COSC 5557
Hyperparameter Optimization

Almountassir Bellah Aljazwe

April 16, 2024

1 Introduction

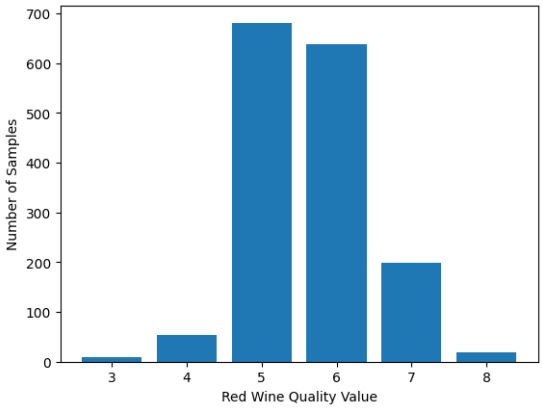
The main objective of finding an optimal predictive model for our "Red Wine Quality" dataset remains in progress. For context, the performance accuracy of five different predictive models were evaluated and compared, in our previous report. What was done, however, is not sufficient in finding an optimal predictive model; this is due to the fact that no hyper-parameter optimization was performed during the previous step. With this in mind, optimizing the hyper-parameters is the necessary upcoming step in accomplishing our main objective.

2 Dataset Description

The 'Red Wine Quality' dataset contains 1599 rows (samples) and twelve columns; eleven are feature columns, while the final twelfth is the target column. The target column contains categorical values; each value is an integer which corresponds to a 'quality' ranking from '1' to '10'. The dataset, however, ranges from '3' to '8' only, with all six values appearing. The dataset contains no missing values for all the provided samples.

A defining characteristic of the 'Red Wine Quality' dataset is its imbalanced distribution of target values. This can be concluded from the following :

3	4	5	6	7	8
10/1559	53/1559	681/1559	638/1559	199/1559	18/1559
$\approx 0.6\%$	$\approx 3\%$	$\approx 44\%$	$\approx 41\%$	$\approx 13\%$	$\approx 1\%$



From the figures, we can clearly observe that the data distribution is cluttered around the middle target values. This, in reality, is reasonable as it may be rare to encounter wine with extremely low quality or extremely high quality; it is reasonable to expect wine with average quality. With regards to our model, the imbalanced nature of this dataset may present obstacles in the performance of our predictions; this is due to the low number of samples for quality values, other than '5' and '6'.

3 Experimental Set-up

Optimizing our models and evaluating the optimized models consists of two independent processes : an evaluation of the optimized models, and a selection of the model hyper-parameters. These processes follow the explanations by Dr. Bernd Bischl in the video titled : "I2ML - Evaluation - Resampling I" and Dr. Sebastian Raschka in the video titled : "11.5 Nested CV for Algorithm Selection (L11 Model Eval. Part 4)". Prior to the processes, no pre-processing was done on the data.

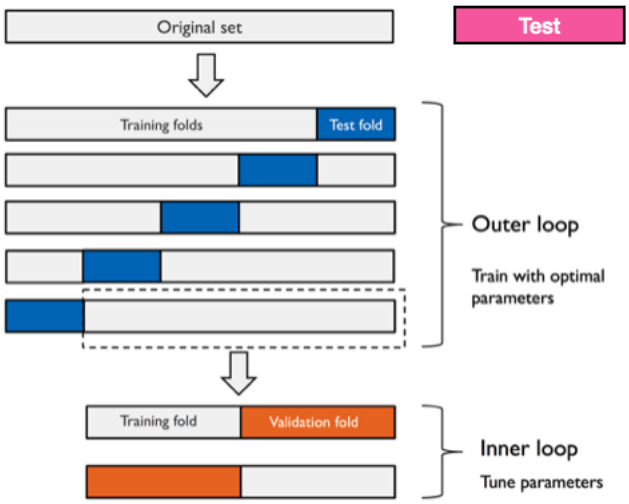


Figure 1: taken from Dr. Sebastian Raschka

3.1 Optimized Model Evaluation

In evaluating the models, the whole data set is used, contrary to the diagram above which leaves out a testing data set. The data set is fed into the nested resampling loops; it consists of an outer 5-fold cross validation loop and an inner 2-fold cross validation loop. For each outer loop, the data set is split up into a 4:1 ratio of training and testing samples. For each inner loop, the training set is split into an "inner-loop" training set and "inner-loop" validation set; the inner loop is used for hyper-parameter optimization.

For each of the five outer loop training sets, a hyper-parameter optimization algorithm is run to find an optimal hyper-parameter configuration. The optimization algorithms used are "Random Search" and "Bayesian Optimization".

In the case of the "Random Search" algorithm, 24 different hyper-parameter configurations are generated randomly from a hyper-parameter space; each model, configured to the randomly generated hyper-parameter configuration, is then evaluated using 2-fold cross validation on the training set of a single outer loop; the hyper-parameter configuration that resulted in the highest average cross validation accuracy score is chosen from the inner loops.

In the case of the "Bayesian Optimization" algorithm, 24 iterations of the algorithm are run; each iteration results in a hyper-parameter configuration that is chosen based off past evaluations (average cross validation accuracy scores from the inner loop folds); once all 24 iterations are run, a single hyper-parameter configuration is chosen from the inner loops.

When a hyper-parameter configuration is returned from the inner loops, the model is configured to the hyper-parameter configuration; then, the model is evaluated on an outer loop testing set; the evaluation result is kept as a single unbiased performance estimate for the single outer loop fold. This process repeats for each of the outer loop folds. At the end, there will be five unbiased performance estimates for each of the five outer loop folds.

Listing 1: Nested Resampling using Random Search for a Single Model

```
from sklearn.model_selection import StratifiedKFold

inner_fold = StratifiedKFold(
    n_splits=INNER_SPLITS,
    shuffle=True,
    random_state=RANDOM_STATE
)

outer_fold = StratifiedKFold(
    n_splits=OUTER_SPLITS,
    shuffle=True,
    random_state=RANDOM_STATE
)

random_search = RandomizedSearchCV(
    random_forest_classifier,
    hyperparameter_distribution,
    cv=inner_fold,
    scoring='accuracy',
    n_iter=RANDOM_SEARCH_ITERATIONS,
    verbose=4
)

rf_rs_performance_estimates = []

for train_index, test_index in outer_fold.split(X, y):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    random_search.fit(X_train, y_train)

    score = random_search.score(X_test, y_test)

    rf_rs_performance_estimates.append(score)
```

Listing 2: Nested Resampling using Bayesian Optimization for a Single Model

```
from sklearn.model_selection import StratifiedKFold, cross_val_score
from baytune import BTBSession

inner_fold = StratifiedKFold(
    n_splits=INNER_SPLITS,
    shuffle=True,
    random_state=RANDOM_STATE
)
outer_fold = StratifiedKFold(
    n_splits=OUTER_SPLITS,
    shuffle=True,
    random_state=RANDOM_STATE
)

rf_bayesian_performance_estimates = []

for train_index, test_index in outer_fold.split(X, y):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    def scoring_function(model_name, hyperparameter_values):
        model_class = models[model_name]
        model_instance = model_class(**hyperparameter_values)
        scores = cross_val_score(
            cv=inner_fold,
            estimator=model_instance,
            X=X_train,
            y=y_train,
            scoring='accuracy',
        )

        return scores.mean()

    session = BTBSession(
        tunables=tunables,
        scorer=scoring_function,
        verbose=True,
    )

    best_result = session.run(BAYESIAN_OPTIMIZATION_ITERATIONS)

    best_model_name = best_result['name']
    best_model_class = models[best_model_name]
```

```
model_instance = best_model_class(**best_result['config'])
model_instance.fit(X_train, y_train)
score = model_instance.score(X_test, y_test)
rf_bayesian_performance_estimates.append(score)
```

3.2 Model Hyper-parameter Selection

Independent of the "Optimized Model Evaluation" process is the hyper-parameter selection process; this process is exclusively focused on the selection of hyper-parameters and is not focused on providing unbiased performance estimates for the optimized models. To select the optimal hyper-parameters for each model, two independent runs of "Random Search" and "Bayesian Optimization" are run on the entire data set and a single hyper-parameter configuration is the result of each .

In the case of the "Random Search" algorithm, 24 different hyper-parameter configurations are generated randomly from a hyper-parameter space; each model, configured to the randomly generated hyper-parameter configuration, is then evaluated using 5-fold cross validation on the entire data set; the hyper-parameter configuration that resulted in the highest average cross validation accuracy score is chosen.

In the case of the "Bayesian Optimization" algorithm, 24 iterations of the algorithm are run; each iteration results in a hyper-parameter configuration that is chosen based off past evaluations (average cross validation accuracy scores from the entire data set); once all 24 iterations are run, a single hyper-parameter configuration is chosen.

Listing 3: HP Selection using Random Search for a Single Model

```
from sklearn.model_selection import StratifiedKFold

cv_fold = StratifiedKFold(
    n_splits=OUTER_SPLITS,
    shuffle=True,
    random_state=RANDOM_STATE
)

random_search = RandomizedSearchCV(
    svc_model,
    hyperparameter_distribution,
    cv=cv_fold,
    scoring='accuracy',
    n_iter=RANDOM_SEARCH_ITERATIONS,
    verbose=4
)

random_search.fit(X, y)

# Output the config with the highest average cv score:
print(random_search.best_params_)

# Output the value of the highest average cv score:
print(random_search.best_score_)
```

Listing 4: HP Selection using Bayesian Optimization for a Single Model

```
from sklearn.model_selection import StratifiedKFold, cross_val_score
from baytune import BTBSession

cv_fold = StratifiedKFold(
    n_splits=OUTER_SPLITS,
    shuffle=True,
    random_state=RANDOM_STATE
)

def scoring_function(model_name, hyperparameter_values):
    model_class = models[model_name]
    model_instance = model_class(**hyperparameter_values)
    scores = cross_val_score(
        cv=cv_fold,
        estimator=model_instance,
        X=X,
        y=y,
        scoring='accuracy',
    )

    return scores.mean()

session = BTBSession(
    tunables=tunables,
    scorer=scoring_function,
    verbose=True,
)

best_result = session.run(BAYESIAN_OPTIMIZATION_ITERATIONS)

# Output the config with the highest average cv score:
print(best_result['config'])

# Output the value of the highest average cv score:
print(best_result['score'])
```

3.3 Hyper-parameter Spaces

The following tables describe the hyper-parameter spaces of which the hyper-parameter optimization algorithms received as inputs, for each predictive model :

Table 1: K-Nearest Neighbors

Hyper-Parameter	Space Type	Range
Number of Neighbors	Integer	[1, 1000]
Weights	Categorical	Uniform or Distance
Algorithm	Categorical	Auto, Ball Tree, KD Tree, Brute
Leaf Size	Integer	[1, 100000]
p	Integer	[1, 100000]

Table 2: Random Forest

Hyper-Parameter	Space Type	Range
Number of Estimators	Integer	[50, 1000]
Criterion	Categorical	Gini, Entropy, Log Loss
Max Depth	Integer	[1, 100]
Minimum Samples Split	Integer	[2, 100]
Minimum Samples Leaf	Integer	[1, 100]
Minimum Weight Fraction Leaf	Real Number	[0.0, 0.5]
Maximum Features	Categorical	Square Root, Log2, None

Table 3: Decision Tree Classifier

Hyper-Parameter	Space Type	Range
Criterion	Categorical	Gini, Entropy, Log Loss
Splitter	Categorical	Best, Random
Max Depth	Integer	[1, 1000]
Minimum Samples Split	Integer	[2, 100]
Minimum Samples Leaf	Integer	[1, 100]
Minimum Weight Fraction Leaf	Real Number	[0.0, 0.5]
Maximum Features	Categorical	Square Root, Log2, None

Table 4: Logistic Regression

Hyper-Parameter	Space Type	Range
Solver	Categorical	LBFGS, Liblinear, Newton-CG, Newton-Cholesky, SAG, SAGA
Penalty	Categorical	L1, L2, Elastic Net, None
C	Real Number	[0.1, 1000]
Maximum Iterations	Integer	[1000, 5000]

Table 5: Support Vector Machine Classifier

Hyper-Parameter	Space Type	Range
C	Real Number	[0.1, 100]
Kernel	Categorical	Linear, RBF, Sigmoid
Degree	Integer	[1, 50]
Gamma	Categorical	Scale, Auto
Coef0	Real Number	[0, 1]
Tolerance	Real Number	[0.001, 1]

3.4 Technologies Used

The source code for all the models, and the "Random Search" algorithm, was taken from the Sklearn Python library. The source code for the "Bayesian Optimization" algorithm was taken from the "baytune" Python library.

4 Results

4.1 Optimized Model Evaluations

As mentioned before, five individual performance estimates (accuracy scores) are stored from each of the five outer loops of the nested resampling evaluation process. For each model, the five performance estimates are plotted :

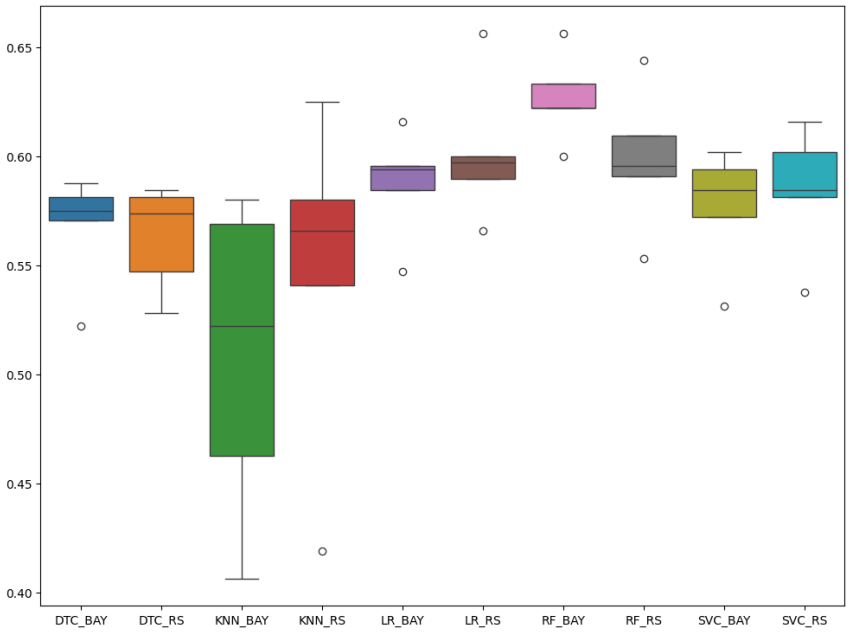


Figure 2: Box-plots of Optimized Model Performance Estimates

To help distinguish the different box-plots, the following is a mapping of each of the labels:

- DTC_BAY : Decision Tree Classifier optimized through the "Bayesian Optimization" algorithm.
- DTC_RS : Decision Tree Classifier optimized through the "Random Search" algorithm.
- KNN_BAY : K-Nearest Neighbors Classifier optimized through the "Bayesian Optimization" algorithm.
- KNN_RS : K-Nearest Neighbors Classifier optimized through the "Random Search" algorithm.

- LR_BAY : Logistic Regression Classifier optimized through the "Bayesian Optimization" algorithm.
- LR_RS : Logistic Regression Classifier optimized through the "Random Search" algorithm.
- RF_BAY : Random Forest Classifier optimized through the "Bayesian Optimization" algorithm.
- RF_RS : Random Forest Classifier optimized through the "Random Search" algorithm.
- SVC_BAY : Support Vector Machine Classifier optimized through the "Bayesian Optimization" algorithm.
- SVC_RS : Support Vector Machine Classifier optimized through the "Random Search" algorithm.

The statistical measurements from the models' performance estimates are divided through the following tables, based on the two optimization algorithms:

Table 6: Random Search

Model	Avg. Accuracy Score	Standard Deviation
Decision Tree Classifier	$\approx 56\%$	≈ 0.02
K-Nearest Neighbors	$\approx 55\%$	≈ 0.07
Logistic Regression	$\approx 60\%$	≈ 0.03
Random Forest	$\approx 60\%$	≈ 0.03
Support Vector Machine	$\approx 58\%$	≈ 0.03

Table 7: Bayesian Optimization

Model	Avg. Accuracy Score	Standard Deviation
Decision Tree Classifier	$\approx 57\%$	≈ 0.02
K-Nearest Neighbors	$\approx 51\%$	≈ 0.07
Logistic Regression	$\approx 59\%$	≈ 0.02
Random Forest	$\approx 63\%$	≈ 0.02
Support Vector Machine	$\approx 58\%$	≈ 0.02

4.2 Model Hyper-parameter Selection

The following tables provide the results of the hyper-parameter selection process that was run for each model through two types of optimization algorithms :

Table 8: Random Search

Model	Avg. 5-CV Accuracy	HP Configuration
Decision Tree Classifier	$\approx 56\%$	Splitter : Best Criterion : Gini Max. Depth : 399 Min. Samples Leaf : 68 Min. Samples Split : 31 Min. Weight Fraction Leaf : ≈ 0.12 Max Features : None
K-Nearest Neighbors	$\approx 57\%$	Neighbors : 223 Weights : Distance Algorithm : Brute Leaf Size : 70588 p : 6277
Logistic Regression	$\approx 60\%$	Solver : LBFGS C : 647.64 Max Iterations : 2046 Penalty : None
Random Forest	$\approx 60\%$	Estimators : 331 Criterion : Log Loss Max Depth : 30 Min. Samples Split : 58 Min. Samples Leaf : 21 Min. Weight Fraction Leaf : ≈ 0.07 Max. Features : Sqrt
SVM Classifier	$\approx 59\%$	C : ≈ 56.57 Coef0 : ≈ 0.26 Degree : 35 Gamma : Auto Kernel : Linear Tolerance : ≈ 0.64

Table 9: Bayesian Optimization

Model	Avg. 5-CV Accuracy	HP Configuration
Decision Tree Classifier	$\approx 60\%$	Splitter : Best Criterion : Gini Max. Depth : 281 Min. Samples Leaf : 9 Min. Samples Split : 36 Min. Weight Fraction Leaf : ≈ 0.003 Max Features : None
K-Nearest Neighbors	$\approx 57\%$	Neighbors : 475 Weights : Distance Algorithm : Brute Leaf Size : 27106 p : 4427
Logistic Regression	$\approx 59\%$	Solver : Newton-CG C : 0.1 Max Iterations : 1000 Penalty : L2
Random Forest	$\approx 66\%$	Estimators : 513 Criterion : Gini Max Depth : 58 Min. Samples Split : 10 Min. Samples Leaf : 4 Min. Weight Fraction Leaf : ≈ 0.0006 Max. Features : Sqrt
SVM Classifier	$\approx 59\%$	C : ≈ 2.33 Coef0 : ≈ 0.66 Degree : 23 Gamma : Auto Kernel : Linear Tolerance : ≈ 0.44

The following table compares the average 5-fold cross validation accuracy scores for the non-optimized predictive models and the average 5-fold cross validation accuracy scores for the (Bayesian) optimized predictive models. The scores from the Bayesian-optimized models were chosen as they had better results, for the majority:

Table 10: Non-optimized vs. Optimized Model Comparison

Model	Non-optimized Accuracy Score	Optimized Accuracy Score
Decision Tree Classifier	$\approx 55\%$	$\approx 60\%$
K-Nearest Neighbors	$\approx 44\%$	$\approx 57\%$
Logistic Regression	$\approx 57\%$	$\approx 59\%$
Random Forest	$\approx 59\%$	$\approx 66\%$
SVM Classifier	$\approx 50\%$	$\approx 59\%$

4.3 Conclusion

From our results, we can analytically observe a performance increase after optimizing each of the models. Most of the models had noticeable improvements in performance after hyper-parameter optimization. The highest optimized score was 66%, which was obtained by the "Random Forest" model; the lowest average optimized performance estimate was 57%, which was obtained by the "K-Nearest Neighbors" model.