COCS 5557: Practical Machine Learning

Hyperparammeter Optimization

Introduction

Specifically, in this study, we aim to optimize the `hyperparameters` of various machine learning algorithms to achieve the best predictive performance on the `White Wine Quality` dataset. We will explore different algorithms and their `hyperparameter` configurations, aiming to find the optimal combination through `hyperparameter` optimization techniques. The primary goal is to demonstrate the effectiveness of `hyperparameter` tuning in improving model performance.

Hyperparameter Tuning

`Hyperparameters` are parameters whose values control the learning process and determine the values of model parameters that a learning algorithm ends up learning.

A model `hyperparameter` is a configuration that is external to the model and whose value cannot be estimated from data.

`Hyperparameter tuning` consists of finding a set of optimal `hyperparameter` values for a learning algorithm while applying this optimized algorithm to any data set. That combination of `hyperparameters` maximizes the model's performance, minimizing a predefined loss function to produce better results with fewer errors.

`Hyperparameter tuning` is an essential part of machine learning.

`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 when implementing 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 need to be tuned, the slower the tuning process would be. 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, on the performance of a machine learning algorithm.

It is important know which `hyperparameters` to focus on to get a good result in a timely manner.

In this Exercise, it will be discovered those `hyperparameters` that are most important for some of the top machine learning algorithms.

Classification Algorithms Overview

Here we look closely at the important `hyperparameters` of the top machine learning algorithms that may be used for classification algorithms. Hyperparameters that need to be focused on are suggested with values to try when tuning the model on the dataset.

The suggestions are based on AI community opinions.

Here are the classification algorithms that would be explored:

† Logistic Regression;

† Ridge Classifier;

† K-Nearest Neighbors (KNN);

† Support Vector Machine (SVM);

† Bagged Decision Trees (Bagging);

† Random Forest; and

† Stochastic Gradient Boosting.

We will consider these algorithms in the context of their scikit-learn implementation (Python). However, the same `hyperparameter` suggestions are usable in the context of other platforms like Weka and R, as well.

Dataset Description

The `White Wine Quality` dataset comprises sensory data of white of the Portuguese “Vinho Verde” wine. The dataset includes physicochemical features such as `acidity, pH`, and `alcohol` content, along with the `quality` rating provided by wine experts. The white wine dataset contains 4898 samples. There are no missing values in the dataset.

Experimental Setup

Programming Languages and Libraries

The following libraries in Python programming language will be used.

† `NumPy`: for numerical computations.

† `Pandas`: for data manipulation and analysis.

† `Scikit-learn`: for machine learning algorithms and `hyperparameter` optimization.

† `Matplotlib` and `Seaborn`: for data visualization.

Data Processing

We will preprocess the data by standardizing numerical features and encoding categorical variables (if any). Additionally, we will split the dataset into training and testing sets with a ratio of 80:20.

Table 1: The First Five Rows of the White Wine Data

	0	1	2	3	4
fixed acidity	7.000000	6.300000	8.100000	7.200000	7.200000
volatile acidity	0.270000	0.300000	0.280000	0.230000	0.230000
citric acid	0.360000	0.340000	0.400000	0.320000	0.320000
residual sugar	20.700000	1.600000	6.900000	8.500000	8.500000
chlorides	0.045000	0.049000	0.050000	0.058000	0.058000
free sulfur dioxide	45.000000	14.000000	30.000000	47.000000	47.000000
total sulfur dioxide	170.000000	132.000000	97.000000	186.000000	186.000000
density	1.001000	0.994000	0.995100	0.995600	0.995600
pH	3.000000	3.300000	3.260000	3.190000	3.190000
sulphates	0.450000	0.490000	0.440000	0.400000	0.400000
alcohol	8.800000	9.500000	10.100000	9.900000	9.900000
quality	6.000000	6.000000	6.000000	6.000000	6.000000

```
quality
6    2198
5    1457
7     880
8     175
4     163
3      20
9       5
Name: count, dtype: int64

X_train shape: (3918, 11)
X_test shape: (980, 11)
```

Logistic Regression

Logistic regression does not have any particular critical `hyperparameters` to tune.

Sometimes, one could see useful differences in performance or convergence with different `solvers` (`solver`):

`solver` in `['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']`

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 `penalty` strength, which can also be effective.

`C` in `[100, 10, 1.0, 0.1, 0.01]`

```
0.5142857142857142
```

```
`LogisticRegression()` Parameters currently in use:

{'C': 1.0,
 'class_weight': None,
 'dual': False,
 'fit_intercept': True,
 'intercept_scaling': 1,
 'l1_ratio': None,
 'max_iter': 1000,
 'multi_class': 'auto',
 'n_jobs': None,
 'penalty': 'l2',
 'random_state': None,
 'solver': 'lbfgs',
 'tol': 0.0001,
 'verbose': 0,
 'warm_start': False}

Best: 0.536001 using {'C': 10, 'penalty': 'l2', 'solver': 'newton-cg'}
0.534776 (0.012027) with: {'C': 100, 'penalty': 'l2', 'solver': 'newton-cg'}
0.523138 (0.009025) with: {'C': 100, 'penalty': 'l2', 'solver': 'lbfgs'}
0.534027 (0.008975) with: {'C': 100, 'penalty': 'l2', 'solver': 'liblinear'}
0.536001 (0.012546) with: {'C': 10, 'penalty': 'l2', 'solver': 'newton-cg'}
0.522390 (0.009798) with: {'C': 10, 'penalty': 'l2', 'solver': 'lbfgs'}
0.533550 (0.008369) with: {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
0.533959 (0.012802) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'newton-cg'}
0.521847 (0.011125) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'lbfgs'}
0.528650 (0.010106) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'liblinear'}
```

```
0.530147 (0.013432) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cg'}
0.515720 (0.009707) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'lbfgs'}
0.511909 (0.006861) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}
0.511228 (0.013399) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cg'}
0.502993 (0.010409) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'lbfgs'}
0.475568 (0.006765) 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]

```
0.5214285714285715
```

```
`RidgeClassifier()` Parameters currently in use:

{'alpha': 1.0,
 'class_weight': None,
 'copy_X': True,
 'fit_intercept': True,
 'max_iter': None,
 'positive': False,
 'random_state': None,
 'solver': 'auto',
 'tol': 0.0001}
```

```
Best: 0.528447 using {'alpha': 0.9}
0.534776 (0.006356) with: {'alpha': 0.1}
0.523138 (0.006434) with: {'alpha': 0.2}
0.534027 (0.006304) with: {'alpha': 0.3}
0.536001 (0.006257) with: {'alpha': 0.4}
0.522390 (0.006064) with: {'alpha': 0.5}
0.533550 (0.006064) with: {'alpha': 0.6}
0.533959 (0.005982) with: {'alpha': 0.7}
0.521847 (0.006014) with: {'alpha': 0.8}
0.528650 (0.005949) with: {'alpha': 0.9}
0.530147 (0.006068) 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’]`

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

`weights` in `[‘uniform’, ‘distance’]`

```
0.55
```

```
`KNeighborsClassifier()` Parameters currently in use:

{'algorithm': 'auto',
 'leaf_size': 30,
 'metric': 'minkowski',
 'metric_params': None,
 'n_jobs': None,
 'n_neighbors': 5,
 'p': 2,
 'weights': 'uniform'}
```

```
Best: 0.624200 using {'metric': 'manhattan', 'n_neighbors': 17, 'weights': 'distance'}
0.577653 (0.012447) with: {'metric': 'euclidean', 'n_neighbors': 1, 'weights': 'uniform'}
0.577653 (0.012447) with: {'metric': 'euclidean', 'n_neighbors': 1, 'weights': 'distance'}
0.487002 (0.009212) with: {'metric': 'euclidean', 'n_neighbors': 3, 'weights': 'uniform'}
0.586089 (0.011984) with: {'metric': 'euclidean', 'n_neighbors': 3, 'weights': 'distance'}
0.477474 (0.010581) with: {'metric': 'euclidean', 'n_neighbors': 5, 'weights': 'uniform'}
0.593235 (0.012953) with: {'metric': 'euclidean', 'n_neighbors': 5, 'weights': 'distance'}
0.468630 (0.013572) with: {'metric': 'euclidean', 'n_neighbors': 7, 'weights': 'uniform'}
0.596299 (0.010838) with: {'metric': 'euclidean', 'n_neighbors': 7, 'weights': 'distance'}
0.466179 (0.014115) with: {'metric': 'euclidean', 'n_neighbors': 9, 'weights': 'uniform'}
0.597999 (0.008691) with: {'metric': 'euclidean', 'n_neighbors': 9, 'weights': 'distance'}
0.468151 (0.011000) with: {'metric': 'euclidean', 'n_neighbors': 11, 'weights': 'uniform'}
0.602763 (0.013321) with: {'metric': 'euclidean', 'n_neighbors': 11, 'weights': 'distance'}
0.466654 (0.011027) with: {'metric': 'euclidean', 'n_neighbors': 13, 'weights': 'uniform'}
0.604396 (0.011761) with: {'metric': 'euclidean', 'n_neighbors': 13, 'weights': 'distance'}
0.460188 (0.009130) with: {'metric': 'euclidean', 'n_neighbors': 15, 'weights': 'uniform'}
0.607867 (0.012859) with: {'metric': 'euclidean', 'n_neighbors': 15, 'weights': 'distance'}
0.462297 (0.011291) with: {'metric': 'euclidean', 'n_neighbors': 17, 'weights': 'uniform'}
0.610656 (0.011691) with: {'metric': 'euclidean', 'n_neighbors': 17, 'weights': 'distance'}
0.462775 (0.011155) with: {'metric': 'euclidean', 'n_neighbors': 19, 'weights': 'uniform'}
0.611678 (0.012783) with: {'metric': 'euclidean', 'n_neighbors': 19, 'weights': 'distance'}
0.586228 (0.013639) with: {'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}
0.586228 (0.013639) with: {'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'distance'}
0.491493 (0.013219) with: {'metric': 'manhattan', 'n_neighbors': 3, 'weights': 'uniform'}
0.592962 (0.012687) with: {'metric': 'manhattan', 'n_neighbors': 3, 'weights': 'distance'}
0.488567 (0.013206) with: {'metric': 'manhattan', 'n_neighbors': 5, 'weights': 'uniform'}
0.602219 (0.012108) with: {'metric': 'manhattan', 'n_neighbors': 5, 'weights': 'distance'}
0.483602 (0.010170) with: {'metric': 'manhattan', 'n_neighbors': 7, 'weights': 'uniform'}
0.605146 (0.011735) with: {'metric': 'manhattan', 'n_neighbors': 7, 'weights': 'distance'}
0.482308 (0.010367) with: {'metric': 'manhattan', 'n_neighbors': 9, 'weights': 'uniform'}
0.609433 (0.011960) with: {'metric': 'manhattan', 'n_neighbors': 9, 'weights': 'distance'}
0.480266 (0.012295) with: {'metric': 'manhattan', 'n_neighbors': 11, 'weights': 'uniform'}
0.615149 (0.011978) with: {'metric': 'manhattan', 'n_neighbors': 11, 'weights': 'distance'}
```

```
0.481355 (0.013384) with: {'metric': 'manhattan', 'n_neighbors': 13, 'weights': 'uniform'}
0.615217 (0.011197) with: {'metric': 'manhattan', 'n_neighbors': 13, 'weights': 'distance'}
0.476728 (0.011735) with: {'metric': 'manhattan', 'n_neighbors': 15, 'weights': 'uniform'}
0.619164 (0.010686) with: {'metric': 'manhattan', 'n_neighbors': 15, 'weights': 'distance'}
0.479176 (0.010684) with: {'metric': 'manhattan', 'n_neighbors': 17, 'weights': 'uniform'}
0.624200 (0.010412) with: {'metric': 'manhattan', 'n_neighbors': 17, 'weights': 'distance'}
0.475910 (0.011965) with: {'metric': 'manhattan', 'n_neighbors': 19, 'weights': 'uniform'}
0.623589 (0.008335) with: {'metric': 'manhattan', 'n_neighbors': 19, 'weights': 'distance'}
0.577653 (0.012447) with: {'metric': 'minkowski', 'n_neighbors': 1, 'weights': 'uniform'}
0.577653 (0.012447) with: {'metric': 'minkowski', 'n_neighbors': 1, 'weights': 'distance'}
0.487002 (0.009212) with: {'metric': 'minkowski', 'n_neighbors': 3, 'weights': 'uniform'}
0.586089 (0.011984) with: {'metric': 'minkowski', 'n_neighbors': 3, 'weights': 'distance'}
0.477474 (0.010581) with: {'metric': 'minkowski', 'n_neighbors': 5, 'weights': 'uniform'}
0.593235 (0.012953) with: {'metric': 'minkowski', 'n_neighbors': 5, 'weights': 'distance'}
0.468630 (0.013572) with: {'metric': 'minkowski', 'n_neighbors': 7, 'weights': 'uniform'}
0.596299 (0.010838) with: {'metric': 'minkowski', 'n_neighbors': 7, 'weights': 'distance'}
0.466179 (0.014115) with: {'metric': 'minkowski', 'n_neighbors': 9, 'weights': 'uniform'}
0.597999 (0.008691) with: {'metric': 'minkowski', 'n_neighbors': 9, 'weights': 'distance'}
0.468151 (0.011000) with: {'metric': 'minkowski', 'n_neighbors': 11, 'weights': 'uniform'}
0.602763 (0.013321) with: {'metric': 'minkowski', 'n_neighbors': 11, 'weights': 'distance'}
0.466654 (0.011027) with: {'metric': 'minkowski', 'n_neighbors': 13, 'weights': 'uniform'}
0.604396 (0.011761) with: {'metric': 'minkowski', 'n_neighbors': 13, 'weights': 'distance'}
0.460188 (0.009130) with: {'metric': 'minkowski', 'n_neighbors': 15, 'weights': 'uniform'}
0.607867 (0.012859) with: {'metric': 'minkowski', 'n_neighbors': 15, 'weights': 'distance'}
0.462297 (0.011291) with: {'metric': 'minkowski', 'n_neighbors': 17, 'weights': 'uniform'}
0.610656 (0.011691) with: {'metric': 'minkowski', 'n_neighbors': 17, 'weights': 'distance'}
0.462775 (0.011155) with: {'metric': 'minkowski', 'n_neighbors': 19, 'weights': 'uniform'}
0.611678 (0.012783) with: {'metric': 'minkowski', 'n_neighbors': 19, '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 take on a range of values and has a dramatic effect on the shape of the resulting regions for each class. A log scale might be a good starting point:

`C` in `[100, 10, 1.0, 0.1, 0.001]`

```
0.5612244897959183
```

```
`SVC()` Parameters currently in use:

{'C': 1.0,
 'break_ties': False,
 'cache_size': 200,
 'class_weight': None,
 'coef0': 0.0,
 'decision_function_shape': 'ovr',
 'degree': 3,
 'gamma': 'scale',
 'kernel': 'rbf',
 'max_iter': -1,
 'probability': False,
 'random_state': None,
 'shrinking': True,
 'tol': 0.001,
 'verbose': False}
```

```
Best: 0.508440 using {'C': 50, 'gamma': 'scale'}
0.508440 (0.008209) with: {'C': 50, 'gamma': 'scale'}
0.450388 (0.004985) with: {'C': 1.0, 'gamma': 'scale'}
0.448754 (0.000276) with: {'C': 0.01, 'gamma': 'scale'}
```

Bagged Decision Trees (Bagging)

The most important parameter for bagged decision trees is the number of trees (`n_estimators`).

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]`

```
0.6153061224489796
```

```
`BaggingClassifier()` Parameters currently in use:

{'bootstrap': True,
 'bootstrap_features': False,
 'estimator': None,
 'max_features': 1.0,
 'max_samples': 1.0,
 'n_estimators': 10,
 'n_jobs': None,
 'oob_score': False,
 'random_state': None,
 'verbose': 0,
 'warm_start': False}
```

```
Best: 0.682999 using {'n_estimators': 1000}
0.677826 (0.011547) with: {'n_estimators': 50}
0.679664 (0.014144) with: {'n_estimators': 100}
0.682999 (0.016121) with: {'n_estimators': 1000}
```

Random Forest

For random forest, the most important parameter is the number of random features to sample at each split point (`max_features`).

A range of integer values, such as 1 to 20, or 1 to half the number of input features could be used:

`max_features [1 to 20]`

Alternately, one could use a suite of different default value calculators:

`max_features` in `[‘sqrt’, ‘log2’]`

Another important parameter for random forest is the number of trees (`n_estimators`). 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]`

```
0.6836734693877551
```

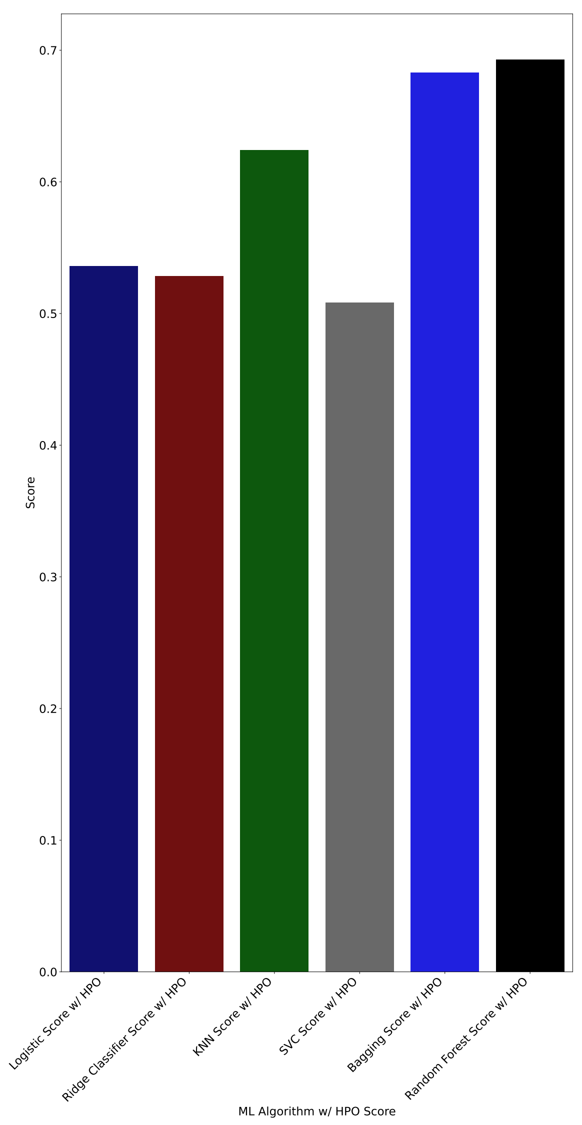
```
`RandomForestClassifier()` Parameters currently in use:

{'bootstrap': True,
 'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': 'sqrt',
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'monotonic_cst': None,
 'n_estimators': 100,
 'n_jobs': None,
 'oob_score': False,
 'random_state': None,
 'verbose': 0,
 'warm_start': False}
```

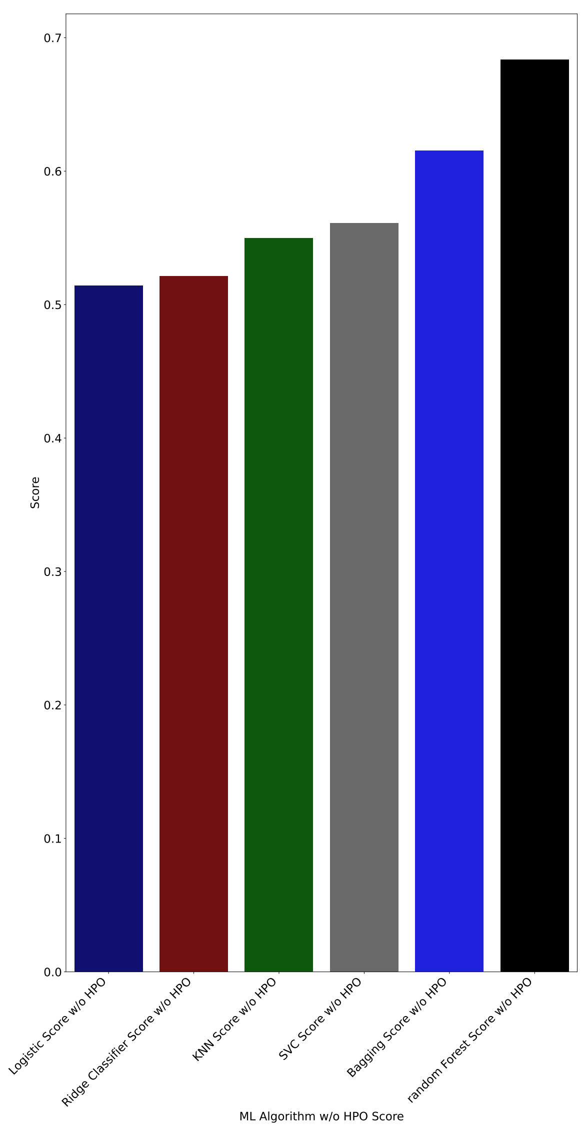
```
Best HPO: 0.692935 using {'max_features': 'log2', 'n_estimators': 5000}
0.690486 (0.015106) with: {'max_features': 'sqrt', 'n_estimators': 100}
0.691710 (0.016215) with: {'max_features': 'sqrt', 'n_estimators': 1000}
0.692459 (0.015545) with: {'max_features': 'sqrt', 'n_estimators': 5000}
0.686401 (0.020138) with: {'max_features': 'log2', 'n_estimators': 100}
0.691574 (0.016333) with: {'max_features': 'log2', 'n_estimators': 1000}
0.692935 (0.015675) with: {'max_features': 'log2', 'n_estimators': 5000}
```

	Score
Logistic Score w/ HPO	0.536001
Ridge Classifier Score w/ HPO	0.528447
KNN Score w/ HPO	0.624200
SVC Score w/ HPO	0.508440
Bagging Score w/ HPO	0.682999
Random Forest Score w/ HPO	0.692935

	Score
Logistic Score w/o HPO	0.514286
Ridge Classifier Score w/o HPO	0.521429
KNN Score w/o HPO	0.550000
SVC Score w/o HPO	0.561224
Bagging Score w/o HPO	0.615306
random Forest Score w/o HPO	0.683673

Visualizing the scores





Conclusion:

After tuning the `hyperparameters`, Random Forest algorithm is the best machine learning algorithm with `{'max_features': 'log2', 'n_estimators': 5000}` `hyperparameters` combination, for predicting the white wine `quality`.

Code:

The code used for this Exerecise will be provided in a separate Quarto Document file for reproducibility.