Practical Machine Learning: Hyperparameter Optimization

Russell Todd[1]

[1]University of Wyoming

Abstract

1 Introduction

In this exercise, I will be looking at a basic version of automated machine learning and choosing the best type of machine learning model and hyperparameters for my task. I will familiarize myself with the process and report on how I progressed through the exercise. I will strive to explain well enough that reproducing my work would be doable even if I did not share my exact code. The dataset I will be looking at is the White Wine Quality dataset so I will be using the features of wines to predict their quality score.

2 Dataset Description

The famous White Wine Quality dataset is comprised of 12 features. The first 11 are measurements of various physical qualities of each wine (fixed acidity, citric acid, residual sugar, pH, alcohol, etc.) and the 12th is the output variable which is a score (0-10) denoting the quality of each wine. There are 4,897 entries (wines) in the dataset and no entries are missing data for any feature. All features are positive numeric values.

After exploring the White Wine Quality dataset, it can be seen that it is quite imbalanced with relatively few entries for low quality and high quality wines. Unfortunately, when I attempted to include an oversampling step in my process, I failed to achieve an improvement in model performance.

It also has two variables, density and residual sugar, that are highly correlated. I decided to drop density before training models as it is less correlated to the score than residual sugar.

Additionally, most of the features have significant outliers present in their data as can be seen from a selection of plots in figure 3.

2.1 Preprocessing

I first split the data into a an 80:20 training test split before applying the scaler so that the test data would not be included when fitting the scaler. I used the Standard Scaler from the sklearn preprocessing package. I fit it to the training data and then applied that fit to both the training data and the testing data.

3 Experimental Setup

The first step was to get the data properly loaded into a pandas package dataframe and ensure that the datatypes of each feature are appropriate. Afterwards, several summary statistics panes are produced using built-in functions of the pandas dataframe class like info(), dtypes, and describe(). I then use these to double check that the data loading process was done correctly. If not, I then corrected them manually.

I then applied the data preprocessing steps as outlined in the data description section above.

The selected regression algorithms were LogisticRegression, DecisionTreeClassifier, RandomForestClassifier, and a Radial Basis Function Support Vector Machine (RBFSVM). All algorithms

were run with parameters as close to default as possible, with the exception of LogisticRegression which required me to raise the max iterations parameter to run it properly. To assess the models performance I used the accuracy score from the sklearn metrics packag on a fivefold cross-validation run of each model to try to avoid overfitting. This basic output was then stored for later comparison to the optimized models after hyperparameter optimization.

3.1 Hyperparameter Optimization Step

I decided to use a pipeline to streamline the process of applying the classifier models to the scaled dataset. On the matter of the scaled dataset I attempted to include it in the pipeline as a preprocessing step but I struggled to implement it in such a way as to fit the scaler after the split. While I could have fitted it before the split, that would introduce data leakage and be cheating in a sense as the scaler would be fitted with the test data included.

At this point I was now able to repeat the basic output runs much more easily so the next step was to define the hyperparameter search space. The I chose the range for each of the hyperparameters from reading the documentation on each model's parameters and selected what I felt were reasonable numbers (basically just multiples of 10). The hyperparameters that were tuned for each model are as follows: classifier__C with a Real range of 0.1 to 10 for LRC. classifier__max_depth with an Integer range of 10 to 100 for DT. classifier__n_estimators with an Integer range of 100 to 1000 and classifier__max_depth with an Integer range of 10 to 100 for RFC. classifier__C with a Real range of 0.1 to 10 and classifier__gamma with a categorical auto scale for RBFSVM.The driving force behind exploring the search space was a bayesian search evaluating performance based on accuracy scores from fivefold cross-validation. The bayes search was implemented using the sklearn optimization package (skopt). To prevent overfitting, nested resampling was performed as part of the bayes search using a 5-fold cross-validation strategy. The bayesian optimization was run for 10 iterations which is on the lower side.

The scores were stored in much the same way as the basic output scores were so that I see the performance difference after the optimization. This data included the training and testing data so that potential overfitting could be seen.

4 Results

From the graphs below it can be seen that the RandomForestClassifier noticeably outperformed the other competing algorithms.

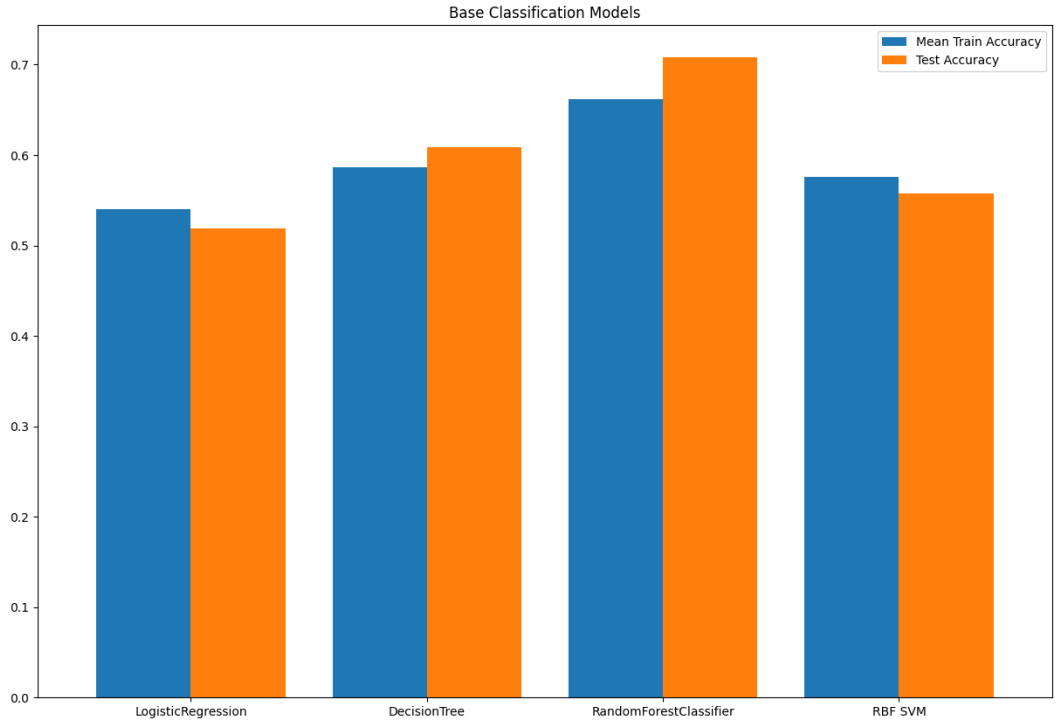

Figure 1: Classification Model Scores Before Tuning

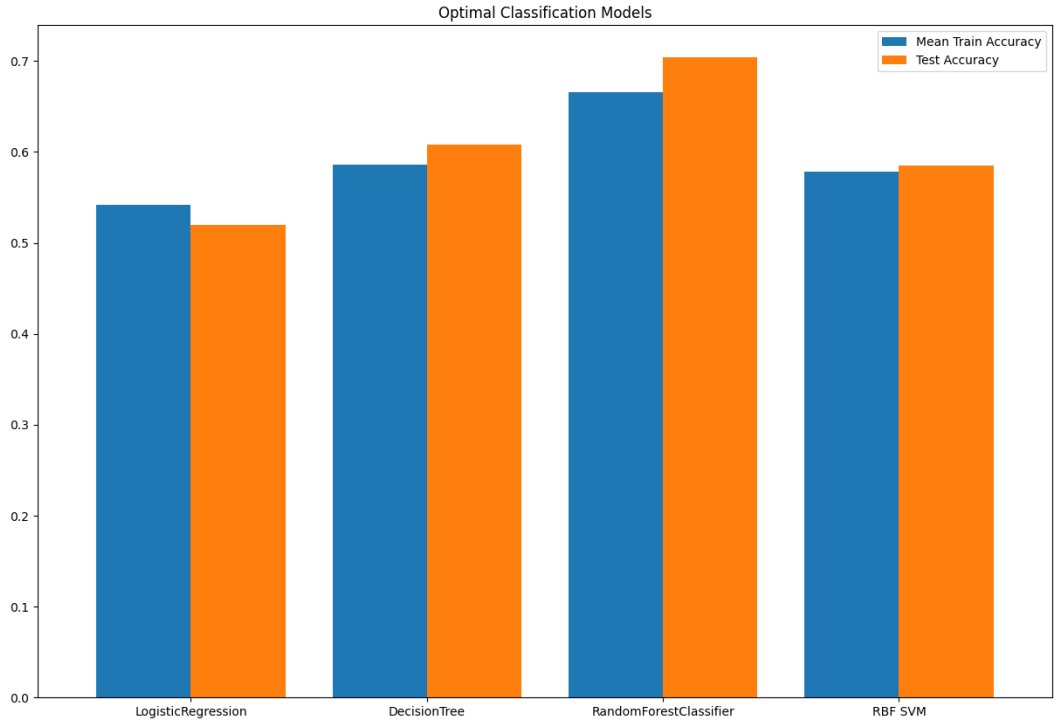

Figure 2: Classification Model Scores After Tuning

The exact scores are shown below:

```
Basic Scores

LRC Train Accuracy: 0.5400669846482654
LRC Test Accuracy: 0.5183673469387755

DT Train Accuracy: 0.5862606536867621
DT Test Accuracy: 0.6081632653061224

RFC Train Accuracy: 0.6618145150780619
RFC Test Accuracy: 0.7081632653061225

RBFSVM Train Accuracy: 0.5758089634321162
RBFSVM Test Accuracy: 0.5571428571428572
----------------------------------------
Optimized Scores

LRC Train Accuracy: 0.5415992258972555
LRC Test Accuracy: 0.5193877551020408
LRC Optimal Parameters:
OrderedDict([('classifier__C', 8.146216961026964)])

DT Train Accuracy: 0.5862606536867621
DT Test Accuracy: 0.6081632653061224
DT Optimal Parameters:
OrderedDict([('classifier__max_depth', 47)])

RFC Train Accuracy: 0.6658967993327599
RFC Test Accuracy: 0.7040816326530612
RFC Optimal Parameters:
OrderedDict([('classifier__max_depth', 47), ('classifier__n_estimators', 755)])

RBFSVM Train Accuracy: 0.5778448927463706
RBFSVM Test Accuracy: 0.5846938775510204
RBFSVM Optimal Parameters:
OrderedDict([('classifier__C', 8.146216961026964), ('classifier__gamma', 'auto')])
```

All optimized models had scores greater than or equal to their basic counterparts with the exception of the RBFSVM. I strong suspect that is due to the low time for which I was able to run it. As I was running this code on google colabs I was not able to run it for long periods of time or on powerful hardware. The documentation for this algorithm stated that it is relatively resource intensive to run.

I did attempt for a while to move my code to a google colabs docker image that I could run locally and leverage my gpu but was unsuccessful in getting it to work.

I also attempted to graph the decision boundaries for each model as I thought it would be a great visualization but I also failed to get that to work properly.