Predicting The Quality Of Red Wine Using Regression Models

Ezgi Nur Alisan
Computer Science Department
Ozyegin University
Cekmekoy, Istanbul
Email: ezgi.alisan@ozu.edu.tr

Emir Mehmet Eryilmaz
Computer Science Department
Ozyegin University
Cekmekoy, Istanbul
Email: emir.eryilmaz@ozu.edu.tr

***Abstract*—Predicting wine quality is a crucial task in the wine business. Since the quality of wine is controlled after the production phase, a wrong prediction of wine quality can be very costly for businesses. To solve this problem, this paper proposes a solution that predicts wine quality using regression models like K-Nearest Neighbors, Decision Tree and Support Vector Regressor. In addition to using these models individualy, ensembling methods such as bagging and stacking were used to combine the results obtained from the three algorithms in order to decrease the errors calculated from the predictions.**

1. Introduction

The wine industry attaches great importance to the wine quality certificate. Wine quality control is conducted after the production of wine. The wine production process is both time-consuming and costly. If the results from quality control are below expectations, restarting the production process can lead to significant cost and time losses. For this reason, the production cost and time of the producers can be reduced by foreseeing the wine quality before the production process. There are lots of components that affect the wine quality. In order to improve the wine quality, these components that influence the quality must be correctly proportioned. Predicting wine quality by using machine learning algorithms to realize this application will provide convenience in the wine industry. As Winers, in this project, it is aimed to predict the quality of red wine by taking into account the physicochemical factors (acidity properties, pH, alcohol content, density, etc.) that affect the wine quality. At the same time, suitable factor combinations that will increase the quality of red wine will be determined by making appropriate modeling. We have treated wine quality estimation as a regression problem. In this context, firstly we had decided to use K-Nearest-Neighbors, Nave-Bayes and Decision Tree as algorithms. However, we have later realized that Nave-Bayes is a classification algorithm and when we applied this algorithm in our own model, we saw that the predicted outputs were the results as if we were doing classification. Because we changed the Nave-Bayes algorithm with the Support Vector Regressor algorithm.

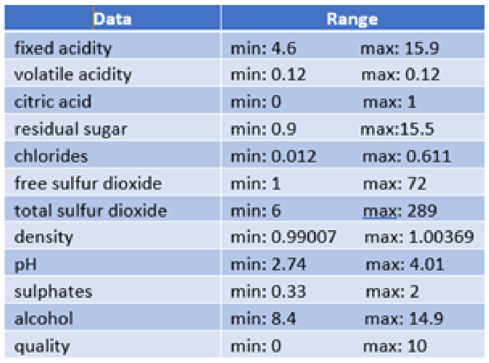
2. Description of Algorithms

In this report, 4 algorithms will be discussed. As we have mentioned in the introduction, although we will be talking about the Nave Bayes, it will not be used in the project.
Naive Bayes is a probabilistic algorithm that applies Bayes' theorem with the assumption of independence between features. It calculates the probability of a class given a set of features by multiplying the individual probabilities of each feature given the class and normalizing the result.
Decision trees are an algorithm that partitions data into subsets based on a series of if-else questions on feature values, creating a tree-like structure to make predictions. Each internal node represents a question, and each leaf node represents a prediction or outcome.
K-nearest neighbors (KNN) is a simple algorithm that classifies new data points based on their proximity to existing labeled data points. It assigns a class label to an unknown sample by taking a majority vote among its k nearest neighbors in the feature space.
The Support Vector Regressor (SVR) is a supervised machine learning algorithm used for regression tasks. It works by mapping the input data into a higher-dimensional feature space and finding the hyperplane that best separates the data points. The SVR aims to minimize the error between the predicted values and the actual targets while considering a margin of tolerance defined by a user-specified parameter.

3. Data Description and Preprocessing

The data to be used was downloaded from the UCL Machine Learning Repository. There are two different datasets, white wine and red wine. This study was done using the red wine dataset which contains 1599 samples. Dataset has 11 input variables (based on physicochemical tests): fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide, density, pH, sulfates, alcohol, and 1 output variable quality. These attributes have different ranges which can be seen in Figure 1.

As seen in the table, some features have a large difference between their minimum and maximum values. Having such a large range for a feature can lead to this



Data	Range	
fixed acidity	min: 4.6	max: 15.9
volatile acidity	min: 0.12	max: 0.12
citric acid	min: 0	max: 1
residual sugar	min: 0.9	max:15.5
chlorides	min: 0.012	max: 0.611
free sulfur dioxide	min: 1	max: 72
total sulfur dioxide	min: 6	max: 289
density	min: 0.99007	max: 1.00369
pH	min: 2.74	max: 4.01
sulphates	min: 0.33	max: 2
alcohol	min: 8.4	max: 14.9
quality	min: 0	max: 10

Figure 1. Data Distribution

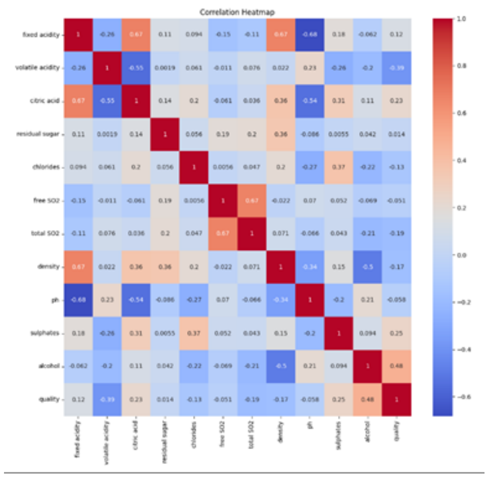


Figure 2. Correlation Heatmap

feature dominating the outcome in some machine learning algorithms. Normalization is not required for Support Vector Regressor and Decision Tree models, which will be implemented in the project. However, K-Nearest-Neighbor is a distance-based algorithm that relies on calculating the distance between data points to make predictions. Because of this, larger ranges can dominate distance calculations easily. In this point normalization can help with bringing the features to a similar scale. Z-score was also used for normalization. It was observed that there was no significant change in mean squared error in Support Vector Regressor and Decision Tree but decreased in KNN.

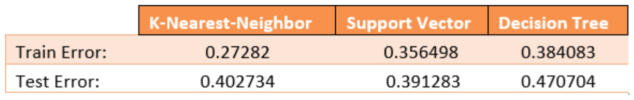
To identify the importance of the features we calculated the Pearson correlation coefficient. Some features appear to be highly correlated while some features have a very low correlation coefficient. Using these results, some features which are not very effective on the output (quality) can be excluded from the dataset. (Figure 1) It can be observed that some pairs of features were highly correlated (either negatively correlated or positively correlated). For example, fixed acidity and ph has the correlation of -0.68 which means they are negatively correlated. Also fixed acidity and density has the correlation of 0.67 which means they are positively correlated. These features could have also been excluded, but this approach was not the focus of our research.

4. Regression Analysis

For the regression process we used 3 algorithms mentioned above. We began by splitting the dataset. When we looked at different studies, we observed that the K-Fold Cross-Validation method and the randomly train-test split method were used in the wine dataset separation. We analyzed the effectiveness of both methods by using them in the data separation part. Our experiments on data separation showed that using the K-Fold Cross-Validation method was less efficient compared to the random train-test split method. According to this observation, we decided to split the data into two pieces (as training set and test set) randomly (test size = 0.2).
Then, we implemented our regression models for the three algorithms which are: KNN, Decision Tree and Support Vector Regressor, using Scikit-learn library. We performed hyperparameter tuning to acquire the best performance of our regression model. In this process, we aimed to minimize the mean squared error on the test data. We obtained the optimal hyperparameter values using grid search method.

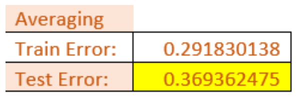
4.1. Tuning Process

For the algorithm K-Nearest-Neighbor, we searched for the optimal k value which indicates the number of nearest neighbors, in the range 1 to 100. The hyperparameter value k is calculated as 11 at the end of the tuning operation.
For the Decision Tree algorithm, we searched for the optimal depth value which indicates tree depth, in the range 1 to 100. The hyperparameter value depth is calculated as 4 at the end of the tuning operation. We realized that when the depth of the Decision Tree is high enough, the training becomes 0. At that point there will be a leaf node in the tree for all outputs in the training data. This implies overfitting on the training data.
For the Support Vector Regressor algorithm, we searched for the optimal pair of the hyperparameters C and epsilon. C hyperparameter has a control over the trade-off between minimizing the training error and allowing deviations from the predicting values. Epsilon ε is a hyperparameter that defines the margin of tolerance for errors in the regression task. These parameters are tuned by finding the pair that has the smallest error on test data. The tuning range for C parameter is from 10^-5 to 10^4 and the optimal value for C is found as 10^4. The tuning range for epsilon parameter is from 0.01 to 1 and the optimal value for epsilon is obtained as 0.19. In addition, we have used radial basis kernel (RBF) because it outperformed other kernels.
Hyperparameter tuning has increased our K-Nearest Neighbors, Decision Tree, and Support Vector Regressor models performance. The fine-tuning process helped us overcome issues such as overfitting in the Decision Tree model and



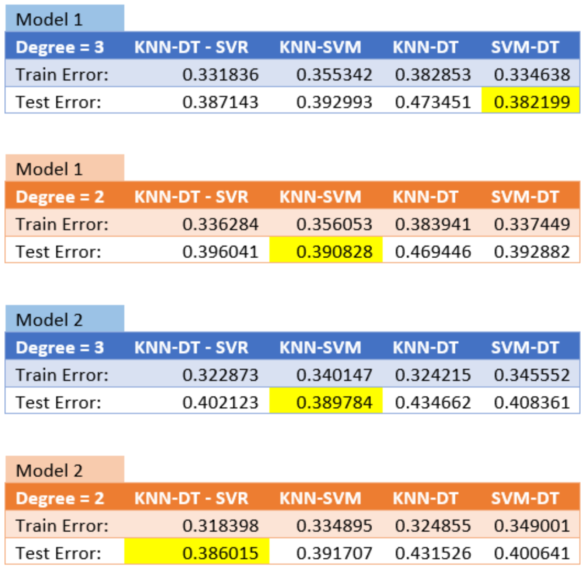
	K-Nearest-Neighbor	Support Vector	Decision Tree
Train Error:	0.27282	0.356498	0.384083
Test Error:	0.402734	0.391283	0.470704

Figure 3. Errors that obtained after Tuning Process



Averaging	
Train Error:	0.291830138
Test Error:	0.369362475

Figure 4. Averaging Results for each Combination



Model 1

Degree = 3	KNN-DT - SVR	KNN-SVM	KNN-DT	SVM-DT
Train Error:	0.331836	0.355342	0.382853	0.334638
Test Error:	0.387143	0.392993	0.473451	0.382199

Model 1

Degree = 2	KNN-DT - SVR	KNN-SVM	KNN-DT	SVM-DT
Train Error:	0.336284	0.356053	0.383941	0.337449
Test Error:	0.396041	0.390828	0.469446	0.392882

Model 2

Degree = 3	KNN-DT - SVR	KNN-SVM	KNN-DT	SVM-DT
Train Error:	0.322873	0.340147	0.324215	0.345552
Test Error:	0.402123	0.389784	0.434662	0.408361

Model 2

Degree = 2	KNN-DT - SVR	KNN-SVM	KNN-DT	SVM-DT
Train Error:	0.318398	0.334895	0.324855	0.349001
Test Error:	0.386015	0.391707	0.431526	0.400641

Figure 5. Stacking Results for each Combination

found the right trade-off between minimizing errors and allowing flexibility in the SVR model.

4.2. Analysis of Algorithms Individually

From the red wine dataset, the train set contains 80% of the data, and test set contains 20% of the data. Using this dataset, each algorithm is trained with their tuned hyperparameters. The analysis of the train and the test errors obtained by the three algorithms has shown in Figure 3. It can be seen that Support Vector Regressor has given better test error compared to the other two algorithms and Decision Tree has given the highest error rate from the three.

4.3. Ensemble Methods

Ensemble methods are a technique used in machine learning to combine the predictions of more than one model to obtain more accurate prediction results. As each model has different features, using all models at the same time can show good results on accuracy. Ensemble methods can improve generalization and prediction accuracy by reducing bias, variability, and overfitting. Since the three models we trained have their own strengths and weaknesses, we believed that utilizing the ensemble learning methods can improve our predictions.

4.3.1. Bagging-Averaging. For the bagging method, averaging is used for combining all three models since we have a regression problem. In the first step, we created regressor models using optimal hyperparameters that were found in tuning part and each one is trained with training data. Predicted values for both train data and test data are obtained. Then, the average of those values for the three models are taken. The averaging results on our data are shown in Figure 4. Compared to each three algorithms individually, it can be seen that averaging method got better results. When we increased the partition of test data, the results got worse. This is because when the partition of training data is smaller it is more likely that the model overfits the training data.

4.3.2. Stacking. Stacking is a very popular ensemble method in machine learning. Multiple models are trained, and their predictions are given to the meta model. Meta model takes predictions of these models as inputs and learns to make a final prediction. Since these predictions are a combination of basic models that can reflect different features and perspectives, when combined with a meta-model, we can obtain a stronger and more comprehensive regression model.
We have done 2 different implementations of stacking model. We have implemented one of them ourselves, not using any library. For the other model, Scikit-learn library is used. In our case, the base models are K-Nearest-Neighbors, Decision Tree, and Support Vector Regressor. Each one is trained, and their predictions are given to the meta model. The meta model that is used was Polynomial Regression. We had tested results for different degree values. These different implementations got better results for different degree values. For instance, the model that we wrote gave better results for the degree value 3 and the library version of stacking model gave better results for the degree value 2. Overall, the best result was obtained by the library version with the degree value 2.

In addition to creating our base model depending on those 3 different algorithms, we have also used all binary combinations of these three algorithms as base models. The results have shown that although there was not much difference in error, the combination of all three algorithms gave better results. All the outcomes can be seen in Figure 5.

5. Conclusion

In conclusion, this study showed that regression techniques in machine learning can be used to predict the wine quality before the production process. The algorithms that we used for this project were K-Nearest-Neighbor, Support Vector Regressor and Decision Tree. Each one showed good results on the test set. When we compared them individually, we observed that Support Vector Regressor outperformed the other two algorithms. In the second phase of the project, bagging and stacking, which are ensemble learning methods were used to get a better result. When using bagging method, averaging is used to combine those three algorithms and it gave more accurate outputs compared to individual algorithms results. When using stacking method, we combined all the dual combinations and combination of three algorithms. The best result was given by the combination of all three algorithms. This proved that the usage of ensemble methods can increase the accuracy of predicting the quality of wine.

References

[1] Dahal, K. , Dahal, J. , Banjade, H. and Gaire, S. (2021) Prediction of Wine Quality Using Machine Learning Algorithms. Open Journal of Statistics, 11, 278-289. https://doi.org/10.4236/ojs.2021.112015

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

[3] Parneeta Dhaliwal, Suyash Sharma, Lakshay Chauhan, ”Detailed Study of Wine Dataset and its Optimization”, International Journal of Intelligent Systems and Applications(IJISA), Vol.14, No.5, pp.35-46, 2022. DOI:10.5815/ijisa.2022.05.04

[4] S. Kumar, K. Agrawal and N. Mandan, ”Red Wine Quality Prediction Using Machine Learning Techniques,” 2020 International Conference on Computer Communication and Informatics (ICCCI), Coimbatore, India, 2020, pp. 1-6, doi: 10.1109/ICCCI48352.2020.9104095.

[5] Sinha, A., Kumar, A. (2020). Wine Quality and Taste Classification Using Machine Learning Model. International Journal of Innovative Research in Applied Sciences and Engineering (IJIRASE), 4(4), 715-721.

[6] UCI Machine Learning Repository. (2023) https://archive.ics.uci.edu/ml/datasets/wine+quality

[7] Yogesh Gupta. (2018) Selection of important features and predicting wine quality using machine learning techniques, Procedia Computer Science, Volume 125, 2018, Pages 305-312. https://doi.org/10.1016/j.procs.2017.12.041 .

The Dataset

The red wine dataset can be found in: https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/

The Implementation

For the stacking.py and averaging.py, we imported knn.py, support_vector.py and decision_tree.py. Before running those files, the user should make sure that they are in the same folder.

Used libraries:

- Scikit-Learn library, it can be found in: https://scikit-learn.org/stable/install.html
  It can be installed with the command: $ `pip install -U scikit-learn`
- Numpy Library, it can be found in: https://numpy.org/
  It can be installed with the command: $ pip install numpy
- Seaborn Library, it can be found in: https://seaborn.pydata.org/installing.html
  It can be installed with the command: $ pip install seaborn
- Matplotlib Library, it can be found in: https://matplotlib.org/stable/users/installing/index.html
  It can be installed with the command: $ python -m pip install -U matplotlib

Heatmap.py

```python
import numpy as np
from scipy.stats import pearsonr
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

data = np.loadtxt(r'C:\Project\winequality-red.csv', delimiter=';',
dtype=np.float32, skiprows=1)

inputs = data[:,:-1]
labels = data[:,-1]

attribute_names = ['fixed acidity', 'volatile acidity', 'citric acid',
'residual sugar', 'chlorides', 'free SO2', 'total SO2', 'density', 'ph',
'sulphates', 'alcohol', 'quality']

correlations = []
for i in range(inputs.shape[1]):
    correlation, _ = pearsonr(inputs[:, i], labels)
    correlations.append(correlation)

for i in range(len(correlations)):
    print(f"Correlation coefficient for attribute {i+1}: {correlations[i]}")
```

```
inputs = pd.DataFrame(inputs)
labels = pd.Series(labels)
totalData = pd.DataFrame(data, columns=attribute_names)
data_with_labels = pd.concat([inputs, labels], axis=1)

# correlation matrix
correlation_matrix = totalData.corr()
plt.figure(figsize=(13, 12))
#the heatmap
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Heatmap')
plt.show()
```

knn.py

```
import math
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
import numpy as np

data = np.loadtxt("winequality-red.csv", delimiter=';', dtype=np.float32,
skiprows=1)
inputs = data[:,:-1]
labels = data[:,-1]

scaler = StandardScaler()
standardized_data = scaler.fit_transform(inputs)

# tuning for parameter k
def tune():
    average_error_test = 0
    average_error_train = 0
    smallest_error = math.inf
    smallest_index = math.inf

    for i in range(100):
        # Split the dataset into training and test sets
        X_train, X_test, y_train, y_test = train_test_split(standardized_data,
labels, test_size=0.2, random_state=1000)
```

```
        # Create a decision tree regressor
        regressor = KNeighborsRegressor(i + 1)

        # Train the model
        regressor.fit(X_train, y_train)

        # Make predictions on the test set
        y_pred_test = regressor.predict(X_test)
        y_pred_train = regressor.predict(X_train)

        # Evaluate the model
        mse_test = mean_squared_error(y_test, y_pred_test)
        mse_train = mean_squared_error(y_train, y_pred_train)
        print("Mean Squared Error For Test:", mse_test)
        print("Mean Squared Error For Train:", mse_train)

        average_error_test += mse_test
        average_error_train += mse_train
        if i + 1 == 11:
            print(y_pred_train)
            print(y_pred_test)
        if smallest_error > mse_test:
            smallest_error = mse_test
            smallest_index = i + 1

    print("Test Error: " + str(average_error_test / 100))
    print("Train Error: " + str(average_error_train / 100))
    print(smallest_index)
    print(smallest_error)
    return smallest_index

# This is used by tuning process for finding the optimal hyperparameter
#optimizedValue = tune()

def errors(optimizedValue):
    regressor = KNeighborsRegressor(optimizedValue)
    X_train, X_test, y_train, y_test = train_test_split(standardized_data,
labels, test_size=0.2, random_state=1000)
    regressor.fit(X_train, y_train)
    y_pred_test = regressor.predict(X_test)
    y_pred_train = regressor.predict(X_train)

    mse_test = mean_squared_error(y_test, y_pred_test)
    mse_train = mean_squared_error(y_train, y_pred_train)
    print("Mean Squared Error For Test:", mse_test)
    print("Mean Squared Error For Train:", mse_train)
```

```python
# using the tuning process, it takes the optimal value for input, and it gives
the train and test results
#errors(optimizedValue)


regressor = KNeighborsRegressor(4)
```

decision_tree.py

```python
import math

from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error
import numpy as np

data = np.loadtxt("winequality-red.csv", delimiter=';', dtype=np.float32,
skiprows=1)

inputs = data[:,:-1]
labels = data[:,-1]

#tuning for parameter depth
def tune():
    average_error_test = 0
    average_error_train = 0
    smallest_error = math.inf
    smallest_index = math.inf
    for i in range(100):
        # Split the dataset into training and test sets
        X_train, X_test, y_train, y_test = train_test_split(inputs, labels,
test_size=0.2, random_state=1000)

        # Create a decision tree regressor
        regressor = DecisionTreeRegressor(max_depth=i + 1, random_state=3)

        # Train the model
        regressor.fit(X_train, y_train)

        # Make predictions on the test set
        y_pred_test = regressor.predict(X_test)
        y_pred_train = regressor.predict(X_train)

        # Evaluate the model
        mse_test = mean_squared_error(y_test, y_pred_test)
        mse_train = mean_squared_error(y_train, y_pred_train)
        print("Mean Squared Error For Test:", mse_test)
        print("Mean Squared Error For Train:", mse_train)

        average_error_test += mse_test
        average_error_train += mse_train
        if smallest_error > mse_test:
            smallest_error = mse_test
            smallest_index = i + 1

    print("Test Error: " + str(average_error_test / 100))
    print("Train Error: " + str(average_error_train / 100))
    print(smallest_index)
    print(smallest_error)

    return smallest_index

# This is used by tuning process for finding the optimal hyperparameter
#optimizedValue = tune()

def errors(optimizedValue):
    X_train, X_test, y_train, y_test = train_test_split(inputs, labels,
test_size=0.2, random_state=1000)
    regressor = DecisionTreeRegressor(max_depth=optimizedValue,
random_state=1000)

    regressor.fit(X_train, y_train)
    y_pred_test = regressor.predict(X_test)
    y_pred_train = regressor.predict(X_train)

    mse_test = mean_squared_error(y_test, y_pred_test)
    mse_train = mean_squared_error(y_train, y_pred_train)
    print("Mean Squared Error For Test:", mse_test)
    print("Mean Squared Error For Train:", mse_train)

# using the tuning process, it takes the optimal value for input, and it gives
the train and test results
#errors(optimizedValue)
```

```
regressor = DecisionTreeRegressor(max_depth=4, random_state=1000)
```

support_vector.py

```
import math
import numpy as np
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error


data = np.loadtxt("winequality-red.csv", delimiter=';', dtype=np.float32,
skiprows=1)

inputs = data[:,:-1]
labels = data[:,-1]

def tune():

    average_error_test = 0
    average_error_train = 0
    smallest_error = math.inf
    smallest_index = (math.inf,math.inf)
    for i in range(10):
        for j in range (100):

            X_train, X_test, y_train, y_test = train_test_split(inputs,
labels, test_size=0.2, random_state=1000)
            regressor = SVR(kernel='rbf', C=10 ** (i - 5), epsilon=(j+1)*0.01)
            regressor.fit(X_train, y_train)

            y_pred_test = regressor.predict(X_test)
            y_pred_train = regressor.predict(X_train)
            mse_test = mean_squared_error(y_test, y_pred_test)
            mse_train = mean_squared_error(y_train, y_pred_train)
            print("Mean Squared Error For Test:", mse_test)
            print("Mean Squared Error For Train:", mse_train)

            average_error_test += mse_test
            average_error_train += mse_train

            if smallest_error > mse_test:
                smallest_error = mse_test
                smallest_index = (i + 1, j + 1)
        average_error_test = average_error_test / 100
        average_error_train = average_error_train / 100
    print("Test Error: " + str(average_error_test / 10))
    print("Train Error: " + str(average_error_train / 10))
    print(smallest_index)
    print(smallest_error)
    return smallest_index

# This is used by tuning process for finding the optimal hyperparameter
#optimizedValue = tune()

def errors(optimizedValue):
    X_train, X_test, y_train, y_test = train_test_split(inputs, labels,
test_size=0.2, random_state=1000)
    regressor = SVR(kernel='rbf', C=10 ** 4, epsilon=19 * 0.01)
    #regressor = SVR(kernel='rbf', C=optimizedValue[0],
epsilon=optimizedValue[1])
    regressor.fit(X_train, y_train)
    y_pred_test = regressor.predict(X_test)
    y_pred_train = regressor.predict(X_train)

    mse_test = mean_squared_error(y_test, y_pred_test)
    mse_train = mean_squared_error(y_train, y_pred_train)
    print("Mean Squared Error For Test:", mse_test)
    print("Mean Squared Error For Train:", mse_train)

# using the tuning process, it takes the optimal value for input, and it gives
the train and test results
#errors(optimizedValue)

regressor = SVR(kernel='rbf', C=10**4, epsilon=19*0.01)
```

Averaging.py

```
import numpy as np
from sklearn.ensemble import BaggingRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
import knn
import support_vector
import decision_tree

data = np.loadtxt('winequality-red.csv', delimiter=';', dtype=np.float32,
skiprows=1)
inputs = data[:,:-1]
labels = data[:,-1]

X_train, X_test, y_train, y_test = train_test_split(inputs, labels,
test_size=0.2, random_state=1000)
scaler = StandardScaler()
```

```
standardized_data = scaler.fit_transform(inputs)

X_trainS, X_testS, y_trainS, y_testS = train_test_split(standardized_data,
labels, test_size=0.2, random_state=1000)

knn = knn.regressor
svr = support_vector.regressor
decision_tree = decision_tree.regressor

knn.fit(X_trainS, y_trainS)
svr.fit(X_train, y_train)
decision_tree.fit(X_train, y_train)

knn_preds = knn.predict(X_trainS)
svr_preds = svr.predict(X_train)
decision_tree_preds = decision_tree.predict(X_train)

knn_preds_test = knn.predict(X_testS)
svr_preds_test = svr.predict(X_test)
decision_tree_preds_test = decision_tree.predict(X_test)

averages = []
for i in range(len(knn_preds)):
    averages.append((knn_preds[i] + svr_preds[i] + decision_tree_preds[i])/3)

mse = mean_squared_error(averages, y_train)

print("Averaging train MSE :" ,mse)

averages = []
for i in range(len(knn_preds_test)):
    averages.append((knn_preds_test[i] + svr_preds_test[i] +
decision_tree_preds_test[i])/3)

mse = mean_squared_error(averages, y_test)

print("Averaging test MSE :", mse)
```

Stacking.py

```
from sklearn.ensemble import StackingRegressor
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
import numpy as np
import support_vector
import decision_tree
import knn

data = np.loadtxt('winequality-red.csv', delimiter=';', dtype=np.float32,
skiprows=1)
inputs = data[:, :-1]
labels = data[:, -1]

scaler = StandardScaler()
standardized_data = scaler.fit_transform(inputs)

X_trainS, X_testS, y_trainS, y_testS = train_test_split(standardized_data,
labels, test_size=0.2, random_state=1000)

X_train, X_test, y_train, y_test = train_test_split(inputs, labels,
test_size=0.2, random_state=1000)

knn = knn.regressor
support_vector = support_vector.regressor
decision_tree = decision_tree.regressor


def model1(x):
    # firstly fitting the train data to model
    knn.fit(X_trainS, y_trainS)
    support_vector.fit(X_train, y_train)
    decision_tree.fit(X_train, y_train)
    # predicting output from the train values
    knnPred = knn.predict(X_train)
    support_vectorPred = support_vector.predict(X_train)
    decisionTreePred = decision_tree.predict(X_train)
    # gathering the train outputs from these 3 models and creating the input
of metamodel
    # we create different metamodels
    metaFeatures = 0
    if(x==0):
        metaFeatures = np.column_stack((knnPred, support_vectorPred,
decisionTreePred))
    elif(x==1):
        metaFeatures = np.column_stack((knnPred, support_vectorPred))
    elif(x==2):
        metaFeatures = np.column_stack((knnPred, decisionTreePred))
    elif(x==3):
        metaFeatures = np.column_stack((support_vectorPred, decisionTreePred))
    # fitting polynomial regression model to our meta model
    degree = 3
    polyFeatures = PolynomialFeatures(degree=degree)
    metaModel = Pipeline([('polynomial_features', polyFeatures),
                             ('linear_regression', LinearRegression())])
    metaModel.fit(metaFeatures, y_train)
    # predicting train values from the metamodel
    train_pred = metaModel.predict(metaFeatures)
    # predicting test outputs for each model
    knn_pred_test = knn.predict(X_test)
    support_vector_pred_test = support_vector.predict(X_test)
    decision_tree_pred_test = decision_tree.predict(X_test)
    # gathering the test outputs from these 3 models and creating the input of
metamodel
    metaFeaturesTest = 0
    if (x == 0):
        metaFeaturesTest = np.column_stack((knn_pred_test,
support_vector_pred_test, decision_tree_pred_test))
    elif (x == 1):
        metaFeaturesTest = np.column_stack((knn_pred_test,
support_vector_pred_test))
    elif (x == 2):
        metaFeaturesTest = np.column_stack((knn_pred_test,
decision_tree_pred_test))
    elif (x == 3):
        metaFeaturesTest = np.column_stack((support_vector_pred_test,
decision_tree_pred_test))

    # predicting test outputs from the metamodel
    test_predict = metaModel.predict(metaFeaturesTest)
    # obtained errors
    mse_train = mean_squared_error(y_train, train_pred)
    mse_test = mean_squared_error(y_test, test_predict)
    if(x==0):
        print("Model 1 K-Nearest-Neighbor - Decision Tree - Support Vector
Regressor MSE TRAIN: ", mse_train)
        print("Model 1 K-Nearest-Neighbor - Decision Tree - Support Vector
Regressor MSE TEST: ", mse_test)
    elif(x==1):
        print("Model 1 K-Nearest-Neighbor - Support Vector Regressor MSE
TRAIN: ", mse_train)
        print("Model 1 K-Nearest-Neighbor - Support Vector Regressor MSE TEST:
", mse_test)
    elif (x == 2):
        print("Model 1 K-Nearest-Neighbor - Decision Tree MSE TRAIN: ",
mse_train)
        print("Model 1 K-Nearest-Neighbor - Decision Tree MSE TEST: ",
mse_test)
    elif (x == 3):
        print("Model 1 Support Vector Regressor - Decision Tree MSE TRAIN: ",
mse_train)
        print("Model 1 Support Vector Regressor - Decision Tree MSE TEST: ",
mse_test)
```

```
def get_stacking_3Model():
    level0 = list()
    level0.append(('knn', knn))
    level0.append(('svr', support_vector))
    level0.append(('dt', decision_tree))
    polynomialFeatures = PolynomialFeatures(degree=2)  # Set the degree of
polynomial features
    estimator = LinearRegression()
    estimatorpip = Pipeline([('polynomial_features', polynomialFeatures),
                                     ('linear_regression', estimator)])
    model = StackingRegressor(estimators=level0, final_estimator=estimatorpip,
cv=None)
    return model

def get_stacking_knn_svr():
    level0 = list()
    level0.append(('knn', knn))
    level0.append(('svr', support_vector))
    polynomialFeatures = PolynomialFeatures(degree=2)  # Set the degree of
polynomial features
    estimator = LinearRegression()
    estimatorpip = Pipeline([('polynomial_features', polynomialFeatures),
                                     ('linear_regression', estimator)])
    model = StackingRegressor(estimators=level0, final_estimator=estimatorpip,
cv=None)
    return model
def get_stacking_knn_dt():
    level0 = list()
    level0.append(('knn', knn))
    level0.append(('dt', decision_tree))
    polynomialFeatures = PolynomialFeatures(degree=2)  # Set the degree of
polynomial features
    estimator = LinearRegression()
    estimatorpip = Pipeline([('polynomial_features', polynomialFeatures),
                                     ('linear_regression', estimator)])
    model = StackingRegressor(estimators=level0, final_estimator=estimatorpip,
cv=None)
    return model

def get_stacking_svr_dt():
    level0 = list()
    level0.append(('svr', support_vector))
    level0.append(('dt', decision_tree))
    polynomialFeatures = PolynomialFeatures(degree=2)  # Set the degree of
polynomial features
    estimator = LinearRegression()
    estimatorpip = Pipeline([('polynomial_features', polynomialFeatures),
                                          ('linear_regression', estimator)])
    model = StackingRegressor(estimators=level0, final_estimator=estimatorpip,
cv=None)
    return model

def model2():
    train_model = get_stacking_3Model().fit(X_train, y_train)
    predicted_values = train_model.predict(X_test)
    predicted_y = train_model.predict(X_train)

    mse_1 = mean_squared_error(y_train, predicted_y)
    mse_2 = mean_squared_error(y_test, predicted_values)

    print("Model 2 K-Nearest-Neighbor - Decision Tree - Support Vector
Regressor MSE TRAIN: ", mse_1)
    print("Model 2 K-Nearest-Neighbor - Decision Tree - Support Vector
Regressor MSE TEST: ", mse_2)

def model3_knn_svr():
    train_model = get_stacking_knn_svr().fit(X_train, y_train)
    predicted_values = train_model.predict(X_test)
    predicted_y = train_model.predict(X_train)

    mse_1 = mean_squared_error(y_train, predicted_y)
    mse_2 = mean_squared_error(y_test, predicted_values)

    print("Model 2 K-Nearest-Neighbor - Support Vector Regressor MSE TRAIN: ",
mse_1)
    print("Model 2 K-Nearest-Neighbor - Support Vector Regressor MSE TEST: ",
mse_2)

def model3_knn_dt():
    train_model = get_stacking_knn_dt().fit(X_train, y_train)
    predicted_values = train_model.predict(X_test)
    predicted_y = train_model.predict(X_train)

    mse_1 = mean_squared_error(y_train, predicted_y)
    mse_2 = mean_squared_error(y_test, predicted_values)

    print("Model 2 K-Nearest-Neighbor - Decision tree MSE TRAIN: ", mse_1)
    print("Model 2 K-Nearest-Neighbor - Decision tree MSE TEST: ", mse_2)

def model3_svr_dt():
    train_model = get_stacking_svr_dt().fit(X_train, y_train)
    predicted_values = train_model.predict(X_test)
    predicted_y = train_model.predict(X_train)

    mse_1 = mean_squared_error(y_train, predicted_y)
    mse_2 = mean_squared_error(y_test, predicted_values)
    print("Model 2 Support Vector Regressor - Decision Tree MSE TRAIN: ",
mse_1)
    print("Model 2 Support Vector Regressor - Decision Tree MSE TEST: ",
mse_2)

model1(0)
model1(1)
model1(2)
model1(3)
model2()
model3_knn_svr()
model3_knn_dt()
model3_svr_dt()
```