Report: Mini Project

Introduction

In this project we aim to predict wine quality from various features (acidity, citric acid, alcohol etc.). The dataset has been downloaded from Kaggle. Wines are being categorized into three different classes (Good, Average, and Below Average) and hence, we model this task as a multiclass classification problem. Python's Scikit-learn library has been used for implementing the ML models and Matplotlib has been used to draw charts. Training has been performed using the API functionalities and therefore, no learning curve over different epochs has been generated. However, we implemented three different approaches (hold-out validation, GridSearchCV, and coordinate descent) for hyperparameter tuning.

Problem Formulation

The dataset contains 6497 instances of red and white wine. Each instance has 12 features (the first twelve columns of winequalityN.csv file). The final column is for quality of the wine (a score between 0 and 10), which we need to predict.

While formulating the task, we faced a critical issue. According to the dataset wine quality is defined as a score between 0 and 10. However, a quick observation reveals that only four classes hold the major portion of the data. Figure 1 shows the scenario:

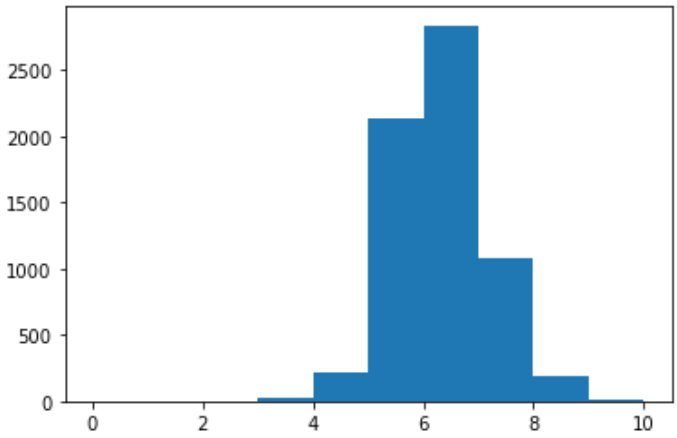


Fig 1: 96% of the dataset lies between quality=4 and quality=7.

Therefore, it is not a feasible idea to use this dataset for a ten-class classification problem. Furthermore, scoring wine using some numbers does not have any real-life implications. So, we decided to formulate the problem in a slightly different manner. Using the numeric scores, we classified the wines in three different classes: Good, Average, and Below Average.

Wine quality score	Class
<=5	2 (Below Average)
6	1 (Average)
>=7	0 (Good)

This approach solves the problem associated with imbalanced distribution of dataset. After grouping the dataset into three classes the histogram is more balanced than the previous one.

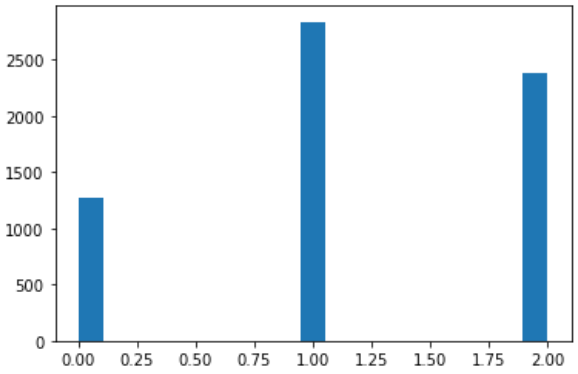


Fig 2: Histogram of three classes.

However, we still have some imbalance in the dataset (*Average* class is around 2.5 times larger than the *Good* class) and as our experiment yields, this eventually affects the model performance.

Approaches and Baseline

We have compared four different ML models to a baseline.

1. Gaussian Naïve Bayes:

Primary hyperparameter for Gaussian Naïve Bayes is the *prior probabilities of the classes*. From the scikit-learn library documentation, the parameters for prior probabilities (mean and variance) are estimated using maximum likelihood. In our case, prior probabilities of classes is just counting. So, we have nothing to tune in this scenario. However, there is another hyperparameter, *var_smoothing*, as specified in the documentation. This is used for calculation stability. We tuned this one using **hold-out validation** (just as the task we did on coding assignment 1, problem 2b).

2. K-nearest Neighbors:

We tuned two hyperparameters for this model. These are:

- n_neighbors: number of neighbors to use for the K-NN algorithm (integer value).
- weights: weight function used in prediction. The values are discrete, either **distance** or **uniform**.

We implemented the ***coordinate descent*** method for tuning multiple hyperparameters. First, *weights* is tuned, assuming *n_neighbors= 5* (the default value for scikit-learn K-NN implementation). In this step we find that, for our task the best value for *weights* is **distance**, which weights points by the inverse of their distance. Then using the tuned value for *weights*, *n_neighbors* is tuned. We did not repeat this tuning process further, since according to our task definition, it makes more sense to have less influence from farther neighbors (using *weights= distance*) instead of uniformly emphasizing all the neighbors (using *weights= uniform*).

3. Decision Tree:

Scikit-learn documentation for decision tree algorithm offers twelve different hyperparameters to be tuned. We choose to tune four hyperparameters and keep the rest as default. We tuned:

- criterion: the function to measure the quality of split. The values are discrete, either **gini** impurity or **entropy** can be used for information gain.
- max_depth: maximum depth of the tree. Longer decision trees are more likely to overfit the training data. Hence, it is important to tune this hyperparameter.
- min_samples_split: the minimum number of samples required to split an internal node. The default value is two, which means if any terminal node has more than two observations and is not a pure node, we will split it further. Since this default value is too small, it may cause unnecessary splitting and eventually overgrowing the tree.
- min_samples_leaf: the minimum number of samples that should be present in the leaf node after splitting a node. Just like the previous one, increasing the value of this hyperparameter until a certain point also helps preventing overfitting.

Again, we used coordinate descent approach. Among the four hyperparameters, *max_depth* affects the model performance more often. So, we decided to tune this one in the first step. The pseudocode for hyperparameter tuning is as follows:

Pseudocode: Hyperparameter tuning for decision tree model

1. Keep all other hyperparameters as default and tune *max_depth.* The tuning has been done using an array of increasing values ([10, 60, 110…, 500]).
2. Using the *max_depth* value obtained from previous step, tune rest three hyperparameters. We used the *GridSearchCV* library function for this task. This is a brute force approach which requires a lot of time.
3. Then using the optimized hyperparameters set from step 2, we tuned *max_depth* again.

We tried repeating step 2 and 3 in a loop, but no noticeable development in accuracy has been obtained. It also requires too much time to execute because of the brute force searching.

4. Random Forest:

Since random forest is just an optimized version of decision tree, we used the optimized hyperparameters from the decision tree implementation to build the forest. We also tuned a crucial hyperparameter for random forest, which is *n_estimators* (number of trees in the forest). As expected, random forest performed much better than decision tree.

Finally, we did a simple experiment. Since, random forest provides the highest performance, we combined the training and validation dataset to train a random forest without any hyperparameter tuning. This resulted into the highest test-accuracy among all of these implemented models, that even without any kind of validation. This shows the necessity of having sufficient training data for better model performance.

Evaluation and Result:

We used accuracy as the evaluation metric. Our task is to classify wines and hence, accuracy is the real goal for this task. We used majority guess as the baseline method. Following table summarizes the result:

SI	Approach	Validation Accuracy	Test Accuracy
1	Predicting the most frequent label (Baseline)	0.43	
2	Gaussian Naïve Bayes	0.43	0.45
3	K-nearest Neighbors	0.62	0.63
4	Decision Tree	0.63	0.59
5.1	Random Forest	0.70	0.68
5.2	Random Forest (larger training set and no validation)	N/A	0.73

Following bar chart shows the performance comparison among the five models-

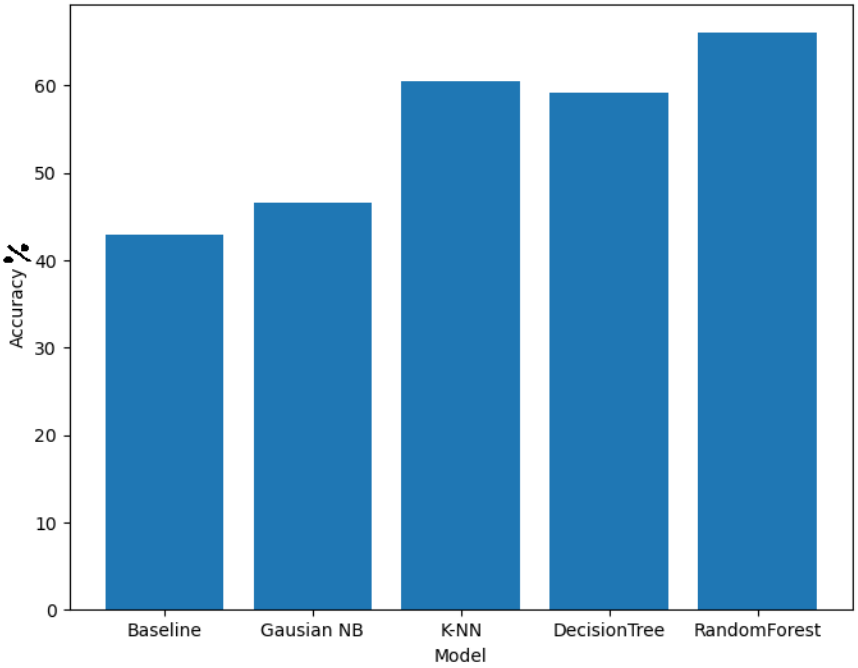
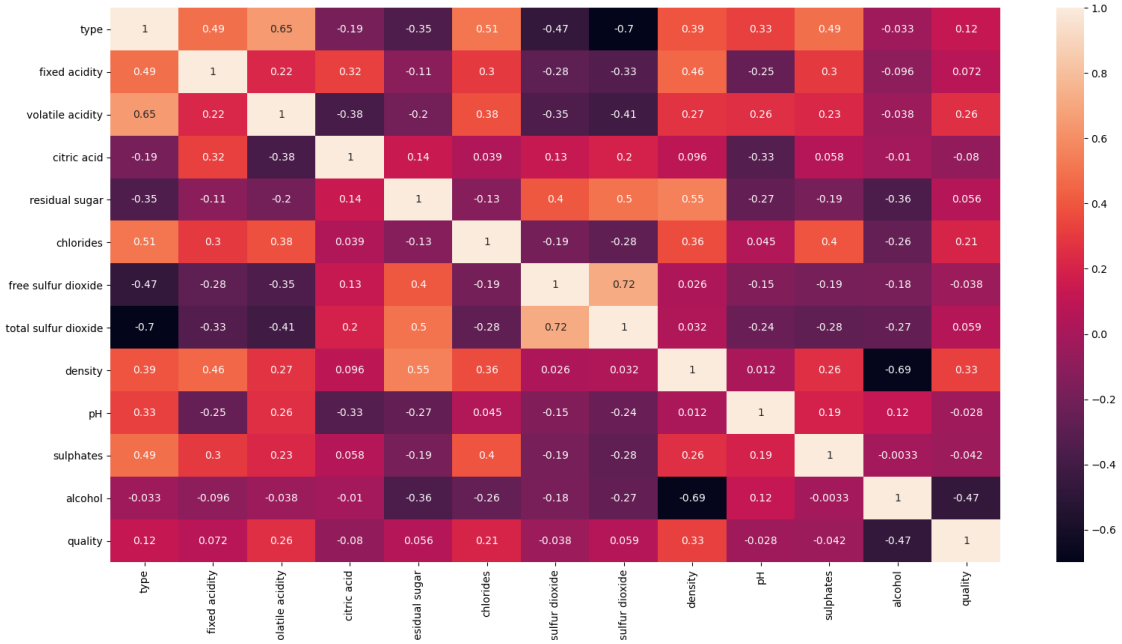


Fig 3: Performance of ML models VS the baseline (majority guess)

Discussion

All ML models except Gaussian Naïve Bayes perform adequately better than the baseline. This shows that these models are actually able to learn 'something' from the dataset. However, the reason behind Naïve Bayes's bad performance is, Naïve Bayes assumes that the input features are independent to each other. For our task this is not the case.

Following heatmap shows the correlation among the features –



As we can see, input features for wine quality prediction are corelated which nullifies the Naïve Bayes assumption. Thus naïve bayes does not perform well for this task.

Rest three ML models have similar performances. Being an optimized version of decision tree, random forest offers the best performance of around 70% accuracy. However, the score is still low. As per my understanding, this is due to the insufficiency of training data and the imbalance in training dataset.