Pipeline Optimization Report

Soudabeh Bolouri

Introduction:

This exercise involves a comprehensive approach to predicting wine quality utilizing multiple ML algorithms including SVM, Random Forest, KNN, and Logistic Regression. The workflow consists of several key steps, including data loading, preprocessing, model creation, evaluation, hyperparameter tuning, and performance comparison visualization.

Dataset Description:

The dataset that we utilized in this exercise is the "Wine Quality" dataset, precisely the "white" wine version. It contains data on different features of white wines, including fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide, density, PH, sulphates, and alcohol. The dataset includes a totality of 4,898 rows and 12 features. The "quality" of the wine is the target variable, which we desire to predict. Also, we did not find any missing values in the dataset.

Experimental Setup:

We used the Python programming language for this exercise. First, we need to import our dataset "winequality-white.csv" using Panda. The dataset is then split into training and testing sets with an 80-20 ratio using the **train_test_split** function from Scikit-Learn. In the next step, the dataset is preprocessed to ensure its suitability for model training. Missing values are replaced with the mean of the column via an imputer (this is a general case for implementing pipelines even though there are no missing values for this specific dataset). As a way of standardizing features, it uses the Standard Scaler to remove the mean and scale to unit variance. Then feature scaling and one-hot encoding are performed using **StandardScaler** and **OneHotEncoder** respectively.
Four classifiers are assessed in this analysis:

- Support Vector Machine (SVM)
- Random Forest
- Logistic Regression
- K-Nearest Neighbors (KNN)

The evaluation of classifiers involves assessing accuracy and training time, both before and after hyperparameter optimization.

The initial models, known as the "simple pipeline," are evaluated using the test set to measure their predictive accuracy. Additionally, the time taken for fitting the simple pipeline to the training data is recorded, offering insights into models' computational efficiency.

As a comparison, another method in hyperparameter optimization is applied to the pipeline to see if there is an improvement in accuracy. RandomizedSearchCV is employed for hyperparameter tuning on each classifier to identify the best combination of hyperparameters.

The hyperparameters and their values to be searched for each model are as follows:

SVM:

Hyperparameter	**Values**	**Description**
classifier__C	[0.001, 0.01, 10, 100]	This parameter controls the trade-off between maximizing the margin and minimizing classification error.
classifier__gamma	[0.1, 0.01, 0.001, 0.0001]	This parameter defines the influence of a single training example's distance, affecting the reach of the kernel
classifier__kernel	['linear', 'rbf', 'poly']	Kernel specifies the type of kernel used in the SVM algorithm, determining the decision boundary shape in the feature space.Three kernel options are considered here: **'linear'**, **'rbf'** (Gaussian kernel), and **'poly'** (polynomial kernel).

Random Forest:

Hyperparameter	**Values**	**Description**
classifier__n_estimators	[5, 20, 50, 100]	number of trees in the rand om forest
classifier__max_features	['auto', 'sqrt']	number of features in consideration at every split
classifier__max_depth	[int(x) for x in np.linspace(10, 120, num = 12)]	maximum number of levels allowed in each decision tre e
classifier__min_samples_split	[2, 6, 10]	minimum sample number t o split a node

classifier__min_samples_leaf	[1, 3, 4]	minimum sample number that can be stored in a leaf node
classifier__bootstrap	[True, False]	method used to sample data points

Logistic Regression:

Hyperparameter	**Values**	**Description**
classifier__penalty	['l1', 'l2', 'elasticnet', 'none']	Regularization term, specifying the nor m used in the penalization
classifier__C	np.logspace(-4, 4, 20)	Inverse of regularization strength, a positive float value. It controls the regularization parameter, where smaller values specify stronger regularization
classifier__solver	['lbfgs','newton-cg','liblinear' ,'sag','saga']	Algorithm to use in the optimization pr oblem
classifier__max_iter	[100, 1000,2500, 5000]	Maximum number of iterations for the solvers to converge

KNN:

Hyperparameter	**Values**	**Description**
classifier__n_neighbors	np.arange(2, 30, 1)	Number of neighbors to consider. It de fines the value of 'k' in the KNN algori thm.
classifier__weights	['uniform', 'distance']	The weight function used in prediction

Results:

For each model, we report:

- The time taken to fit the pipeline without optimizing hyperparameters.
- The accuracy was achieved without the use of hyperparameters.
- Hyperparameters with the best performance identified after optimization.
- The time taken to optimize the pipeline with hyperparameters.
- Achieved accuracy after tuning hyperparameters.

After going through each of these hyperparameters using Random Search, the best hyperparameters have been printed as follows:

Models	**Best Hyperparameters**
SVM	• classifier__kernel: 'rbf ‘ • classifier__gamma: 0.1 • classifier__C: 100
Random Forest	• classifier__n_estimators: 50 • classifier__min_samples_split: 6 • classifier__min_samples_leaf: 3 • classifier__max_features: 'sqrt' • classifier__max_depth: 110 • classifier__bootstrap: False
Logistic Regression	• classifier__solver: 'saga' • classifier__penalty: 'l1' • classifier__max_iter: 100 • classifier__C: 29.763514416313132
KNN	• classifier__weights: 'distance' • classifier__n_neighbors: 18

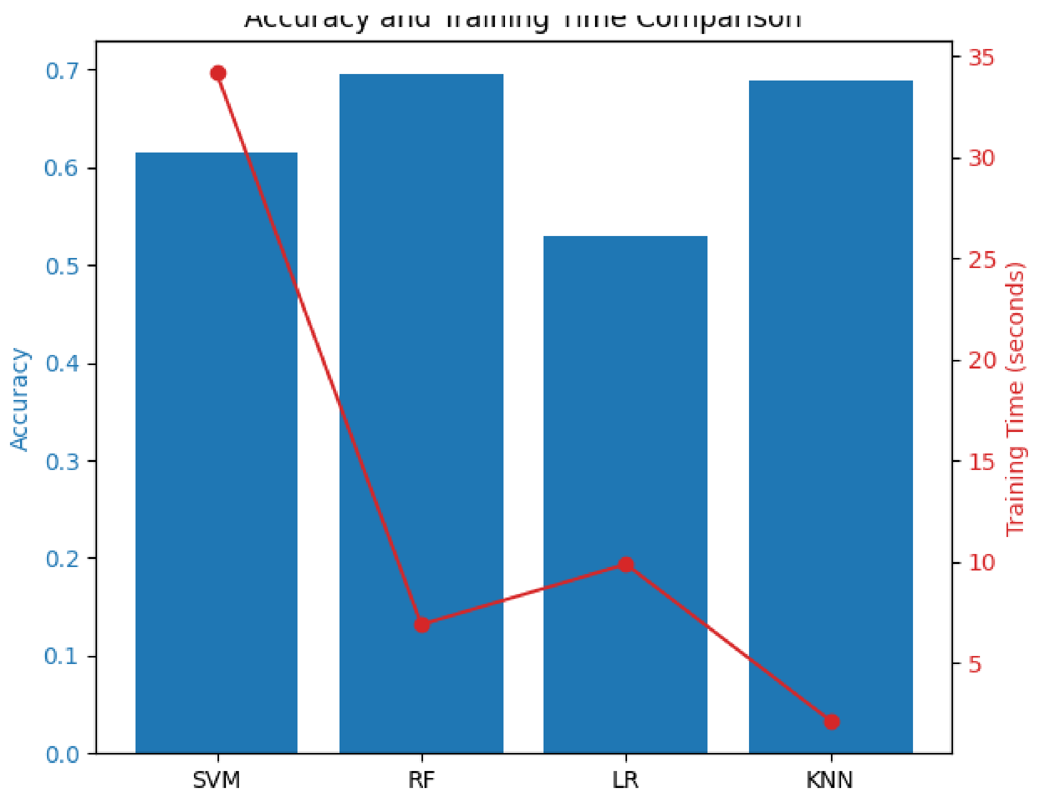
SVM and KNN models show a significant improvement in accuracy after hyperparameter tuning compared to the initial results while the most accurate classifier is Random Forest. In the table below, the results are presented as follows:

Models	**Accuracy (Percent)**	**Time (Sec)**
SVM without HPO	0.5612	0.5364
SVM with HPO	0.6153	34.1713
RF without HPO	0.6959	0.5539
RF with HPO	0.6949	6.8960
LR without HPO	0.5306	0.0824
LR with HPO	0.5296	9.8787
KNN without HPO	0.5429	0.0111

KNN with HPO	0.6878	2.1306

We generated informative visualizations to present the findings (The simple pipeline means the one without hyperparameter optimization, while the optimized pipeline uses Random Search as HPO):

- The bar chart visually compares the accuracy scores achieved by simple and optimized pipelines, offering a clear view of their performance differences.
- The line plot illustrates the training times for both pipelines, highlighting the trade-off between model accuracy and computational cost.



References:

[1] https://towardsdatascience.com/step-by-step-tutorial-of-sci-kit-learn-pipeline-62402d5629b6

[2] https://medium.com/mlearning-ai/how-to-use-sklearns-pipelines-to-optimize-your-analysis-b6cd91999be

[3] https://scikit-learn.org/stable/modules/grid_search.html

[4] https://towardsdatascience.com/hyper-parameter-tuning-and-model-selection-like-a-movie-star-a884b8ee8d68

[5] https://towardsdatascience.com/nested-cross-validation-hyperparameter-optimization-and-model-selection-5885d84acda

[6] https://machinelearningmastery.com/nested-cross-validation-for-machine-learning-with-python/

[7] https://www.kaggle.com/code/arjunprasadsarkhel/simple-random-forest-with-hyperparameter-tuning

[8] https://www.kaggle.com/code/funxexcel/p2-logistic-regression-hyperparameter-tuning

[9] https://medium.com/@agrawalsam1997/hyperparameter-tuning-of-knn-classifier-a32f31af25c7