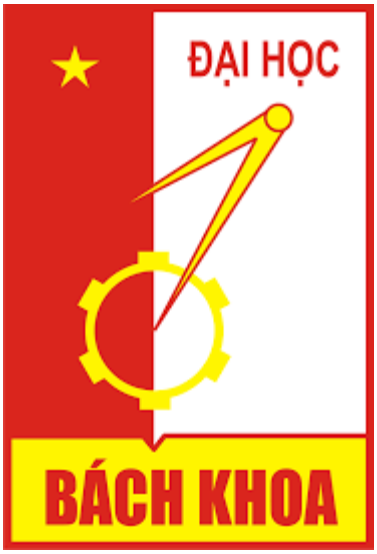
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
School of Information and Communication Technology



Mini Project Report

Topic: Gemstones price prediction

Instructor: Than Quang Khoat

Team 13

Student's name	**Student's ID**
Nguyen Viet Minh	20214917
Phan Duc Hung	20214903
Truong Gia Bach	20210087
Tran Duong Chinh	20210122

Hanoi – July 6, 2023

Gemstone Price Prediction

Nguyen Viet Minh	*Phan Duc Hung*	*Truong Gia Bach*	*Tran Duong Chinh*
20214917	*20214903*	*20210087*	*20210122*

I Introduction

Gemstones have captivated humanity for centuries with their exquisite beauty, rarity, and enduring allure. From the brilliant sparkle of diamonds to the mesmerizing hues of sapphires and emeralds, these precious stones hold a special place in our hearts and culture. However, determining the true value of a gemstone has always been a complex and intricate task, relying on a myriad of factors such as cut, clarity, carat weight, and color. To tackle this challenge and unlock new insights into pricing, the world of gemology has turned to the cutting-edge field of machine learning.

In this exploration of predicting gemstone prices using machine learning, we delve into the fascinating world of gemology, where the artistry of nature converges with the analytical power of technology. We will uncover the key factors that influence gemstone value, including the famous four Cs—cut, clarity, carat weight, and color—and examine how machine learning algorithms can learn from these factors to develop robust pricing models.

Join us on this captivating journey as we combine the timeless beauty of gemstones with the cutting-edge power of machine learning. Together, we will unravel the mysteries of gemstone valuation, unlocking new horizons in the fascinating world of precious gemstones.

`Note:` Remember that we are working with `cubic zirconia`. We are provided with the dataset containing the prices and other attributes of almost 193,600 cubic zirconia (which is an inexpensive diamond alternative with many of the same qualities as a diamond). The company is earning different profits on different prize slots. We have to help the company in predicting the price for the stone on the basis of the details given in the dataset so it can distinguish between higher profitable stones and lower profitable stones so as to have a better profit share `jewels`.

II Data Explanation

1 Data Collections

The data is collected from Kaggle: https://www.kaggle.com/competitions/playground-series-s3e8/data

This data set (both train and test) was generated from a deep learning model trained on the Gemstone Price Prediction dataset. Feature distributions are close to, but not exactly the same, as the original.

Overview of the data:

id	carat	cut	color	clarity	depth	table	x	y	z	price
0	1.52	Premium	F	VS2	62.2	58.0	7.27	7.33	4.55	13619
1	2.03	Very Good	J	SI2	62.0	58.0	8.06	8.12	5.05	13387
2	0.70	Ideal	G	VS1	61.2	57.0	5.69	5.73	3.50	2772
3	0.32	Ideal	G	VS1	61.6	56.0	4.38	4.41	2.71	666
4	1.70	Premium	G	VS2	62.6	59.0	7.65	7.61	4.77	14453
...	...	...	...	...	...	...	...	...	...	...
193568	0.31	Ideal	D	VVS2	61.1	56.0	4.35	4.39	2.67	1130
193569	0.70	Premium	G	VVS2	60.3	58.0	5.75	5.77	3.47	2874
193570	0.73	Very Good	F	SI1	63.1	57.0	5.72	5.75	3.62	3036
193571	0.34	Very Good	D	SI1	62.9	55.0	4.45	4.49	2.81	681
193572	0.71	Good	E	SI2	60.8	64.0	5.73	5.71	3.48	2258

193573 rows × 10 columns.

2 Attribute Comprehension

The data contains 10 attributes include:

- ***Carat***: It is the weight of the cubic zirconia.
- ***Cut***: It describes the cut quality of the cubic zirconia. Quality is in increasing order `Fair`, `Good`, `Very Good`, `Premium`, `Ideal`.
- ***Color***: It is the color of the cubic zirconia. `D` being the best and `J` the worst.
- ***Clarity***: It refers to the absence of the Inclusions and Blemishes. (In order from Best to Worst, `FL` = flawless, `I3`= level 3 inclusions) `FL`, `IF`, `VVS1`, `VVS2`, `VS1`, `VS2`, `SI1`, `SI2`, `I1`, `I2`, `I3`.
- ***Depth***: It describes the height of the cubic zirconia, measured from the Culet to the table, divided by its average Girdle Diameter.
- ***Table***: It describes the width of the cubic zirconia's Table expressed as a Percentage of its Average Diameter.
- ***Price***: It describes the Price of the cubic zirconia.

- ***X***: It describes the length of the cubic zirconia in mm.
- ***Y***: It describes the width of the cubic zirconia in mm.
- ***Z***: It describes the height of the cubic zirconia in mm.

For more details we will take a deeper look in our data

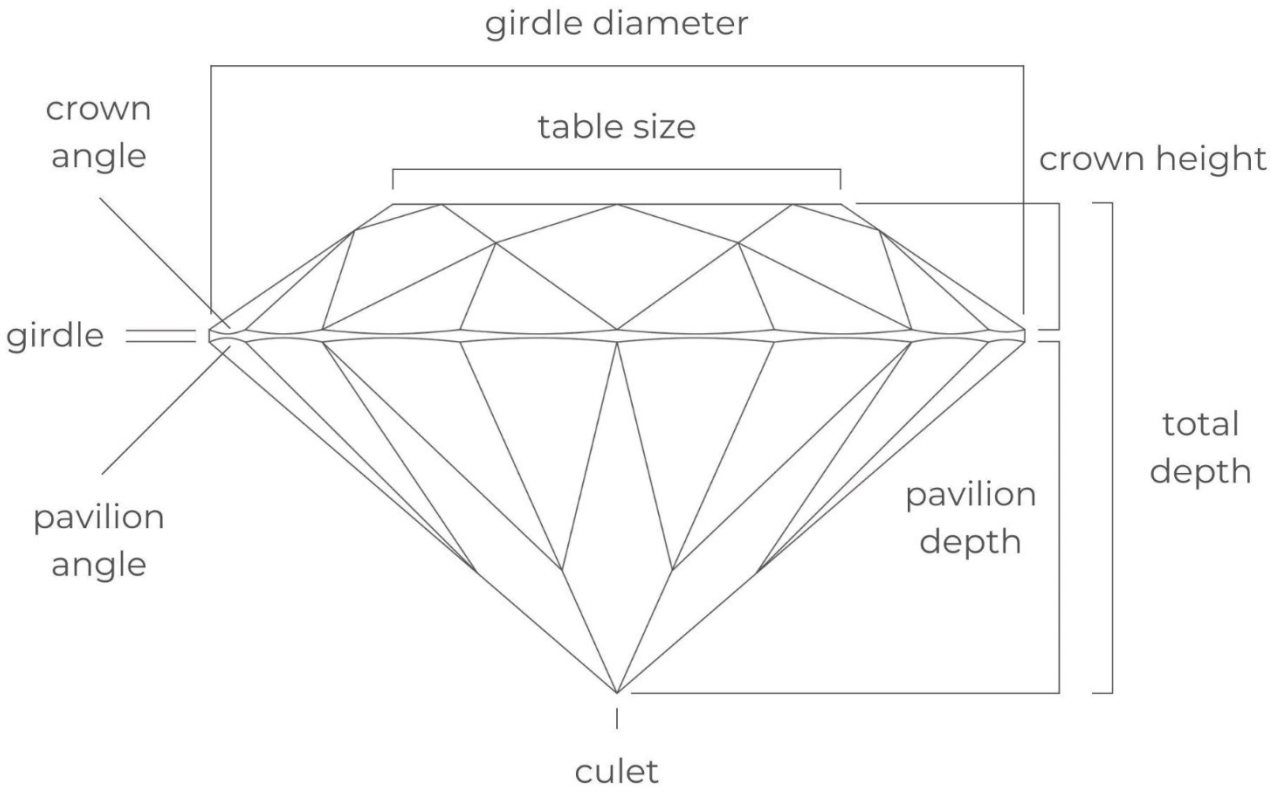


Figure 2.1: Details of a gemstone

For the attribute carat, we have the unit is carat, 1 carat = 0.2 gram

For the attribute Depth and Table (unit %), we have the following formular:

$$\texttt{Average girdle diameter} = \frac{\texttt{minimum diameter + maximum diameter}}{2} = \frac{\texttt{X} + \texttt{Y}}{2}$$

$$\texttt{Depth} = \frac{\texttt{Z}}{\texttt{Average girdle diameter}} \times 100$$

$$\texttt{Table} = \frac{\texttt{Average table size}}{\texttt{Average girdle diameter}} \times 100$$

3 Feature Classification: Categorical and Numerical

- Categorical values in data are values that can be placed into categories. These values are not numerical and cannot be compared or added together. In our data, the attributes ***cut***, ***color*** and ***clarity*** contain categorical values.
- Numerical values in data are values that can be represented as numbers. These values can be compared and added together, and they can be used to perform mathematical operations. Our data has the attributes ***carat***, ***depth***, ***table***, ***x***, ***y***, ***z*** containing numerical values.

III Exploratory Data Analysis (EDA)

1 Continuous feature

1.1 Continuous feature distribution

We will show the histograms of all attributes represented by continuous values:

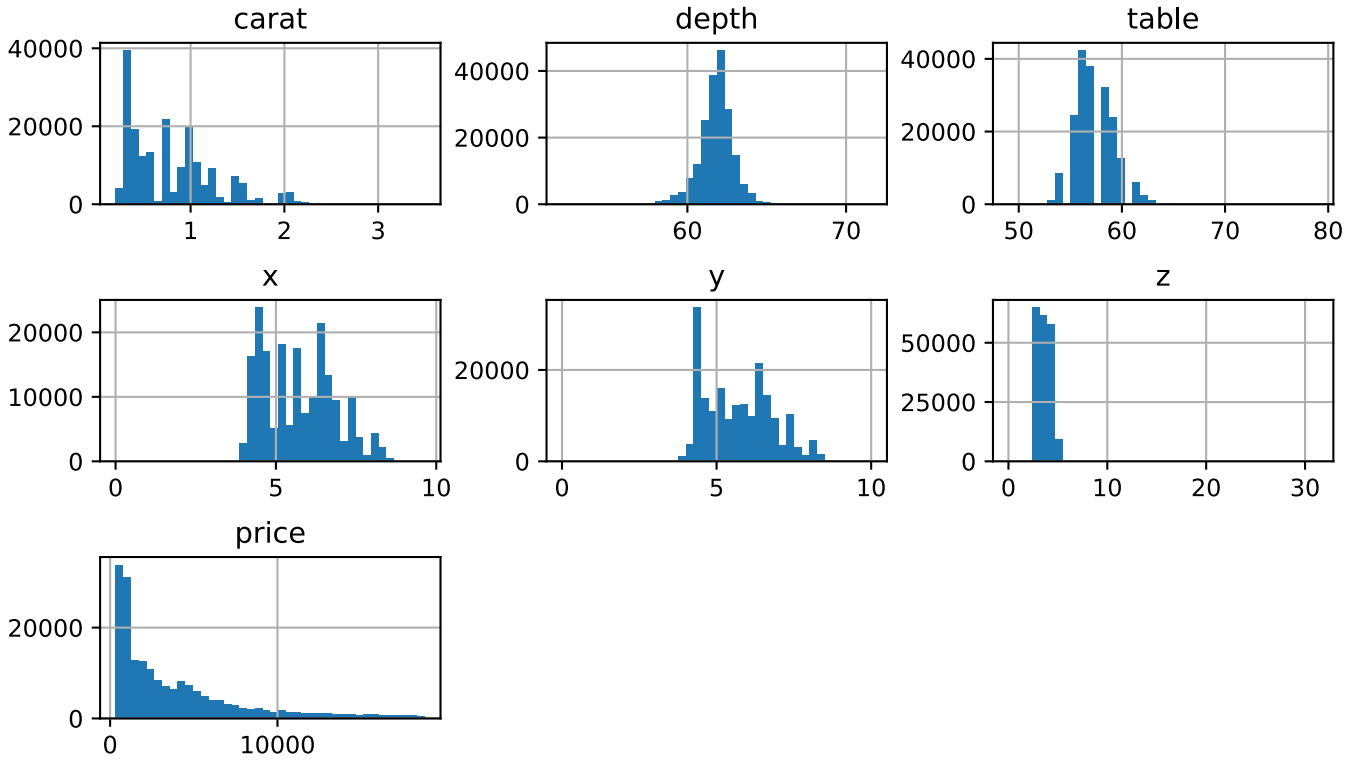


Figure 3.1: Histograms of all attributes with continuous values

For some attributes, the distribution is not so good, and follow normal distribution, we cannot avoid that, but for some attributes (not the ***price*** attribute, which we need to predict), we may try to adjust their skewness.

1.2 Continuous feature correlation

A higher ***carat*** value is equivalent to a higher price, as shown in Figure 3.2. This is easy to understand, because normally cubic that has higher weight can be sold for higher price. Notice here ***carat*** is an important feature that we might think about stratified it later to make our data smaller for validation process.

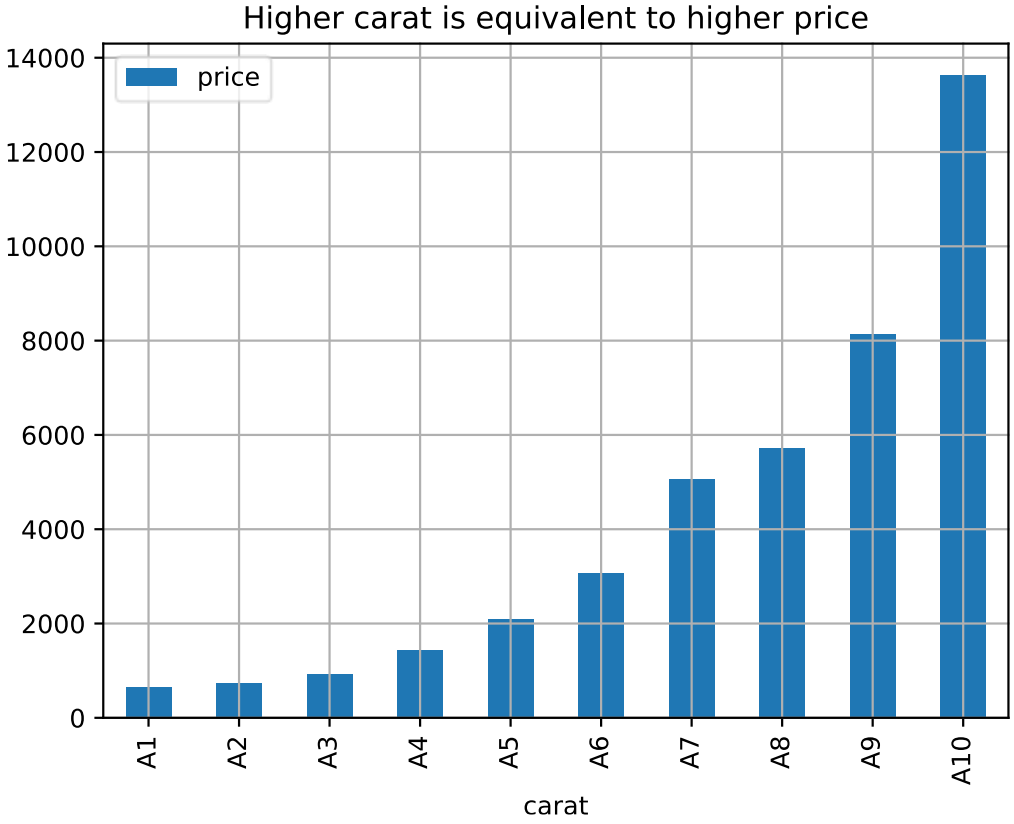


Figure 3.2: Mean price of carat, divided in 10 labels

The distribution of gemstones based on ***carat*** and ***price*** is shown in Figure 3.3. We can see the high correlation of ***price*** and ***carat***. Need to remember, ***price*** is the attribute that we work to predict so it is very important to detect important attributes of data that affect significantly the ***price*** value.

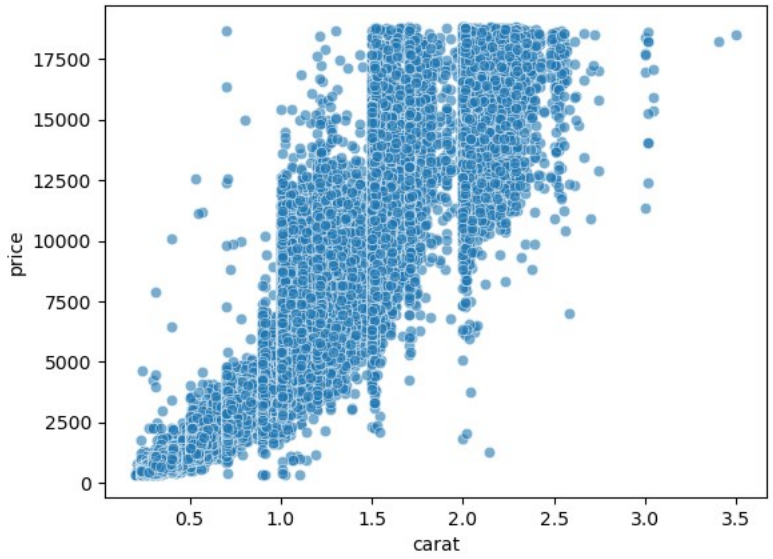


Figure 3.3: Distribution of gemstone based on carat and price

To have an overview of all the correlations may have in data, we will show a pair plot of all continuous features to each other, as seen in Figure 3.4. The shade of color of each point will be based on the ***cut*** quality

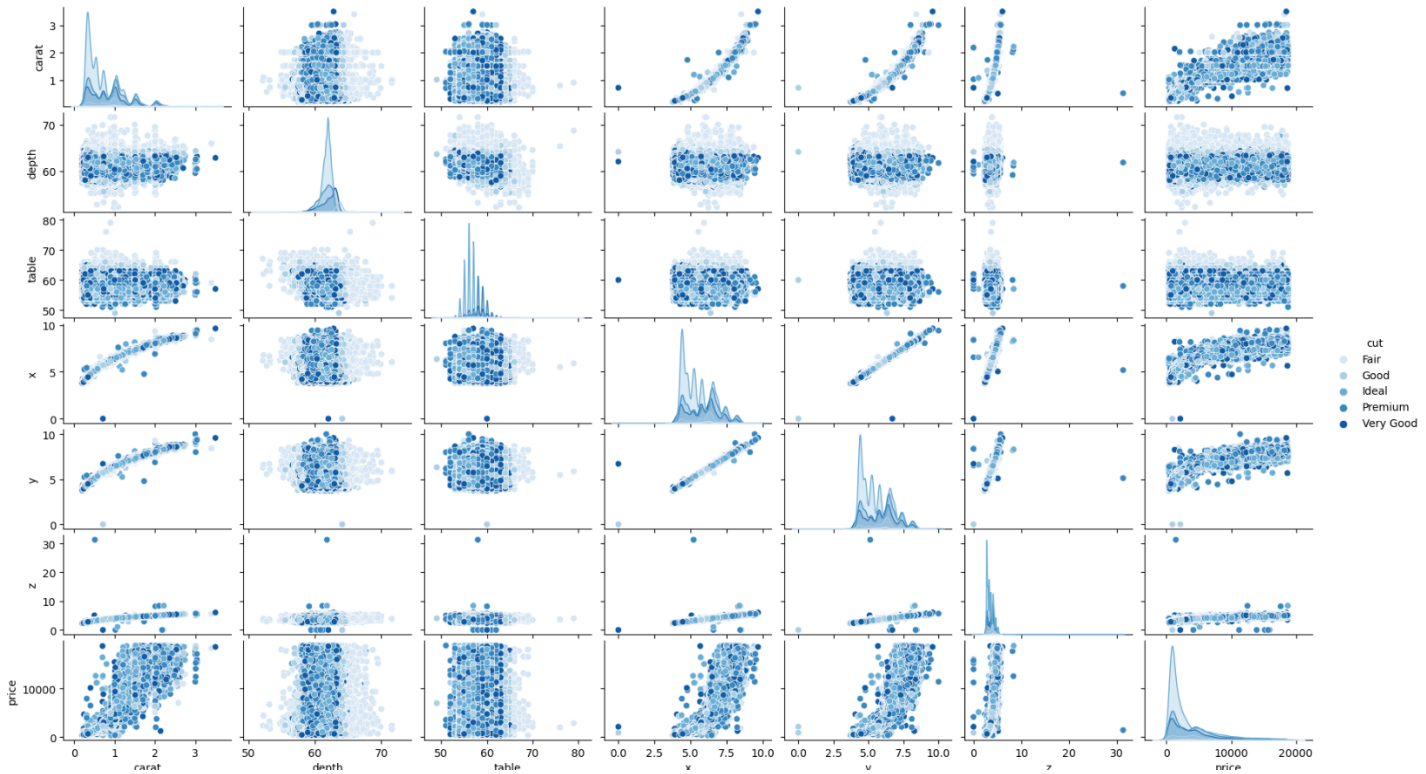


Figure 3.4: Pair-plots of all continuous features

We will take a deeper look on each pair-plot to see the correlation.

In Figure 3.4, the value of ***carat*** attributes rises as the value of ***x*** attributes increases, also, the ***price***, which represented by the color of the point, changed from cyan to violet as the price is higher. This shows a great correlation of 2 attribute ***x*** and ***carat***, and also, we have information that they affect directly to the ***price*** value.

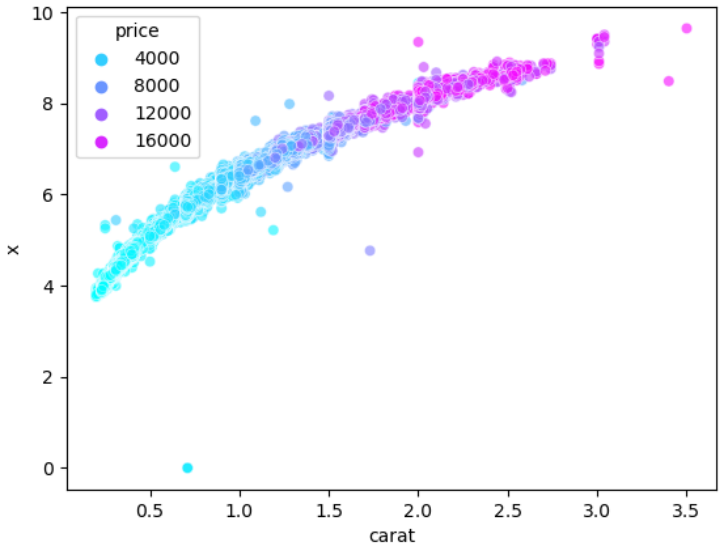


Figure 3.5: Distribution of data by ***carat*** and ***x***, with ***price*** represented by the shade of color

We also see some datapoints that have dimensional outliers (values of ***x***, ***y*** or ***z***), which lie very far from the distribution of other points. This will be taken more seriously when we work with outlier detection in Section III.1.2 and Section IV.

From the pair-plots table, we also notice that the ***depth*** and ***table*** features seem to take no effect on the ***price*** or important attribute that we assign. Which means they have no or very small weights when working on the regression algorithms.

The correlation of all continuous attributes to each other will be calculated based on the formula:

$$\rho\ X, Y\ = E\frac{(X-\mu_x)(Y-\mu_y)}{\sigma_x \cdot \sigma_y}$$

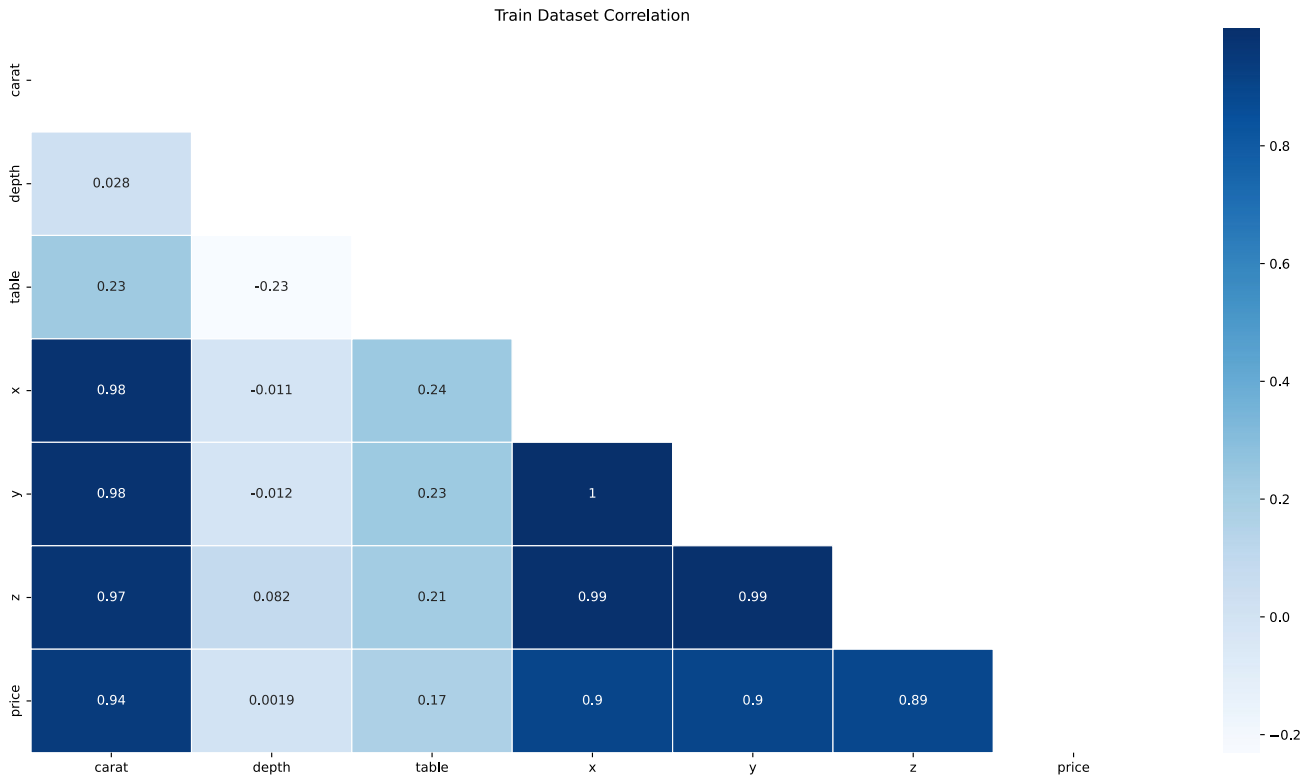


Figure 3.6: Correlation heatmap of the data

All the above plots indicate some understandable findings:

The dimensions of a diamond (i.e: ***x***, ***y***, and ***z***) show a strong correlation with its ***price*** and ***carat*** weight. As the dimensions of a diamond increase, so does its volume, which in turn affects its mass, as mass is equal to volume multiplied by density. We also think about making a feature named ***volume*** but don't have enough attributes so ignore it. For more explanation, a gem seems not to be in a cube shape so that we can calculate the ***volume*** by multiplying all the values of ***x***, ***y*** and ***z***. Its shape is complicated and the ***volume*** value that we want to evaluate is not exactly accurate.

And because our data is quite big so we will split it to stratified data based on ***carat*** in order to make the learning faster with cv and also make our data more equally.

The correlation of ***x*** and ***y*** is highest, which we can think about fusing it into average girdle diameter in the future, also is ***z*** but still have some outliers in ***z*** which we might have to take care of it.

The attributes ***depth***, ***cut*** and ***table*** show low correlations. For the regression algorithms, these attributes seem to have no or very small weights or effects. For this reason, we can consider dropping them to optimize the data.

1.3 Continuous outliers detection

Outliers in the dataset can affect the overall performance of the model. Here we analyze the dataset in order to determine and remove outliers in the continuous features.

We will show the plot of each continuous features:

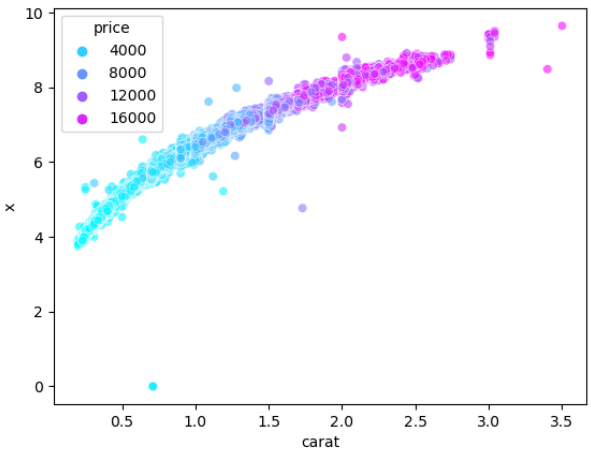


Figure 3.7: Distribution of data by ***carat*** and ***x***, with ***price*** represented by the shade of color

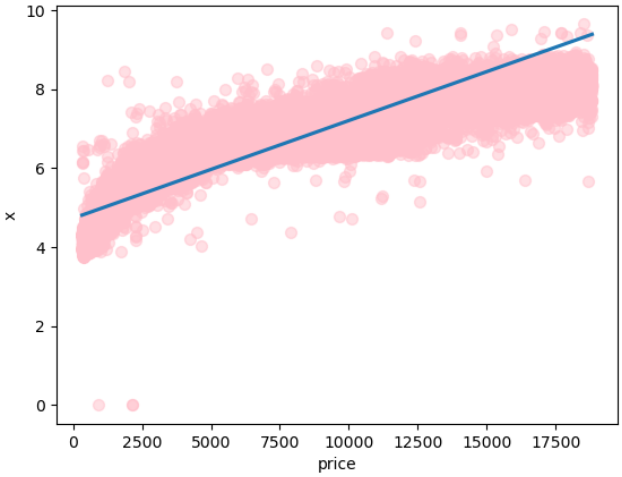


Figure 3.8. Pair-plot of ***price*** and ***x***

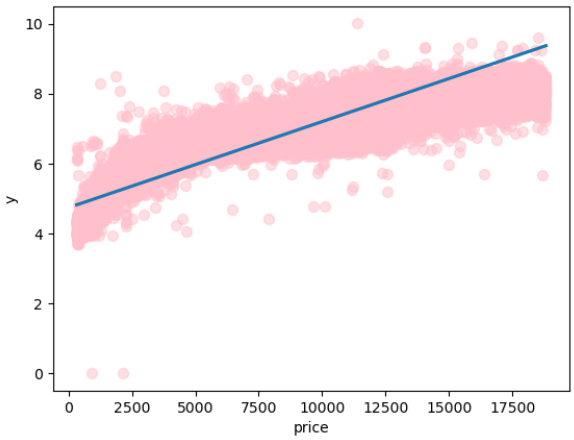


Figure 3.9: Pair-plot of ***price*** and ***y***

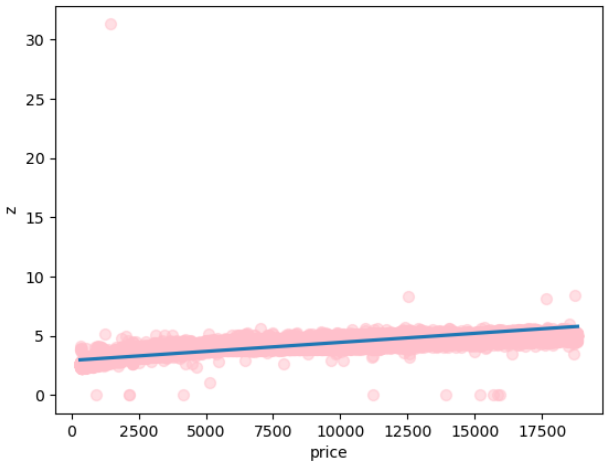


Figure 3.10: Pair-plot of ***price*** and ***z***

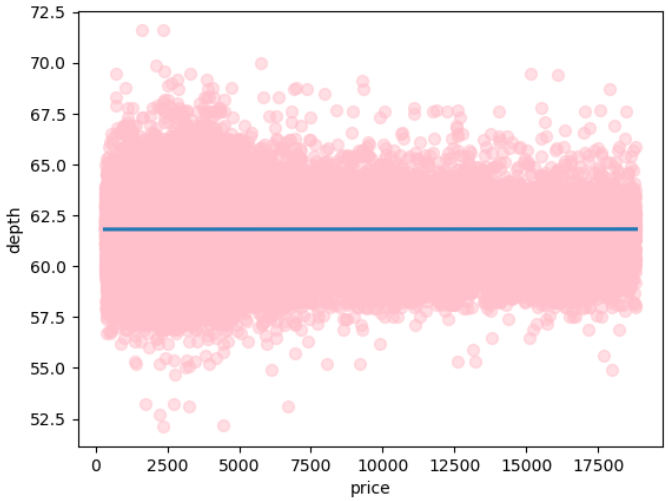


Figure 3.11: Pair-plot of ***price*** and ***depth***

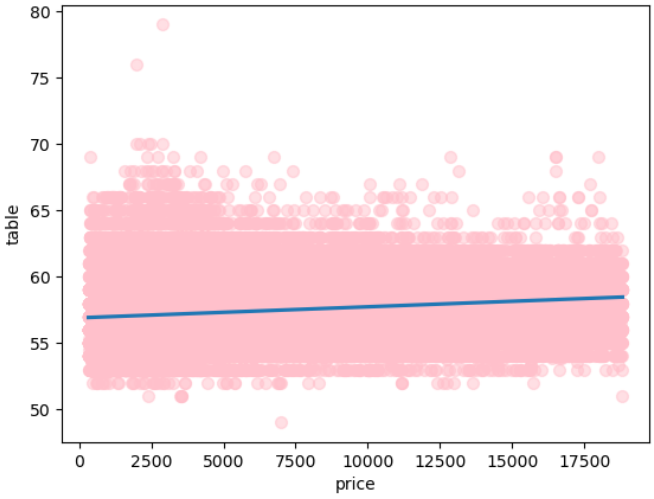


Figure 3.12: Pair-plot of ***price*** and ***table***

From the above figures, there are a few points we can notice. First there are some instances in the dataset that are far from the rest which will affect the performance of our regression model.

Taking a closer look at Figure 1, Figure 2 and Figure 3 we can clearly see that ***x, y*** and ***z*** have some dimensional outliers that need to be removed from the data set. We can inspect one such instance where the value for the ***z*** feature is unexpectedly high, exceeding 30.

id	carat	cut	color	clarity	depth	table	x	y	z	price
167044	0.51	Premium	G	VS2	61.8	58.0	5.2	5.13	31.3	1447

For the ***depth*** value, we need to cap off the instances too far from the rest of the data but we must examine the regression line to be sure. The ***table*** feature should also be capped.

These outliers will be removed from the dataset for the following sections.

2 Categorial features distribution

There are 3 categorial features in our dataset, in this section we are going to investigate further these attributes.

The 3 categorial features includes ***cut***, ***colo***r and ***clarity***. ***cut*** represents the quality of a cubic zirconia cut, with values ranging from worst quality to best, 'Fair', 'Good', 'Very Good', 'Premium' and 'Ideal'. The following figure shows us a closer insight into how the data is distributed by ***cut*** and by ***price***.

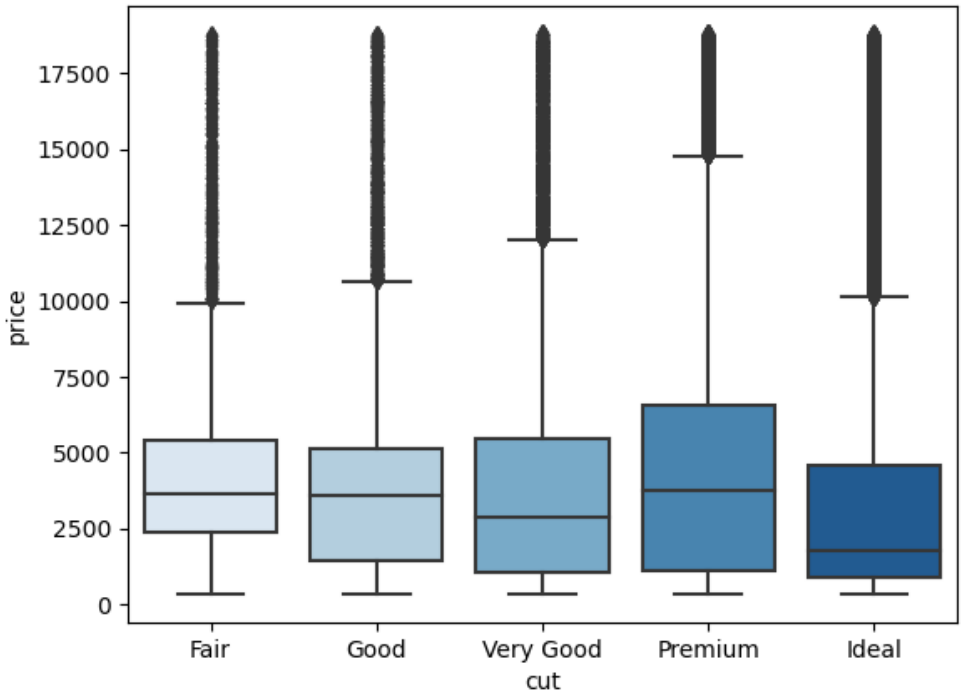


Figure 3.13: Data distribution by ***price*** and ***cut***

We can do the same for the remaining categorial features.

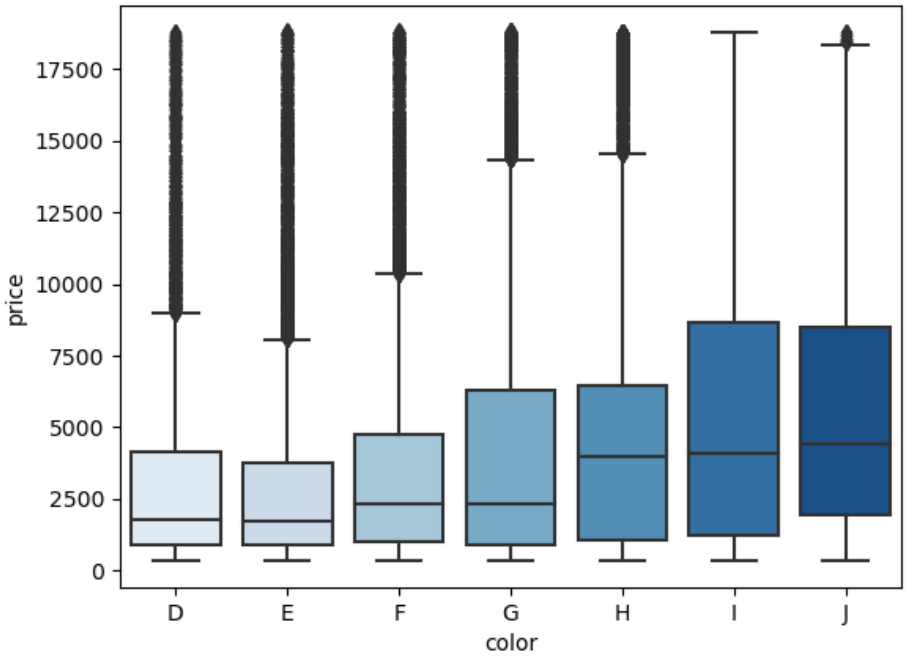


Figure 3.14: Data distribution by ***price*** and ***color***

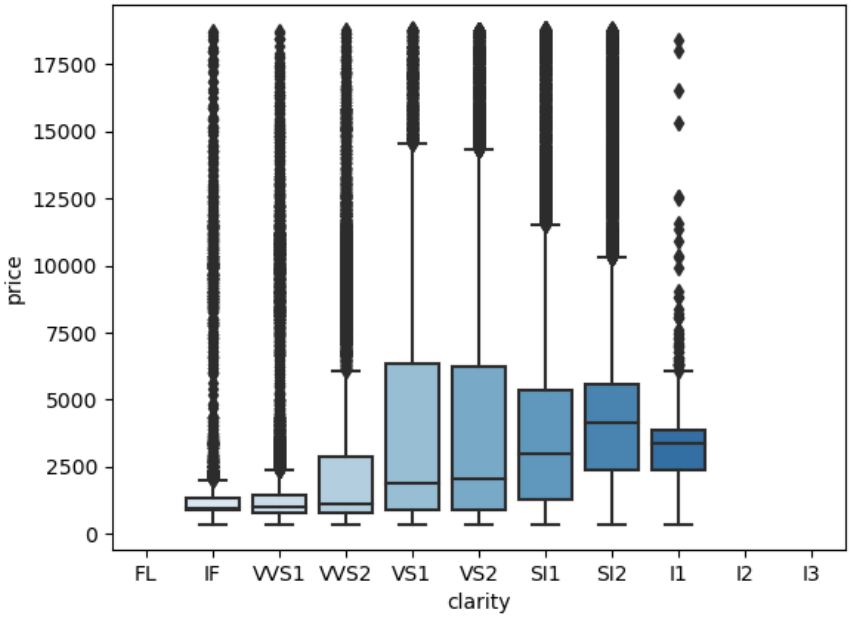


Figure 3.15: Data distribution by ***price*** and ***clarity***

We don't see any noticeable patterns in the distribution of categorial features. Perhaps if we limit the range of the other features we can see better.

IV Feature Engineering & Data Wrangling

1 Add new attributes to data

There are many instances where the value for x, y or z is 0, which we don't want. After eliminating all the value 0 in dimension, we will calculate the attribute. Average gridle diameter and add to data, we will shorten the name to ***diameter***. The formula for ***diameter*** is calculated as:

$$diameter = \frac{x + y}{2}$$

We have the attribute ***depth*** which can be calculated as the formula:

$$depth_{adj} = \frac{z}{diameter} \cdot 100$$

We have the attribute ***esti_volume*** which can be calculated as the formula:

$$volume_{estimate} = x \cdot y \cdot z$$

We will compare the attributes ***depth*** and ***depth_adj*** by absolute mean value of the difference between.

```
abs(train_df['depth_adj'] - train_df['depth']).mean()
```

```
0.10831632837563883
```

With the mean value of ***depth*** around 62, the error seems to be not so much that can affect badly our prediction, but to improve the accuracy, we will remove the ***depth*** attribute and replace by ***depth_adj***.

The ***esti_volume*** is finally got rid of because of qualifying of models. In testing phase, it just makes our prediction worse in every case.

2 Skewness Modification

First, we will calculate the skewness of all attributes with continuous values

carat	**0.993911**
depth_adj	-0.046152
table	0.619147
x	0.362721
y	0.357705
z	0.354970
diameter	0.359856

The skewness of ***carat*** distribution is high, we will reduce it by transformation. We use the square-root transformation to transform all the values of ***carat***.

```
train_df['carat'] = np.sqrt(train_df['carat'])
```

The new skewness value of ***carat*** attribute is 0.497810.

We have a new correlation heatmap to illustrate all the correlations of our new attributes in data, as shown in Figure 4.1.

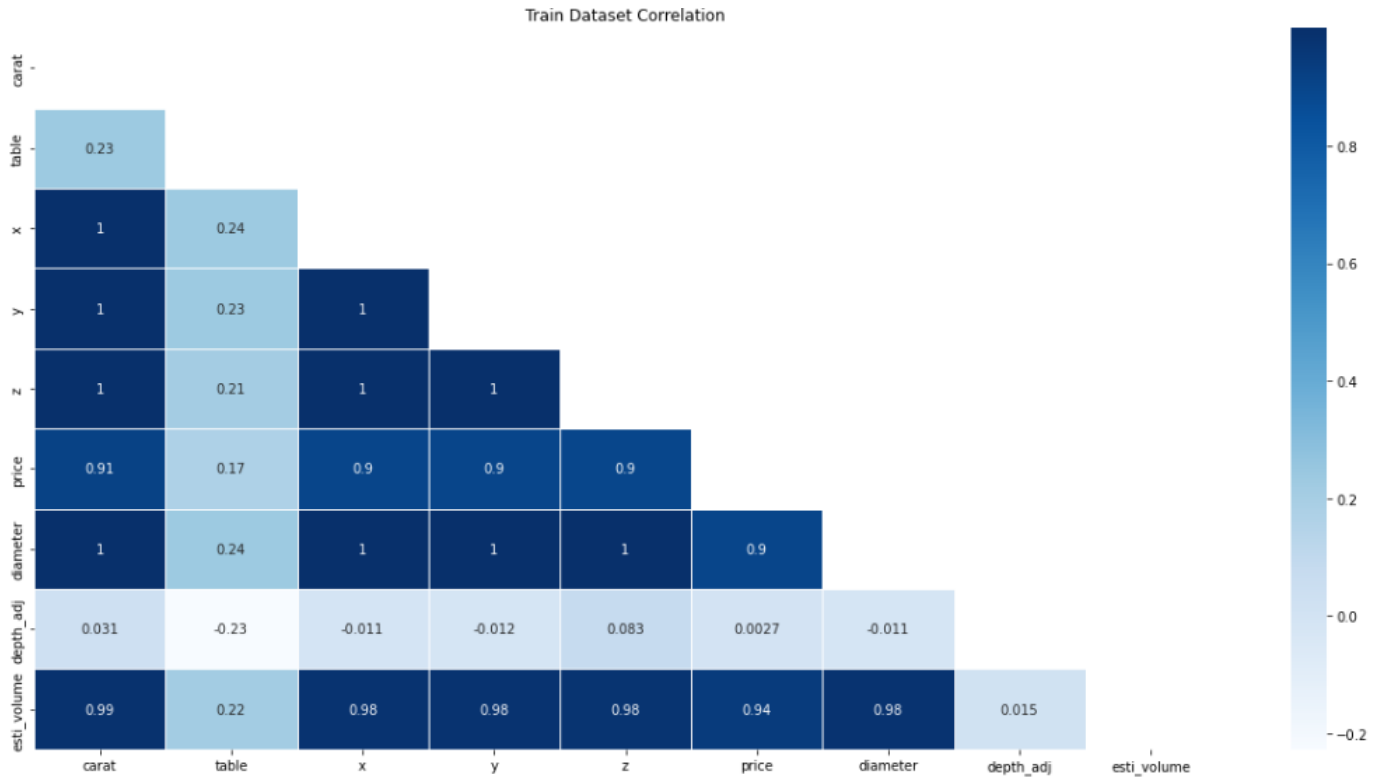


Figure 4.1: Correlation heatmap of data with new attributes

Although the esti_volume shows a high correlation and taking square root of carat makes the correlation to be lower but our testing said that not all high correlation will make our best model which is XGB better. Not surely with other methods, maybe Linear Regression algorithms will take advantage of this higher correlation.

3 Splitting data stratified based on carat attribute

We will create new data that is smaller in size than the original data, with the aim of using it for complex regression algorithms to make them run faster. We will split the data collected from the file train.csv to 3 groups, then collect from each group a percentage of data with respect to the original percentage of data. This can ensure that our new data can represent various categories of incomes from the whole dataset.

We will split the data collected from the file train.csv to train set and validation set to evaluate the quality of each regression algorithm and make conclusion. So here, to ensure that both the train set and validation set can represent various categories of incomes from the whole dataset.

We will split data based on the ***carat*** values, into 3 ranges, $0 - 0.5$, $0.5 - 1.5$ and $1.5 - \infty$, which are sequentially labeled 1, 2 and 3. We will keep track of each data point by adding a new attribute called ***weight_cat***, with values categorized to 1, 2 and 3.

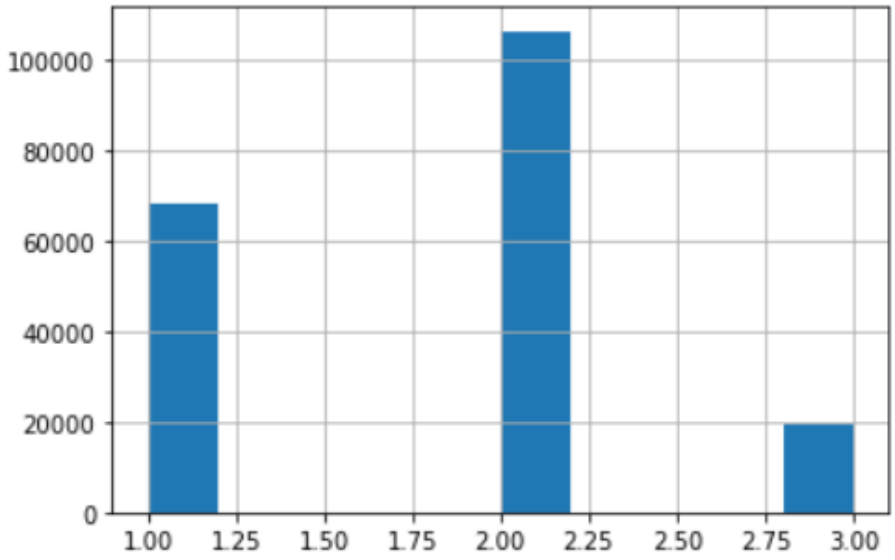


Figure 4.2: Split data to 3 groups based on ***carat*** value

We will use method StratifiedShuffleSplit from sklearn.model_selection to split the data

```
from sklearn.model_selection import StratifiedShuffleSplit

spliting = StratifiedShuffleSplit(n_splits=1, test_size=0.6, random_state=21)
for use_index, leave_index in spliting.split(train_df, train_df['weight_cat'])
    stra_use_set = train_df.loc[use_index]
    stra_leave_set = train_df.loc[leave_index]
```

We set the n_splits = 1 to have 1 set only, the test_size = 0.6, which means our new data will have the size equals to 60% of the original data size, the random_state that needs to be the same for each randomization we create, we will set it to be 21. So here we will have the stra_use_set to be new training data.

4 Data Wrangling

Regression only works with continuous value, so for all the category attributes, we will label each category value to a new integer. The way to label needs to be carefully determined from the better of the category value can affect our generalized function value, which in our data is the price attribute. Otherwise, the data wrangling will seem to be non-sense cause the category is assigned to number that shows no distribution.

We will use the OrdinalEncoder from sklearn.preprocessing to label each category value to number, with order will be assigned by our choice.

For the ***cut*** value, we have 5 values to be 'Fair', 'Good', 'Very Good', 'Premium', 'Ideal', the order of them should be 'Fair' for the worst, 'Good', 'Very Good', 'Premium' with sequentially increase better and 'Ideal' to be the best. We will assign them to 5 values, from 0 to 4:

- 'Fair' assigned to 0
- 'Good' assigned to 1
- 'Very Good' assigned to 2
- 'Premium' assigned to 3
- 'Ideal' assigned to 4

Similarly, the ***color*** values will be assigned with respect to the order: 'D', 'E', 'F', 'G', 'H', 'I', 'J', and ***clarity*** values with order 'IF', 'VVS1', 'VVS2', 'VS1', 'VS2', 'SI1', 'SI2', 'I1'.

5 Standard Scale for continuous features

We will standardize features by removing the mean and scaling to unit variance. We have the new score z as the formula

$$z = \frac{x-u}{\sigma} \text{ with } u \text{ denotes the mean and } \sigma \text{ denotes standard deviation}$$

u will be assigned to 0 and s assigned to 1 as we want to scale the distribution to Standard Normal Distribution (z-distribution)

We use the StandardScaler from sklearn.preprocessing for scaling.

```
category_order = [['Fair', 'Good', 'Very Good', 'Premium', 'Ideal'],
                  ['D', 'E', 'F', 'G', 'H', 'I', 'J'],
                  ['IF', 'VVS1', 'VVS2', 'VS1', 'VS2', 'SI1', 'SI2', 'I1'],
                  ['1', '2', '3']]
```

```
# preprocess pipeline
from sklearn.preprocessing import OrdinalEncoder, StandardScaler
#from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline


num_transformer = Pipeline(steps =[
    ('scaler', StandardScaler())
])

cat_transformer = Pipeline(steps=[
    ('encoder', OrdinalEncoder(categories = category_order))
])
```

```
preprocessor = ColumnTransformer(transformers=[
    ('num', num_transformer, num_features),
    ('cat', cat_transformer, cat_features)
])
```

```
preprocessor.fit(X)
```

Figure 4.3: Code implementation for Data Wrangling and Standard Scaling

The data will then be converted to numpy array to be used directly in algorithms.

```
array([[-1.01684101,  0.18350305, -1.15765084, ...,  5.        ,
         3.        ,  0.        ],
       [-0.19524163, -0.15308456, -0.63755093, ...,  2.        ,
         5.        ,  1.        ],
       [ 0.79932605, -0.05691667,  0.40264887, ...,  4.        ,
         5.        ,  1.        ],
       ...,
       [ 0.60473672, -0.20116851, -0.63755093, ...,  1.        ,
         1.        ,  1.        ],
       [-0.17362059, -1.35518317,  1.96294858, ...,  1.        ,
         6.        ,  1.        ],
       [-0.88711479,  0.47200672, -0.63755093, ...,  1.        ,
         5.        ,  0.        ]])
```

V Model Training

1 Training and Evaluating on the Training Set

As we denoted at the Section **IV.3 Spliting data stratified based on carat attribute,** we will split data based on carat attribute to new training data and validation data. The idea is to avoid unexpectedly bad outcomes when using hold-out with uncontrolled randomness. In fact, we actually worked with the hold-out at the first time for tuning parameters, but the differences of results when changing parameters are hard to control, as it will keep changing when we change the random_state of the splitting method using hold-out from the sklearn, so, we cannot choose the best parameter for all the regression algorithms.

Using our splitting method, we can keep the diversity of ***carat,*** the attribute that affects the most to the ***price*** value, from both training and validation data set. We will see in the next parts that the parameters that we get using GridSearchCV for cross validation in the training set will be the same, or close to the parameters that make algorithms perform the best on validation set.

The ratio of the training set and validation set are a little bit surprising, as we make only 40% for the training set and 60% for the validation set. This comes from the reasons. First, our data is huge (193573 data points when not pre-processing) and the data training is too big may make our models cost too much time to fit, an adequate amount of data for training not only makes the model cost less time to run, but also creates a good generalized function, which not only can avoid overfitting (because less data learned than the original data), but also learn only the data points that are more crucial, or in other words, avoid duplication of data points. The 60% data for validation also reflects very well how the regression algorithms works on unseen data and make predictions. We will choose the alpha that performs the best on validation for the regression algorithms for comparison.

For the comparison model, we will use cross-validation on the new training data (split stratified from the original data) with the parameters that we choose above to see the efficiency of each algorithm to our problem. We will choose the best models for the submission phase, when we will let the algorithms fit all the data that we have, then make predictions for the test.csv file, with data doesn't contain values of ***price*** attribute. The result will then be submitted to the competition Regression with a Tabular Gemstone Price Dataset | Kaggle to show the loss value (RMSE) to original results. This phase can be called our testing phase.

2 Tuning for simple models

2.1 Linear Regression

2.1.1 Ordinary Least Square Linear Regression (OLS)

We model the price of cubic zirconia to be a linear function of continuous features. We can perform OLS to get our function:

$$f\ \boldsymbol{x_i}\ = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

We wish to learn the weights that minimize the Root Mean Square Error (RMSE) over the entire training data:

$$\boldsymbol{w}^* = \text{argmin}_{\boldsymbol{w}} \sqrt{\frac{1}{M}\sum_{i=1}^{M} y_i - w_0 - w_1 x_{i1} - w_2 x_{i2} - \cdots - w_n x_{in}\ ^2}$$

Using this method, we obtain a function with the following result:

```
RMSE: 1010.6101376835952
R2 Score: 0.9331235065116853
```

2.1.2 Ridge Regression

We also try Ridge Regression for our problem.

The optimized objective of Ridge Regression:

$$\boldsymbol{w}^* = \text{argmin}_{\boldsymbol{w}} \left(\sum_{i=1}^{M} y_i - \boldsymbol{A_i w}\ ^2 + \alpha \sum_{i=1}^{M} w_i^2 \right)$$

We use GridSearchCV for some alpha values that we have chosen to find the α parameter value that performs best on the train set when we use cross validation, with k-fold = 5.

```
param_grid = {'alpha': [0, 0.05, 0.1, 0.2, 0.5, 1, 1.5, 2]}
grid_search = GridSearchCV(Ridge(), param_grid, scoring='neg_mean_squared_error', cv=5)
grid_search.fit(X_train, y_train)
print(grid_search.best_params_)
print(grid_search.cv_results_)
predictions = grid_search.predict(X_val)
print('RMSE:', np.sqrt(np.mean((predictions - y_val)**2)))
```

The best α value that the method returns is 0.1.

We will show how the loss values of all the α parameters when using Ridge Regression to predict validation set changing.

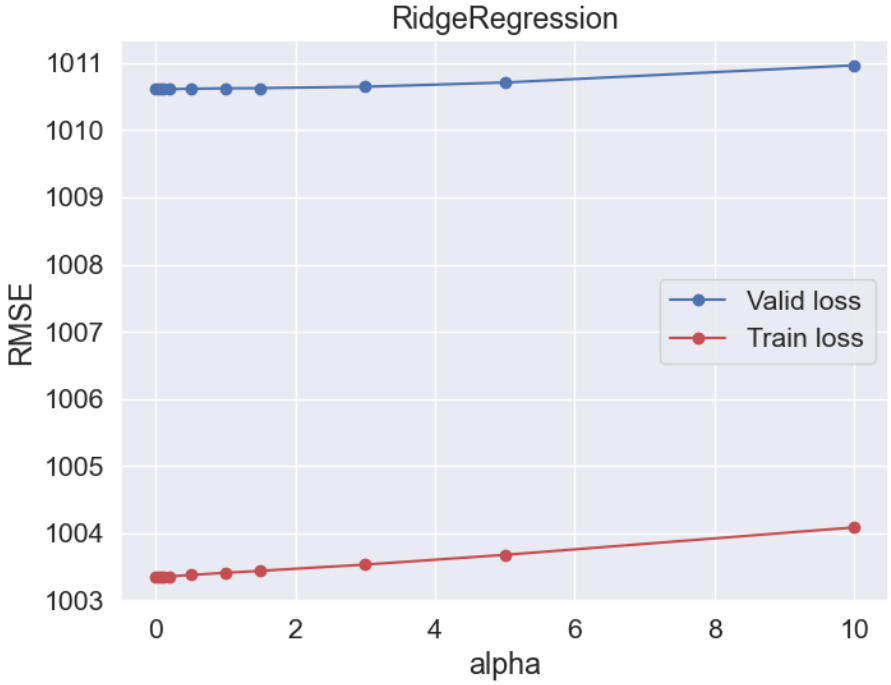


Figure 5.1. Loss values on train set and valid set for different values of α

Though GridSearchCV returns the alpha value that performs the best is 0.1, when working on validation set, it returns that the Ordinary Least Square Linear Regression or Ridge Regression when $\alpha = 0$ has better performance on validation set. We will denote it in the conclusion for all regression algorithms.

2.1.3 Lasso Regression

The optimized objective of Lasso Regression:

Similar to the Ridge Regression, first we will use GridSearchCV to find the α parameter that cross validation in the train set would have best average result.

We will use search on the parameters [0.05, 0.1, 0.2, 0.5, 1, 1.5, 2], and got 0.05 to have the best results

To check the efficiency of all the parameters, we will show the plot of parameters and the loss values of the Lasso Regression with α value equals to the parameter when fits the X_train to predict the y_train and predict the y_valid based on X_valid.

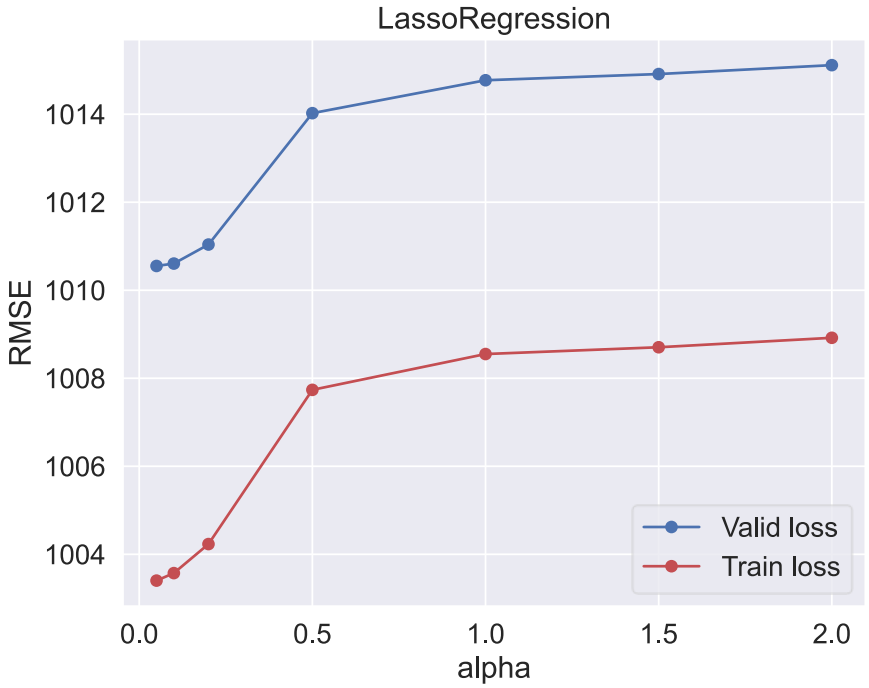


Figure 5.2: Loss values on train set and valid set for different values of α

The plot also shows that $\alpha = 0.05$ returns the lowest loss value for validation.

2.1.4 ElasticNet

ElasticNet is the combination of Ridge Regression and Lasso Regression, which idea is to put the penalty terms of both Ridge Regression and Lasso Regression together.

The minimized objective of ElasticNet:

$$\boldsymbol{w} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \ \|y - \boldsymbol{X}\boldsymbol{w}\|^2 + \lambda_2\|\boldsymbol{w}\|^2 + \lambda_1\|\boldsymbol{w}\|_1$$

Using GridSearchCV and re-evaluate using plot with training loss and validation loss, it shows that the validation loss and training loss don't have lot of differences between as those of Ridge and Lasso Regression.

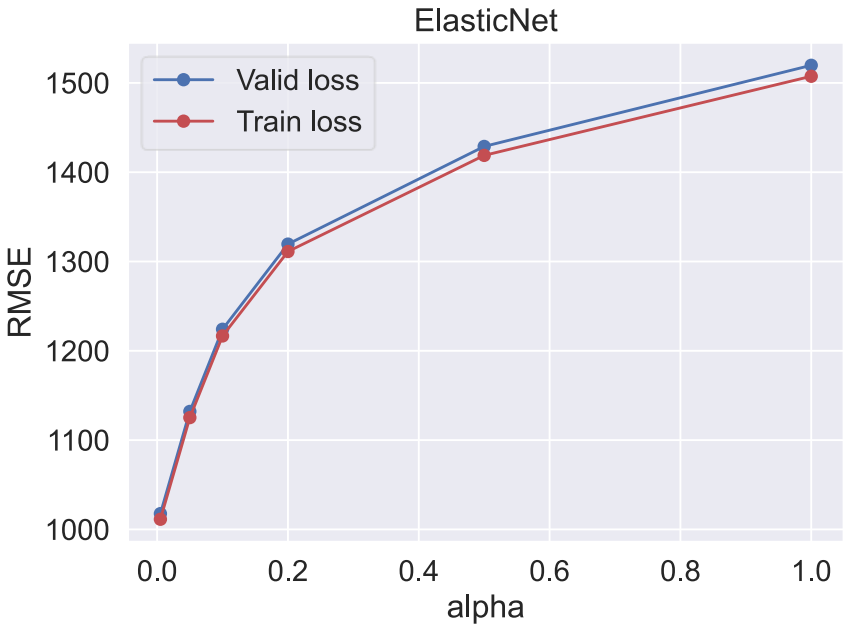


Figure 5.3: Loss values on train set and valid set for different values of α

2.1.5 Conclusion for all Linear Regression algorithms

For our problem, we notice that the data seems not to have a form likely linear distribution. So for the Linear Regression algorithms which can only create a linear generalized function, they are not reliable to make prediction and their loss is quite high. As we see from above, Ridge and Lasso seem to have better results when $\alpha = 0$, or in other words, the penalty terms are removed, making no difference from the Ordinary Least Square Linear Regression. For the ElasticNet, which aim is to combine Ridge and Lasso to gain better results, seems to be worse because it cannot make both their penalty term parameters λ_1 and λ_2 equal 0.

2.2 KNeighbors Regression

The KNeighbors Regression uses k nearest values of a new data point, which is from the training data, that are closest to the data point with respect to a distance metric (we will use Euclidean distance). The value of the new data point is then predicted by taking the average of all target values of the k nearest neighbors.

So, the parameter that we need to focus on is the number of k nearest neighbors. Implementing using sklearn library, the number of k will be denoted by **n_neighbor**.

We will use GridSearchCV to find the **n_neighbor** that that cross validation on the training set returns the best average result. Using grid-search for **n_neighbor** on [1, 2, 5, 10, 15, 20, 50], the best is 10.

Showing the plot of parameters and their validation loss and training loss, with **n_neighbor** on [1, 2, 5, 7, 10, 13, 14, 15, 20, 30] to have a close look to the parameters that are near to 10, we actually got the **n_neighbor** = 13 to have the best performance.

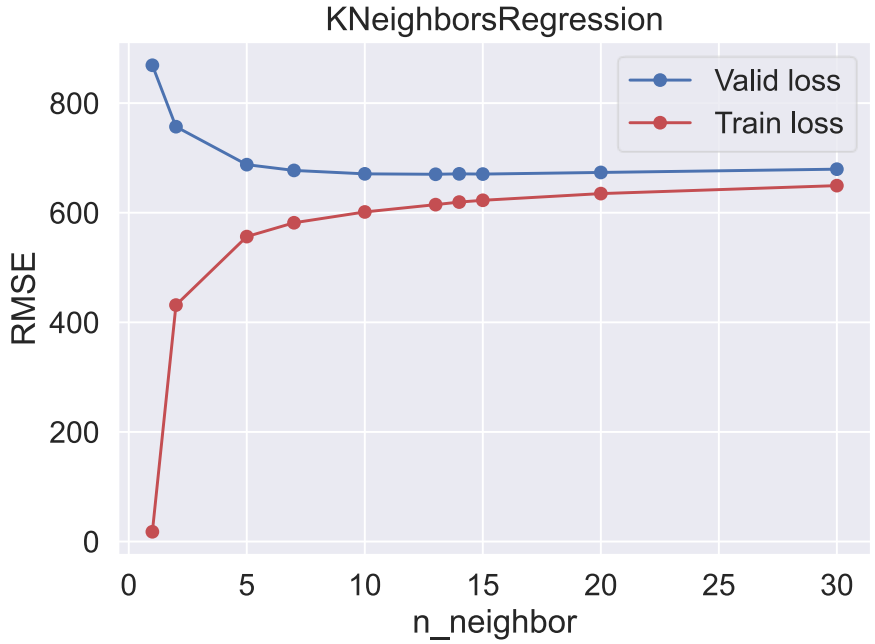


Figure 5.4: Loss values on train set and valid set for different values of n_neighbors

2.3 Decision Tree Regression

The DecisionTreeRegression will build a learning tree by splitting training data to smaller and smaller subsets. When we make predictions by using it, we will go from the initial node, root node following the condition of each branch to the leaf node that contains our predicted values. This method is good to learn the training set with high accuracy, but it has the problem of overfitting. So, we will need to choose its parameter very carefully to avoid this.

The DecisionTreeRegression has a crucial parameter, **max_depth**, which represents the maximum depth of the regression tree. With a higher value of **max_depth**, the tree will learn more closely to the data from the training set. With a too low value of **max_depth**, the regression tree learns nothing.

Firstly, we use GridSearchCV to find **max_depth** for the set [1, 2, 5, 10, 15, 20, 50] to find the parameter that makes the regression has the best average results of cross validation. The parameter **max_depth** = 10 returns the best result.

We will check it again using plot, showing the training loss and validation loss of each value of parameter when working with DecisionTreeRegression.

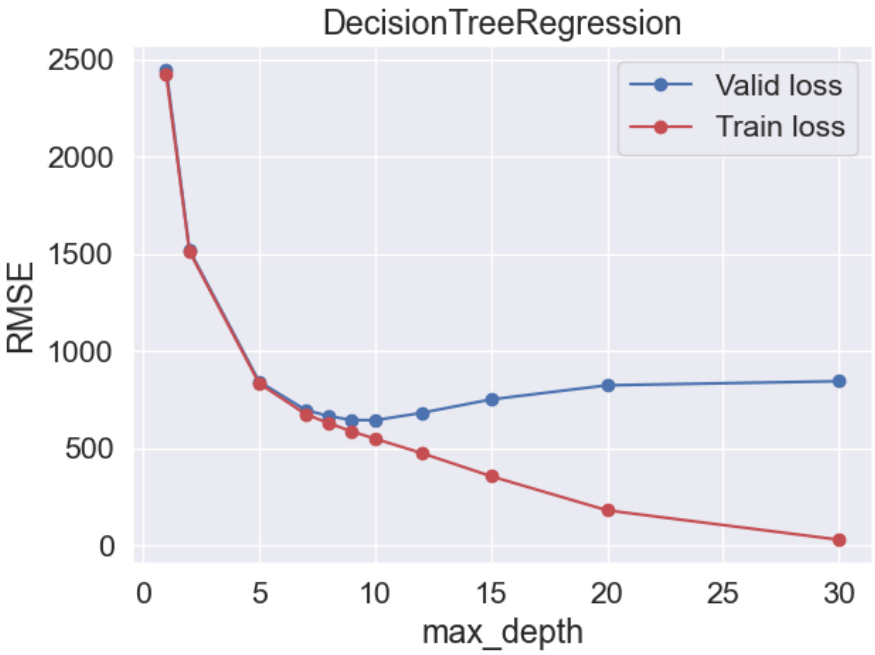


Figure 5.5: Loss values on train set and valid set for different values of ***max_depth***

The plot also shows that ***max_depth*** = 10 has the best performance.

3 Fine-Tune for Ensemble Learning models

We will go for tuning 3 Ensemble Learning models: Gradient Boosting Regressor, Random Forrest Regressor and XGBoost (Extreme Gradient Boosting Regressor).

First we use Randomized Search to minimize searching space for hyper-parameter and then using Grid Search to finding best parameter for some parameters we know the most, specific tune hyper-parameter.

3.1 GradientBoosting

We experiment on 2 familiar parameters which are **learning_rate** and **n_estimator**

After doing 2 Searches, we will check it again using plot, showing the training loss and validation loss of each value of parameter **n_estimator** with **learning_rate** = 0.3 or 0.4

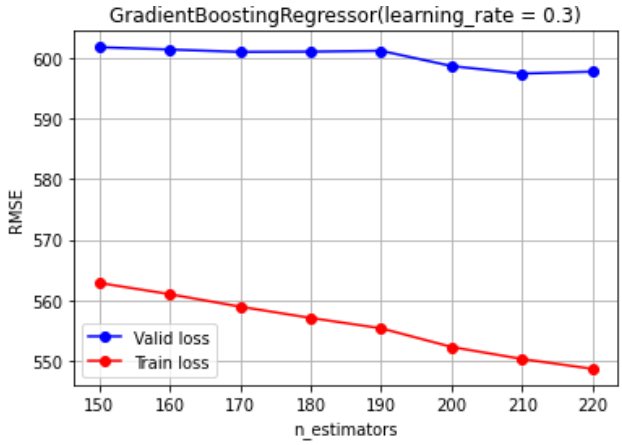


Figure 5.6: Loss values on train set and valid set for different values of **n_estimators** with **learning_rate** = 0.3

The plot also shows that **n_estimators** = 210 has the best performance for **learning_rate** = 0.3

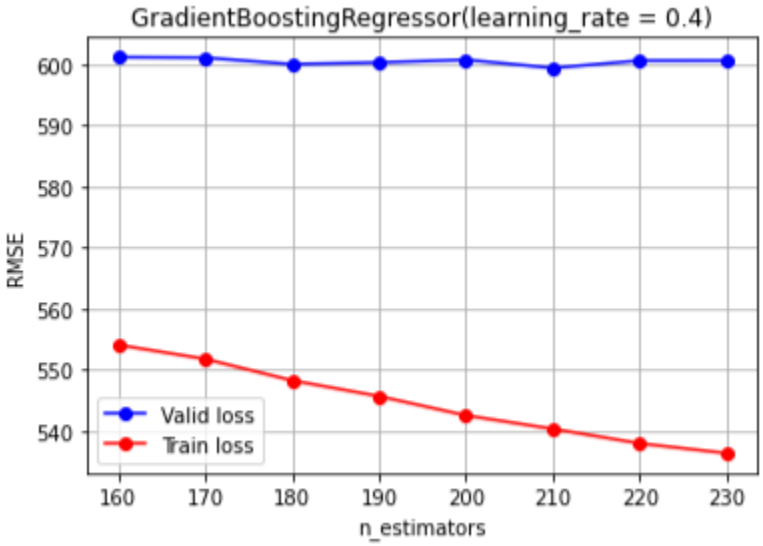


Figure 5.7: Loss values on train set and valid set for different values of **n_estimators** with **learning_rate** = 0.4

The plot also shows that **n_estimators** = 210 has the best performance for **learning_rate** = 0.4

Fusion 2 plots, we choose best parameters **n_estimators** = 210 and **learning_rate** = 0.3

3.2 RandomForest

We experiment on 2 familiar parameters which are **max_features** and **n_estimators**

After doing 2 Searches, we will check it again using plot, showing the training loss and validation loss of each value of parameter **n_estimators** with **max_features**= 'sqrt' or None

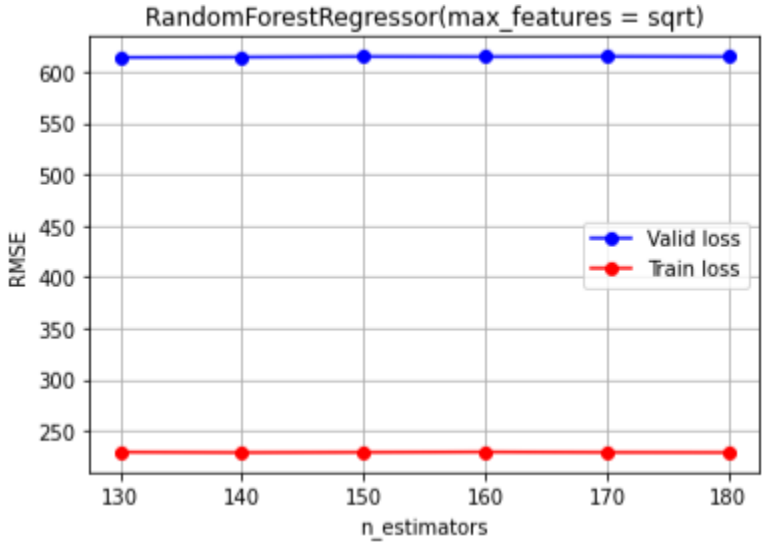


Figure 5.8: Loss values on train set and valid set for different values of **n_estimators** with **max_features** = sqrt

The plot also shows that **n_estimators** = 130 has the best performance for **max_features** = sqrt

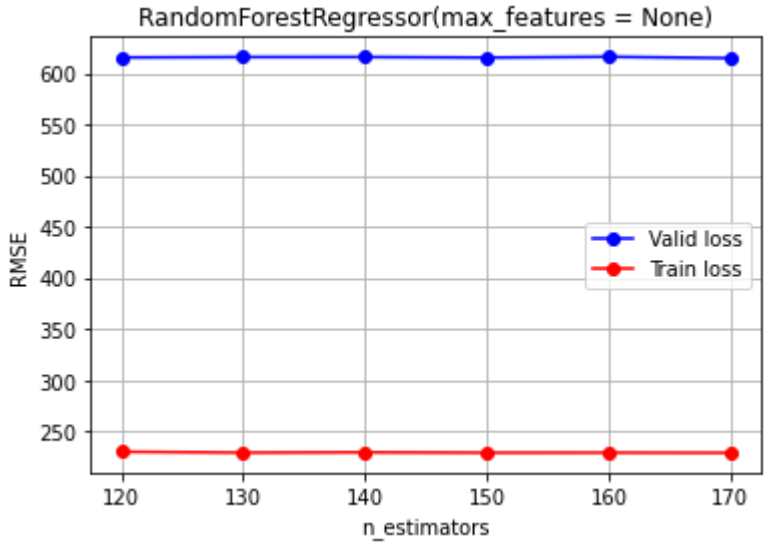


Figure 5.9: Loss values on train set and valid set for different values of **n_estimators** with **max_features** = None

The plot also shows that **n_estimators** = 170 has the best performance for **max_features** = None

Fusion 2 plots, we choose best parameters **n_estimators** = 10 and **max_features** = sqrt

3.3 XGBoosting

We experiment on 2 familiar parameters which are **learning_rate** and **n_estimator**

After doing 2 Searches, we will check it again using plot, showing the training loss and validation loss of each value of parameter **n_estimator** with **learning_rate** = 0.05

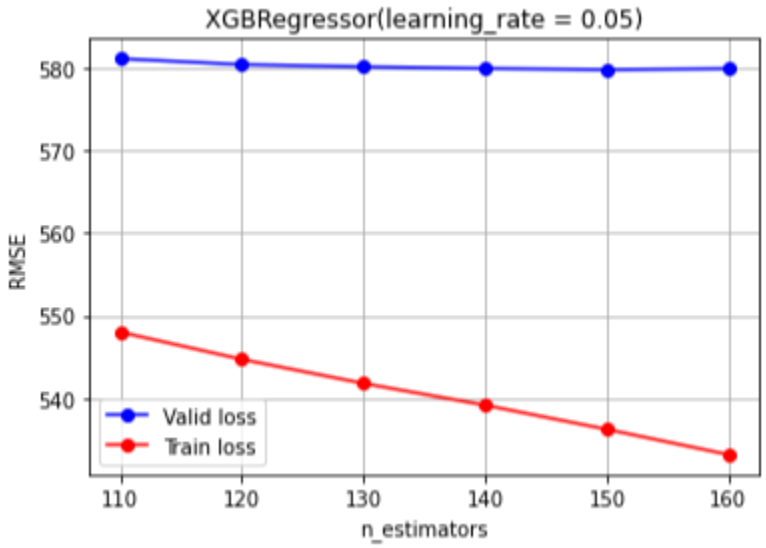
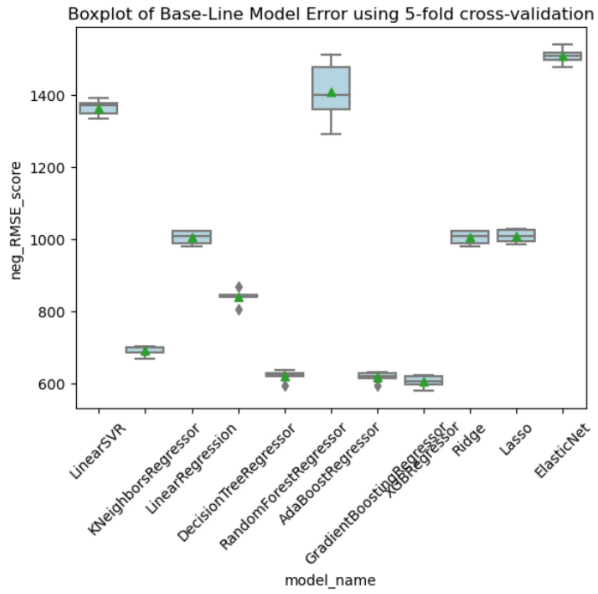


Figure 5.10: Loss values on train set and valid set for different values of **n_estimators** with **learning_rate** = 0.05

The plot also shows that **n_estimators** = 150 has the best performance for **learning_rate** = 0.05

4 Model Comparison

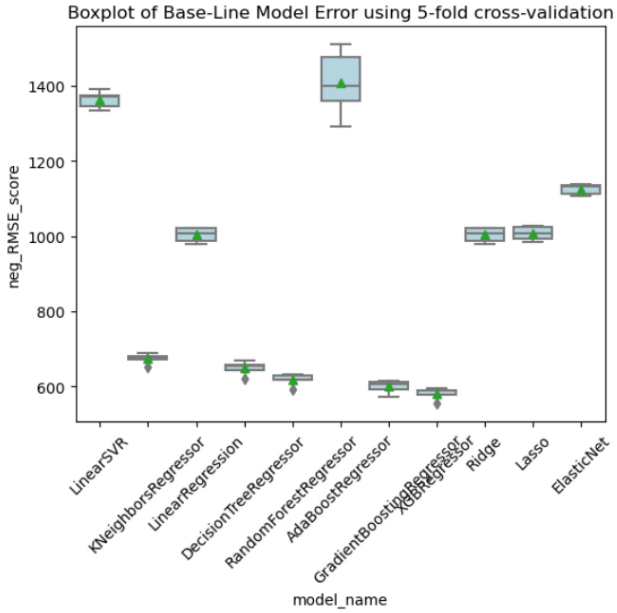
We use 5-fold cross validation for model assessment. To compare between all learning methods, we calculate the root mean square error (RMSE) of each one. Initially, we compare the results before tunning some of our learning methods and get the table below:



model_name	Mean	Standard Deviation
ElasticNet	1506.906838	22.861094
AdaBoostRegressor	1406.959134	87.786349
LinearSVR	1363.051870	23.026913
Lasso	1006.910365	18.886002
Ridge	1003.697447	19.221565
LinearRegression	1003.629774	19.229036
DecisionTreeRegressor	839.142958	21.865984
KNeighborsRegressor	690.495496	14.128458
RandomForestRegressor	619.468620	15.918911
GradientBoostingRegressor	617.157650	14.686335
XGBRegressor	603.471304	17.325353

Result before tunning

After applying tuning for simple models and also fine-tuning for ensemble learning methods, we reassess the scoring of all methods and get the results below:



model_name	Mean	Standard Deviation
AdaBoostRegressor	1406.959134	87.786349
LinearSVR	1363.051870	23.026913
ElasticNet	1124.822213	13.982078
Lasso	1006.787951	18.823909
Ridge	1003.630130	19.232641
LinearRegression	1003.629774	19.229036
KNeighborsRegressor	674.388220	14.131043
DecisionTreeRegressor	649.142652	17.892841
RandomForestRegressor	618.187231	15.148785
GradientBoostingRegressor	600.430182	17.232717
XGBRegressor	581.758428	16.591634

Result after tunning

As we can see, the AdaBoostRegressor and LinearSVR are by far the models with the worst accuracy now (scoring 1406.96 and 1363.05 respectively). All LinearRegression models still

perform fairly bad (around 1000 scoring on RMSE). KNeighborsRegressor, DecisionTreeRegressor, RandomForestRegressor, GradientBoostingRegressor, XGBRegressor all reach significantly better accuracy (scoring around 600 – 700). The latter three, RandomForestRegressor at 618.19, GradientBoostingRegressor at 600.43 and XGBRegressor at 581.76, are chosen to be the best 3 methods with lowest error scoring.

VI Model Submission

The data set for our project is taken from a competition on Kaggle. Here is the link to more information on the competition Regression with a Tabular Gemstone Price Dataset | Kaggle. The competition also provides a test data set that we can run our prediction on. In this section we are going to run our models on the test data set and submit our results to the competition to see how they perform.

First look at the test dataset.

id	carat	cut	color	clarity	depth	table	x	y	z
193573	0.35	Ideal	D	VS2	62.3	56.0	4.51	4.54	2.82
193574	0.77	Very Good	F	SI2	62.8	56.0	5.83	5.87	3.68
193575	0.71	Ideal	I	VS2	61.9	53.0	5.77	5.74	3.55
193576	0.33	Ideal	G	VVS2	61.6	55.0	4.44	4.42	2.73
193577	1.20	Very Good	I	VS2	62.7	56.0	6.75	6.79	4.24
...	...	...	...	...	...	...	...	...	...
322618	0.72	Ideal	D	VVS2	62.0	56.0	5.75	5.78	3.57
322619	0.70	Premium	D	SI1	59.6	62.0	5.77	5.74	3.43
322620	1.01	Premium	G	VVS2	62.3	58.0	6.44	6.41	4.01
322621	1.35	Ideal	D	I1	62.0	56.0	7.05	7.08	4.38
322622	1.07	Premium	H	SI2	62.6	60.0	6.49	6.45	4.06

129050 rows × 9 columns

Because ***depth*** made by divided by ***diameter*** so it cannot be zero but in the test dataset has some zeros so we need to be careful, take the original ***depth***.

We will add our derived attributes ***diameter***, ***depth_adj*** and ***weight-cat*** to the test data. We will need to be carefull of null entries.

```
test_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 129050 entries, 193573 to 322622
Data columns (total 12 columns):
 #   Column      Non-Null Count   Dtype
---  ------      --------------   -----
 0   carat       129050 non-null  float64
 1   cut         129050 non-null  object
 2   color       129050 non-null  object
 3   clarity     129050 non-null  object
 4   depth       129050 non-null  float64
 5   table       129050 non-null  float64
 6   x           129050 non-null  float64
 7   y           129050 non-null  float64
 8   z           129050 non-null  float64
 9   diameter    129050 non-null  float64
 10  depth_adj   129047 non-null  float64
 11  weight_cat  129050 non-null  category
dtypes: category(1), float64(8), object(3)
memory usage: 11.9+ MB
```

There are some null entries in the ***depth__adj*** feature, we replace these with the original ***depth*** from the data.

Below we show an example of our models running on the first 10 instances of the test dataset.

	id	carat	cut	color	clarity	depth	table	x	y	z	Predicted price
0	193573	0.35	Ideal	D	VS2	62.3	56.0	4.51	4.54	2.82	863.245130
1	193574	0.77	Very Good	F	SI2	62.8	56.0	5.83	5.87	3.68	2481.920780
2	193575	0.71	Ideal	I	VS2	61.9	53.0	5.77	5.74	3.55	2354.564436
3	193576	0.33	Ideal	G	VVS2	61.6	55.0	4.44	4.42	2.73	851.522958
4	193577	1.20	Very Good	I	VS2	62.7	56.0	6.75	6.79	4.24	5782.712319
5	193578	0.30	Premium	G	VS1	62.7	58.0	4.30	4.25	2.68	658.948056
6	193579	1.52	Premium	G	VS2	62.9	59.0	7.31	7.26	4.59	12326.697583
7	193580	0.72	Very Good	E	VS2	59.2	59.0	5.85	5.88	3.46	2988.271382
8	193581	2.01	Ideal	I	VS2	65.0	57.0	7.84	7.91	5.15	15123.118662
9	193582	0.60	Premium	D	SI1	60.0	58.0	5.44	5.50	3.29	1917.764980

My Major Voting method

Our stategy is to take 3 best methods and to fuse it by mathematics below to give our final prediction (the ideal is based on bagging of ensemble).

We have min(RMSE) : meaning of best model and max(RMSE) and middle(RMSE)

We take percentage error of RMSE by: p_error = (RMSE - min(RMSE))/min(RMSE)*100 : unit (%)

$$p_{error} = \frac{RMSE - \min RMSE}{\min RMSE}.100\ \%$$

Then based on our best model prediction:

If prediction of best model > prediction(p) of model: $p + p \cdot p_{error}$

If prediction of best model < prediction(p) of model: $p - p \cdot p_{error}$

Finally we take the average of 3 models prediction.

Our result

No tuning	• xgb: (586.17, 592.46): 587.43 • Ensemble bagging xgb and rf: (585.43, 591.5): 562.64
Tuning by hand	• xgb: (576.09,583.62): 577.60 • Ensemble bagging xgb and rf: (582.01, 588.16): 583.24 • myMVmethod(use 2 models xgb and gb: (578.72, 587.42): 580.46 • myMVmethod(use 3 models): (582.31, 589.18): 583.68
Optuna	• xgb: (575.59, 583.07): 577.09 • Ensemble bagging xgb and gb: (574.1, 581.36): 575.55 • myMVmethod(use 2 models xgb and gb):(574.07, 581.25): 575.51
h2o	• h2o_partial: (575.33, 579.44): 576.15 • h2o_full: (575.35, 582.45): 576.77 • h2o_pre: (574.99, 581.79): 576.35 • h2o_pre2(not applying sqrt for *carat*): (574.45, 580.87): 575.73

Note:

(private score, public score): 80%*private score + 20%*public score

Private score is calculated by 80% of data test while public is other 20%

- xgb: XGBoosting, rf: RandomForest, gb: GradientBoosting
- myMVmethod: My Major Voting method
- Ensemble bagging: calculate by taking average

- h2o_partial: using 80% of data for training
- h2o_full: using 100% of data for training
- h2o_pre: preprocessing before putting to h2o_full
- h2o_pre2: remove step taking square root for ***carat*** and continue like h2o_pre

Bonus: Using Tools autoML

1 AutoML help choosing algorithms

We will choose h2o for this part because of its integration with Jupyter notebook

```
print(h2o.estimators.xgboost.H2OXGBoostEstimator.available())
```

```
Cannot build an XGBoost model - no backend found.
False
```

Unfortunately, windows and macOS do not support XGB so our h2o cannot execute it, we need linux for that. It may our model prediction a little bit weaker because XGB is quite a strong model

We fix the time search is 180s = 3 minutes for h2o

After fit in model, we will have some results like this

model_id	rmse	mse	mae	rmsle	mean_residual_deviance
StackedEnsemble_AllModels_3_AutoML_8_20230703_213009	572.389	327629	292.974	0.106634	327629
StackedEnsemble_AllModels_2_AutoML_8_20230703_213009	572.412	327655	292.976	0.106635	327655
StackedEnsemble_AllModels_1_AutoML_8_20230703_213009	572.663	327942	292.853	0.106386	327942
StackedEnsemble_BestOfFamily_3_AutoML_8_20230703_213009	573.054	328391	294.591	0.108016	328391
GBM_2_AutoML_8_20230703_213009	574.11	329602	294.4	0.107462	329602
StackedEnsemble_BestOfFamily_2_AutoML_8_20230703_213009	574.187	329691	294.553	0.107587	329691
GBM_3_AutoML_8_20230703_213009	574.471	330017	293.383	0.106326	330017
GBM_5_AutoML_8_20230703_213009	574.856	330459	296.242	0.108991	330459
GBM_1_AutoML_8_20230703_213009	576.016	331795	292.253	0.104701	331795
StackedEnsemble_BestOfFamily_1_AutoML_8_20230703_213009	576.128	331923	292.699	0.105096	331923

We also will try preprocessing before putting to h2o to see that if results get better. For the source code we need to make sure to transform between *panda* and *h2o* form.

	carat	depth_adj	table	x	y	z	diameter	cut	color	clarity	weight_cat	price
0	1.518495	0.444636	0.402574	1.401939	1.461007	1.482353	1.431587	3.0	2.0	4.0	2.0	13619.0
1	2.286157	0.536302	0.402574	2.114283	2.177890	2.212002	2.146311	2.0	6.0	6.0	2.0	13387.0
2	-0.066541	-0.472033	-0.118584	-0.022749	0.009094	-0.049908	-0.006907	4.0	3.0	3.0	1.0	2772.0
3	-1.150539	-0.105366	-0.639742	-1.203978	-1.188734	-1.202753	-1.201129	4.0	3.0	3.0	0.0	666.0
4	1.802352	0.627969	0.923732	1.744585	1.715092	1.803399	1.730143	3.0	3.0	4.0	2.0	14453.0
...	...	...	...	...	...	...	...	...	...	...	...	...
193551	-1.186178	-0.655367	-0.639742	-1.231029	-1.206883	-1.261125	-1.219224	4.0	0.0	2.0	0.0	1130.0
193552	-0.066541	-1.480368	0.402574	0.031353	0.045392	-0.093687	0.038328	3.0	3.0	2.0	1.0	2874.0
193553	0.004427	1.177970	-0.118584	0.004302	0.027243	0.125207	0.020234	2.0	2.0	5.0	1.0	3036.0
193554	-1.080893	0.994637	-1.160900	-1.140859	-1.116138	-1.056823	-1.128752	2.0	0.0	5.0	0.0	681.0
193555	-0.042719	-0.930367	3.529523	0.013319	-0.009054	-0.079094	0.002140	1.0	1.0	6.0	1.0	2258.0

2 AutoML help tuning hyper-parameter

Helping with 3 best model so far we have tuning parameter so we can see if we can make it better

We will choose **Optuna** for this part because of its integration with Jupyter notebook

Using Meta-heuristic approaches

That being said, as data volume gets larger and models get more complex, the cost of randomly training a set of hyperparameters without any information increases tremendously, and these advanced algorithms greatly outperform traditional methods

```
study = optuna.create_study(direction='minimize')
study.optimize(objective_xgb, n_trials=20, show_progress_bar=True)
```

We fix the number of trials is 20, the result behind will look like this

```
best_params_xgb = study.best_trial.params
best_params_xgb
```

```
{'tree_method': 'gpu_hist',
 'reg_lambda': 2.553793950847241,
 'colsample_bytree': 0.9,
 'colsample_bylevel': 0.8,
 'subsample': 0.9,
 'learning_rate': 0.04480245593299995,
 'n_estimators': 150,
 'max_depth': 9,
 'min_child_weight': 3,
 'grow_policy': 'lossguide'}
```

Note that we only tune 2 of 3 best model is GradientBoosting and XGBoosting, ignore RandomForest with Optuna because time running issue, also we use HistGradientBoosting

replace for GradientBoosting because HistGradientBoosting is the improvement of GradientBoosting, especially in time cause we fit in all the datasets

Something specially that we test with our best model which is **myMVmethod(use 2 models xgb and gb)**.

In our experiment, we have try add esti_volume or remove x, y or remove weight_cat. Fusion 2 of 3 or 3 of it but just make the result worse so decide to stop. Also we change the order of encoder of cut to make categorical go from best to worst by 0 to ... And weight_cat also too check in 'Experiment.ipynb'. Best result have so far is 574.

VII REFERENCES

[1] *Regression with a tabular gemstone price dataset | Kaggle*. (2023). Kaggle: Your Machine Learning and Data Science Community. https://www.kaggle.com/competitions/playground-series-s3e8

[2] *XGBoost documentation — xgboost 1.7.1 documentation*. (2022). XGBoost Documentation — xgboost 1.7.1 documentation. https://xgboost.readthedocs.io/en/stable/

[3] *Scikit-learn*. (2023). scikit-learn: machine learning in Python — scikit-learn 0.16.1 documentation. Retrieved July 5, 2023, from https://scikit-learn.org/stable/

[4] *H2O AutoML: Automatic machine learning — H2O 3.42.0.1 documentation*. (2023). H2O.ai Documentation. https://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html

[5] *Diamond properties and characteristics*. (2023, June 26). Diamond Buzz. https://diamondbuzz.blog/diamond-properties-and-characteristics/

[6] Chirag Choudhary. (2023, March 4). 💎 *Plotly EDA | Feature engineering | Optuna* 💎. Kaggle: Your Machine Learning and Data Science Community. https://www.kaggle.com/code/ch124uec/plotly-eda-feature-engineering-optuna

[7] Joaquin Vanschoren. (2023). *Lecture 6. Data preprocessing*. Machine Learning for Engineers. https://ml-course.github.io/master/notebooks/06%20-%20Data%20Preprocessing.html

[8] *A consumer's guide to gem grading*. (2022, October 17). International Gem Society. https://www.gemsociety.org/article/a-consumers-guide-to-gem-grading/

[9] Gurucharan, K. M. (2020, July 18). *Machine learning basics: Decision tree regression*. Medium. https://towardsdatascience.com/machine-learning-basics-decision-tree-regression-1d73ea003fda

[10] Singh, A. (2023, March 13). *KNN algorithm: Introduction to k-nearest neighbors algorithm for regression*. Analytics Vidhya.

https://www.analyticsvidhya.com/blog/2018/08/k-nearest-neighbor-introduction-regression-python/

[11] Shenmare, K., & Vaidya, D. (2023). *Elastic Net*. WallStreetMojo. https://www.wallstreetmojo.com/elastic-net/

[12] *SciKit learn SVR runs very long*. (2017). Stack Overflow. Retrieved July 6, 2023, from https://stackoverflow.com/questions/47460201/scikit-learn-svr-runs-very-long

[13] *SVM using scikit learn runs endlessly and never completes execution*. (2014). Data Science Stack Exchange. Retrieved July 6, 2023, from https://datascience.stackexchange.com/questions/989/svm-using-scikit-learn-runs-endlessly-and-never-completes-execution

VIII CONTRIBUTIONS

Contributors	Main Responsibility
Nguyen Viet Minh 20214917	**Ensemble Learning, Model Result**
Phan Duc Hung 20214903	**Linear Regression, KNeighbors Regression, Decision Tree Regression**
Truong Gia Bach 20210087	**Data Explanation, Exploratory Data Analysis**
Tran Duong Chinh 20210122	**Data-Preprocessing, Report, PowerPoint**

Note: The table only shows main responsibility, all contributors had responsibility to help others finish the work, together.

For more details of our work, please refer to our source code: **GemstoneProject.ipynb**

and experiment from: **Experiment.ipynb**

In Min-KiD/GemstonePredictionPrice (github.com)