Final Project

Saleh Mansour M Alawaji, Qiyuan Wang, Hanyu Wang

December 16, 2022

1 Introduction

The following report details how we use the machine learning technique to assess the vinho verde red wine quality. Machine learning is a subset of artificial intelligence that can learn from experience and make predictions based on what it learns (Zhou 2021). Generally, the machine learning technique has a learning algorithm capable of building a computational model once the algorithm is fed with a training dataset. The computational model can perform predictions when there are new observations. In this report, the machine learning technique is applied to establish different models using three classification algorithms.

The dataset analyzed in this report is provided by Cortez et al. (2019), which contains 1599 Vinho verde red wine samples collected from the Minho region of Portugal in 2004. As we know, the quality of a wine is determined by its taste, which can be verified by a sensory test or supported by a physicochemical test. Obviously, it is much easier to qualify wines by inspecting their physicochemical properties than relying on human tastes. In this case, 11 physicochemical features of the Vinho Verde red wine samples are selected and analyzed together with the quality of the wine samples. The samples are divided into a training dataset and a testing dataset. The training dataset is fed into the Support Vector Classifier (SVC) algorithm, the Multilayer Perceptron Classifier (MLP) algorithm, and the Random Forest Classifier algorithm separately to train different models. Then the test dataset is applied to the models to measure their performances.

Among all, the SVC algorithm is a soft margin classifier based on the Support Vector Machine (SVM) method. It uses parameter C to control the bias-variance tradeoff and therefore optimizes the margin in the hyperplane. SVC works by mapping data into a high-dimensional feature space, allowing data points to be classified even when the data cannot otherwise be linearly separable Delimiters between categories are detected, and the data is transformed so that the delimiters are drawn as hyperplanes.

Multilayer Perceptron (MLP) is an extension of feedforward neural networks. It consists of three types of layers (an input layer, a hidden layer, and an output layer). The hidden layer can consist of multiple layers. The feature vectors feed to the input layer. The classification task is performed by the output layer. Any number of hidden layers located between the input and output layers is the actual computational engine of MLP. MLP is a forward feed where the features flow forward from the input layer to the output layer. MLP neurons are trained using a backpropagation learning algorithm. One of the advantages of MLP is that it's considered universal expressive power, which means any continuous function can be approximated as close as needed by a trained three-layer ANN with a sufficient number of nodes in the hidden layer. Furthermore, MLP doesn't relay any probability density or probabilistic information. It yields the classification directly via training. Contrastingly, MLP disregards spatial information, and the model parameter increases exponentially with increasing hidden layers.

Random forest is an ensemble learning method for classification that consist of multiple decision trees, where classification is based on voting or the average of the result of multiple decision trees. Each Decision tree in the ensemble is comprised of a bootstrap sample, and Out-of-bag samples are used to evaluate the model. During the bagging process, an instance of randomness is injected. Random forest tries to overcome some of the challenges that exist in a decision tree classifier. First, a Decision tree classifier is prone to overfitting. A random forest classifier overcomes this by fitting multiple decision trees. Second, a random forest classifier can be used for both classification and regression with high accuracy compared to a Decision tree classifier. Third, Random forests make it easier to measure the importance and impact of a specific feature on a model.

The goal of this report is to establish a classification model with the highest possible accuracy to predict the quality of Vinho Verde red wine when given the physicochemical properties of the wine. Among the three classification algorithms implemented, the MLP Classifier manifests the best performance with an accuracy of 76%.

2 Methods

First and foremost, preprocessing processes: resampling, dimension reduction, and feature scaling are applied to the dataset to reduce computation cost and improve accuracy. Upon investigation of the distribution plot on prediction label quality, it can be observed that most of the wine data have the quality of 5, 6, and 7. As to make models better generalized to different labels, resampling is applied to data partitions with lower frequency.

Moreover, there are 12 features in the dataset, as shown in table 1, which can lead to extremely computationally expensive and poor performance when applied to sophisticated models or hyperparameter settings. For feature selection, PCA and LDA graphs (Figure 1) are examined along with SelectKBest (Figure 2), which scores based on chi-square p-value. Combined with the above methods, features "total sulfur dioxide", "free sulfur dioxide", "alcohol", "volatile acidity", "citric acid", and "fixed acidity" are selected for model input. Furthermore, it can be observed from the distribution that feature perimeters vary in scale and range. To balance the weight that each feature contributes to the models, MinMaxScaler, StandardScaler, and RobustScaler, are applied. Among these scaling methods, StandardScaler has the best performance. Outlier samples that significantly differ from the rest of the dataset often skew the data distribution and are caused by erroneous observation or inconsistent data entry. To further improve performance and make sure models grasp the pattern instead of noise, samples are filtered with bounds around 25 and 75 percentiles for each selected feature. Models are fitted with reduced clean datasets and original for comparison.

As for the classification method, the One-vs-One algorithm is implemented for fitting samples with, Support Vector Machine, Multi-layer Perceptron and Random Forest classifier. One-vs-One not only makes it applicable for binary models (SVM), which requires meta-strategies, but also optimizes the performance of each subproblem. With the combination of all models from One-vs-One classifiers, a comprehensive algorithm with better performance could be obtained. For Random Forest, number of estimators of 90 and class weight of balanced turn out to be the best combination of hyperparameters setting with GridSearchCV exhaustive method. Similarly, for the SVC classifier, grid search is applied for determining hyperparameter estimator gamma, estimator kernel, and C; for MLP, estimator activation, estimator solver, and estimator hidden layer sizes combination are selected based on accuracy.

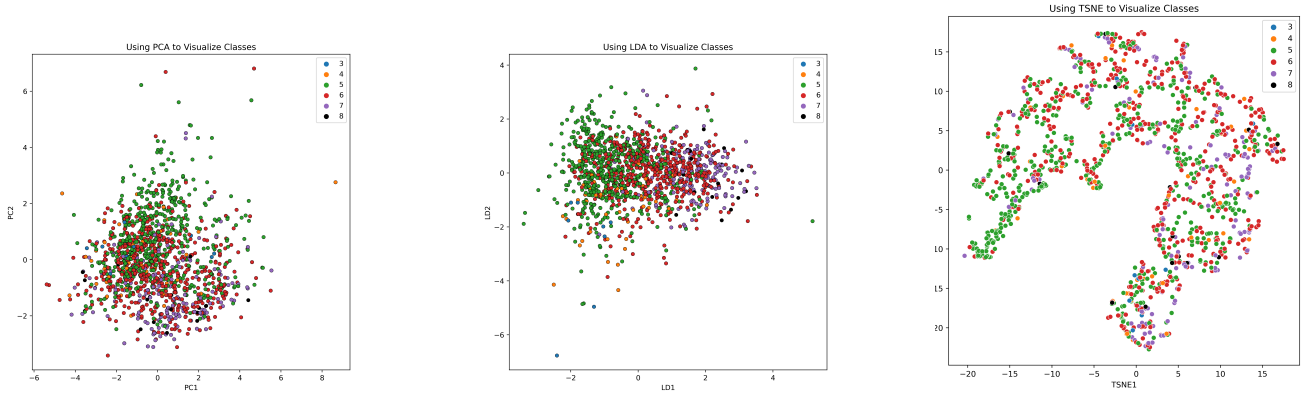
Table 1: The physicochemical data statistics for red wine .

Feature	Red Wine		
	(Min)	(Max)	(Mean)
Fixed acidity	4.6	15.9	8.3
Volatile acidity	0.1	1.6	0.5
Citric acid	0.1	1.6	0.5
Residual sugar	0.9	15.5	2.5
Chlorides	0.01	0.61	0.08
Free sulfur dioxide	1	72	14
Total sulfur dioxide	6	289	46
pH	2.7	4	3.3
Sulphates	0.3	2	0.7
Alcohol (vol.%)	8.4	14.9	10.4

It is worth mentioning that our test and implementation weren't limited to the technique discussed above. Several methods were implemented, such as Ensemble and Resample. The Ensemble is a popular meta-approach to machine learning that aims to improve prediction performance by combining predictions from multiple models. The Ensemble consists of three techniques:

1. Bagging combines bootstrapping and aggregation techniques. Bootstrapping is a method to create random samples of the dataset with replacement. Aggeration tion consists of combining the result of multiple models.

2. Stacking, where various models are trained sequentially, such as that of the first model on the whole dataset. The model prediction output feeds to the second model.



(a) PCA (b) LDA (c) T-SNE

Figure 1: Dataset visualization

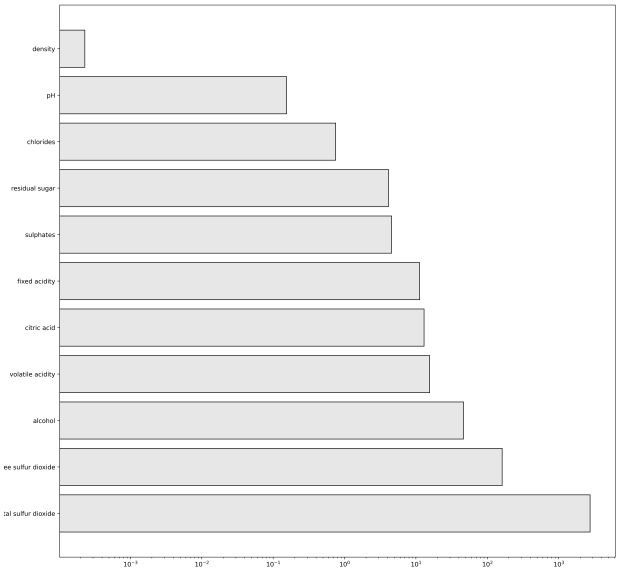


Figure 2: SelectKBest Scores comparison

3. Boosting, where the model is trained on the entire original dataset, then we re-weight the data points that the model poorly predicts. train the model with the new weights.

Out of these techniques, Stacking, aggregation with random sampling, bagging, and upsample were implemented, as shown in the code in Appendix B. However, We decided only to include Bagging and upsample only in the result section due to the nature of the remaining techniques, where randomness has an impact of producing unstable accuracy over multiple runs. In some cases, we observed accuracy drops below our threshold. In addition to that, finding the optimal number of ensembles is a non-trivial task. Regardless, Ensemble is still considered a powerful technique, and this issue can be due to our decision to implement it from scratch. This observation is confirmed by how RandomForest and Bagging, which is an Ensemble method, performed well, which is discussed in the next section.

Our methodology for evaluating and tuning the models can be summarized in 6 steps:

1. Duplicate the dataset into four datasets.
2. for each dataset, apply a single preprocessing technique, i.e., resample, grouping, and feature selection. Standardization is applied to all datasets.
3. Randomly split the dataset into training and test datasets, where the training consist of 80% and the test consists of 20% of the dataset.
4. for each model, select 2 hyperparameters to tune.
5. Use GridSearch/RandomSearch with 5-fold cross-validation to train the model and tune the parameters on each dataset.
6. Evaluate the performance of each model based on the test dataset using the confusion matrix and execution time for both fitting and testing.

3 Result

Table 2: Performance for SVM & MLP

Criteria / Dataset	SVM Original	SVM Grouped	SVM Reduced	MLP Original	MLP Grouped	MLP Reduced
Accuracy	72%	85%	69%	71%	83%	70%
F1_Score	0.70	0.81	0.68	0.70	0.78	0.69
Recall	0.72	0.85	0.69	0.71	0.83	0.70
Precision	0.69	0.84	0.67	0.71	0.78	0.69
Fitting Time(s)	6.4	3.1	4.2	6.2	0.5	4.9
Test time (ms)	283	148	243	9.6	1.7	8.2
Methods	OvO GridSearch	OvO GridSearch	OvO GridSearch	OvO	OvO	OvO

Table 3: Performance for Random Forest

Criteria/ Dataset	Original				Grouped	Reduced	
Accuracy	72%	73%	72%	73%	86%	72%	70%
F1_Score	0.70	0.71	0.71	0.71	0.83	0.70	0.68
Recall	0.72	0.73	0.72	0.73	0.86	0.72	0.70
Precision	0.69	0.70	0.69	0.69	0.82	0.68	0.66
Fitting Time(s)	33.8	0.59	0.33	4.39	0.435	0.263	0.03
Test time (ms)	160	27.3	255	11.3	24.5	262	28.9
Methods	OvO GridSearch	None	OvO	GridSearch	GridSearch	OvO	None

Table 4: Random Forest & MLP-Bagging & Resample

Algorithm/Criteria	Random Forest	MLP
Accuracy	75%	76%
F1_Score	0.73	0.74
Recall	0.75	0.76
Precision	0.73	0.75
Parameters	criterion: Gini, Estimator:90	Activation : Tanh, Hidden_layer_size:60
Fitting Time(s)	1.599	N/A [1]
Test time (ms)	15.057	N/A [1]
Methods	Upsample	Bootstrap, Bagging

[1] We haven't recorded the fitting and test time due to that it was run on a different machine.

The Classification models are evaluated based on a confusion matrix. In this paper, we evaluate the performance of each model according to precision, recall, f-score, and accuracy based on the confusion matrix and formulas shown below. Since this project's scope didn't specify the application of the models, the execution time while fitting and testing has a substantial impact on selecting the optimal model for specific applications. Therefore, we've included the execution time in the result.

Confusion Matrix

actual value	p	n	total
p′	True Positive	False Negative	P′
n′	False Positive	True Negative	N′
total	P	N	

$$\text{Precision} = \frac{TP}{TP + FP} \quad \text{Recall} = \frac{TP}{TP + FN}$$

$$F_{\text{score}} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

Table 2 shows the performance of SVM and MLP based on 3 datasets (Original, Grouped, and Reduced). For the Original dataset, we have applied only the standardization technique. OVO refers to the usage of a One-vs-One classifier, which in our case, 15 binary classifiers were created to able to classify multi-class labels. One thing that can be observed is that the execution time on the test data using MLP is considerably faster than SVM, which indicates that MLP can be more suitable for online algorithms. Table 3 shows the RandomForest performance. Even though we showed the result of Random Forest using a One-vs-One classifier, we expected that the accuracy would be worse compared to Random forest alone, as shown in the result in Table 3. Table 4 shows the performance of Random Forest and MLP using the bagging and upsample techniques. Our approach here is to trade time complexity for the sake of accuracy, and this is indicated by the accuracy gain for the MLP classifier compared to other methods. However, this technique increased the fitting time significantly. Furthermore, bagging is the only technique that is able to achieve at least a single correct prediction for quality level 4, as shown in Figure 3.

Grid Search is an exhaustive method that iterates over specified parameter values for an estimator and determines the best combination. As the model performance suggests, Grid Search CV cannot evidently improve the accuracy compared to the best hyperparameters, however, it is significantly more computationally expensive. The One-vs-One classifier can reach convergence in the training phase in a more efficient way, for it further splits the problem into subproblems that deal with binary classification. However, when fitting new samples into the model, it is more computationally expensive. Reduced models generally have a slightly lower but acceptable accuracy compared to the whole dataset, but it is less computationally expensive and corresponding models are more interpretable. Using a grouped target that merges quality into three categories instead of six can improve accuracy. However, grouped models have a narrower application, and it does not solve the problem of classifying wine into exact labels. the purpose of including grouped models is to show the complexity of the problem. Another reason is that we tried stacking classifiers, where the first classifier is based on grouped labels, and its prediction is fed to other classifiers to predict the exact label. However, it didn't yield acceptable accuracy, so it's not included in the result section but mentioned in the code in Appendix B.

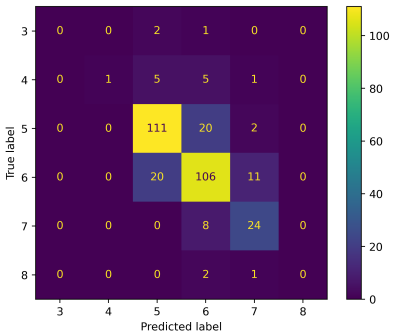


Figure 3: MLP(bagging) confusion matrix

4 Conclusions

As can be seen from Table 2, Table 3, and Table 4 in the Results section, in order to improve the accuracy of the model, each of the classification methods has been implemented with up to three sets of hyperparameters and three datasets. The three sets of parameters are the hyperparameters specifically set for the Random Forest Classifier and the hyperparameter sets generated by the One-vs-One classifier and generated by the Grid Search, respectively. The three datasets are the original dataset, the reduced clean dataset, which eliminates the outliers and some trivial features, and the grouped dataset. The original and reduced datasets are composed of the six quality levels (from level 3 to level 8) as given by Cortez et al. (2019), and the grouped dataset is composed of three quality levels, which are low (levels 3 and 4), medium (level 5 and level 6), and high qualities (level 7 and level 8) to show how the original is more complex compared to specifying categories

The two figures indicate that the accuracy is always higher when grouping the wine qualities and is always lower when using the reduced dataset. This is because the classification types decrease when using the grouped dataset, and it reduces overfitting. Also, when using the reduced dataset, the number of quality level 3 and quality level 8 wine samples significantly decreases, the training data become insufficient, and hence the overfitting of the model increases. In addition, GridSearchCV doesn't significantly improve accuracy in most instances but always significantly increases running time since it iterates through all the provided hyperparameters to look for the potentially most suitable ones. As a conclusion to the results, when a higher model performance is pursued, it is better to use the grouped dataset instead of the original and reduced ones and avoid using Grid Search to find hyperparameters.

All in all, as shown in Table 3, the model with the highest accuracy is the Random Forest Classifier when using the grouped wine qualities and the specific hyperparameters. Precisely, there are 250 trees, and the classes are weighted inversely proportional to their frequencies. The highest accuracy of the Random Forest Classifier is then 86%. The highest accuracy of the Support Vector Classifier is 85% when using the grouped wine qualities and the Grid Search. The highest accuracy of the Multilayer Perceptron Classifier is 83% when using the grouped wine qualities and the One-vs-One Classifier. Suppose it is required to classify the wine qualities into six levels from 3 to 8. In that case, the highest Multilayer Perceptron Classifier accuracy is 76%, the highest Random Forest Classifier accuracy is then 75%, and the highest Support Vector Classifier accuracy is 72%.

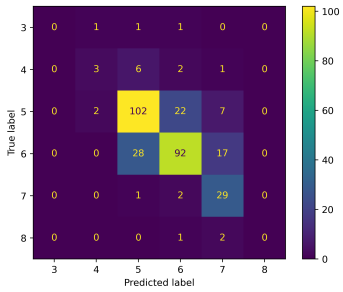
Worthwhile mentioning work has been divided into three parts and completed by every group member separately in this project. Saleh Mansour M Alawaji has styled the report, implemented all classification algorithms, and generated the figures as shown in the Appendix and the Results. Qiyuan Wang has drafted the report and made comparisons among models and methods by completing the Methods and Results sections. Hanyu Wang has drafted the report by completing the Introduction, Conclusion, and Appendix parts.

5 References

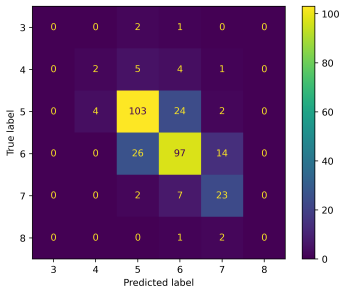
1. Z. Zhou. Machine learning. Springer Nature, 2021.

2. P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 47(4):547-553, 2009.

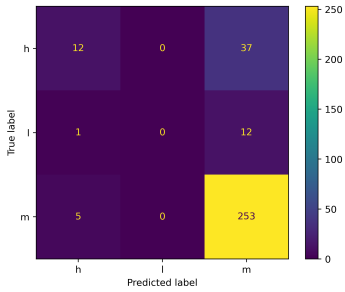
6 Appandix A: Confusion Matrices



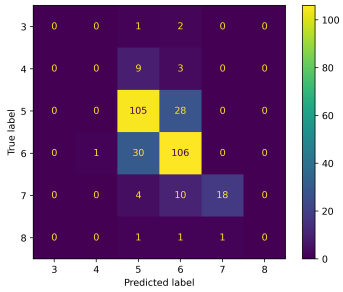
(a) MLP (OvO)- Original Dataset



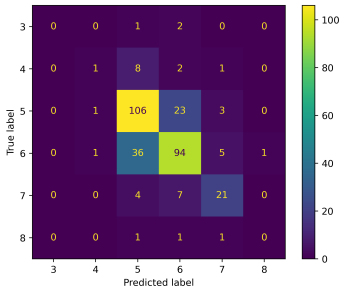
(b) MLP(OvO)- reduced dataset



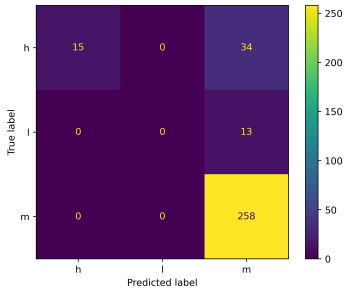
(c) SVC (OvO)- Grouped Label



(d) SVC (OvO)- Original Dataset

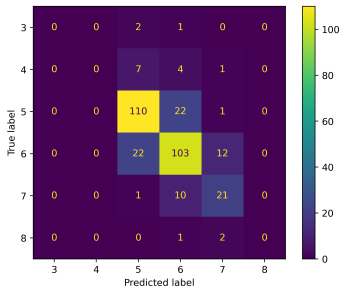


(e) SVC(OvO)- reduced dataset

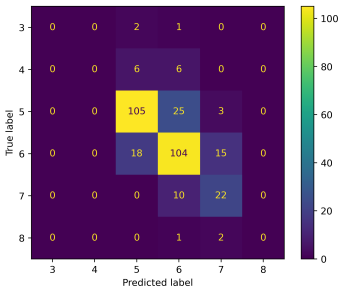


(f) SVC (OvO) - Grouped Labels

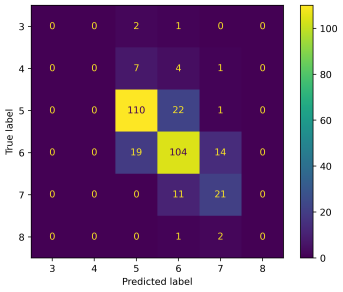
Figure 4: MLP & SVC confusion matrices



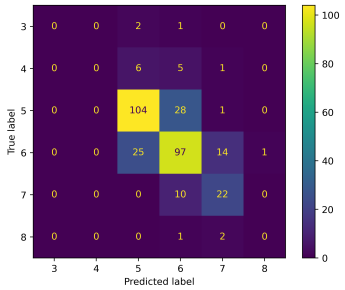
(a) GridSearch - Original Dataset



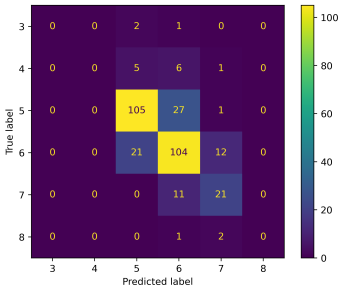
(b) OvO- Original Dataset



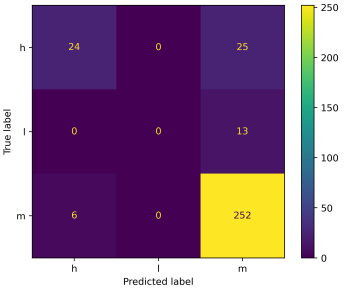
(c) RandomForest alone - Original Dataset



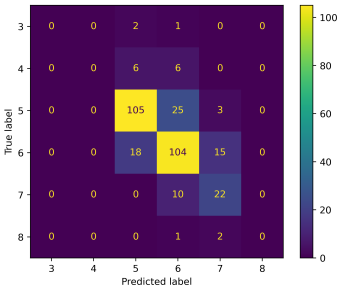
(d) Random Forest - Reduced Dataset



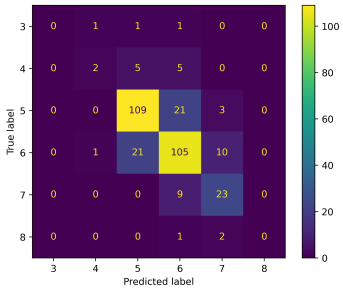
(e) OvO- reduced dataset



(f) Random Forest - Grouped Labels



(g) GridSearch & OvO - Original Dataset



(h) GridSearch & - Upsample Dataset

Figure 5: Random Forest confusion matrices

7 Appandix B: Code

Listing 1: The Code for All classifiers

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import joblib
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, accuracy_score, ConfusionMatrixDisplay, confusion_matrix
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split, KFold, cross_val_score
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScaler, LabelBinarizer
from sklearn.multiclass import OneVsOneClassifier, OneVsRestClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.utils._testing import ignore_warnings
from sklearn.exceptions import ConvergenceWarning
from joblib import Parallel, delayed
from sklearn.multiclass import _fit_binary
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier, RandomForestRegressor
from sklearn.feature_selection import SelectKBest, chi2
from sklearn.svm import SVC, SVR
from sklearn.manifold import TSNE
from sklearn import preprocessing
import os
import time
from sklearn.utils import resample

os.makedirs("./Graphs/", exist_ok=True)


class CustomOneVsRestClassifier(OneVsRestClassifier):
    """
    Not completed
    This part of the code to allow each one vs res classifier to have different parameters
    """

    def __init__(self, estimators, n_jobs=1):
        self.estimators = estimators
        self.n_jobs = n_jobs

    @ignore_warnings(category=ConvergenceWarning)
    def fit(self, X, y):
        self.label_binarizer_ = LabelBinarizer(sparse_output=True)
        Y = self.label_binarizer_.fit_transform(y)
        Y = Y.tocsc()
        self.classes_ = self.label_binarizer_.classes_
        columns = (col.toarray().ravel() for col in Y.T)

        self.estimators_ = Parallel(n_jobs=self.n_jobs)(delayed(_fit_binary)(
            estimator, X, column, classes=[
                "not %s" % self.label_binarizer_.classes_[i],
                self.label_binarizer_.classes_[i]])
                                                        for i, (column, estimator) in
                                                        enumerate(zip(columns, self.estimators)))
        return self

```

```python

# region plot
def plot_pca(x, y):
    x_pca = StandardScaler().fit_transform(x)
    pca = PCA(n_components=2)
    principalComponents = pca.fit_transform(x_pca)
    fig = plt.figure(figsize=(8, 8))
    sns.scatterplot(
        x=principalComponents[:, 0], y=principalComponents[:, 1], hue=y.ravel(), marker="o", s=25,
        ↪  edgecolor="k",
        palette=["C0", "C1", "C2", "C3", "C4", "k"]).set_title("Using PCA to Visualize Classes")
    plt.xlabel("PC1")
    plt.ylabel("PC2")
    plt.savefig("./Graphs/PCA_plot.png", dpi=400)
    plt.close("all")


def plot_tsne(x, y):
    tsne = TSNE(n_components=2, perplexity=40, n_iter=300)
    tsne_results = tsne.fit_transform(x)

    plt.figure(figsize=(8, 8))
    sns.scatterplot(
        x=tsne_results[:, 0], y=tsne_results[:, 1],
        hue=y.ravel(),
        palette=["C0", "C1", "C2", "C3", "C4", "k"]
    ).set_title("Using TSNE to Visualize Classes")
    plt.xlabel("TSNE1")
    plt.ylabel("TSNE2")
    plt.savefig("./Graphs/TSNE_plot.png", dpi=400)
    plt.close("all")


def plot_lda(x, y):
    x_lda = StandardScaler().fit_transform(x)
    lda = LDA(n_components=2)
    lda_ds = lda.fit_transform(x_lda, y.ravel())
    fig = plt.figure(figsize=(8, 8))
    sns.scatterplot(
        x=lda_ds[:, 0], y=lda_ds[:, 1], hue=y.ravel(), marker="o", s=25, edgecolor="k",
        palette=["C0", "C1", "C2", "C3", "C4", "k"]
    ).set_title("Using LDA to Visualize Classes")
    plt.xlabel("LD1")
    plt.ylabel("LD2")
    plt.savefig("./Graphs/LDA_plot.png", dpi=400)
    plt.close("all")


def plot_confusion_matrix(matrix, title, classes):
    cm_display = ConfusionMatrixDisplay(confusion_matrix=matrix, display_labels=classes)
    cm_display.plot()
    plt.savefig("./Graphs/{}.png".format(title), dpi=400)
    plt.close("all")


def plot_feature_score(ds):
    scores = ds.iloc[:, -1:].values
    features = ds.iloc[:, 0:-1].values
    fig = plt.figure(figsize=(10, 10))
    plt.barh(features.ravel(), scores.ravel(), color=(0.1, 0.1, 0.1, 0.1), edgecolor='black')
```

```
    plt.xscale("log")
    plt.savefig("./Graphs/Features_scores.png", dpi=400)
    plt.close("all")


# endregion

# region classification function
@ignore_warnings(category=ConvergenceWarning)
def oneVsOne(model, xtrain, ytrain, xtest, ytest, model_name, hyper_para=None, label=False):
    if hyper_para is None:
        clfs = OneVsOneClassifier(model, n_jobs=-1)
        start_time_fitting = time.perf_counter()
        clfs.fit(xtrain, ytrain.ravel())
        end_time_fitting = time.perf_counter()
        start_time_test = time.perf_counter()
        yhat = clfs.predict(xtest)
        end_time_test = time.perf_counter()
        print("classification report one vs one classifiers ({}):".format(model_name))
        print(f"Execution Time For fitting :{end_time_fitting - start_time_fitting :0.6f} sec")
        print(f"Execution Time For Test : {end_time_test - start_time_test:0.6f} sec")
        title = "onevsone classifiers({})".format(model_name)
        if not label:
            print(classification_report(ytest, yhat, zero_division=0))
            matrix = confusion_matrix(ytest, yhat)
            plot_confusion_matrix(matrix, title, [3, 4, 5, 6, 7, 8])
        else:
            print(classification_report(ytest, yhat, zero_division=0, target_names=['high', 'low', 'mid']))
            matrix = confusion_matrix(ytest, yhat)

            plot_confusion_matrix(matrix, title, le.classes_)
        return clfs
    else:
        clfs = OneVsOneClassifier(model, n_jobs=-1)
        grids_clf = RandomizedSearchCV(clfs, hyper_para, n_iter=40, cv=5, n_jobs=-1, random_state=50)
        # grids_clf = GridSearchCV(clfs, hyper_para, cv=5, verbose=5, n_jobs=-1)
        start_time_fitting = time.perf_counter()
        clf_opt = grids_clf.fit(xtrain, ytrain.ravel())
        end_time_fitting = time.perf_counter()
        start_time_test = time.perf_counter()
        yhat = clf_opt.predict(xtest)
        end_time_test = time.perf_counter()
        print("classification report one vs one classifiers ({}) with GridSearch:".format(model_name))
        print(f"Execution Time For fitting :{end_time_fitting - start_time_fitting :0.6f} sec")
        print(f"Execution Time For Test : {end_time_test - start_time_test:0.6f} sec")
        title = "onevsone classifiers({})_GridSearch".format(model_name)
        if not label:
            print(classification_report(ytest, yhat, zero_division=0))
            matrix = confusion_matrix(ytest, yhat)
            plot_confusion_matrix(matrix, title, [3, 4, 5, 6, 7, 8])
        else:
            print(classification_report(ytest, yhat, zero_division=0, target_names=['high', 'low', 'mid']))
            matrix = confusion_matrix(ytest, yhat)
            plot_confusion_matrix(matrix, title, le.classes_)
        return clf_opt.best_estimator_


def ensemble_predictions(members, testX):
    """
       (there are some issues with this function)
```

```
        This method only used for  ensample_classifiers function
        :param members:
        :param n_members:
        :param testX:
        :param testy:
        :return:
    """
    # make predictions
    yhats = [model.predict_proba(testX) for model in members]
    yhats = np.array(yhats)
    # sum across ensemble members
    summed = np.sum(yhats, axis=0)
    # argmax across classes
    result = np.argmax(summed, axis=1)
    result[result == 5] = 8
    result[result == 4] = 7
    result[result == 3] = 6
    result[result == 2] = 5
    result[result == 1] = 4
    result[result == 0] = 3
    return result


def evaluate_n_members(members, n_members, testX, testy):
    """
    This method only used for  ensample_classifiers function
    :param members:
    :param n_members:
    :param testX:
    :param testy:
    :return:
    """
    # select a subset of members
    subset = members[:n_members]
    # make prediction
    yhat = ensemble_predictions(subset, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)


@ignore_warnings(category=ConvergenceWarning)
def evaluate_model(model, xtrain, ytrain, x_validation, y_validation):
    """
     This method only used for  ensample_classifiers function
    :param model:
    :param xtrain:
    :param ytrain:
    :param x_validation:
    :param y_validation:
    :return:
    """
    model.fit(xtrain, ytrain.ravel())
    yhat = model.predict(x_validation)
    valid_acc = accuracy_score(y_validation, yhat)
    return model, valid_acc


def ensample_classifers(xtrain, ytrain, xtest, ytest, hyper_para, grid_status=False):
    """
    (not complete)
```

```
    :param model:
    :param xtrain:
    :param ytrain:
    :param xtest:
    :param ytest:
    :param hyper_para:
    :param grid_status:
    :return:
    """
    n_splits = 10
    scores, members = list(), list()
    for value in hyper_para:
        print("=" * 40)
        print(value)
        random_state = np.random.randint(100, 1000, 10)
        for _, s in zip(range(n_splits), random_state):
            print("random state:", s)
            model = MLPClassifier(max_iter=3000, random_state=55, hidden_layer_sizes=value)
            # split data
            trainX, validX, trainy, validy = train_test_split(xtrain, ytrain, test_size=0.10,
            ↪  random_state=s)
            # evaluate model
            model, test_acc = evaluate_model(model, trainX, trainy, validX, validy)
            print('>%.3f' % test_acc)
            scores.append(test_acc)
            members.append(model)

        indices = [ind for ind, score in enumerate(scores) if score < 0.70]
        for i in sorted(indices, reverse=True):
            del scores[i]
            del members[i]
        print(members)
        print("=" * 40)
    single_scores, ensemble_scores = list(), list()
    if members:
        for i in range(1, len(members) + 1):
            ensemble_score = evaluate_n_members(members, i, xtest, ytest)
            single_yhat = members[i - 1].predict(xtest)
            single_score = accuracy_score(ytest, single_yhat)
            print('> %d: single=%.3f, ensemble=%.3f' % (i, single_score, ensemble_score))
            ensemble_scores.append(ensemble_score)
            single_scores.append(single_score)


@ignore_warnings(category=ConvergenceWarning)
def stacking(group_clf, ds, xtest, ytest):
    # raise NotImplemented
    yhat = group_clf.predict(xtest)
    opt_yhat = []

    low_label_condition = ds.iloc[:, -1].isin([3, 4])
    mid_label_condition = ds.iloc[:, -1].isin([5, 6])
    high_label_condition = ds.iloc[:, -1].isin([7, 8])

    low_labels = pd.DataFrame(ds[low_label_condition])
    mid_labels = pd.DataFrame(ds[mid_label_condition])
    high_labels = pd.DataFrame(ds[high_label_condition])

    X_low = low_labels.iloc[:, :-1].values
    Y_low = low_labels.iloc[:, -1:].values
```

```
    X_mid = mid_labels.iloc[:, :-1].values
    Y_mid = mid_labels.iloc[:, -1:].values
    X_high = high_labels.iloc[:, :-1].values
    Y_high = high_labels.iloc[:, -1:].values
    Xtrain_high, _, Ytrain_high, _ = train_test_split(X_high, Y_high, test_size=0.2, random_state=100)
    Xtrain_mid, _, Ytrain_mid, _ = train_test_split(X_mid, Y_mid, test_size=0.2, random_state=100)
    Xtrain_low, _, Ytrain_low, _ = train_test_split(X_low, Y_low, test_size=0.2, random_state=100)

    clf_low = MLPClassifier(max_iter=3000, random_state=55)
    clf_mid = MLPClassifier(max_iter=3000, random_state=55)
    clf_high = MLPClassifier(max_iter=3000, random_state=55)

    clf_low.fit(Xtrain_low, Ytrain_low.ravel())
    clf_mid.fit(Xtrain_mid, Ytrain_mid.ravel())
    clf_high.fit(Xtrain_high, Ytrain_high.ravel())

    for ind, pred in enumerate(yhat):
        if pred == 2:  # mid
            pred2 = clf_mid.predict(xtest[ind, :].reshape(1, -1))
            opt_yhat.append(pred2)
        elif pred == 0:  # high
            pred2 = clf_high.predict(xtest[ind, :].reshape(1, -1))
            opt_yhat.append(pred2)
        elif pred == 1:  # low
            pred2 = clf_low.predict(xtest[ind, :].reshape(1, -1))
            opt_yhat.append(pred2)

    print("classification report Stack classification :")
    print(classification_report(ytest, opt_yhat, zero_division=0))


def rf_classification(xtrain, ytrain, xtest, ytest, data_type, hyper_para=None, label=False):
    if hyper_para is None:
        model = RandomForestClassifier(random_state=55, n_estimators=250, class_weight='balanced')
        start_time_fitting = time.perf_counter()
        model.fit(xtrain, ytrain.ravel())
        end_time_fitting = time.perf_counter()
        start_time_test = time.perf_counter()
        y_hat = model.predict(xtest)
        end_time_test = time.perf_counter()
        print("Classification report for RandomForest on {}:".format(data_type))
        print(f"Execution Time For fitting :{end_time_fitting - start_time_fitting :0.6f} sec")
        print(f"Execution Time For Test : {end_time_test - start_time_test:0.6f} sec")
        title = "RandomForest on {}:".format(data_type)
        if not label:
            print(classification_report(ytest, y_hat, zero_division=0))
            matrix = confusion_matrix(ytest, y_hat)
            plot_confusion_matrix(matrix, title, [3, 4, 5, 6, 7, 8])
        else:
            print(classification_report(ytest, y_hat, zero_division=0, target_names=['high', 'low',
            ↪  'mid']))
            matrix = confusion_matrix(ytest, y_hat)
            plot_confusion_matrix(matrix, title, le.classes_)
        return model
    else:
        grids_clf = GridSearchCV(RandomForestClassifier(random_state=55, class_weight='balanced'),
        ↪  hyper_para, cv=5,
                                 n_jobs=-1)
        start_time_fitting = time.perf_counter()
        clf_opt = grids_clf.fit(xtrain, ytrain.ravel())
```

```
        end_time_fitting = time.perf_counter()
        start_time_test = time.perf_counter()
        yhat = clf_opt.predict(xtest)
        end_time_test = time.perf_counter()
        print("classification report RandomForest with GridSearch {}:".format(data_type))
        print(f"Execution Time For fitting :{end_time_fitting - start_time_fitting :0.6f} sec")
        print(f"Execution Time For Test : {end_time_test - start_time_test:0.6f} sec")
        title = "GridSearch RandomForest on {}:".format(data_type)
        if not label:
            print(classification_report(ytest, yhat, zero_division=0))
            matrix = confusion_matrix(ytest, yhat)
            plot_confusion_matrix(matrix, title, [3, 4, 5, 6, 7, 8])
        else:
            print(classification_report(ytest, yhat, zero_division=0, target_names=['high', 'low', 'mid']))
            matrix = confusion_matrix(ytest, yhat)
            plot_confusion_matrix(matrix, title, le.classes_)
        return clf_opt.best_estimator_


# endregion

# region data preprocessing

def feature_selection(x, y):
    # apply SelectKBest class to extract top best features
    bestFeatures = SelectKBest(score_func=chi2, k='all')
    bestFeaturesFit = bestFeatures.fit(x, y)
    dfscores = pd.DataFrame(bestFeaturesFit.scores_)  # Store predictor scores in a column
    dfcolumns = pd.DataFrame(x.columns)  # Store predictor variable names in a column

    # #concatenate scores with predictor names
    predScores = pd.concat([dfcolumns, dfscores], axis=1)
    predScores.columns = ['Predictor', 'Score']  # naming the dataframe columns
    print(predScores.nlargest(13, 'Score'))  # print top (by score) 10 features
    # df_selected = x.drop('sulphates', 1)
    # df_selected = df_selected.drop('residual sugar', 1)
    df_selected = x.drop(columns='chlorides')
    df_selected = df_selected.drop(columns='pH')
    df_selected = df_selected.drop(columns='density')
    plot_feature_score(predScores.nlargest(13, 'Score'))
    return df_selected.to_numpy()


def group_label(ds):
    low_label_condition = ds.iloc[:, -1].isin([3, 4])
    mid_label_condition = ds.iloc[:, -1].isin([5, 6])
    high_label_condition = ds.iloc[:, -1].isin([7, 8])

    low_labels = pd.DataFrame(ds[low_label_condition])
    mid_labels = pd.DataFrame(ds[mid_label_condition])
    high_labels = pd.DataFrame(ds[high_label_condition])

    # low_labels['quality'].values.astype(str)
    # mid_labels['quality'].values.astype(str)
    # high_labels['quality'].values.astype(str)
    l, m, h = 'l', 'm', 'h'
    low_labels['quality'].replace([3, 4], l, inplace=True)
    mid_labels['quality'].replace([5, 6], m, inplace=True)
    high_labels['quality'].replace([7, 8], h, inplace=True)
    df_grouped = pd.concat([low_labels, mid_labels, high_labels])
```

```
    return df_grouped.to_numpy()[:, :-1], df_grouped.to_numpy()[:, -1:]


def upsample_training(ds_unbalanced,n):
    class3_minority_condition = ds_unbalanced.iloc[:, -1].isin([3])
    class4_minority_condition = ds_unbalanced.iloc[:, -1].isin([4])
    class8_minority_condition = ds_unbalanced.iloc[:, -1].isin([8])
    class7_minority_condition = ds_unbalanced.iloc[:, -1].isin([7])
    majority_condition = ds_unbalanced.iloc[:, -1].isin([5, 6,7])
    df_class3_minority = pd.DataFrame(ds_unbalanced[class3_minority_condition])
    df_class4_minority = pd.DataFrame(ds_unbalanced[class4_minority_condition])
    df_class7_minority = pd.DataFrame(ds_unbalanced[class7_minority_condition])
    df_class8_minority = pd.DataFrame(ds_unbalanced[class8_minority_condition])
    df_train_majority = pd.DataFrame(ds_unbalanced[majority_condition])
    df_class3_upsampled = resample(df_class3_minority, replace=True, n_samples=n, random_state=123)
    df_class4_upsampled = resample(df_class4_minority, replace=True, n_samples=n, random_state=123)
    #df_class7_upsampled = resample(df_class7_minority, replace=True, n_samples=200, random_state=123)
    df_class8_upsampled = resample(df_class8_minority, replace=True, n_samples=n, random_state=123)
    df_upsampled = pd.concat([df_train_majority,
                              df_class3_upsampled,
                              df_class4_upsampled,
                              df_class8_upsampled])
    df_upsampled=df_upsampled.sample(frac=1,random_state=1)
    return df_upsampled.to_numpy()[:, :-1], df_upsampled.to_numpy()[:, -1:]


def outliers_detection(data):
    # raise NotImplemented
    data = np.array(data)
    percentile_25 = np.percentile(data, 25)
    percentile_50 = np.percentile(data, 50)
    percentile_75 = np.percentile(data, 75)
    lower_bound = percentile_25 - 1.5 * (percentile_75 - percentile_25)
    upper_bound = percentile_75 + 1.5 * (percentile_75 - percentile_25)
    outliers = []
    for point in list(data):
        if point < lower_bound or point > upper_bound:
            outliers.append(point)
        else:
            outliers.append('not a outlier')

    return outliers


def clean_outliers(ds):
    # raise NotImplemented
    d_outliers_focused = {}
    for name in list(ds):
        d_outliers_focused.setdefault(name, outliers_detection(ds[name]))
    df_outliers_focused = pd.DataFrame(data=d_outliers_focused)

    series_list = []
    for index, row in df_outliers_focused.iterrows():
        for name in list(df_outliers_focused):
            if type(row[name]) == np.float64:
                series_list.append(row)
                break

    df_outliers = pd.DataFrame(series_list, columns=list(df_outliers_focused))
    outliers_indices = df_outliers.index.tolist()
```

```
    cleaned_dataset = ds.drop(ds.index[outliers_indices])
    return cleaned_dataset


df = pd.read_csv("./winequality-red.csv", sep=';')
cleaned_df = clean_outliers(df)

X = df.iloc[:, :-1].values
Y = df.iloc[:, -1:].values
X2 = df.iloc[:, :-1]
Y2 = df.iloc[:, -1:]
X_cleaned = cleaned_df.iloc[:, :-1].values
Y_cleaned = cleaned_df.iloc[:, -1:].values
X_grouped, Y_grouped = group_label(df)

le = preprocessing.LabelEncoder()
le.fit(Y_grouped.ravel())
Y_grouped = le.transform(Y_grouped.ravel())

# print(cleaned_df.iloc[:, -1].value_counts())

selected_X = feature_selection(X2, Y2)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=100)
X_train_red, X_test_red, Y_train_red, Y_test_red = train_test_split(selected_X, Y, test_size=0.2,
↪  random_state=100)
X_train_cleaned, X_test_cleaned, Y_train_cleaned, Y_test_cleaned = train_test_split(X_cleaned, Y_cleaned,
↪  test_size=0.2,
                                                                                    random_state=100)
X_train_grouped, X_test_grouped, Y_train_grouped, Y_test_grouped = train_test_split(X_grouped, Y_grouped,
↪  test_size=0.2,
                                                                                    random_state=100)

# ressample original training Dataset
# standardize the datasets
norm = StandardScaler().fit(X_train)
X_train = norm.transform(X_train)
X_test = norm.transform(X_test)


# train_unbalanced = X_train
# train_unbalanced = np.append(train_unbalanced, Y_train, axis=1)
# df_train_unbalanced = pd.DataFrame(train_unbalanced)
# X_train_upsample, Y_train_upsample = upsample_training(df_train_unbalanced)


train_unbalanced = X_train
train_unbalanced = np.append(train_unbalanced, Y_train, axis=1)
df_train_unbalanced = pd.DataFrame(train_unbalanced)
X_train_upsample, Y_train_upsample = upsample_training(df_train_unbalanced,464)



# standardize the dataset without outlier
norm = StandardScaler().fit(X_train_cleaned)
X_train_cleaned = norm.transform(X_train_cleaned)
X_test_cleaned = norm.transform(X_test_cleaned)


norm = StandardScaler().fit(X_train_red)
```

```
X_train_red = norm.transform(X_train_red)
X_test_red = norm.transform(X_test_red)
# endregion

# region hyperparameters
activation = ['identity', 'logistic', 'tanh', 'relu']
solver = ['lbfgs', 'adam']
hidden_layer_sizes = [i for i in range(10, 150, 10)]
# hidden_layer_sizes.extend([(i, math.floor(i / 2)) for i in range(10, 200, 10)])
hyperparameters_mlp = dict(estimator__activation=activation, estimator__solver=solver,
                           estimator__hidden_layer_sizes=hidden_layer_sizes)

n_estimators = [40, 60, 70, 90, 100, 150, 200, 250]
criterion = ['gini', 'entropy', 'log_loss']

hyperparameters_RF2 = dict(estimator__criterion=criterion, estimator__n_estimators=n_estimators)
hyperparameters_RF = dict(criterion=criterion, n_estimators=n_estimators)

kernel = ["rbf", "sigmoid"]
gamma = [2 ** 3, 2 ** 1, 2 ** 0]
gamma.extend([2 ** i for i in range(-15, 0)])
C = [i for i in range(1, 20)]
hyperparameters_SVC = dict(estimator__gamma=gamma, estimator__kernel=kernel, estimator__C=C)
hyperparameters_svc = dict(gamma=gamma, kernel=kernel, C=C)
# endregion

# region running scripts
# region Baggin

# bag = BaggingClassifier(MLPClassifier(max_iter=10000, random_state=55, activation='tanh',
↪  hidden_layer_sizes=60), n_estimators=300, random_state=0, bootstrap_features=True,
↪  n_jobs=-1).fit(X_train, Y_train.ravel())
# yyyy=bag.predict(X_test)
# print("Classification report for MLP using Bagging")
# print(classification_report(Y_test,yyyy,zero_division=0))
# file="MLP_accuracy.pkl"
# joblib.dump(bag,file)


file_name="MLP_accuracy_updatd.pkl"
if os.path.exists(file_name):
    mlp_best = joblib.load(file_name)
    yyyy=mlp_best.predict(X_test)
    print("Classification report for MLP using Bagging")
    print(classification_report(Y_test,yyyy,zero_division=0))
    matrix = confusion_matrix(Y_test, yyyy)
    plot_confusion_matrix(matrix, "MLP with Bagging", [3, 4, 5, 6, 7, 8])
else:
    print("The pickle files '{}' must be in the same directory to show the best result".format(file_name))
    print("if you want to train the classifer from the scratch uncomment line# 558-563,  \n"
          "and remove the 'MLP_accuracy_updatd.pkl' from the directory. It will take long time to train ")


#endregion




# region SVC
```

```
# SVC
oneVsOne(SVC(random_state=55, class_weight='balanced', decision_function_shape='ovo'), X_train, Y_train, X_test, Y_test,
         "SVC", hyperparameters_SVC)

# SVC Grouped label
oneVsOne(SVC(random_state=55, class_weight='balanced', decision_function_shape='ovo'), X_train_grouped,
         Y_train_grouped, X_test_grouped, Y_test_grouped, "SVC - Grouped Label", hyperparameters_SVC, label=True)

# SVC reduced Dataset

oneVsOne(SVC(random_state=55, class_weight='balanced', decision_function_shape='ovo'), X_train_red, Y_train_red,
         X_test_red,
         Y_test_red, "SVC - Reduced Dataset", hyperparameters_SVC)


oneVsOne(SVC(random_state=55, class_weight='balanced', decision_function_shape='ovo'), X_train_upsample, Y_train_upsample,
         X_test,
         Y_test, "SVC - upsample Dataset", hyperparameters_SVC)

# endregion

# region MLP
# MLP
oneVsOne(MLPClassifier(max_iter=3000, random_state=55), X_train,
         Y_train, X_test, Y_test, "MLP")  # best MLP


# oneVsOne(MLPClassifier(max_iter=3000, random_state=55), X_train,
#          Y_train, X_test, Y_test,"MLP" , hyperparameters_mlp)

oneVsOne(MLPClassifier(max_iter=10000, random_state=55), X_train_upsample,
         Y_train_upsample, X_test, Y_test,"MLP - Upsample DS")

# MLP with reduced feature

oneVsOne(MLPClassifier(max_iter=3000, random_state=55), X_train_red,
         Y_train_red, X_test_red, Y_test_red, "MLP - reduced DS")

# MLP with cleaned Dataset
# oneVsOne(MLPClassifier(max_iter=3000, random_state=55), X_train_cleaned,
#          Y_train_cleaned, X_test_cleaned, Y_test_cleaned, "MLP - Dataset without Outliers")

# MLP with Grouped labels
oneVsOne(MLPClassifier(max_iter=3000, random_state=55), X_train_grouped,
         Y_train_grouped, X_test_grouped, Y_test_grouped, "MLP - Grouped Labels", label=True)
# endregion

# region RandomForest
# Random Forest
oneVsOne(RandomForestClassifier(random_state=55), X_train,
         Y_train, X_test, Y_test, "RandomForest - Original DS")

oneVsOne(RandomForestClassifier(random_state=55), X_train,
         Y_train, X_test, Y_test, "Random Forest - Original DS", hyperparameters_RF2)

rf_classification(X_train, Y_train, X_test, Y_test, "original DS")
```

```
643 rf_classification(X_train, Y_train, X_test, Y_test, "Original DS", hyperparameters_RF)
644
645 # Random Forest with cleaned Dataset
646 # rf_classification(X_train_cleaned, Y_train_cleaned, X_test_cleaned, Y_test_cleaned,data_type="Cleaned
    ↪  DS")
647 # rf_classification(X_train_cleaned,Y_train_cleaned, X_test_cleaned, Y_test_cleaned, hyperparameters_RF,
    ↪  data_type="Cleaned DS")
648
649 # RandomForest reduced feature
650 oneVsOne(RandomForestClassifier(random_state=55), X_train_red,
651          Y_train_red, X_test_red, Y_test_red, "RandomForest Reduced Dataset")
652 #
653 # oneVsOne(RandomForestClassifier(random_state=55), X_train_red,
654 #          Y_train_red, X_test_red, Y_test_red,"RandomForest - Reduced features DS" ,hyperparameters_RF2)
655 rf_classification(X_train_red, Y_train_red, X_test_red, Y_test_red, data_type="reduced features DS")
656 rf_classification(X_train_upsample, Y_train_upsample, X_test, Y_test, "upsample DS", hyperparameters_RF)
657 # rf_classification(X_train_red, Y_train_red, X_test_red, Y_test_red, hyperparameters_RF,
    ↪  data_type="reduced features DS")
658
659 # RandomForest  grouped label
660 clf_group = rf_classification(X_train_grouped, Y_train_grouped, X_test_grouped, Y_test_grouped,
    ↪  data_type="Grouped DS",
661                               label=True)
662
663
664 n_estimators = [40,90]
665 criterion = ['gini', 'entropy']
666 hyperparameters_RF = dict(criterion=criterion, n_estimators=n_estimators)
667 rf_classification(X_train_upsample, Y_train_upsample, X_test, Y_test, "upsample DS", hyperparameters_RF)
668
669 # endregion
670
671
672 stacking(clf_group, df, X_test, Y_test)
673
674 plot_pca(X, Y)
675 plot_lda(X, Y)
676 plot_tsne(X, Y)
677 # endregion
678
```