Practical Machine Learning

Exercise 4
Spring 2024

Lisa Stafford

March 15, 2024

Abstract

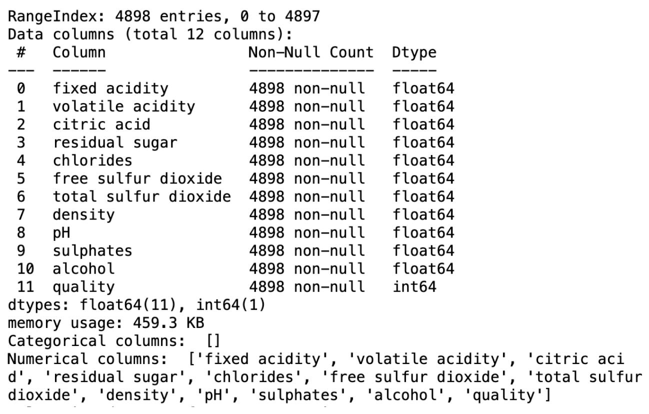
Hyper-parameter selection is a common problem for machine learning architects where one selects the model that does the best job of generalizing and most appropriate for the given problem, but without proper knowledge of hyper-parameters within the model and what they actually do to affect the model performance, many users new to machine learning may be unaware how to select or tune hyper-parameters correctly, and most non-data scientists don't know (or may not be aware of without significant trial and error) how those hyper-parameters affect the overall runtime and performance of their model. After selecting a base model, hyper-parameter selection remains an important part of the machine learning process.

Introduction

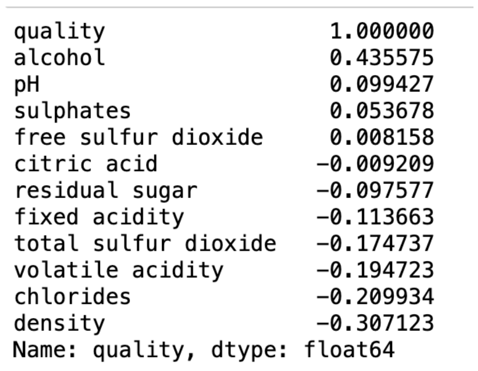
In order to learn and determine which hyper-parameters have the greatest overall effect upon the model run-time, resource use, and performance, we seek to isolate hyper-parameter selection by isolating variables in which to evaluate the performance, resource use, and runtime without changing underlying factors that may also affect model performance. Throughout our investigation, we used a wine data set obtained from the University of Irvine [1]. Training is performed on only the white wine within the data set. Several models are trained implementing hyper-parameter optimization libraries directly from scikitlearn [4] and then obtaining total runtime, and training and testing performance scores and tuned hyper-parameter choices for a number of hyper-parameter tuning mechanisms.

Dataset Description

Our wine data set is actually comprised of two different wine subsets of data. While both are related to variants of [1] "Portuguese Vhinho Verde wine" the data is actually split in two subsets - one red subset and one white subset. In this exercise, we will only be performing machine learning tasks on the white wine data set since the parameter tuning time should be extensive for each run, and instructions indicate we must not be able to use a grid search on the hyperparameter space. All wine subset contains 11 features and 1 label for wine quality. All features are continuous float values. Within the model the "wine quality" labels are recognized as discrete categorical integer values ranging from 3 to 9 for the white wines. White wine contains a total of 4898 total data instances, as shown in the following table images. Figure 1 displays feature values, statistics, and total instances for white wine. The image to the right shows feature correlation to the output label.

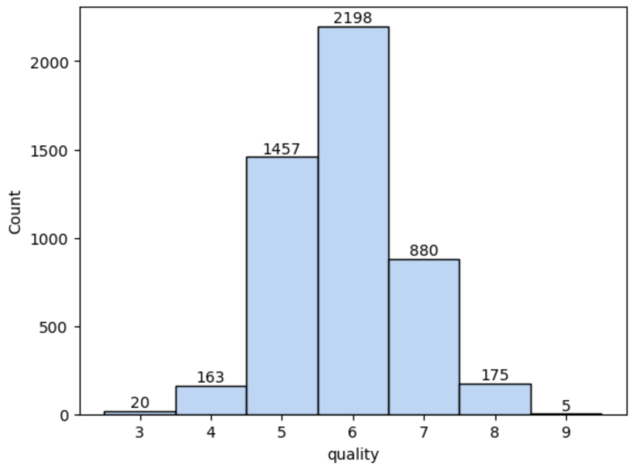


```
RangeIndex: 4898 entries, 0 to 4897
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   fixed acidity         4898 non-null   float64
 1   volatile acidity      4898 non-null   float64
 2   citric acid           4898 non-null   float64
 3   residual sugar        4898 non-null   float64
 4   chlorides             4898 non-null   float64
 5   free sulfur dioxide   4898 non-null   float64
 6   total sulfur dioxide  4898 non-null   float64
 7   density               4898 non-null   float64
 8   pH                    4898 non-null   float64
 9   sulphates             4898 non-null   float64
 10  alcohol               4898 non-null   float64
 11  quality               4898 non-null   int64
dtypes: float64(11), int64(1)
memory usage: 459.3 KB
Categorical columns:  []
Numerical columns:  ['fixed acidity', 'volatile acidity', 'citric aci
d', 'residual sugar', 'chlorides', 'free sulfur dioxide', 'total sulfur
dioxide', 'density', 'pH', 'sulphates', 'alcohol', 'quality']
```



```
quality                 1.000000
alcohol                 0.435575
pH                      0.099427
sulphates               0.053678
free sulfur dioxide     0.008158
citric acid            -0.009209
residual sugar         -0.097577
fixed acidity          -0.113663
total sulfur dioxide   -0.174737
volatile acidity       -0.194723
chlorides              -0.209934
density                -0.307123
Name: quality, dtype: float64
```

As mentioned from prior exercises, white wine data is somewhat "pre-cleaned". The feature values are all continuous and contain no missing data values, however these features are not evenly distributed. The data values are somewhat bell-shaped with a larger number of data instances with labels closer to the mean. As described on the data set website [1] both "data set... classes are ordered and []unbalanced] so there are many more normal wines than excellent or poor ones.... [also], both datasets have 11 physiochemical features... and a sensory output label, ('quality')". The following histogram shows the distribution of class instances within the dataset:



Standard Scaling worked best as a data transformation method in Exercise 2, so we will use it in this exercise since we would like to automate and optimize our performance on a Random Forest Classifier from the ensemble library in ScikitLearn [4] and K-Nearest Neighbors classifier from the neighbors library. Though the whole of the search relies on a decision tree, sub-estimators are reliant on other classifiers like SVMs which have been shown in previous exercises to be sensitive to unscaled data. We want to experiment with subsets of most important features obtained from an updated version of our data set exploration exercise since this could have significant impact on overall performance for any classifier. The Random Forest Classifier also automates feature correlations noted in a prior chart when deciding which features to extract or drop or group from the dataset per it's 'max_features' parameter which we'll set as a hyperparameter for optimization. Overall, dropping unimportant or redundant features will not only help with the speed at which the model optimization completes, it may also provide better performance if the features are not that important in label prediction, and may help performance if the feature is redundant or very similar to another feature already contained within the dataset. This would need to be programmed into capabilities for k-neighbors.

Experimental Setup

We are looking at the wine quality data set, specifically, the white wines in the data set. From there, we're trying to predict wine quality for items within the data set to determine the best hyper-parameters and methods for finding them are done so in the most optimized way.

We know that this particular data set is supervised and the problem is a classification problem with a known number of categories. Therefore we know that the best ML models will be in that realm. In prior exercises Random Forest Classifiers and KNN worked best and most consistently therefore we will attempt to optimize the hyperparameters for these 2 classification model types. We have a large number of hyperparameters to search through given the two model options. For larger continuous hyperparameters, we will iterate through certain values manually selected. Method of optimizing our hyperparameters will include:

1. Initially perorming a Randomized Search with 5-Fold Cross Validation ($RandomizedSearchCV()$) on the model and listed hyperparameters in order to pre-find some hyperparameters that perform well. This should be easy to do since the $RandomizedSearchCV()$ appears to take much less

time on average than other hyperparameter optimization searchers. On average, the Random search optimizer takes around 1 minute to complete, therefore we will run it 10 times and see if we can note any patterns.

2. Attempting (but possibly failing) when pretuning Hyperparameters (Metalearning) by finding an initial list of the best subsets of hyperparameters by performing a 5-Fold Cross Validated Grid Search (*GridSearchCV*) on hyperparameters to find those that perform most optimally on smaller subsets of the dataset. This may fail because the algorithm for grid search may still be too complex and time consuming to perform, even on a smaller dataset.

3. Running the out of the box Bayesian Search with 5-Fold Cross Validation (*BayesSearchCV*()) available in the skoptimizer library.

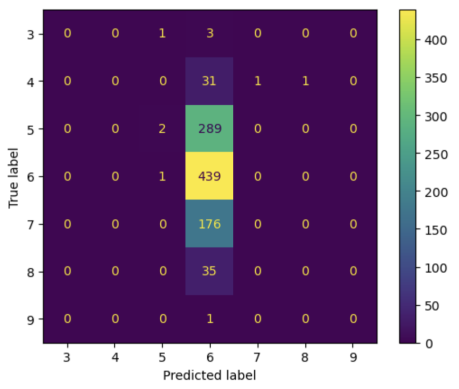
Searchers were conducted on a list of classifier hyperparameters populated from a python *dict*() as follows.

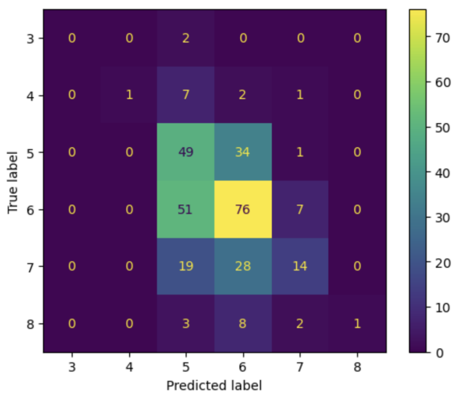
Random Forest Classifier Hyper-parameter Options (left): KNN (right)

n_estimators:	50, 100, 200, 300
criterion:	'gini', 'entropy', 'log_loss'
max_depth:	None, 15, 25, 50
bootstrap:	True, False
min_samples_split:	5, 10
min_samples_leaf:	1, 2, 5
min_weight_fraction_leaf:	0.001, 0.01, 0.1
class_weight	'balanced', 'bal_sub
max_features:	'sqrt', 'log2', None
n_jobs:	-1

n_neighbors:	2,5,10,25,50
weights:	uniform, distance
algorithm:	ball_tree, kd_tree, auto
leaf_size:	5,10,15,20,30,40,50
n_jobs:	-1

Since the RandomSearchCV for hyperparameter optimization performs very quickly (our average run for a Random Search is under 1 minute), we use it as a base with our full dataset to get a very general idea of which hyperparameters perform the best by running it over 10 iterations and reporting resulting "best hyperparameters". After only a few runs, it becomes very clear that certain hyperparameter settings (such as setting minimum sample leafs to anything over 2 or min_weight_fraction_leaf over .1) resulted in a vastly worse model and notable drop in accuracy scores. Similar performance is noted with KNN for all less common classes within the dataset, but also an overall lack of accuracy was noted for KNN. Examples of improper fits are shown below for Random Forest on the left and KNN on the right:



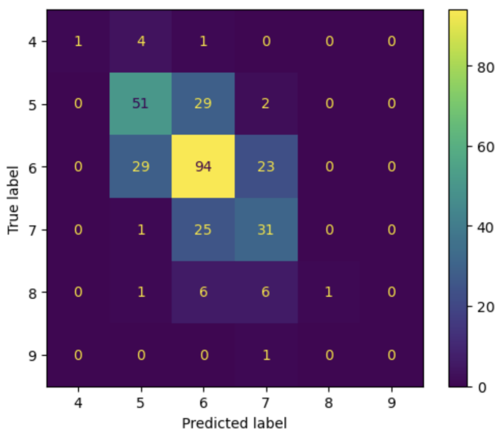


The resulting performance when setting sample leafs to higher values is now obvious in hindsight. This is because several outlier label classes (like class 3 or class 9) had very few instances to begin with and therefore the model becomes extremely biased towards the most commonly "seen" labels (in this dataset, that's wine qualities 5 and 6). For RandomForestClassifier(), Setting the min_sample_leaf to anything over 5 or min_weight_fraction_leaf over .1 will result in the uncommon classes simply being disregarded as a possibility. Therefore these options are completely removed before running any further hyperparameter tuning. Since the overall performance shown between these two models seems to be more consistent and promising with Random Forest Classifiers, we proceed with this particular model, and pitch our work on the KNN model.

To follow up, we attempt to perform a grid search upon a subset of data beginning with a 25% random sample. It is in hopes that this will help us gain more information through "metalearning". Unfortunately, this fails to pay off, as the grid search never completes, despite only being fed 25% of the total dataset. We again attempt to metalearn with the smaller red wine dataset since it has 1600 instances instead of the white wine's 4900, however this also fails to complete. The total number of hyperparameters in our search space is simply too large, regardless of the total number of instances in the dataset. Therefore, we attempt metalearning again using the bayesian search cv library in SciKit-Optimizer [5] and again perform the optimization on a 25% subset of the full white wine dataset. This metalearning variant of the BayesSearchCV run results in a 58% accuracy rate with the following recommended hyperparameters:

Training Accuracy:	99.5%
Test Accuracy:	58.17%
N Estimators:	300
Class Weight:	Balanced Subsample
Bootstrap:	True
Criterion:	Entropy
Max Depth:	50
Min Sample Leaf:	1
Min Samples Split:	5
Max Features:	log2
Min Weight Fraction Leaf:	0.001
N Jobs:	-1
Fit Runtime:	128.78 seconds

While this particular search did not perform well with outlier labels, this is expected, since a subsample of the full dataset will have very few outlier labels available for training. A confusion matrix for this bayesian search on hyperparameters shows the following performance:



Given the speed at which this completes, we re-sample the subset of data from the full dataset and perform the same search again over 10 iterations to see if there are any patterns of note. Over the course of 10 runs we get the following results:

Run #:	1	2	3	4	5	6	7	8	9	10
Train Acc:	.995	.995	.9945	.993	.999	.997	.997	.993	.998	.997
Test Acc:	.5817	.572	.542	.621	.56	.6	.621	.621	.56	.54
#Estims:	300	300	50	100	300	300	300	300	300	300
Class Wt:	BS	BS	BS	B	BS	BS	BS	BS	B	BS
Bootstrap:	True	" "	" "	" "	" "	" "	" "	" "	" "	" "
Criterion:	Ent.	LL	LL	Ent	Ent	LL	Ent	Ent	LL	Ent
Mx Depth:	50	50	-1	-1	50	-1	-1	50	-1	50
Min Smpl Lf:	1	2	1	1	1	2	2	1	1	1
Min Smpl Split:	5	" "	" "	" "	" "	" "	" "	" "	" "	" "
Mx Feat:	log	$\sqrt{f}$	log	$\sqrt{f}$	log	$\sqrt{f}$	log	$\sqrt{f}$	$\sqrt{f}$	$\sqrt{f}$
Min Wt Frac Lf:	0.001	" "	" "	" "	" "	" "	" "	" "	" "	" "
#Jobs:	-1	" "	" "	" "	" "	" "	" "	" "	" "	" "
Fit Time (sec:)	128.8	119.6	149.4	93.17	141.4	136.5	115.7	130.54	121.8	146.35

From this we may make some reasonable assumptions:

1. We can remove the smallest value from the number of estimators since that never resulted in an optimal score over the course of any of our 10 searches.
2. Setting bootstraping to True always resulted in the optimal hyperparameter over the course of 10 searches of the hyperparameter space, therefore we remove the False option (which is the default) and set it to True for all cases.
3. The model always performed best when Minimum Weight Fraction Leaf hyperparameter was set to the lowest value so we remove it from our search space since the default for a Random Forest Classifier model will set it to 0.
4. The search always found the minimum sample split optimal value to be 5 for all 10 runs, therefore we remove values higher than 5, and add a lower value to see if that results in any change to performance.
5. The model always performs better with a higher maximum depth and having it set to None does not appear to have any significant affect on the overall runtime, therefore we remove the hyperparameters from the search space so the model may take simply the default - None.

After making these changes to our hyperparameter search space, we run 5 runs of our Random Search CV on the new subset. Two things are notable in the results of our random search:

1. Model test accuracy has improved on average from .58 in our original run to .685 in the updated randomized search run. This is a significant improvement.
2. Average run time for fitting using randomized search remains around 1 minute or less, so we do not appear to have negatively impacted overall runtime for hyperparameter tuning.
3. We have yet to ever see 'gini' criterion perform optimally, despite being the default hyperparameter for the model. While this is the case, because we don't fully understand the difference between 'entropy' and 'gini' and entropy has performed well in prior instances, we leave it in the search space.

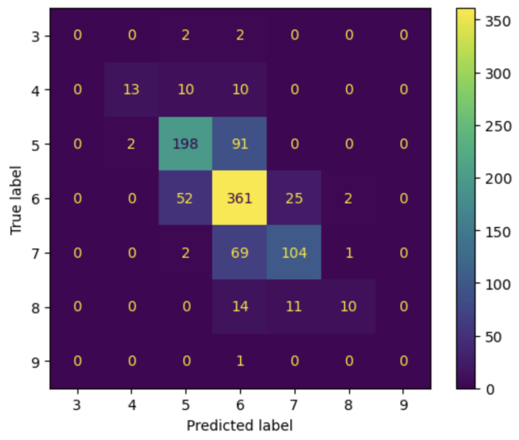
Given our results and improvements with our updated hyperparameter search space, we perform our bayesian search on our data subsets another 10 iterations, again noting performance.

Run #:	1	2	3	4	5	6	7	8	9	10
Train Acc:	1	.998	.997	.99	1	.992	.999	.995	.999	.998
Test Acc:	.59	.575	.61	.593	.595	.585	.588	.628	.614	.58
#Estims:	300	300	100	300	300	300	300	300	300	300
Class Wt:	BS	B	BS	B	B	BS	B	B	BS	B
Bootstrap:	True	” ”	” ”	” ”	” ”	” ”	” ”	” ”	” ”	” ”
Criterion:	LL	Ent	LL	Ent	Ent	LL	LL	LL	LL	LL
Min Smpl Lf:	1	2	1	2	1	2	1	2	2	1
Min Smpl Split:	2	2	2	5	4	5	2	2	5	2
Mx Feat:	$\sqrt{f}$	log	$\sqrt{f}$	$\sqrt{f}$	log	$\sqrt{f}$	$\sqrt{f}$	$\sqrt{f}$	$\sqrt{f}$	$\sqrt{f}$
#Jobs:	-1	” ”	” ”	” ”	” ”	” ”	” ”	” ”	” ”	” ”
Fit Time (sec:)	139.46	129.04	119.8	106.5	122.56	120.22	121.61	128.46	138.9	126.11

This time we don't notice much of a change to performance, nor to runtime. The number of estimators is mostly consistent with best results using 300 estimators, so we remove the other options from the hyperparameter search space. With this in place, we can now run the Bayesian Search on the entire dataset. Our final model results in the following for 3 runs:

Run #:	1	2	3
Training Accuracy:	1.0	.999	.999
Test Accuracy:	.69	.69	.69
Class Weight:	Bal Subsample	Bal Subsample	Bal Subsample
Bootstrap:	True	True	True
Criterion:	Entropy	Entropy	LL
Min Sample Leaf:	1	1	2
Min Samples Split:	5	5	2
Max Features:	$\sqrt{f}$	$\sqrt{f}$	$\sqrt{f}$
Fit Runtime (sec):	232.4	254.7	214.33

Finalized Performance looks as follows:



```
Roc_Auc_Score for wine quality 3 is: 0.8041752049180327
Roc_Auc_Score for wine quality 4 is: 0.8898595244952161
Roc_Auc_Score for wine quality 5 is: 0.9007825475438781
Roc_Auc_Score for wine quality 6 is: 0.8369381313131313
Roc_Auc_Score for wine quality 7 is: 0.9109177125734962
Roc_Auc_Score for wine quality 8 is: 0.9023431594860166
Roc_Auc_Score for wine quality 9 is: 1.0
```

References

[1] A. Asuncion, D. Newman, UCI Machine Learning Repository, University of California, Irvine (2007). Obtained from https://archive-beta.ics.uci.edu/dataset/186/wine+quality.

[2] C. Harris, K. Millman, S. van der Walt, Array programming with NumPy. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2. https://numpy.org/doc/stable/reference/generated/numpy.isnan.html

[3] M. Waskom, (2021). seaborn: statistical data visualization. Journal of Open Source Software, 6(60), 3021, https://doi.org/10.21105/joss.03021.

[4] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

[5] ScikitLearn-Optimizer. (2021). obtained from: https://scikit-optimize.github.io/stable/modules/generated/skopt.BayesSearchCV.html