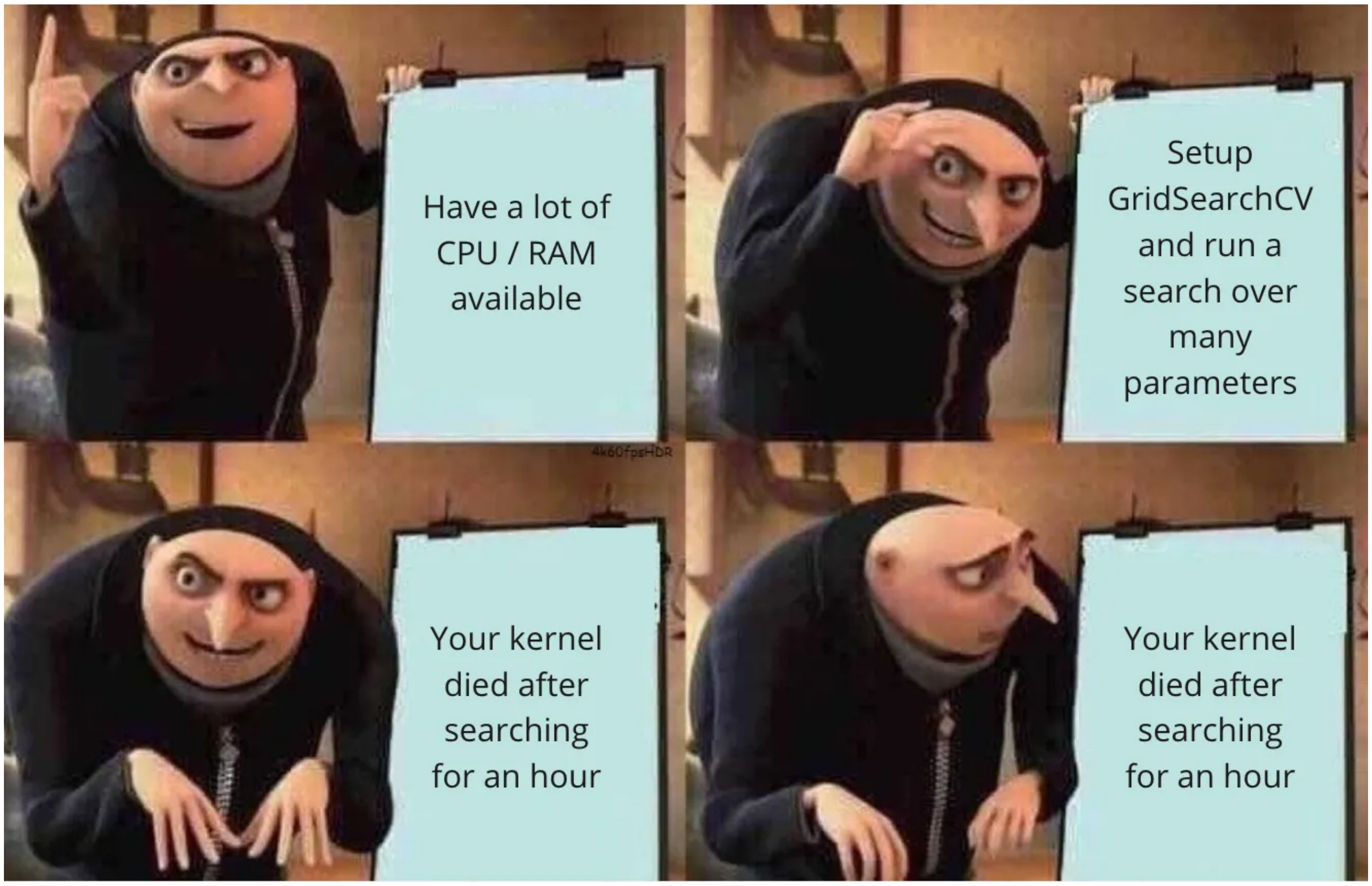
Grid search and random search are outdated. This approach outperforms both.

If you're a data scientist, there is a good chance you have used "Grid Search" to fine-tune the hyperparameters of your model. This is a standard approach available in scikit-learn, which takes a set of values for every parameter as a search space, tries all the possible combinations of those parameters and chooses the ones that result in the best cross-validation performance. Or you may have used Random Search, which randomly selects a few of the values in the search space for a predefined number of times.

Computation power and time

Applicable to Grid Search

The main problem with grid search is that it takes a lot of time and computing power. Since it checks every possible combination of settings, it can mean evaluating many models, especially when there are a lot of settings to play with and a wide range of values for each one. This can make finding the best settings drag on and on, making it too slow for some models and data.

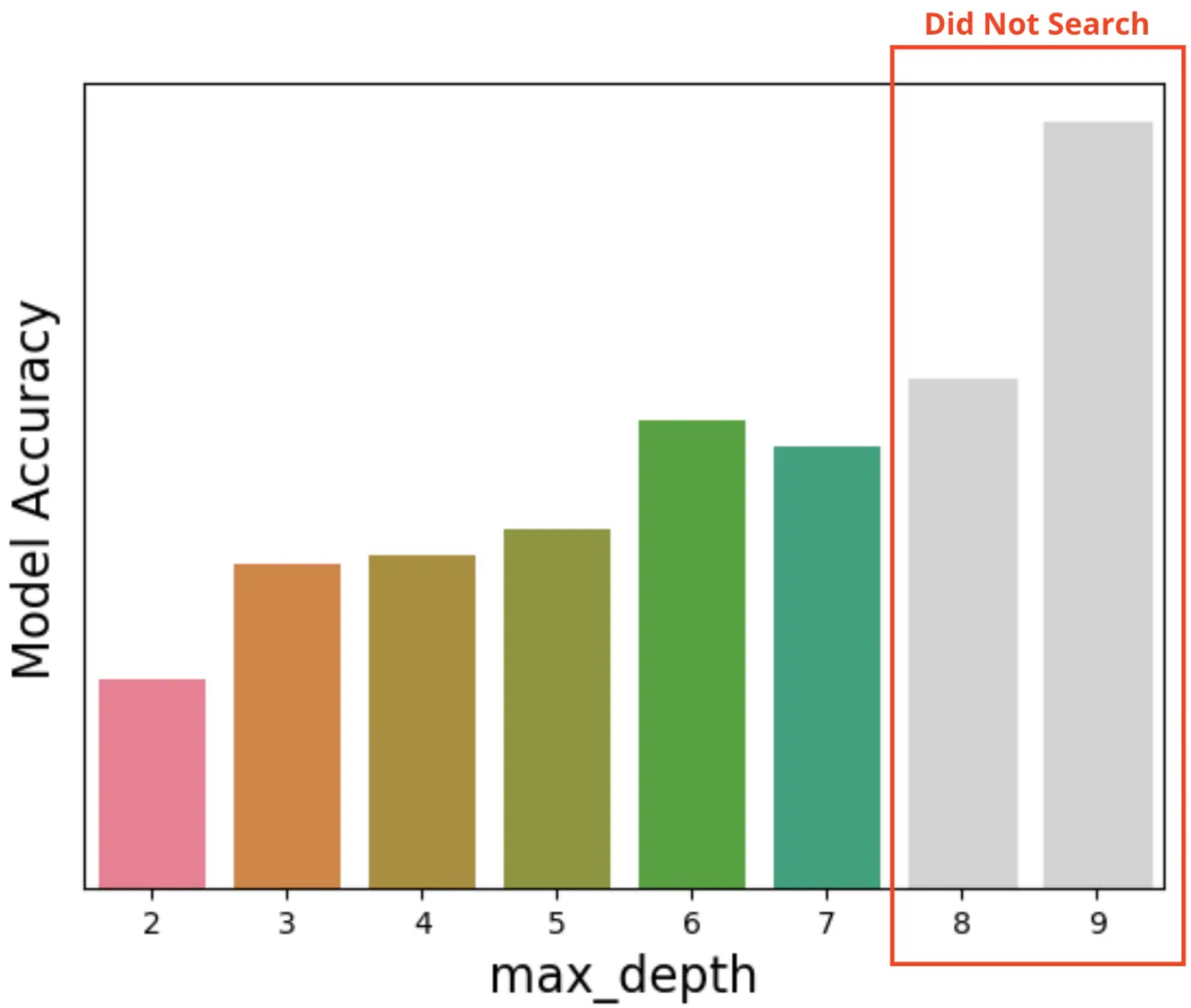

Have a lot of CPU / RAM available
Setup GridSearchCV and run a search over many parameters
Your kernel died after searching for an hour
Your kernel died after searching for an hour

We’ve all been there!

Limited and Biased Search

Applicable to both Grid and Random Search

But wait! What if the best hyperparameters for the model are not even in your search space? What if the best max_depth for that XGBoost model is 9, but you searched over [4, 5, 6, 7]?



Plot by the author. Values are only for demonstration purposes

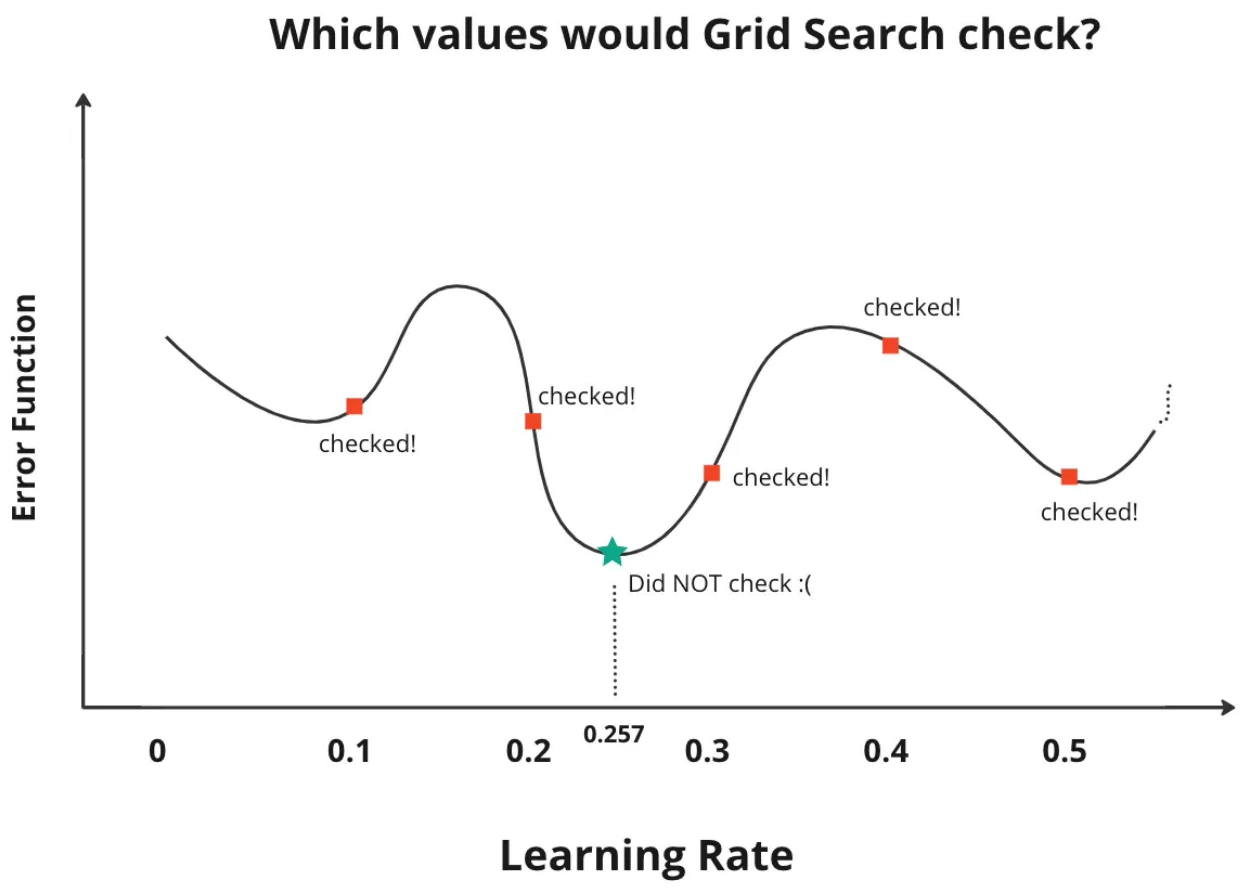
When choosing the search space for Grid Search, data scientists usually reflect on their previous successful hyperparameter tuning jobs and tend to repeatedly choose the same parameters and ranges for different projects. As a result, Grid Search inherits the users' bias in their perception of the optimal search space for the best hyperparameters. On the other hand, **expanding the search space further will exponentially increase the processing time for Grid Search**.

Discrete Search

Applicable to both Grid and Random Search

Many of the hyperparameters are continuous in nature, such as the learning rate for gradient-boosted trees and neural networks or ccp_alpha for any of the tree-based models. The only way we can use Grid Search to fine-tune these parameters is to choose a few discrete values and feed them into Grid Search. This method inevitably misses out on all the values in between the chosen values in the grid, possibly missing promising ones for the hyperparameters.

For example, if we want to search for optimal "learning_rate" for an XGBoost model over the range of (0, 1), we may choose a few values for a search like this:
'learning_rate' : np.arange(0.1, 1, 0.1)
But what if the best value for your learning_rate is 0.257?



Plot by the author. The red dots are the discrete values that Grid Search tries, but the best value will not be discovered.

A Better Approach: Bayesian Search

One alternative to grid search is Bayesian Optimization.

In Layman's terms, Bayesian Optimization uses Bayesian statistics to estimate the distribution of the best hyperparameters for the model instead of just using grid search or random search. During the tuning process, the algorithm updates its beliefs about the distribution of the best hyperparameters based on each hyperparameter's observed impact on the model's performance. This allows it to gradually converge on the optimal set of hyperparameters, resulting in better performance on the test set.

Once you see this in action, it makes a lot more sense. (Credit to Andrew Ng, our ML Master)

Case Study

Method 1: Grid Search

This case study will use the bike-sharing dataset you can find here.

After basic data cleaning, it's time to train an XGBoost regressor and perform a Grid Search on a few of the hyperparameters. We will also time the process to compare.

```
%%time
# Choose the type of classifier.
xgb_tuned = XGBRegressor(random_state=1)

# Grid of parameters to choose from
```

```
parameters = {
    "n_estimators": [10, 50, 100],
    "subsample":[0.6, 0.8, 1],
    "learning_rate":[0.01, 0.1, 0.5, 1],
    "gamma":[0.01, 0.1, 1, 5],
    "colsample_bytree":[0.5, 0.7, 0.9, 1],
    "alpha":[0, 0.1, 0.5]
}

# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.r2_score)

# Run the grid search
grid_obj = GridSearchCV(xgb_tuned, parameters, scoring=scorer,
                        cv=5, n_jobs=-1, verbose=2)

grid_obj = grid_obj.fit(X_train, y_train)
```

Method 2: Randomized Search

we will use the same parameters for Randomized Search, and we will run it for 20 iterations. We expect this to be fast, but not as accurate as Grid Search.

```
# Run the grid search
rand_obj = RandomizedSearchCV(xgb_tuned, parameters, scoring=scorer,
                              n_iter=20, n_jobs=-1, cv=5, verbose=1)

rand_obj = rand_obj.fit(X_train, y_train)
```

Method 3: Bayesian Search on the same search space as Grid Search

Now it’s time to use Bayesian Search with just a few lines of code. Make sure to install scikit-optimize first.

```
pip install scikit-optimize
```

```
%%time
from skopt import BayesSearchCV
bayes = BayesSearchCV(XGBRegressor(random_state=1),
                      search_spaces=parameters, # same space as GridSearch
                      n_iter=20, cv=5)

bayes.fit(X_train, y_train)
```

Note that for the Bayesian Search, we are using the exact same hyperparameter search space as the Grid Search to compare the processing time apples to apples.

We are only running this for 10 iterations. The Bayesian Search will estimate the model's performance based on the prescribed hyperparameters during each of the 10 iterations and use that estimate to select the hyperparameters for the subsequent iteration.

Method 4: Using continuous Search Spaces for Bayesian Search

As mentioned before, using discrete values to represent a continuous hyperparameter is suboptimal. Scikit-Optimize allows us to define a continuous range with a predefined distribution (such as `'log-uniform'`) and specify upper and lower bounds for each hyperparameter.

```
from skopt.space import Real, Integer
from skopt.utils import use_named_args
from sklearn.model_selection import cross_val_score

bayes_space = XGBRegressor()

space  = [Integer(1, 20, name='max_depth'),
          Real(10**-5, 10**0, "log-uniform", name='learning_rate'),
          Real(0.5, 1,"uniform", name='subsample'),
          Real(10**-5, 10**1, "uniform", name='gamma'),
          Real(10**-5, 10**0, "uniform", name='alpha'),]

# The decorator below enables the objective function
# to receive the parameters as keyword arguments.
@use_named_args(space)
def objective(**params):
    '''
```

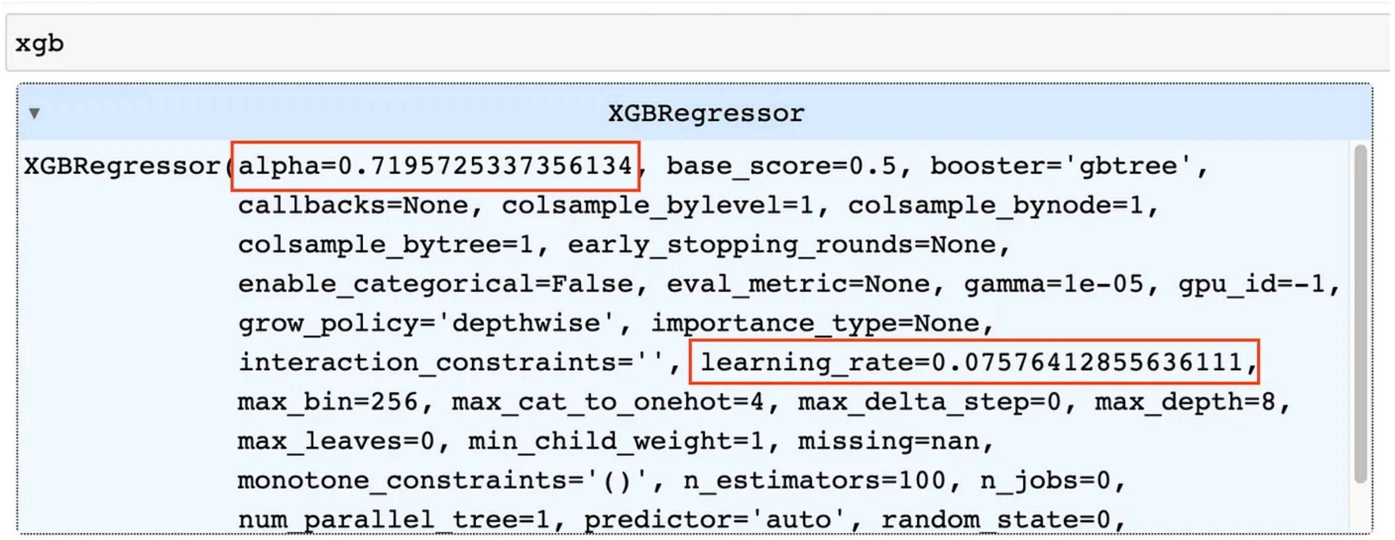
```
        Scitkit Learn Optimize requires an objective function to minimize.
        We use the average of cross-validation mean absolute errors as
        the objective function (also called cost function in optimization)
        '''
        xgb.set_params(**params)

        return -np.mean(cross_val_score(xgb, X, y, cv=5, n_jobs=-1,
                                        scoring="neg_mean_absolute_error"))
```

Every optimization problem has an objective function (sometimes called cost function or error function). In the code above, we defined our objective function as the mean absolute error, so the Bayesian Search will try to minimize that. Once we have an objective function, we need to perform the search to find the set of hyperparameters that minimize our mean absolute error. We can use gp_minimize from scikit-optimize to run this optimization.

```
from skopt import gp_minimize
res_gp = gp_minimize(objective, space, n_calls=20, random_state=0)
```

Once the search is finished, we can inspect the best hyperparameters:



```
xgb
```

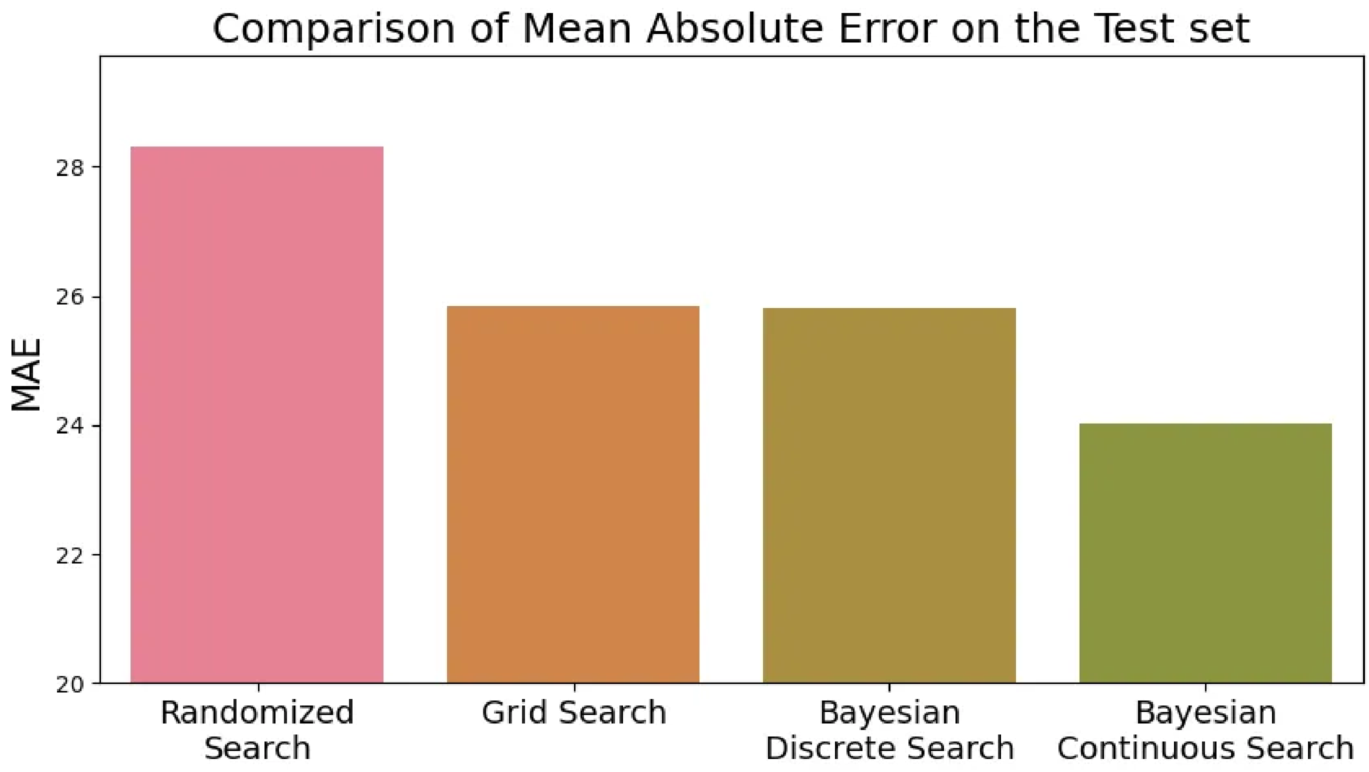
```
▾                              XGBRegressor
XGBRegressor(alpha=0.7195725337356134, base_score=0.5, booster='gbtree',
             callbacks=None, colsample_bylevel=1, colsample_bynode=1,
             colsample_bytree=1, early_stopping_rounds=None,
             enable_categorical=False, eval_metric=None, gamma=1e-05, gpu_id=-1,
             grow_policy='depthwise', importance_type=None,
             interaction_constraints='', learning_rate=0.0757641285563611,
             max_bin=256, max_cat_to_onehot=4, max_delta_step=0, max_depth=8,
             max_leaves=0, min_child_weight=1, missing=nan,
             monotone_constraints='()', n_estimators=100, n_jobs=0,
             num_parallel_tree=1, predictor='auto', random_state=0,
```

As you can see, the best values for learning_rate or alpha (L1 regularization parameter of XGBoost) are decimals that one would never choose in their Grid

Search, yet they ended up being the best values chosen by Bayesian Optimization.

Performance Comparison

The error of the XGBoost model tuned by each of the methods discussed is plotted below. As you can see, the best result was obtained by using Bayesian Search on a continuous search space. It is noteworthy that a MAE of 24 was associated with adjusted R squared of 95%.

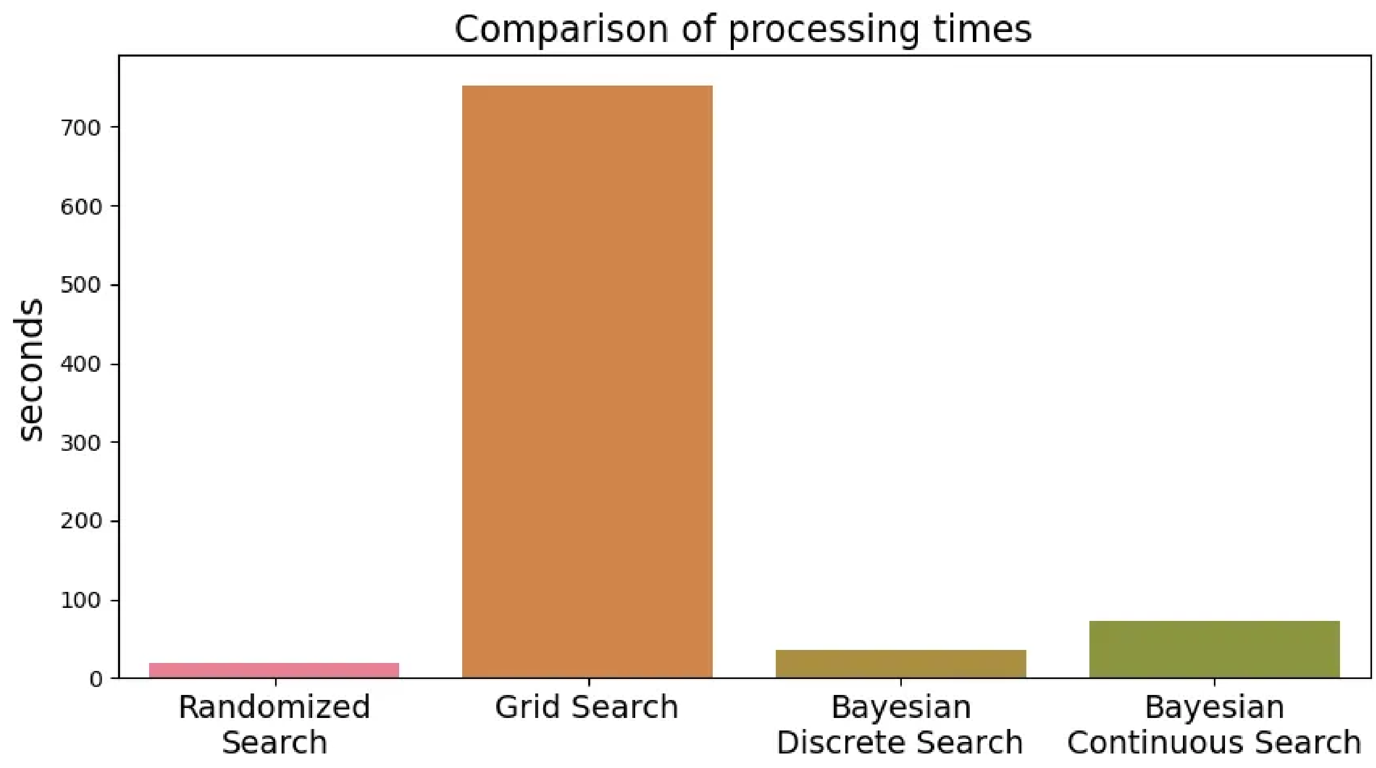


Comparison of mean absolute errors (lower is better. duh…). Plot by the author

Also, doing Bayesian Search on the same search space as Grid Search resulted in the same (within 1%) accuracy as well, and it achieved the same performance as Grid Search in only 10 iterations as opposed to trying all the possible combinations. The next section highlights the impact of this difference on the processing time.

Processing Time Comparison

As shown in the plot below, Grid Search took roughly 20x more time than Bayesian search to find the best set of hyperparameters in the SAME search space. Even Bayesian Search over a continuous space finished 10x faster than Grid Search while resulting in a much better performance.



Comparing processing times. Plot by the author

Summary

This article discussed the main drawbacks of Grid Search and Random Search in scikit-learn. We introduced Bayesian Search as a viable alternative, which instead of trying all the possible combinations of hyperparameters or a random selection thereof, uses a Bayesian probability-based model to predict the best hyperparameters from a given search space.

	Model Accuracy	Waiting Time
Grid Search	High	Very High
Bayesian Search on Discrete Parameters	High	Very Low
Bayesian Search on Contnuous Parameters	Very High	Low