Tune Hyperparameters for Classification Machine Learning Algorithms

by **Jason Brownlee** on December 13, 2019 in **Python Machine Learning**

Last Updated on August 28, 2020

Machine learning algorithms have hyperparameters that allow you to tailor the behavior of the algorithm to your specific dataset.

Hyperparameters are different from parameters, which are the internal coefficients or weights for a model found by the learning algorithm. Unlike parameters, hyperparameters are specified by the practitioner when configuring the model.

Typically, it is challenging to know what values to use for the hyperparameters of a given algorithm on a given dataset, therefore it is common to use random or grid search strategies for different hyperparameter values.

The more hyperparameters of an algorithm that you need to tune, the slower the tuning process. Therefore, it is desirable to select a minimum subset of model hyperparameters to search or tune.

Not all model hyperparameters are equally important. Some hyperparameters have an outsized effect on the behavior, and in turn, the performance of a machine learning algorithm.

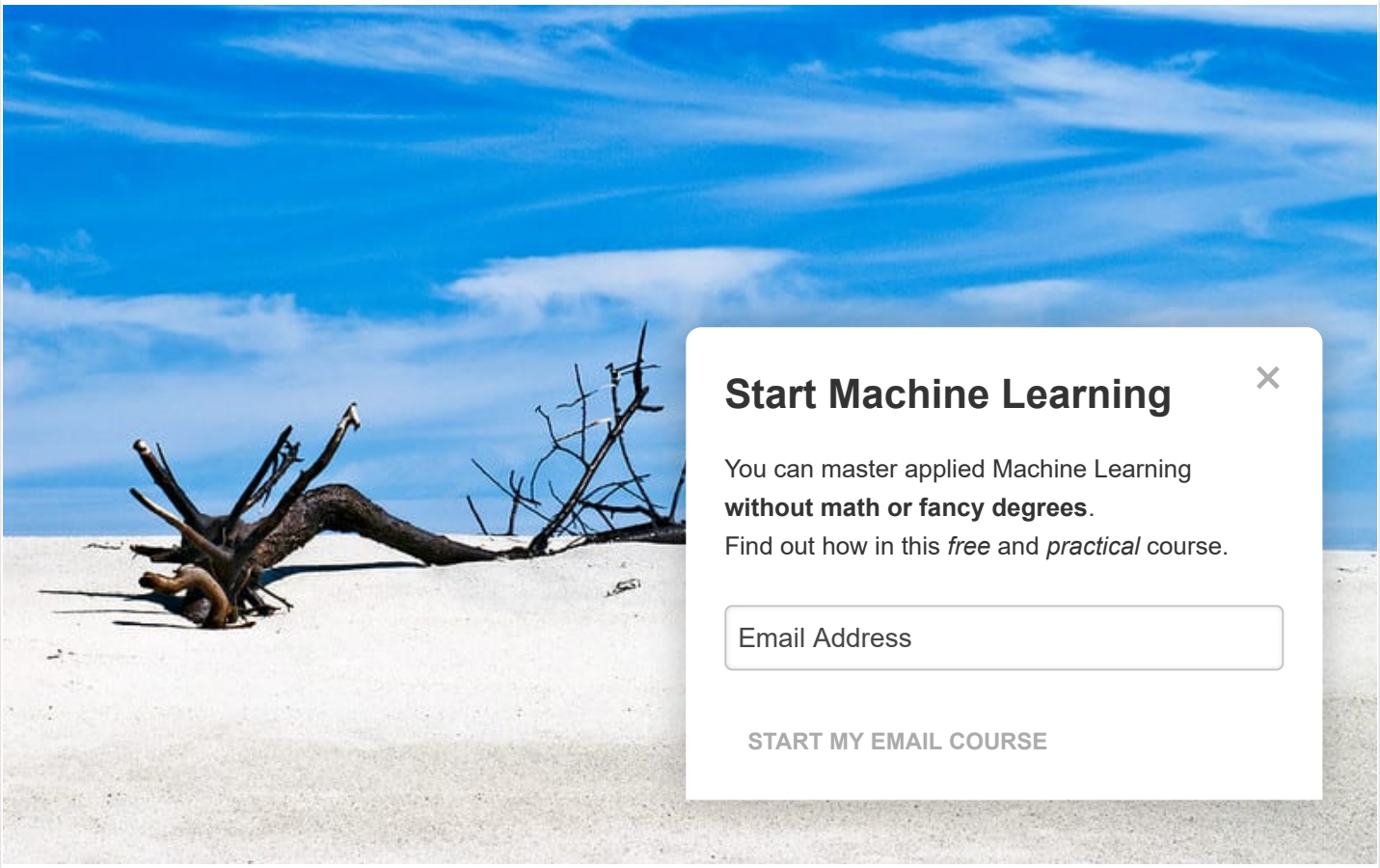
As a machine learning practitioner, you must know which hyperparameters to focus on to get a good result quickly.

In this tutorial, you will discover those hyperparameters that are most important for some of the top machine learning algorithms.

Kick-start your project with my new book Machine Learning Mastery With Python, including *step-by-step tutorials* and the *Python source code* files for all examp

Let's get started.

- **Update Jan/2020**: Updated for changes in scikit-learn v0.22 API.



Hyperparameters for Classification Machine Learning Algorithms
Photo by shuttermonkey, some rights reserved.

Classification Algorithms Overview

We will take a closer look at the important hyperparameters of the top machine learning algorithms that you may use for classification.

We will look at the hyperparameters you need to focus on and suggested values to try when tuning the model on your dataset.

The suggestions are based both on advice from textbooks on the algorithms and practical advice suggested by practitioners, as well as a little of my own experience.

The seven classification algorithms we will look at are as follows:

1. Logistic Regression
2. Ridge Classifier
3. K-Nearest Neighbors (KNN)
4. Support Vector Machine (SVM)
5. Bagged Decision Trees (Bagging)

6. Random Forest
7. Stochastic Gradient Boosting

We will consider these algorithms in the context of their scikit-learn implementation (Python); nevertheless, you can use the same hyperparameter suggestions with other platforms, such as Weka and R.

A small grid searching example is also given for each algorithm that you can use as a starting point for your own classification predictive modeling project.

Note: if you have had success with different hyperparameter values or even different hyperparameters than those suggested in this tutorial, let me know in the comments below. I'd love to hear about it.

Let's dive in.

Logistic Regression

Logistic regression does not really have any critical hy

Sometimes, you can see useful differences in perform).

- **solver** in ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'sa

Regularization (*penalty*) can sometimes be helpful.

- **penalty** in ['none', 'l1', 'l2', 'elasticnet']

Note: not all solvers support all regularization terms.

The C parameter controls the penality strength, which can also be effective.

- **C** in [100, 10, 1.0, 0.1, 0.01]

For the full list of hyperparameters, see:

- sklearn.linear_model.LogisticRegression API.

The example below demonstrates grid searching the key hyperparameters for LogisticRegression on a synthetic binary classification dataset.

Some combinations were omitted to cut back on the warnings/errors.

```
# example of grid searching key hyperparametres for logistic regression
from sklearn.datasets import make_blobs
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
# define dataset
X, y = make_blobs(n_samples=1000, centers=2, n_features=100, cluster_std=20)
# define models and parameters
model = LogisticRegression()
solvers = ['newton-cg', 'lbfgs', 'liblinear']
```

```
penalty = ['l2']
c_values = [100, 10, 1.0, 0.1, 0.01]
# define grid search
grid = dict(solver=solvers,penalty=penalty,C=c_values)
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='accurac
grid_result = grid_search.fit(X, y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running th
outcome.

Running the example prints the best result as well as

```
Best: 0.945333 using {'C': 0.01, 'penalty': '
0.936333 (0.016829) with: {'C': 100, 'penalty
0.937667 (0.017259) with: {'C': 100, 'penalty
0.938667 (0.015861) with: {'C': 100, 'penalty
0.936333 (0.017413) with: {'C': 10, 'penalty'
0.938333 (0.017904) with: {'C': 10, 'penalty'
0.939000 (0.016401) with: {'C': 10, 'penalty'
0.937333 (0.017114) with: {'C': 1.0, 'penalty
0.939000 (0.017195) with: {'C': 1.0, 'penalty
0.939000 (0.015780) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'liblinear'}
0.940000 (0.015706) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cg'}
0.940333 (0.014941) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'lbfgs'}
0.941000 (0.017000) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}
0.943000 (0.016763) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cg'}
0.943000 (0.016763) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'lbfgs'}
0.945333 (0.017651) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'liblinear'}
```

Ridge Classifier

Ridge regression is a penalized linear regression model for predicting a numerical value.

Nevertheless, it can be very effective when applied to classification.

Perhaps the most important parameter to tune is the regularization strength (*alpha*). A good starting point might be values in the range [0.1 to 1.0]

- **alpha** in [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]

For the full list of hyperparameters, see:

- sklearn.linear_model.RidgeClassifier API.

The example below demonstrates grid searching the key hyperparameters for RidgeClassifier on a synthetic binary classification dataset.

```
# example of grid searching key hyperparametres for ridge classifier
from sklearn.datasets import make_blobs
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import RidgeClassifier
# define dataset
X, y = make_blobs(n_samples=1000, centers=2, n_features=100, cluster_std=20)
# define models and parameters
model = RidgeClassifier()
alpha = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
# define grid search
grid = dict(alpha=alpha)
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='accurac
grid_result = grid_search.fit(X, y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_sc
stds = grid_result.cv_results_['std_test_scor
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, pa
    print("%f (%f) with: %r" % (mean, stdev,
```


Note: Your results may vary given the stochastic natu
differences in numerical precision. Consider running th
outcome.

Running the example prints the best result as well as

```
Best: 0.974667 using {'alpha': 0.1}
0.974667 (0.014545) with: {'alpha': 0.1}
0.974667 (0.014545) with: {'alpha': 0.2}
0.974667 (0.014545) with: {'alpha': 0.3}
0.974667 (0.014545) with: {'alpha': 0.4}
0.974667 (0.014545) with: {'alpha': 0.5}
0.974667 (0.014545) with: {'alpha': 0.6}
0.974667 (0.014545) with: {'alpha': 0.7}
0.974667 (0.014545) with: {'alpha': 0.8}
0.974667 (0.014545) with: {'alpha': 0.9}
0.974667 (0.014545) with: {'alpha': 1.0}
```

K-Nearest Neighbors (KNN)

The most important hyperparameter for KNN is the number of neighbors (*n_neighbors*).

Test values between at least 1 and 21, perhaps just the odd numbers.

- **n_neighbors** in [1 to 21]

It may also be interesting to test different distance metrics (*metric*) for choosing the composition of the neighborhood.

- **metric** in ['euclidean', 'manhattan', 'minkowski']

For a fuller list see:

- sklearn.neighbors.DistanceMetric API

It may also be interesting to test the contribution of members of the neighborhood via different weightings (*weights*).

- **weights** in [‘uniform’, ‘distance’]

For the full list of hyperparameters, see:

- sklearn.neighbors.KNeighborsClassifier API.

The example below demonstrates grid searching the key hyperparameters for KNeighborsClassifier on a synthetic binary classification dataset.

```
# example of grid searching key hyperparametres f
from sklearn.datasets import make_blobs
from sklearn.model_selection import RepeatedStrat
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifie
# define dataset
X, y = make_blobs(n_samples=1000, centers=2, n_fe
# define models and parameters
model = KNeighborsClassifier()
n_neighbors = range(1, 21, 2)
weights = ['uniform', 'distance']
metric = ['euclidean', 'manhattan', 'minkowski']
# define grid search
grid = dict(n_neighbors=n_neighbors,weights=weigh
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='accuracy',
grid_result = grid_search.fit(X, y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```


Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example prints the best result as well as the results from all combinations evaluated.

```
Best: 0.937667 using {'metric': 'manhattan', 'n_neighbors': 13, 'weights': 'uniform'}
0.833667 (0.031674) with: {'metric': 'euclidean', 'n_neighbors': 1, 'weights': 'uniform'}
0.833667 (0.031674) with: {'metric': 'euclidean', 'n_neighbors': 1, 'weights': 'distance'}
0.895333 (0.030081) with: {'metric': 'euclidean', 'n_neighbors': 3, 'weights': 'uniform'}
0.895333 (0.030081) with: {'metric': 'euclidean', 'n_neighbors': 3, 'weights': 'distance'}
0.909000 (0.021810) with: {'metric': 'euclidean', 'n_neighbors': 5, 'weights': 'uniform'}
0.909000 (0.021810) with: {'metric': 'euclidean', 'n_neighbors': 5, 'weights': 'distance'}
0.925333 (0.020774) with: {'metric': 'euclidean', 'n_neighbors': 7, 'weights': 'uniform'}
0.925333 (0.020774) with: {'metric': 'euclidean', 'n_neighbors': 7, 'weights': 'distance'}
0.929000 (0.027368) with: {'metric': 'euclidean', 'n_neighbors': 9, 'weights': 'uniform'}
0.929000 (0.027368) with: {'metric': 'euclidean', 'n_neighbors': 9, 'weights': 'distance'}
...
```

Support Vector Machine (SVM)

The SVM algorithm, like gradient boosting, is very popular, very effective, and provides a large number of hyperparameters to tune.

Perhaps the first important parameter is the choice of kernel that will control the manner in which the input variables will be projected. There are many to choose from, but linear, polynomial, and RBF are the most common, perhaps just linear and RBF in practice.

- **kernels** in ['linear', 'poly', 'rbf', 'sigmoid']

If the polynomial kernel works out, then it is a good idea to dive into the degree hyperparameter.

Another critical parameter is the penalty (*C*) that can t
on the shape of the resulting regions for each class. A

- **C** in [100, 10, 1.0, 0.1, 0.001]

For the full list of hyperparameters, see:

- sklearn.svm.SVC API.

The example below demonstrates grid searching the k
classification dataset.


```
# example of grid searching key hyperparametres for SVC
from sklearn.datasets import make_blobs
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
# define dataset
X, y = make_blobs(n_samples=1000, centers=2, n_features=100, cluster_std=20)
# define model and parameters
model = SVC()
kernel = ['poly', 'rbf', 'sigmoid']
C = [50, 10, 1.0, 0.1, 0.01]
gamma = ['scale']
# define grid search
grid = dict(kernel=kernel,C=C,gamma=gamma)
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='accuracy',
grid_result = grid_search.fit(X, y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example prints the best result as well as the results from all combinations evaluated.

```
Best: 0.974333 using {'C': 1.0, 'gamma': 'scale', 'kernel': 'poly'}
0.973667 (0.012512) with: {'C': 50, 'gamma': 'scale', 'kernel': 'poly'}
0.970667 (0.018062) with: {'C': 50, 'gamma': 'scale', 'kernel': 'rbf'}
0.945333 (0.024594) with: {'C': 50, 'gamma': 'scale', 'kernel': 'sigmoid'}
0.973667 (0.012512) with: {'C': 10, 'gamma': 'scale', 'kernel': 'poly'}
0.970667 (0.018062) with: {'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}
0.957000 (0.016763) with: {'C': 10, 'gamma': 'scale', 'kernel': 'sigmoid'}
0.974333 (0.012565) with: {'C': 1.0, 'gamma': 'scale', 'kernel': 'poly'}
0.971667 (0.016948) with: {'C': 1.0, 'gamma': 'scale', 'kernel': 'rbf'}
0.966333 (0.016224) with: {'C': 1.0, 'gamma': 'scale', 'kernel': 'sigmoid'}
0.972333 (0.013585) with: {'C': 0.1, 'gamma': 'scale', 'kernel': 'poly'}
0.974000 (0.013317) with: {'C': 0.1, 'gamma': 'scale', 'kernel': 'rbf'}
0.971667 (0.015934) with: {'C': 0.1, 'gamma': 'scale', 'kernel': 'sigmoid'}
0.972333 (0.013585) with: {'C': 0.01, 'gamma': 'scale', 'kernel': 'poly'}
0.973667 (0.014716) with: {'C': 0.01, 'gamma'
0.974333 (0.013828) with: {'C': 0.01, 'gamma'
```


Bagged Decision Trees (Baggin

The most important parameter for bagged decision tre

Ideally, this should be increased until no further impro

Good values might be a log scale from 10 to 1,000.

- **n_estimators** in [10, 100, 1000]

For the full list of hyperparameters, see:

- sklearn.ensemble.BaggingClassifier API

The example below demonstrates grid searching the key hyperparameters for BaggingClassifier on a synthetic binary classification dataset.

```
# example of grid searching key hyperparameters for BaggingClassifier
from sklearn.datasets import make_blobs
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import BaggingClassifier
# define dataset
X, y = make_blobs(n_samples=1000, centers=2, n_features=100, cluster_std=20)
# define models and parameters
model = BaggingClassifier()
n_estimators = [10, 100, 1000]
# define grid search
grid = dict(n_estimators=n_estimators)
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='accurac
grid_result = grid_search.fit(X, y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, pa
    print("%f (%f) with: %r" % (mean, stdev,
```

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example prints the best result as well as the results from all combinations evaluated.

```
Best: 0.873667 using {'n_estimators': 1000}
0.839000 (0.038588) with: {'n_estimators': 10}
0.869333 (0.030434) with: {'n_estimators': 100}
0.873667 (0.035070) with: {'n_estimators': 1000}
```

Random Forest

The most important parameter is the number of rando
(*max_features*).

You could try a range of integer values, such as 1 to 2

- **max_features** [1 to 20]

Alternately, you could try a suite of different default va

- **max_features** in ['sqrt', 'log2']

Another important parameter for random forest is the

Ideally, this should be increased until no further improvement is seen in the model.

Good values might be a log scale from 10 to 1,000.

- **n_estimators** in [10, 100, 1000]

For the full list of hyperparameters, see:

- sklearn.ensemble.RandomForestClassifier API.

The example below demonstrates grid searching the key hyperparameters for BaggingClassifier on a synthetic binary classification dataset.


```
# example of grid searching key hyperparameters for RandomForestClassifier
from sklearn.datasets import make_blobs
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
# define dataset
X, y = make_blobs(n_samples=1000, centers=2, n_features=100, cluster_std=20)
# define models and parameters
model = RandomForestClassifier()
n_estimators = [10, 100, 1000]
max_features = ['sqrt', 'log2']
# define grid search
grid = dict(n_estimators=n_estimators,max_feature
cv = RepeatedStratifiedKFold(n_splits=10, n_repea
```

```
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='accuracy',
grid_result = grid_search.fit(X, y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example prints the best result as well as the results from all combinations evaluated.

```
Best: 0.952000 using {'max_features': 'log2',
0.841000 (0.032078) with: {'max_features': 'sq
0.938333 (0.020830) with: {'max_features': 'sq
0.944667 (0.024998) with: {'max_features': 'sq
0.817667 (0.033235) with: {'max_features': 'lo
0.940667 (0.021592) with: {'max_features': 'lo
0.952000 (0.019562) with: {'max_features': 'lo
```


Stochastic Gradient Boosting

Also called Gradient Boosting Machine (GBM) or nam ... ost.

The gradient boosting algorithm has many parameters to tune.

There are some parameter pairings that are important to consider. The first is the learning rate, also called shrinkage or eta (*learning_rate*) and the number of trees in the model (*n_estimators*). Both could be considered on a log scale, although in different directions.

- **learning_rate** in [0.001, 0.01, 0.1]
- **n_estimators** [10, 100, 1000]

Another pairing is the number of rows or subset of the data to consider for each tree (*subsample*) and the depth of each tree (*max_depth*). These could be grid searched at a 0.1 and 1 interval respectively, although common values can be tested directly.

- **subsample** in [0.5, 0.7, 1.0]
- **max_depth** in [3, 7, 9]

For more detailed advice on tuning the XGBoost implementation, see:

- How to Configure the Gradient Boosting Algorithm

For the full list of hyperparameters, see:

- sklearn.ensemble.GradientBoostingClassifier API

The example below demonstrates grid searching the key hyperparameters for GradientBoostingClassifier on a synthetic binary classification dataset.

```
# example of grid searching key hyperparameters for GradientBoostingClassifier
from sklearn.datasets import make_blobs
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import GradientBoostingClassifier
# define dataset
X, y = make_blobs(n_samples=1000, centers=2, n_features=100, cluster_std=20)
# define models and parameters
model = GradientBoostingClassifier()
n_estimators = [10, 100, 1000]
learning_rate = [0.001, 0.01, 0.1]
subsample = [0.5, 0.7, 1.0]
max_depth = [3, 7, 9]
# define grid search
grid = dict(learning_rate=learning_rate, n_es
cv = RepeatedStratifiedKFold(n_splits=10, n_r
grid_search = GridSearchCV(estimator=model, p
grid_result = grid_search.fit(X, y)
# summarize results
print("Best: %f using %s" % (grid_result.best
means = grid_result.cv_results_['mean_test_sc
stds = grid_result.cv_results_['std_test_scor
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, pa
    print("%f (%f) with: %r" % (mean, stdev,
```


Note: Your results may vary given the stochastic natu
differences in numerical precision. Consider running t
outcome.

Running the example prints the best result as well as the results from all combinations evaluated.

```
Best: 0.936667 using {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 1000, 'subsample':
0.803333 (0.042058) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 10, 'subsamp
0.783667 (0.042386) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 10, 'subsamp
0.711667 (0.041157) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 10, 'subsamp
0.832667 (0.040244) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 100, 'subsam
0.809667 (0.040040) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 100, 'subsam
0.741333 (0.043261) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 100, 'subsam
0.881333 (0.034130) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 1000, 'subsa
0.866667 (0.035150) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 1000, 'subsa
0.838333 (0.037424) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_estimators': 1000, 'subsa
0.838333 (0.036614) with: {'learning_rate': 0.001, 'max_depth': 7, 'n_estimators': 10, 'subsamp
0.821667 (0.040586) with: {'learning_rate': 0.001, 'max_depth': 7, 'n_estimators': 10, 'subsamp
0.729000 (0.035903) with: {'learning_rate': 0.001, 'max_depth': 7, 'n_estimators': 10, 'subsamp
0.884667 (0.036854) with: {'learning_rate': 0.001, 'max_depth': 7, 'n_estimators': 100, 'subsam
0.871333 (0.035094) with: {'learning_rate': 0.001, 'max_depth': 7, 'n_estimators': 100, 'subsam
0.729000 (0.037625) with: {'learning_rate': 0.001, 'max_depth': 7, 'n_estimators': 100, 'subsam
0.905667 (0.033134) with: {'learning_rate': 0.001, 'max_depth': 7, 'n_estimators': 1000, 'subsa
...
```

Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- scikit-learn API
- Caret List of Algorithms and Tuning Parameters

Summary

In this tutorial, you discovered the top hyperparameters and how to configure them for top machine learning algorithms.

Do you have other hyperparameter suggestions? Let me know in the comments below.

Do you have any questions?
Ask your questions in the comments below and I will do my best to answer.

Tweet Share Share

About Jason Brownlee

Jason Brownlee, PhD is a machine learning specialist who teaches developers how to get results with modern machine learning methods via hands-on tutorials.
View all posts by Jason Brownlee →

‹ How to Develop Super Learner Ensembles in Python

How to Transform Target Variables for Regression in Python ›

44 Responses to *Tune Hyperparameters for Classification Machine Learning Algorithms*

Dazhi December 13, 2019 at 6:07 am #

REPLY

Thanks for the useful post! A quick question here: why do you set `n_repeats=3` for the cross validation? As far as I understand, the cv will split the data into folds and calculate the metrics on each fold and take the average. Is it necessary to repeat this process for 3 times?

Jason Brownlee December 13, 2019 at 6:29

Excellent question.

Repeats help to smooth out the variance in some r
datasets. Repeated CV compared to 1xCV can oft
model.

Doug Dean December 13, 2019 at 7:09 am #

Thanks for the great article.

Changing the parameters for the ridge classifier did not change the outcome. Is that because of the synthetic dataset or is there some other problem with the example?

Jason Brownlee December 13, 2019 at 1:40 pm #

REPLY

Yes, likely because the synthetic dataset is so simple.

Jonathan Mackenzie December 13, 2019 at 10:20 am #

REPLY

I normally use TPE for my hyperparameter optimisation, which is good at searching over large parameter spaces. Hyperas and hyperopt even let you do this in parallel!
https://github.com/maxpumperla/hyperas

Also, keras recently introduced their own HO tool called keras-tuner, which looks easy to use:

https://github.com/keras-team/keras-tuner

Jason Brownlee December 13, 2019 at 1:42 pm #

REPLY

Thanks for sharing.

Oren December 13, 2019 at 11:50 am #

REPLY

Dear Jason,

How about an article about generalization abilities of ML models? For instance, we train and tune a specific learning algorithm on a data set (train + validation set) from a distributon X and apply it to some data that origins from another distribution Y. In practise, the lear
how to counteract the problem besides basic stuff like

Best regards

Jason Brownlee December 13, 2019 at 1:44

Thanks for the suggestion.

Yes, I have tens of tutorials on the topic. Perhaps s
https://machinelearningmastery.com/introduction-t
generalization-error/

Rich Larrabee December 21, 2019 at 9:23 am #

REPLY

Thanks for the article Jason. I have a follow-up question. Which one of these models is best when the classes are highly imbalanced (fraud for example)? And why?

Thanks,

Rich

Jason Brownlee December 22, 2019 at 6:05 am #

REPLY

There is no best model in general. I recommend testing a suite of different techniques for imbalanced classification and discovering what works best for your specific dataset.

adip32 January 5, 2020 at 10:49 pm #

REPLY

xgboost not included? why only 7 algorithms?

Jason Brownlee January 6, 2020 at 7:12 am #

REPLY

This are the popular algorithms in sklearn.

For tuning xgboost, see the suite of tutorials, perhaps starting here:
https://machinelearningmastery.com/start-here/#xgboost

Aned Esquerra Arguelles April 1, 2020 at 1:49 am #

REPLY

Hi Jason, great tut as ever! I have a question,
all examples if those cases aren't supposedly imbalan

Jason Brownlee April 1, 2020 at 5:53 am #

Thanks!

It's a good practice, perhaps a best practice.

John White April 30, 2020 at 9:12 am #

REPLY

Hi Jason!

Question on tuning RandomForest. For my hypertuning results, the best parameters' precision_score is very similar to the spot check. I am having a hard time understanding how is this possible.

From the spot check, results proved the model already has little skill, slightly better than no skill, so I think it has potential. However the best parameters says otherwise.

I am currently trying to tune a binary RandomForestClassifier using RandomizedSearchCV (… refit='precision'). Precision being: make_scorer(precision_score, average = 'weighted'). Dataset is balanced.

Jason Brownlee April 30, 2020 at 11:37 am #

REPLY

Perhaps the difference in the mean results is no statistically significant. So the numbers look different, but the behavior is not different on average.

John White April 30, 2020 at 12:13 pm #

Ah I see. Is there a way to get to the bottom of this? I am currently looking into feature selection as given here: https://machinelearningmastery.com/feature-selection-with-real-and-categorical-data/

Jason Brownlee April 30, 2020 at 1:32 pm #

REPLY

Yes, here is some advice on how to use hypothesis tests to compare results: https://machinelearningmastery.com/statistical-significance-tests-for-comparing-machine-learning-algorithms/

Or perhaps you can change your test harness, e.g. more repeats, more folds, to help better expose differences between algorithms.

John White April 30, 2020 at 1:42 pm #

Thank you. I'll start there. When I was models, they also returned similar very similar just the similar: slightly better than no skill.

Jason Brownlee May 1, 2020 at 6:

It is possible that your problem is not predictable in its current form/framing.

John White May 9, 2020 at 5:13 am #

I am going to try out different models. Features are correlated and important through different feature selection and feature importance tests. Also coupled with industry knowledge, I also know the features can help determine the target variable (problem). I won't give up!

Jason Brownlee May 9, 2020 at 6:25 am #

Sounds great!

Skylar May 16, 2020 at 5:04 am #

REPLY

Hi Jason,

Nice post, very clear! You mainly talked about algorithms for classification problems, do you also have the summary for regression? Or it is more or less similar? Thanks!

Jason Brownlee May 16, 2020 at 6:24 am # REPLY

Thanks!

Not at this stage, perhaps soon.

Skylar May 16, 2020 at 2:32 pm #

That would be great, I will definitely ke

Jason Brownlee May 17, 2020 at 6

Thanks.

sukhpal June 1, 2020 at 1:10 am # REPLY

sir what technique we apply after hyper-parameter optimization to furthur refine the results

Jason Brownlee June 1, 2020 at 6:25 am # REPLY

See this:
http://machinelearningmastery.com/machine-learning-performance-improvement-cheat-sheet/

Vinayak Shanawad July 10, 2020 at 6:42 pm # REPLY

Thank you so much.

1. Why do you set random_state=1 for the cross validation?

cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

As per my understanding, in test_train_split with different random state we get different accuracies and to avoid that we will do cross validation.

2. Is it necessary to set the random_state=1 for the cro

3. In your all examples above, from gridsearch results we are getting accuracy of Training data set. Is that right?

# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))

I think from grid_result which is our best model and using that calculate the accuracy of Test data set.

Jason Brownlee July 11, 2020 at 6:07 am # REPLY

The random seed is fixed to ensure we get the same result each time the code is run – helpful for tutorials.

You can set any value you like:
https://machinelearningmastery.com/faq/single-faq

We are not using a train/test split, we are using rep
performance of each config.

Mihai July 16, 2020 at 11:22 pm #

Regarding the parameters for Random Forest
0.22: The default value of n_estimators changed from 10 to 100 in 0.22." – In your code you have up to 1000, in case you want to update your code ☺

Jason Brownlee July 17, 2020 at 6:17 am # REPLY

Thanks.

More is better to a limit, when it comes to RF.

Amilkar August 11, 2020 at 12:20 am # REPLY

I love your tutorials. I think you do a great job. I have learned so much from you. I've been considering buying one of your books, but you a so many that I don't know which one to buy. I have come to realize how important hyperparameter tuning is and I have noticed that each model is different and I need a summarized source of information that gives me a general idea of what hyperparameters to try for each model and techniques to do the process as fast and efficiently as possible. I've heard about Bayesian hyperparameter optimization techniques. Would be great if I could learn how to do this with scikitlearn. Also, I'm particularly interested in XGBoost because I've read in your blogs that it tends to perform really well. Which one of your books would you recommend me to learn how to do hyperparameter tuning fast and efficiently using python (special mention on XGBoost if

Jason Brownlee August 11, 2020 at 6:34 am # REPLY

Thanks!

I recommend using the free tutorials and only get a book if you need more information or want to systematically work through a topic.

Golo August 15, 2020 at 4:50 pm # REPLY

Hi Jason, thanks for the post. Regarding this
the same results in each split and repetition? Or where

Jason Brownlee August 16, 2020 at 5:48 am

When random_state is set on the cv objec
hyperparameter configuration is evaluated on the s

Sreeram August 16, 2020 at 11:52 pm # REPLY

why Repeated Stratified K fold is used?

Jason Brownlee August 17, 2020 at 5:47 am # REPLY

It is a best practice for evaluating models on classification tasks.

jenny November 18, 2020 at 12:52 pm # REPLY

what are the best classification algorithms to use in the popular (fashion mnist) dataset and also which hyperparameters are preferable?

Jason Brownlee November 18, 2020 at 1:08 pm # REPLY

A CNN.

BInnan November 19, 2020 at 3:14 pm #

REPLY

Hi Jason, it's a great article!

I am just wondering that since grid search implement through cross-validation, once the optimal combination of hyperparameters are selected, is it necessary to perform cross-validation again to test the model performance with optimal parameters?

Jason Brownlee November 20, 2020 at 6:42 am #

REPLY

No, but you can if you like to confirm the f

Hadi Sabahi June 12, 2021 at 12:01 am #

Hi Jason, thanks for your post, I have a quest
tune a classifier, we should find its Operating Point, wh
intersection with Y=-X. The ROC curve is calculated us
corresponding hyperparameters of the Operating Point
My question is that why do you use (scoring='accuracy
In other words, why don't you consider sensitivity and
curve?

Jason Brownlee June 12, 2021 at 5:35 am #

REPLY

I recommend optimizing the ROC AUC and use roc curve as a diagnostic.

Leave a Reply

Name (required)

Email (will not be published) (required)

Website

SUBMIT COMMENT

Welcome!
I'm *Jason Brownlee* PhD
and I **help developers** get results with **machine learning**.
Read more

Never miss a tutorial:

Picked for you:

Your First Machine Learning Project in Python Step-

How to Setup Your Python Environment for Machine Learning with Anaconda

Feature Selection For Machine Learning in Python

Python Machine Learning Mini-Course

Save and Load Machine Learning Models in Python with scikit-learn

Loving the Tutorials?

The Machine Learning with Python EBook is
where you'll find the