Systematic hyperparameter tuning using GridSearchCV and RandomizedSearch CV

In the wonderful MLZoomCamp course, we had many occasions where we needed to tune the hyperparameters of a machine learning model to understand and improve its performance on a given dataset. The way we usually went about it was something like the following:

- Select one hyperparameter
- Iterate through a range of values of this hyperparameter
- Evaluate the performance for each value, using a specific metric (e.g. accuracy)
- Plot & analyze results, choosing the “optimal” value of the parameter
- Repeat for all hyperparameters you want to optimize
- Build model with the combination of individually “optimal” hyperparameters

This is a very valuable approach in practice, and is especially useful to learn about individual hyperparameters and their impact, in line with the goals of the course. However, when it comes to real world, there are a few drawbacks if we need to scale this up:

- The combination of the individually best values of hyperparameters need not be the same as the best combination of all hyperparameters, which is what we actually need
- Manually coding loops to evaluate all possible combinations of hyperparameters is tedious and error-prone, and
- Depending on a single training and validation split is not very robust, as the performance of models might depend on a particular split, and not generalize well.

In the following sections, we’ll discuss GridSearchCV and RandomizedSearchCV, which are methods provided by Scikit-Learn to address the above issues. In order to illustrate the use of GridSearchCV and RandomizedSearchCV, we’ll build upon the code and also use the dataset from Week 6 of the MLZoomCamp course. I shall skip the steps involved in reading in and preprocessing the data, as they can be found in the official Week 6 notebook (https://github.com/alexeygrigorev/mlbookcamp-code/blob/master/course-zoomcamp/06-trees/notebook.ipynb).

Model	Chosen Hyperparameters	Accuracy on validation set
Decision Tree	max_depth=6, min_samples_leaf=15	0.7853
Random Forest	n_estimators=200,max_depth=10,min_samples_leaf=3,random_state=1	0.8246
Xgboost	‘eta’: 0.1, ‘max_depth’: 3, ‘min_child_weight’: 1, ‘objective’: ‘binary:logistic’, ‘eval_metric’: ‘auc’, ‘nthread’: 8, ‘seed’: 1, ‘verbosity’: 1	0.8360

Xgboost was the best model, and after retraining using the full training set, achieved 0.8323 accuracy on the test set. We’ll use these results for reference as we look at the results obtained after tuning hyperparameters using GridSearchCV and RandomizedSearchCV.

Hyperparameter Tuning with GridSearchCV

GridSearchCV (https://scikit-learn.org/stable/modules/grid_search.html#grid-search) is a Scikit-Learn function to exhaustively evaluate all combinations of specific hyperparameter values (specified via input) - “Grid” refers to the points comprising the finite subspace of the hyperparameter space represented by the input values, and “CV” refers to the fact that GridSearchCV evaluates hyperparameter combinations in a cross-validation setting, making performance estimates more robust. We can specify discrete values or ranges for each individual hyperparameter using dictionaries, and we can evaluate multiple models, as shown in the code snippets below:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
```

```
model_params = {
    'random_forest': {
        'model': RandomForestClassifier(),
        'params': {
            'n_estimators': [10,50,100,150,200],
            'max_depth' : [10,50,100,150,200],
            'min_samples_leaf':[1, 3, 5, 10, 50]
        }
    },
    'xgboost': {
        'model': XGBClassifier(),
        'params': {
            'max_depth' : [3, 6, 10],
            'eta' : [0.01,0.1,0.3],
            'min_child_weight' : [None , 1 , 2 , 3]
        }
    },
    'decision_tree': {
        'model': DecisionTreeClassifier(),
        'params': {
            'max_depth' : [1, 3, 5, 7, 10, 15, 20, None],
            'min_samples_leaf' :[1, 5, 10, 15, 20, 500, 100, 200],
        }
    }
}
```

```
scores = []

for model_name, mp in model_params.items():
    clf =  GridSearchCV(mp['model'], mp['params'], cv=5, return_train_score=False, scori
ng = 'roc_auc')
    clf.fit(X_train, y_train)
    scores.append({
        'model': model_name,
        'best_score': clf.best_score_,
        'best_params': clf.best_params_
    })

gridsearch_results = pd.DataFrame(scores,columns=['model','best_score','best_params'])
gridsearch_results
```

	model	best_score	best_params
0	random_forest	0.844150	{'max_depth': 10, 'min_samples_leaf': 3, 'n_es…
1	xgboost	0.853449	{'eta': 0.1, 'max_depth': 3, 'min_child_weight…
2	decision_tree	0.794940	{'max_depth': 7, 'min_samples_leaf': 100}

```
gridsearch_results
```

	model	best_score	best_params
0	random_forest	0.844150	{'max_depth': 10, 'min_samples_leaf': 3, 'n_es…
1	xgboost	0.853449	{'eta': 0.1, 'max_depth': 3, 'min_child_weight…
2	decision_tree	0.794940	{'max_depth': 7, 'min_samples_leaf': 100}

```
gridsearch_results['best_params'][0]
```

```
{'max_depth': 10, 'min_samples_leaf': 3, 'n_estimators': 100}
```

```
rf_best_grid = RandomForestClassifier(criterion='gini', max_depth= 10,
                                     min_samples_leaf= 5,
                                     n_estimators=150)
```

```
rf_best_grid.fit(X_train, y_train)
```

```
▾ RandomForestClassifier
RandomForestClassifier(max_depth=10, min_samples_leaf=5, n_estimators=150)
```

```
y_pred = rf_best_grid.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)
```

```
0.8246379356276188
```

```
gridsearch_results['best_params'][1]
```

```
{'eta': 0.1, 'max_depth': 3, 'min_child_weight': 2}
```

```
xgbc_best_grid = XGBClassifier(max_depth=3, min_child_weight=2,eta=0.1)
xgbc_best_grid.fit(X_train, y_train)
```

```
▾                          XGBClassifier
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eta=0.1, eval_metric=None,
              feature_types=None, gamma=None, gpu_id=None, grow_policy=None,
              importance_type=None, interaction_constraints=None,
              learning_rate=None, max_bin=None, max_cat_threshold=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=3,
              max_leaves=None, min_child_weight=2, missing=nan,
              monotone_constraints=None, n_estimators=100, n_jobs=None,
              num_parallel_tree=None, predictor=None, ...)
```

```
y_pred = xgbc_best_grid.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)
```

```
0.8286218546414473
```

Let’s define the Decision Tree with best parameters & evaluate on validation data:

```
gridsearch_results['best_params'][2]
```

```
{'max_depth': 7, 'min_samples_leaf': 100}
```

```
dt_best_grid = DecisionTreeClassifier(max_depth=7, min_samples_leaf=100,criterion='gin
i')
dt_best_grid.fit(X_train, y_train)
```

```
▾             DecisionTreeClassifier
DecisionTreeClassifier(max_depth=7, min_samples_leaf=100)
```

```
y_pred = dt_best_grid.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)
```

```
0.7795827177835364
```

xgboost is clearly the best model, let's evaluate it on test data:

```
y_pred = xgbc_best_grid.predict_proba(X_test)[:, 1]
roc_auc_score(y_test, y_pred)
```

```
0.8304029617320756
```

Finally, let's train xgboost on the full training data & then run it on test data:

```
df_full_train = df_full_train.reset_index(drop=True)
y_full_train = (df_full_train.status == 'default').astype(int).values
del df_full_train['status']

dicts_full_train = df_full_train.to_dict(orient='records')

dv = DictVectorizer(sparse=False)
X_full_train = dv.fit_transform(dicts_full_train)
```

```
xgbc_best_grid.fit(X_full_train, y_full_train)
```

▾ **XGBClassifier**

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eta=0.1, eval_metric=None,
              feature_types=None, gamma=None, gpu_id=None, grow_policy=None,
              importance_type=None, interaction_constraints=None,
              learning_rate=None, max_bin=None, max_cat_threshold=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=3,
              max_leaves=None, min_child_weight=2, missing=nan,
              monotone_constraints=None, n_estimators=100, n_jobs=None,
              num_parallel_tree=None, predictor=None, ...)
```

```
y_pred = xgbc_best_grid.predict_proba(X_test)[:, 1]
roc_auc_score(y_test, y_pred)
```

```
0.8334331166609648
```

We see that the xgbclassifier trained on the full dataset achieves an accuracy of 83.34% on the test dataset, which is very close to (& slightly higher than) the 83.27% achieved in the official Week 6 notebook by xgboost, which is a more highly optimized version of the same algorithm.

In summary, GridsearchCV allows us to systematically traverse the grid representing all hyperparameter choices that we specify, guaranteeing that we get the combination yielding the best performance according to robust cross-validation based estimates of the performance metric we choose. However, combinatorial growth means that GridSearchCV is computationally expensive - we next consider RandomizedSearchCV, which trades performance for speed, and often yields performance that is acceptably close to that provided by GridSearchCV.

Faster, not better - but probably good enough : Hyperparameter Tuning with RandomizedSearchCV

RandomizedSearchCV is similar to GridsearchCV, but it only searches a random subspace of the hyperparameter space, which makes it faster than GridSearchCV at the cost of a drop in performance

that is often acceptable in practice. We can define the parameter grid exactly like we did for GridSearchCV.

```
model_params = {
    'random_forest': {
        'model': RandomForestClassifier(),
        'params': {
            'n_estimators': [10,50,100,150,200],
            'max_depth' : [10,50,100,150,200],
            'min_samples_leaf':[1, 3, 5, 10, 50]
        }
    },
    'xgboost': {
        'model': XGBClassifier(),
        'params': {
            'max_depth' : [3, 6, 10],
            'eta' : [0.01,0.1,0.3],
            'min_child_weight' : [None , 1 , 2 , 3]
        }
    },
    'decision_tree': {
        'model': DecisionTreeClassifier(),
        'params': {
            'max_depth' : [1, 3, 5, 7, 10, 15, 20, None],
            'min_samples_leaf' :[1, 5, 10, 15, 20, 500, 100, 200],
        }
    }
}
```

```
scores_rand = []

for model_name, mp in model_params.items():
    clf =  RandomizedSearchCV(mp['model'], mp['params'], cv=5, return_train_score=False,
scoring = 'roc_auc')
    clf.fit(X_train, y_train)
    scores_rand.append({
        'model': model_name,
        'best_score': clf.best_score_,
        'best_params': clf.best_params_
    })

randsearch_results = pd.DataFrame(scores_rand,columns=['model','best_score','best_param
s'])
randsearch_results
```

	model	best_score	best_params
0	random_forest	0.842455	{‘n_estimators’: 200, ‘min_samples_leaf’: 10, …
1	xgboost	0.852028	{‘min_child_weight’: 3, ‘max_depth’: 3, ‘eta’:…

	model	best_score	best_params
2	decision_tree	0.794712	{‘min_samples_leaf’: 100, ‘max_depth’: None}

```
randsearch_results['best_params'][0]
```

```
{'n_estimators': 200, 'min_samples_leaf': 10, 'max_depth': 100}
```

```
rf_best_rand = RandomForestClassifier(max_depth= 50,
                                      min_samples_leaf= 5,
                                      n_estimators=150)
```

```
rf_best_rand.fit(X_train, y_train)
```

```
▾ RandomForestClassifier
RandomForestClassifier(max_depth=50, min_samples_leaf=5, n_estimators=150)
```

```
y_pred = rf_best_rand.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)
```

```
0.8243594245719406
```

```
randsearch_results['best_params'][1]
```

```
{'min_child_weight': 3, 'max_depth': 3, 'eta': 0.1}
```

```
xgbc_best_rand = XGBClassifier(max_depth=3, min_child_weight=1,eta=0.1)
xgbc_best_rand.fit(X_train, y_train)
```

```
▾ XGBClassifier
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eta=0.1, eval_metric=None,
              feature_types=None, gamma=None, gpu_id=None, grow_policy=None,
              importance_type=None, interaction_constraints=None,
              learning_rate=None, max_bin=None, max_cat_threshold=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=3,
              max_leaves=None, min_child_weight=1, missing=nan,
              monotone_constraints=None, n_estimators=100, n_jobs=None,
              num_parallel_tree=None, predictor=None, ...)
```

```
y_pred = xgbc_best_rand.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)
```

```
0.8319155506042479
```

```
randsearch_results['best_params'][2]
```

```
{'min_samples_leaf': 100, 'max_depth': None}
```

```
dt_best_rand = DecisionTreeClassifier(max_depth=15, min_samples_leaf=100)
dt_best_rand.fit(X_train, y_train)
```

▾ DecisionTreeClassifier

```
DecisionTreeClassifier(max_depth=15, min_samples_leaf=100)
```

```
y_pred = dt_best_rand.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)
```

```
0.7797250006054588
```

```
y_pred = xgbc_best_rand.predict_proba(X_test)[:, 1]
roc_auc_score(y_test, y_pred)
```

```
0.8293460730169591
```

```
xgbc_best_rand.fit(X_full_train, y_full_train)
```

▾ XGBClassifier

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eta=0.1, eval_metric=None,
              feature_types=None, gamma=None, gpu_id=None, grow_policy=None,
              importance_type=None, interaction_constraints=None,
              learning_rate=None, max_bin=None, max_cat_threshold=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=3,
              max_leaves=None, min_child_weight=1, missing=nan,
              monotone_constraints=None, n_estimators=100, n_jobs=None,
              num_parallel_tree=None, predictor=None, ...)
```

```
y_pred = xgbc_best_rand.predict_proba(X_test)[:, 1]
roc_auc_score(y_test, y_pred)
```

```
0.8346916084257856
```

Thus, we see that the results obtained using RandomizedSearchCV are in general competitive with those obtained using GridSearchCV, even though the latter is guaranteed to find the best combination of the specified input hyperparameter values. Moreover, since RandomizedSearchCV is much faster, we can provide it a wider range of input hyperparameter values, increasing the chancing of finding combinations that result in even better performance.

So far, we have only talked of tuning classical machine learning models - what about deep learning models? It turns out that GridSearchCV and RandomizedSearchCV can be used to tune deep learning models as well, but the Keras library provides a bespoke method, called Keras-Tuner, that is specifically designed to make tuning deep learning models easier - and it can even tune classical ML models. However, that's a story for another day - hope you found this brief article useful, and are now motivated to try out GridSearchCV, RandomizedSearchCV, and other hyperparameter tuning methods provided by Scikit-Learn. Have fun (Machine) learning!