STU44003

Random Forest Regression Analysis

Lecturer – Dr. Myra O’Regan

Gabriel Ogundipe

1. Introduction

This report aims to predict potential prices of diamonds, using the Random forest regression technique. The dataset contains information about different attributes of diamonds and the price each diamond was sold for. We will conduct explanatory data analysis to gain a deeper understanding of the data, before delving into the techniques used to predict diamond prices.

To complete this project, the following were used:

- **Python**: A high level, general programming language, with a good reputation for machine learning due to its libraries and speed.
- **Pandas**: A Python library used for the manipulation and analysis of data.
- **Numpy:** A Python library for scientific computing.
- **Seaborn**: A Python library for data visualization.
- **Scikit-Learn**: An open-source machine learning library for Python.

2. Diamond Data

The variables contained in the diamond dataset and a description of their characteristics is shown in the table below.

Variable Name	Description	Scale
Carat	Carat weight of the diamond	
Cut	Cut quality of the diamond	Fair, Good, Very Good, Premium, Ideal
Color	The color of the diamond	D – J, best to worst respectively
Clarity	How obvious inclusions are within the diamond	In order from best to worst. FL,IF, VVS1, VVS2, VS1, VS2, SI1, SI2, I1, I2, I3
Depth	The height of a diamond, measured from the culet to the	43-79

	table, divided by its average girdle diameter	
Table	The width of the diamond's table expressed as a percentage of its average diameter	43-95
Price	The price of the diamond in US dollars.	$326 - 18823
x	Diamond length in mm	0 -1 0.74
y	Diamond width in mm	0 – 58.9
z	Diamond depth in mm	0 – 3.18

3. Data Pre-processing

i. Missing and Incorrect Data

- Before analysis a dataset, it is important to check whether that dataset has any missing values as that can affect the results of our analysis.
- Using Pandas' "isnull()" function, it was confirmed that no missing values in the data were present.

ii. Data Cleaning

- The first column "nid", identification number of the diamonds was dropped as we already have an index of rows in our dataset.
- 20 rows were dropped that had minimum values measurement values of 0 for x, y, and z. It doesn't make sense to have either length, width or height as a measurement 0.
- These rows only accounted for only 0.00037% of the dataset, as total number of diamonds was 53,940 - so dropping them is a good option.

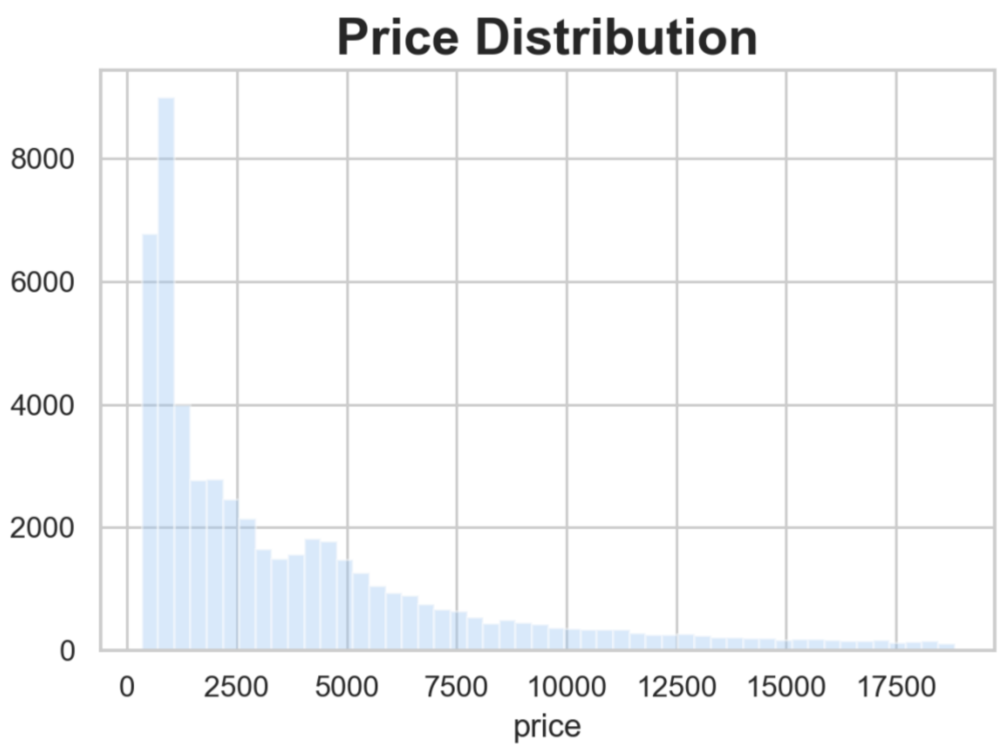
4. Preliminary Analysis

In this section we will analyse the results of the initial analysis of the data. This will mainly consist of graphs of the independent variables plotted against the target variable. Summary statistics of different variables will also be explored.

iii. A quick snapshot of our data

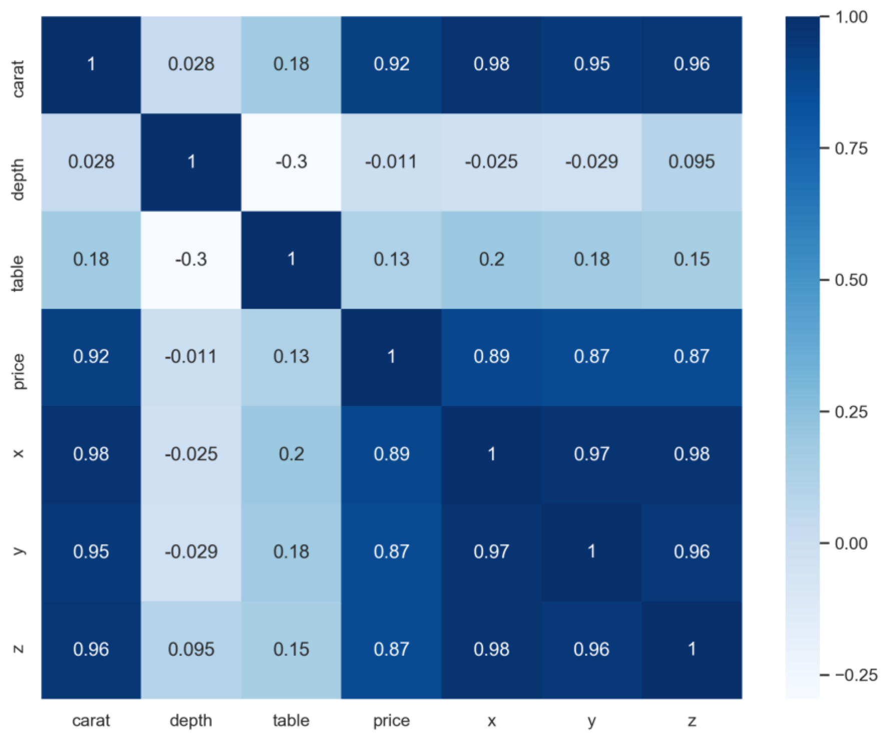
Here we have a snapshot of our data using Pythons describe function. One thing to notice is that the variables ‘cut’, ‘color’, and ‘clarity’ are not included in the descriptive statistics below. This is because Python automatically excludes categorical variables from its describe function, therefore this is something we must pay attention to when we begin to implement our regression technique.

	carat	depth	table	price	x	y	z
count	53920.00	53920.00	53920.00	53920.00	53920.00	53920.00	53920.00
mean	0.80	61.75	57.46	3930.99	5.73	5.73	3.54
std	0.47	1.43	2.23	3987.28	1.12	1.14	0.70
min	0.20	43.00	43.00	326.00	3.73	3.68	1.07
25%	0.40	61.00	56.00	949.00	4.71	4.72	2.91
50%	0.70	61.80	57.00	2401.00	5.70	5.71	3.53
75%	1.04	62.50	59.00	5323.25	6.54	6.54	4.04
max	5.01	79.00	95.00	18823.00	10.74	58.90	31.80



Here we have a distribution plot of the diamond price variable. We can see that the majority of prices are between the $326 - $5000, the tail of the distribution begins to curtail at this point.

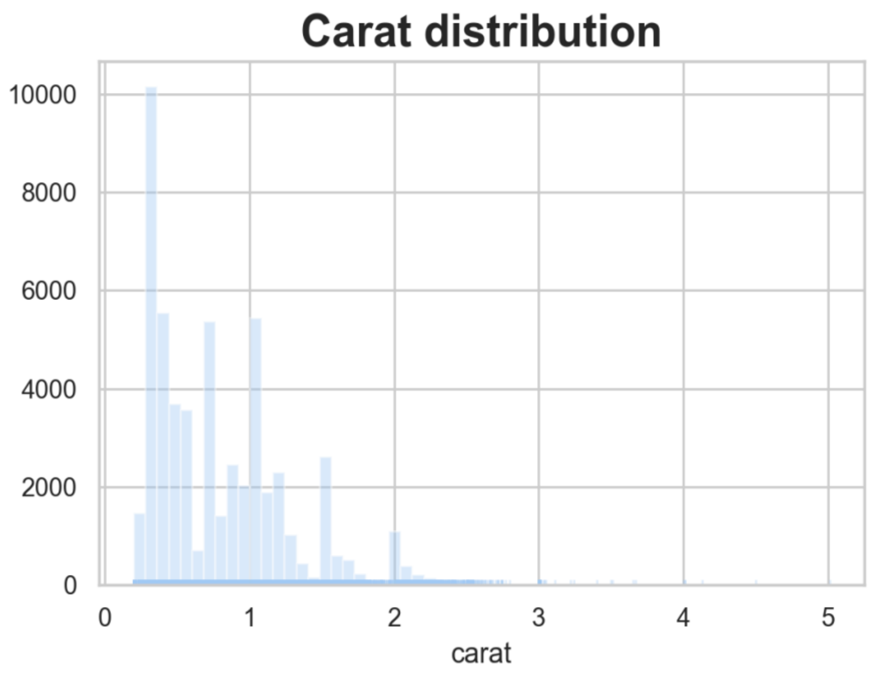
Intuitively, this makes sense economically – as prices increases, demand naturally decreases as fewer people may be able to afford higher prices.



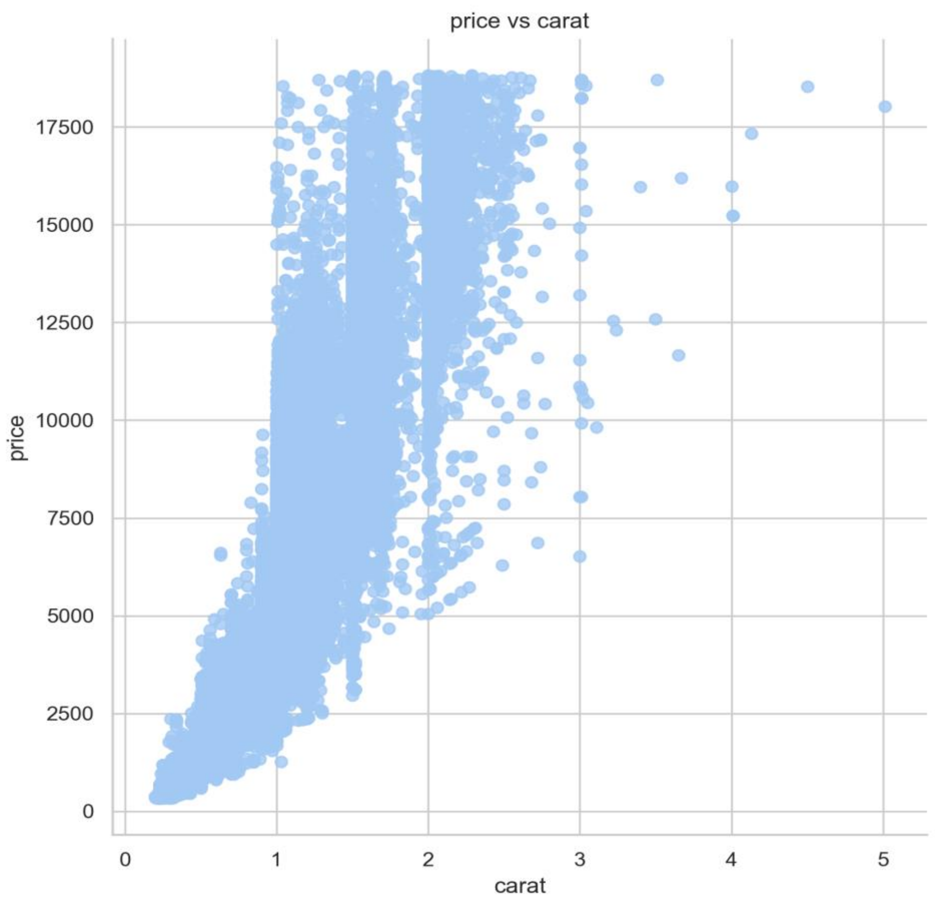
From this correlation heatmap, we can immediately observe that the price of a diamond is highly correlated with its weight (carat) and its dimensions. The carat of a diamond has the most significant impact on price, with a correlation value of 0.92. The price is also highly correlated with dimensions x, y and z of a diamond.

iv. Exploring Price and Carat

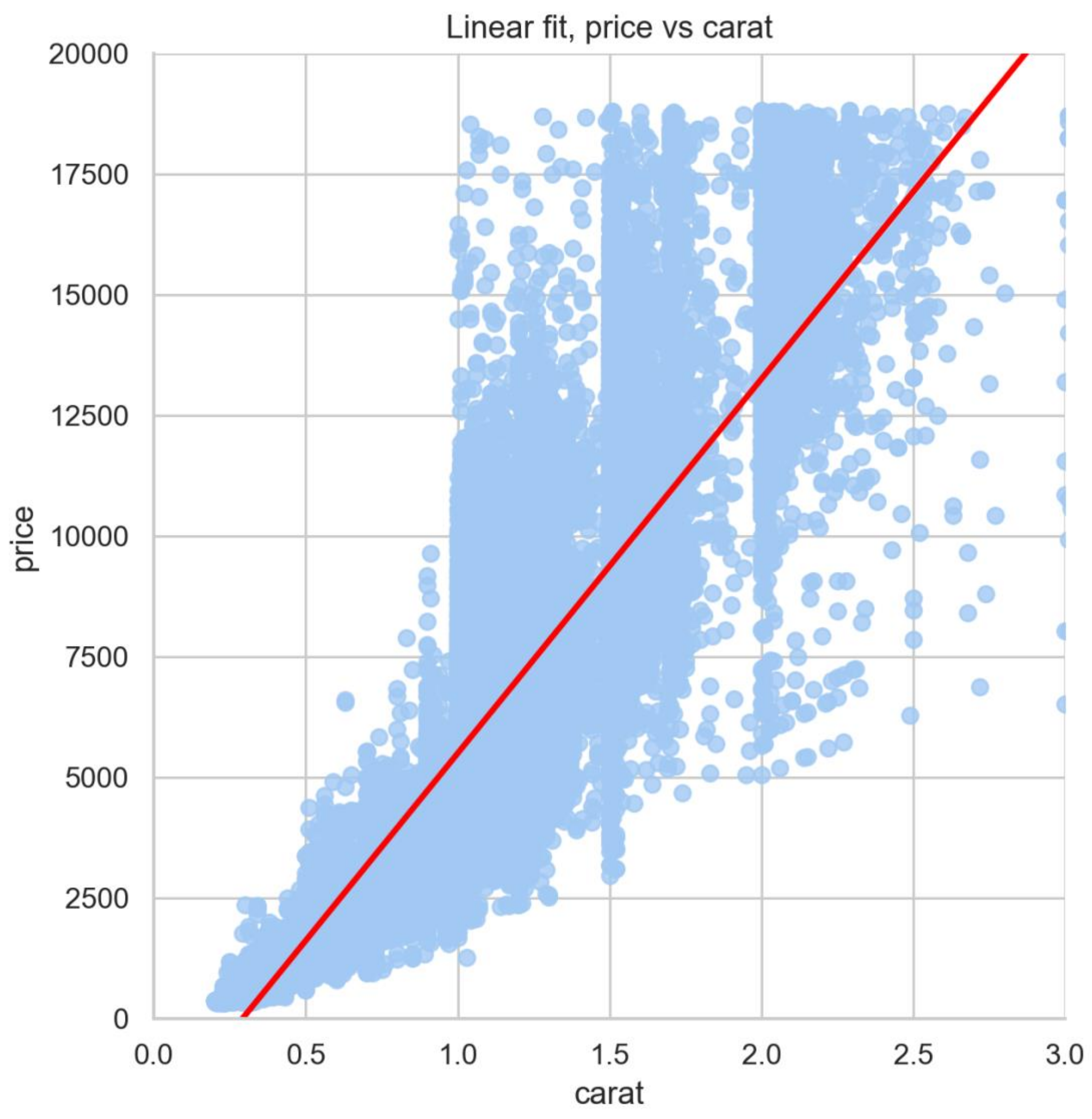
Taking a step back and approaching the effect of a diamonds carat on price, one would think that the weight (carat) of a diamond would directly influence the price. A distribution plot of the carat variable can be observed



above. This distribution plot illustrates that the distribution of a diamond's carat is relatively similar to the distribution of the price. It can be seen that the majority of carats in this data set are below 1 carat. This follows the same principle that the higher the number of a diamond's carat, the higher the price the diamond will be. To further analyse this relationship, a scatter plot between diamond price and carat can be seen below.

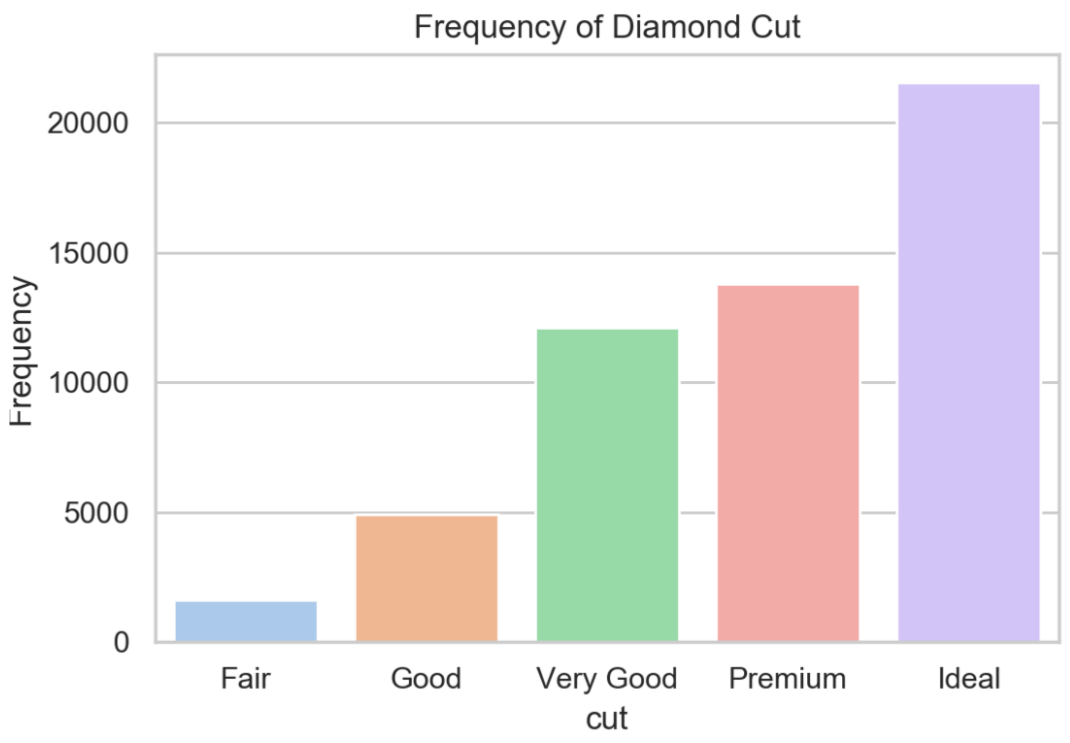


From this scatter plot of price vs carat, it is clear that a relationship exists between both variables. Carat has a significant relationship with the price of a diamond, which validates are initial thoughts. Diamonds with a carat of less than 2 appear to follow a linear progression, and as the carat size increases, the data points appear to be varying. As a general rule, the more a diamond weighs, the pricier it will become. However not all diamonds are created equal, a higher carat value does not necessarily mean it will be more expensive than a diamond with a lower value as this can be due to other factors like colour, cut etc.



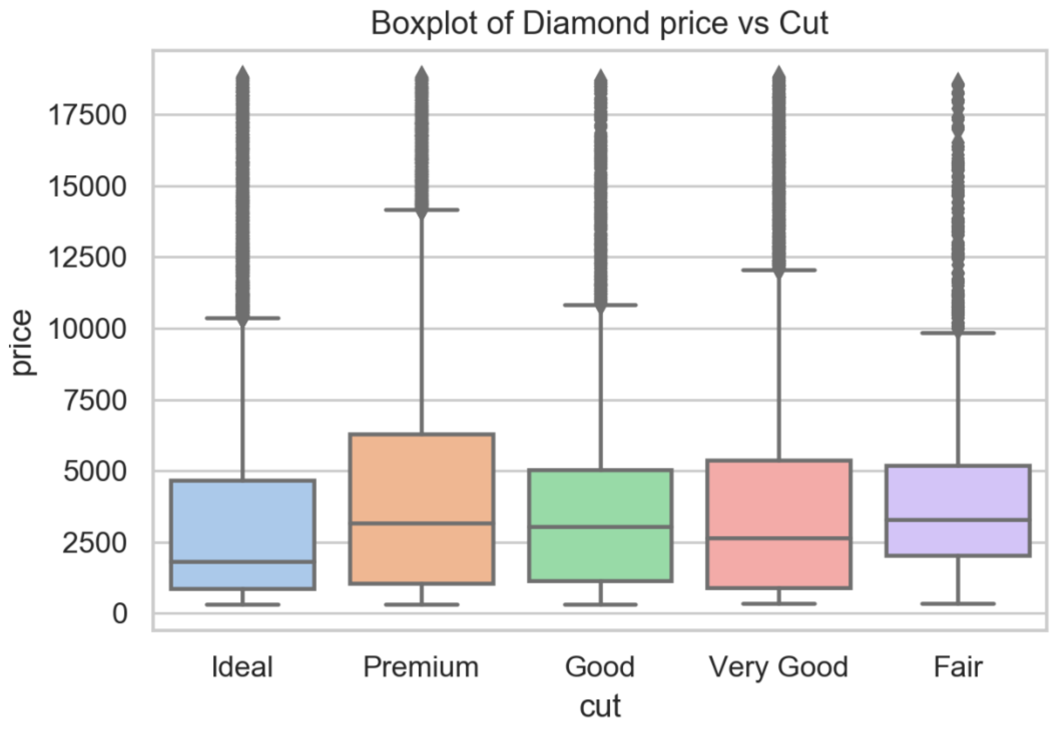
Looking at the graph above, it confirms that there is a linear fit to diamond price and diamond carat, especially when the carat size is under 3. However, this linear relationship fluctuates as carat size increases, but overall still fits relatively well in a linear relationship. Thus, we can make an interim conclusion to say that a relationship exists between carat and price, and that the diamond carat size will be key in determining the price of a diamond.

v. Exploring Price and Cut

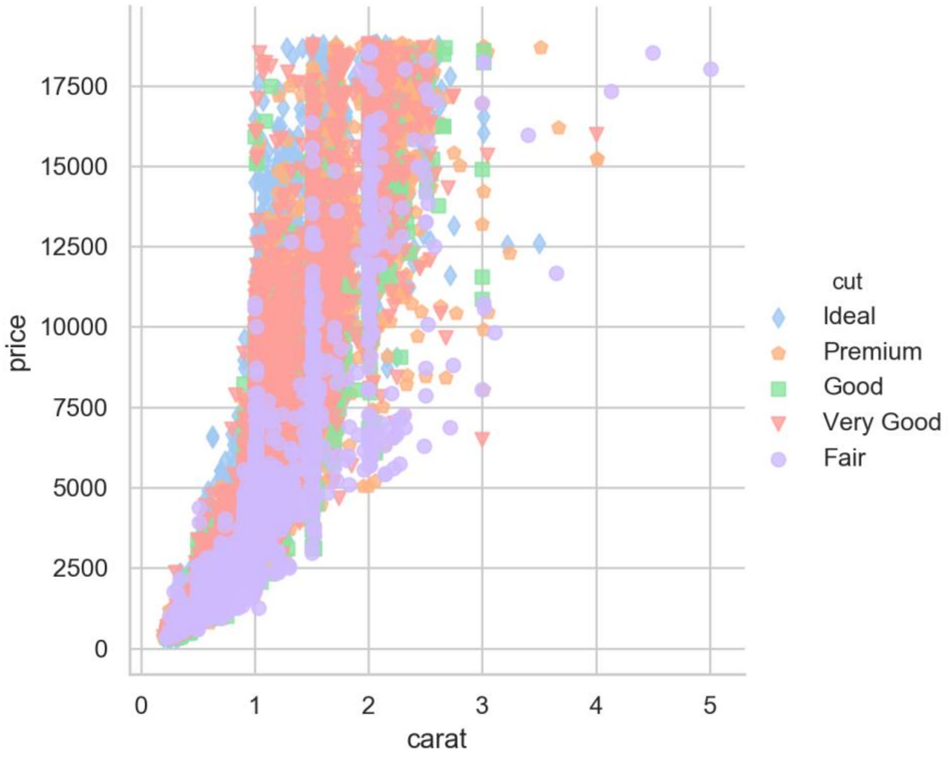


We can see above from the frequency plot generated from each different kind of cut, that the majority of diamonds have a cut quality equal to “Very Good” or Higher. For obvious reasons, people would be more inclined to obtain higher quality diamond cuts! But how do the diamond cuts relate to price?

The boxplot below shows us that premium diamond cuts are the most expensive, followed by cuts that are very good. In terms of the median value of the cut of diamonds, they are all relatively similar except for the ideal category. This may be due to the fact that the Ideal cuts are the highest counting cuts in the dataset. All in all, looking at cut solely in comparison with the price variable does not give us a lot of insight into how it affects the price. After this fact, I then decided to look at the cut of a diamond in relation to price and carat, as we have already established that carat is an important variable.

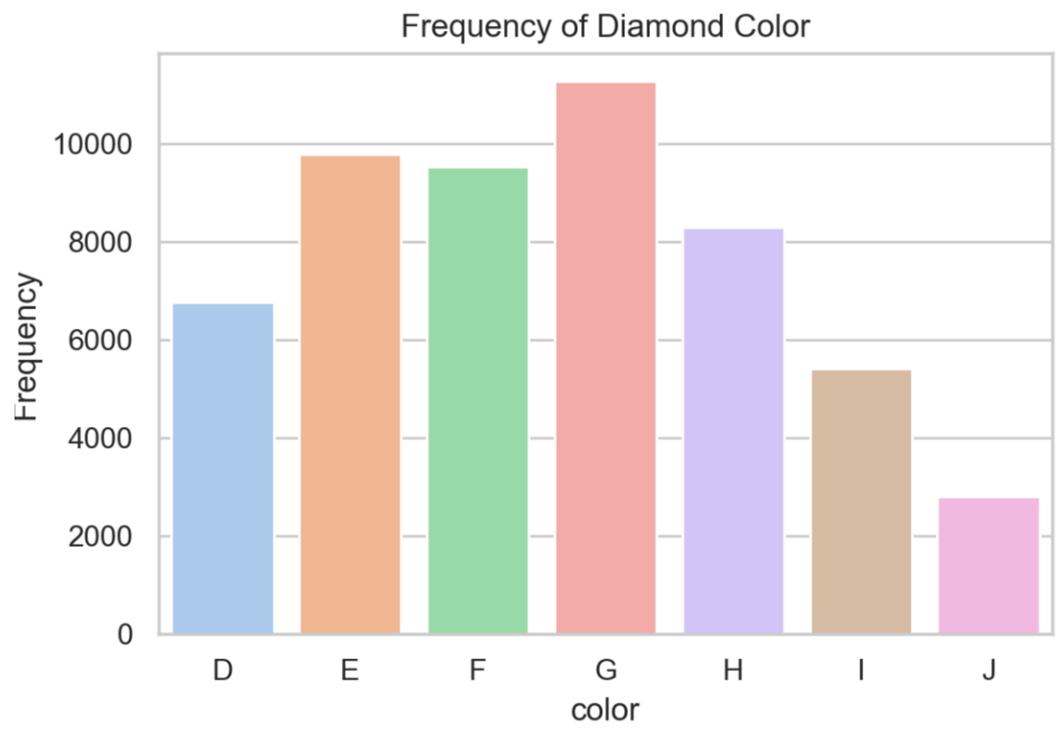


From this graph, it can be seen that fair cut diamonds weigh the most, but they are not the most expensive diamonds – this is a really interesting observation as one would automatically think the more a diamond weighs, the more costly it will become. I also noticed that ideal cuts were not found in the high carat region, perhaps this is showing that finding an ideal cut diamond with a high carat size is rare. Ultimately, we can assume that cut has an effect on diamond prices.

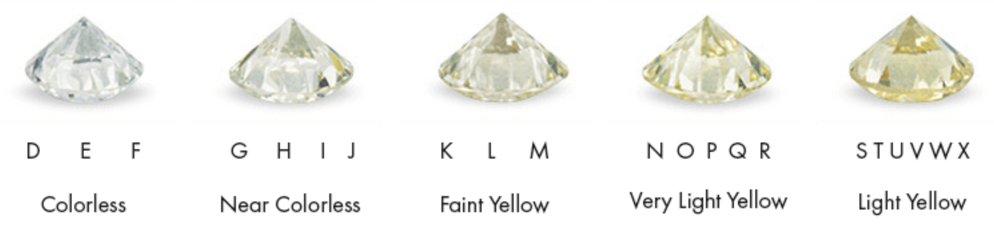


vi. Price vs Color

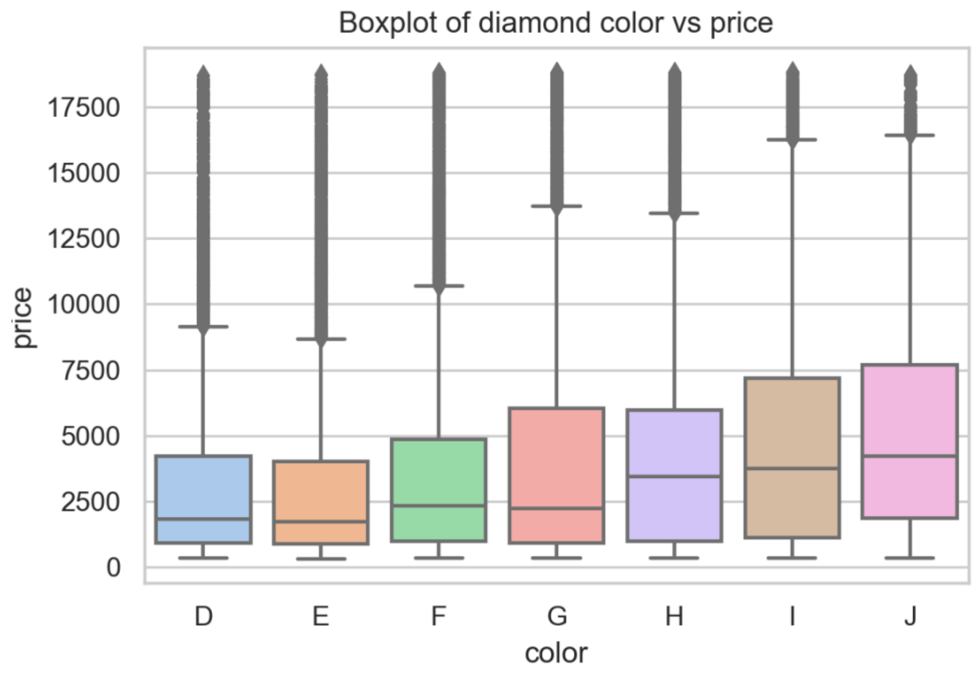
We will first look the frequency of each type of colour that is in the dataset.



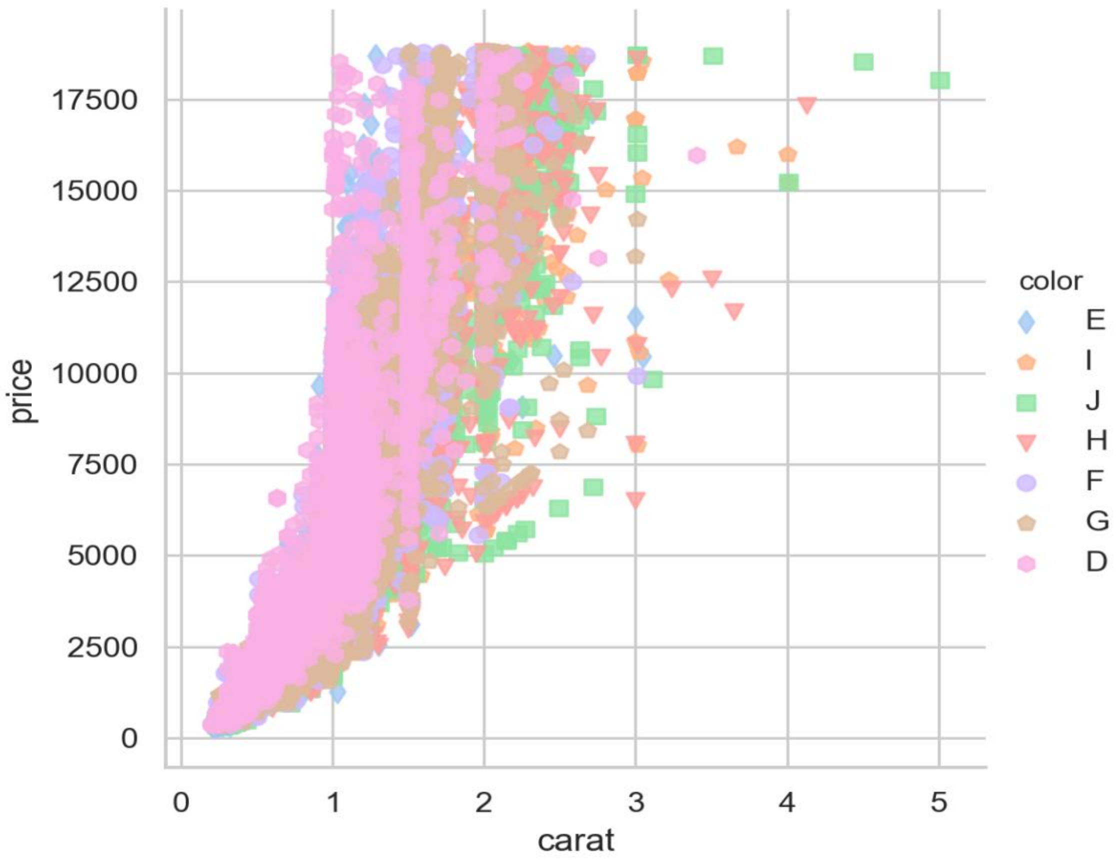
From the above plot we can observe that the majority of diamonds fall into the higher better color quality bracket. Looking at the image below, we can gain a deeper understanding into how diamond colour is graded.



It makes sense that the majority of diamonds fall in the color range of D – H, as these colors are described as ‘colorless’ and ‘near colorless’. Of course, it makes logical sense that most people would like their diamond color to fall into this range too as nobody wants a diamond that is faint in yellow. Now, how does colour affect diamond prices? In the boxplot below, we can see that the higher prices range from colors G-J. I found this fact intriguing because as observed, these colours are at the lower end in the scale of colors in our dataset. Although granted, in the grand scheme of things the colors G-J are still in fact high quality in the overall scale seen above. I then decided to explore diamond color and price along with diamond carat.



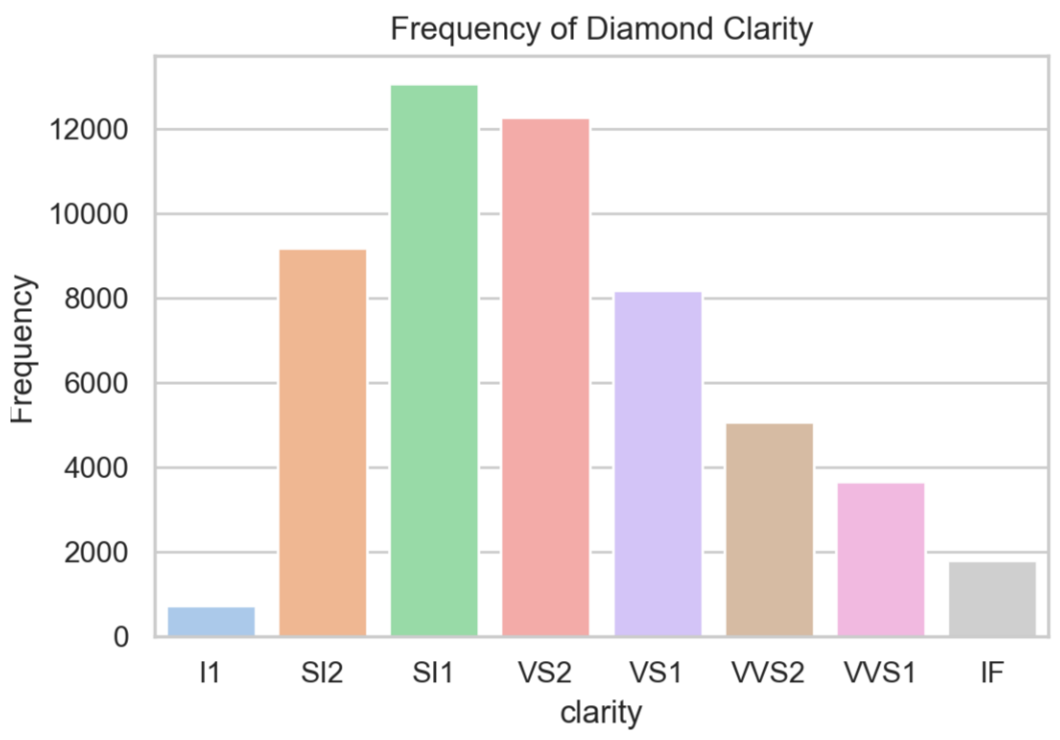
In the plot below, we can see that colors H, I, G are the heaviest diamonds. Although they are not the best color, their carat size is a big influence. It makes sense that these diamonds are overall more expensive, it can be seen from the graph that diamond J has one of the higher prince points of



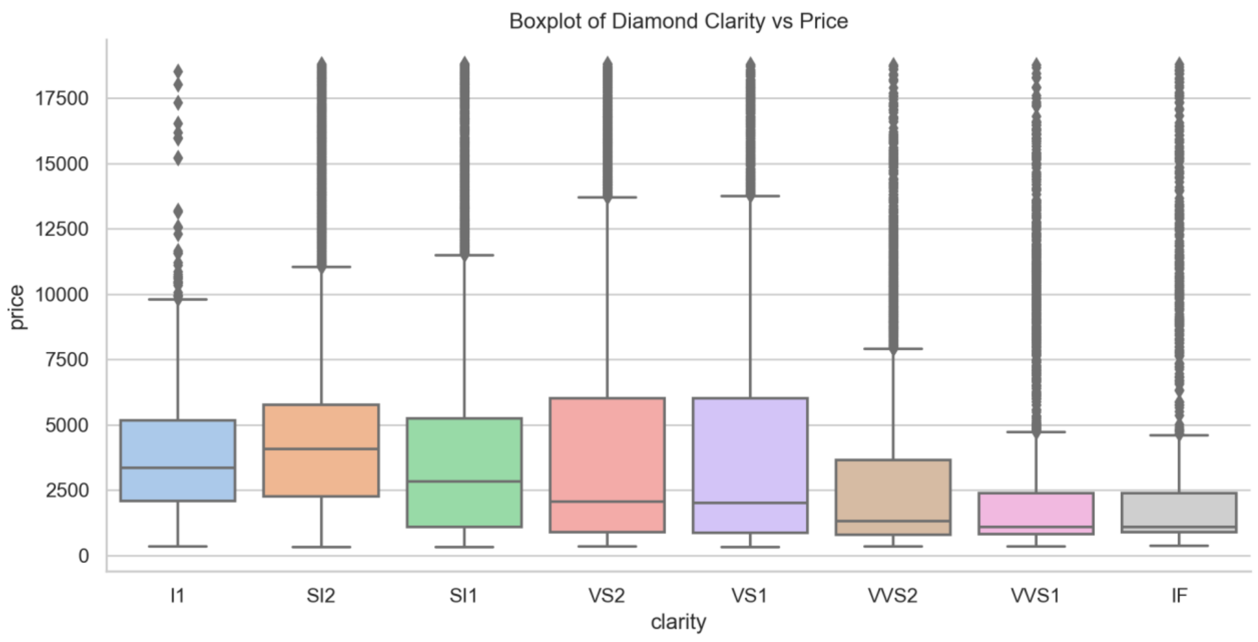
diamonds, with a big carat size. To me, the difference in ‘colorless’ to ‘near colorless’ is minuscule and I would most likely pick a larger carat J over a smaller carat F for example. At this stage, I think it’s clear that the carat size is a clear influence on diamond prices!

vii. Price vs clarity

Again, we will begin by looking at the frequency of the different types of diamond clarities in the data.

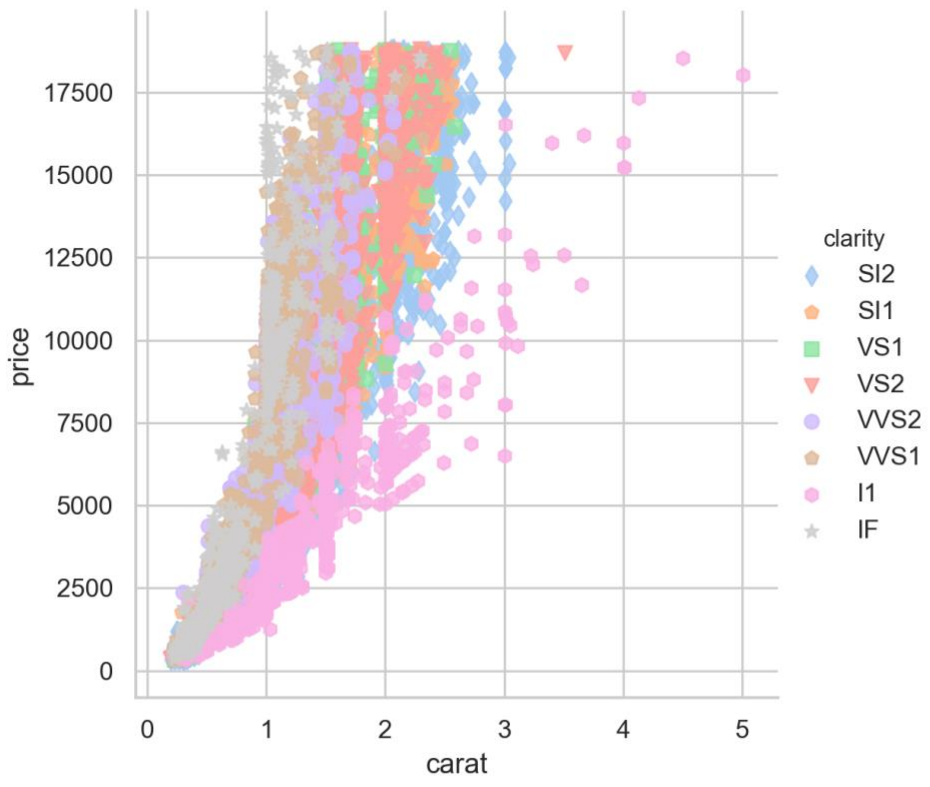


It obvious from this plot that the majority of diamonds fall within the lower to mid-scale clarity range. Could this be because there is more demand for diamonds of this type? Perhaps. We have seen from previous analysis that the majority of diamonds fall below the 3 carat range, so it’s a possibility that a lot of diamonds in this range have the clarity levels in the mid-scale range as seen above. Now, we can examine how clarity levels interacts with the price of a diamond.



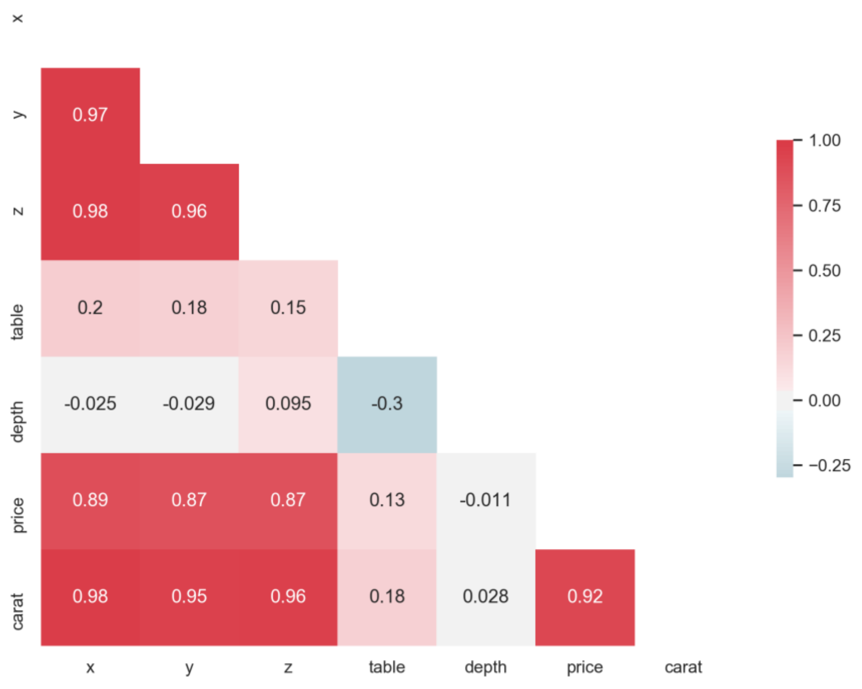
Straight away from the graph, we can observe that the diamonds with the highest quality clarity have a lower median price than diamonds with a lower quality of clarity. Naturally, this may quiz the non-diamond expert as one would expect that diamonds of a higher clarity quality would cost more. Upon this observation, I then decided to explore the relationship between diamond clarity, price and carat.

Here we see an intriguing plot. It appears that the higher quality clarity diamonds have a lower carat value than the lower quality clarity diamonds. From this representation of the relationship between diamond clarity, price and carat – one can conclude that although the clarity of a diamond may indicate the price, the carat of a diamond holds greater weight in predicting the price of a diamond.

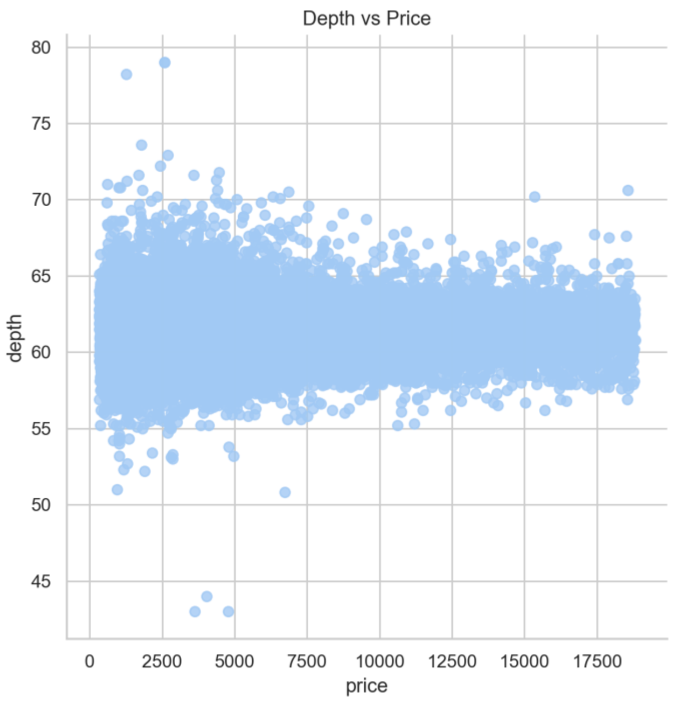


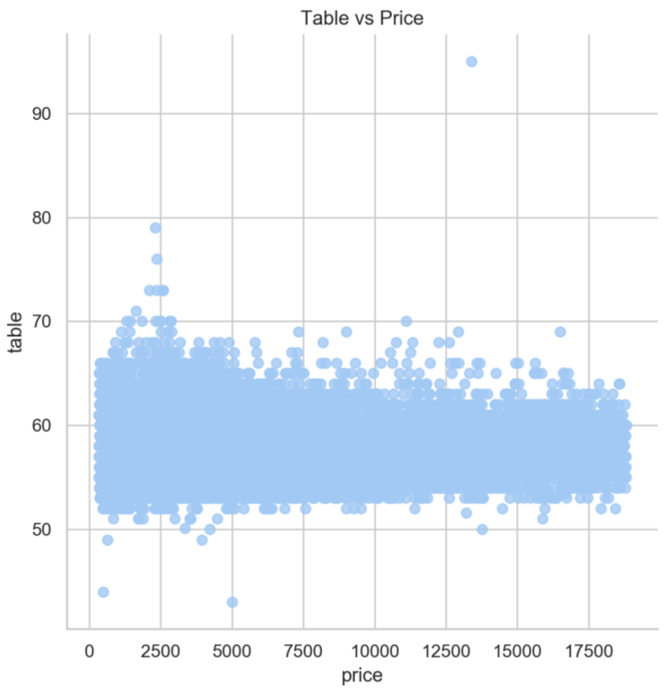
viii. Price vs Dimensions

To analyse the impact of a diamonds dimensions on its price, we will look at the remaining variables which are depth, table, x, y, and z.



From the above correlation heatmap, we can see that x, y, and z are extremely correlated with the carat of a diamond. Inherently, this makes sense as because x, y and z are used to describe the volume of a diamond, clearly, they would be associated with the weight of a diamond also. Not only are they highly correlated with the carat, they are also very correlated with the price of a diamond. From this information, to me it makes sense to exclude x, y, and z when we run our model to predict diamond prices – as carat is highly correlated with the dimensional variables. We do not want an instance where we have multiple cariables explained by one variable. When we look at look at the table and depth of a diamond, we see an unexpected low correlation to price. Since the table and depth are calculated from the dimensional values of a diamond, I was confused as to why it was not highly correlated with the price. Furthermore, examining the graphs below also showed how little correlated it was to the price of a diamond.





This then led me to conduct further research on the table and depth of a diamond. The table and depth of a diamond determine how much light passes through a diamond to give it that 'shiny factor'. There is no one size fits all for the depth and table, however, it can't be too small, neither can it be too large otherwise it greatly alters the appearance of a diamond. From this analysis, it makes sense that table and depth are inversely correlated to the price of a diamond. Perhaps the reason why they appear not to influence price is because they are derived from other variables.

ix. Recap of preliminary analysis of data

- It seems like carat holds the most weight in predicting the price of a diamond
- Cut and colour also play a role in predicting, the price – although it is only when you analyse both of these variables with the carat variable that you see the true interaction between variables
- The dimension variables x, y, and z are highly correlated with price. This makes a lot of sense as these variables are naturally highly correlated with carat size which in turn influences the price as we have already determined.
- I therefore decided that since carat is a direct consequence of the dimension variables, it made sense not to include variables x, y and z in our model as these variables can be explained through the carat variable.

- At initial glance, the table and depth of a diamond did not seem to influence the price, but upon further research, it emerged that these variables can highly alter the appearance of a diamond.

5. Random Forest Regression

x. Theory behind the Random Forest Regression

Random forests are an ensemble learning method that is used for classification and regression methods, in this case for predicting the diamond price, the random forest regressor method will be used. How a random forest works is that It builds multiple decision trees, and it combines them. This is what "bagging" is referred to. The key idea behind "bagging" is that the calculation and combination of all these models will help increase the reliability, stability and accuracy of a model. Individually, predictions made my each decision tree may not be accurate, but when combined together, the prediction will be closer to the true value on average. When a node is split, each tree looks for the best features in the dataset. Each tree also only has access to a random sample of training data. This results in an increase in diversity in a random forest, so it is better at dealing with over fitting compared to a single decision tree.

xi. Parameters of a model

There are multiple parameters to consider when you are fitting a random forest model. The aim of any aspiring data analyst or scientist is to find the perfect balance between bias-variance trade off. An ideal situation would be to minimise both. When a model has a bias, it over simplifies the model. Consequently, performance on training and test data will be poor. When a model has high variance, it does not perform well at the generalisation of new data. They perform well on the training data but do not perform well with test data.

When a model is overfitting, it contains more parameters that can be justified by the data. I.E – when a model captures too much noise in the data. Overfitted models have high variance and low bias. When a model is underfitting, it indicates that a model has cannot adequately capture the underlying structure of the data. Ideally, you would find a balance that is optimal that doesn't overfit, or underfit the data. When tuning parameters (manipulating parameters to change the behaviour of a model) it is important that this is kept in mind.

For our Random Forest Regressor model, parameters tuned that affect it are:

- N_estimators: This represents the number of trees in the random forest. A model with higher number of trees results in better predictions, however this comes at a price! A model with many trees will require more computational power and will therefore take longer to run. The aim of tuning this parameter is to find the optimal level between the predictions/speed trade off.
- Max_depth: Max depth represents how deep the trees will be. The deeper a tree goes, the more splits it has. This can help to capture interactions between the predictor variables. The downside to this is that have a tree with a lot of depth runs the risk of overfitting, however adding the 'None' parameter will leave said trees unpruned.
- Max_features: This represents the number of features considered when splitting a node. This parameter will be set to either 'None' (considers all features) or 'sqrt' (takes the square root of the total number of features). Again, an increase of this parameter has a direct impact on the computational power required for analysis.

We will also be using parameters that train the model at a fast rate, while also improving it. The following parameters are:

- N_jobs: This represents the number of processors used in the training process. To speed up the process, this was set to a value of '-1'. This means that there is no restriction on the amount of processors to be used.
- Random_state: This parameter allows results to be easily replicable. This parameter was set at '85'.
- Oob_score: 'Out of bag' samples allow you to estimate how accurate your model is. It uses observations on trees that were not used in the training data, as a general rule – roughly 34% of trees aren't used when a model is being trained. OOB samples use these remaining trees to test the accuracy, it is a cross validation technique.

xii. Growing an Initial Model

Before training model, several steps had to be taken to ensure the data was prepared to undergo a random forest regressor. To make sure all the Random Forest algorithm could be run on all variables, the categorical variables were transformed to numerical ones. This one done by one-hot encoding the categorical variables. This was mandatory, otherwise we could not have run a Random Forest Regressor model on our data through Scikit-Learn.

Training and Test Data

The data was split into a 80:20 ratio, 80 being the training data and 20 being the testing data.

Before we proceeded to tune hyperparameters, a base model was first created so the effects of the tuned hyperparameters could be tracked. The parameters used in the base model are the default values from the Scikit-Learn Package. This can be seen below.

Base model parameters (default values)	
N_estimators	10
Max_features	'auto'
Max_depth	'None'

Below, we can look at the results from evaluating the base model against the test set of data. As Diamond price is a continuous variable, the following will be used to compare and contrast different model iterations in order to see the impact of tuning and success of the model.:

- R-Squared
- RMSE
- Percentage Accuracy (1-MAPE)

For our base model, the results were:

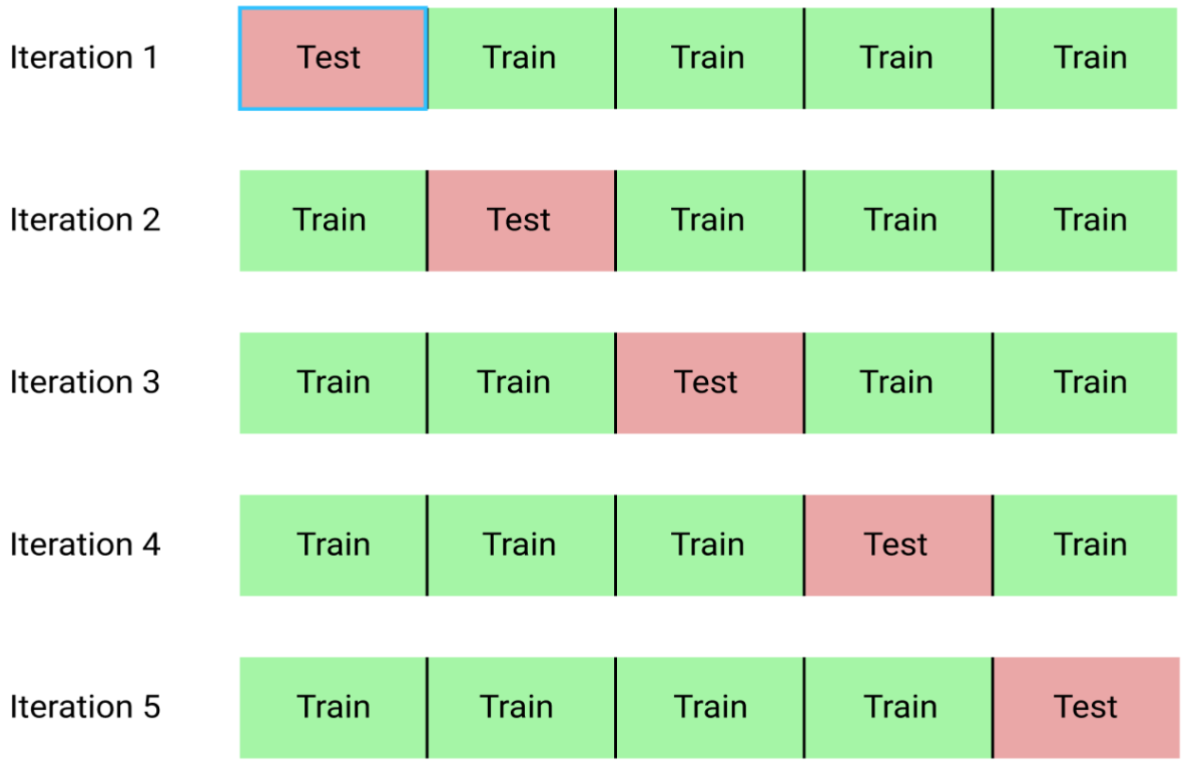
Base model Performance	
R-Squared	0.98
RMSE	536.78
Accuracy	91.62%

xiii. Hyperparameter Tuning – Random Search

To improve the performance of a model, more data could always be collected. Unfortunately for this project, this option isn't feasible. Another option for a better performing model is to tune the hyperparameters. As a default, the parameters set by Scikit learn may work well the majority of the time, but often times these parameters may not be the best for optimising the model. Hyperparameter tuning requires you to set the parameters before training the data. It is mainly a process of tweaking different parameters to see what works best.

To start the tuning process, a random search was used to find the best combination of parameters from a pre-defined parameter grid. The number the grid is searched is set by the user. Upon running random search, a n estimate of the best range of parameters are returned. These parameters will be used to define the thorough grid search that takes place after. To carry out the random search, the function RandomizedSearchCV function was used.

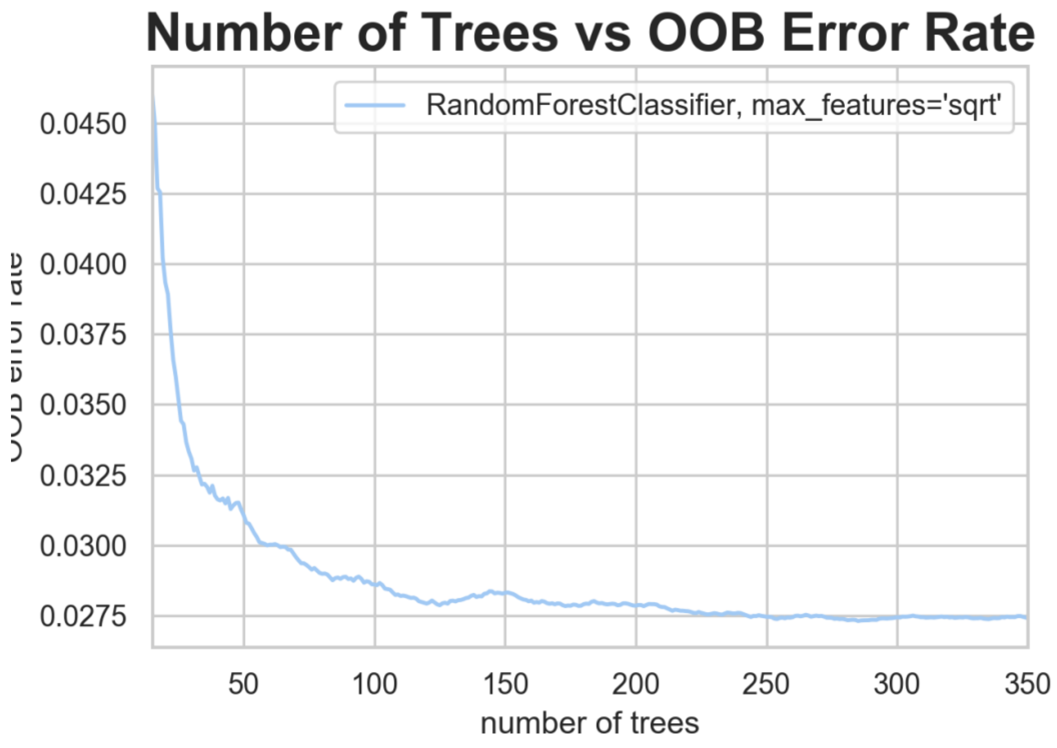
The RandomizedSearchCV function uses K-fold cross validation, this is used to reduce the risk of the model overfitting the data when training the model. It is usually advised to set a fold between 5-10. A training dataset is split into K number of subsets. For example, let's assume that K is 5, the first iteration trains on 4 folds and evaluates the 5th fold. The process begins again for K = 2 and so on. The average performance on each fold is then taken. An example illustrating this process can be observed below.


Iteration 1
Test Train Train Train Train
Iteration 2
Train Test Train Train Train
Iteration 3
Train Train Test Train Train
Iteration 4
Train Train Train Test Train
Iteration 5
Train Train Train Train Test

The iterations of the K-fold Cross Validation process is performed during the tuning of the hyperparameters. After every iteration, a different combination of parameters is generated. Once the best model is selected, it is then trained on the full training-set. When using Scikit-Learn, you can specify the number of folds and interactions to perform. For this model, these were set at:

- K-Folds (CV) = 5
- Number of iterations = 80

It is known that a higher number of trees usually improves model performance. However as stated earlier, more trees mean more computational power, consequently increasing the time needed for the analysis. To determine the range of trees to set in the grid for the randomised search, an out of bag error rate was plotted against the number of trees. In the graph below, we can see the error rate begins to flatten at around 200 trees. From this information, a range of trees was set from 180 trees to 230 trees.



The following table describes the sample that was used for the parameter grid for the Random Search:

Random Search parameters	
N_estimators	(180,190,200,210,220,230)
Max_features	(‘sqrt’, None)
Max_depth	(10,20,30,40,50,60,70,80,90,100,110,None)

Following the running of the Random Search, the following parameters were identified to be the best performing:

Random Search parameters (Best parameters)	
N_estimators	180
Max_features	None
Max_depth	100

From this result, it skews towards the already established reasoning that more trees and more depth will result in better predictions. I found it interesting how the Random Search picked the minimum number of trees passed into the parameter grid, while a figure in the upper scale was chosen for max depth. The results of evaluating the model against the test data is as follows:

First Hyperparameter tuning (Random Search)	
R-Squared	0.98
RMSE	518.15
Accuracy	91.87%

We can see that the accuracy score marginally increased from the base model by 0.25%, but a more notable improvement is in the RMSE. 536.78 from the base model to 518.15 to the first tune, a decrease of 4%.

xiv. Hyperparameter tuning – Grid Search

Now that we have performed a Random Search, the next step is to perform a Grid Search. Thanks to Random Search, we were able to narrow down the range of values for each parameter. This gives us flexibility in explicitly defining the parameters to be used in Grid Search. How Grid Search works is that it evaluates all parameters passed by the user to identify the best one. To carry this out, the GridSearchCV function from Scikit-Learn was used. GridSearchCV also uses the same K-Fold cross-validation as seen in Random Search. The number of K-Fold cross validations was also set at 5.

The grid search will run through all the parameters derived from Random Search. As you can see in the table below, the grid of parameters is more streamlined than seen in the Random Search grid.

Grid Search Parameters	
N_estimators	(170,180,190)
Max_features	(‘sqrt’, None)
Max_depth	(80,90,100,110)

After running the GridSearch algorithm, the results of evaluating the model against the test data, the best identified parameters were as follows:

Grid Search Parameters (Best Parameters)	
N_estimators	180
Max_features	None
Max_depth	80

The results of these parameters are the same as Random Search, except for max depth which happens to be 80 for GridSearch. The results of evaluating the model were:

Second Hyperparameter tuning (Grid Search)	
R-Squared	0.98
RMSE	518.15
Accuracy	91.87%

As seen in the table above, these results are the exact same as the Random Search result.

6. Model Interpretability & Evaluation

xv. Model Results

As mentioned in the previous section, the key metrics that are used in this report to evaluate the Random Forest Models are:

- R-Squared
- RMSE
- Percentage Accuracy (1-MAPE)

For regression problems, we aim to maximise R-Squared, minimise RMSE and maximise the accuracy of the model. Below we can see a comparison of the base model with the GridSearch model (only GridSearch is being shown as the results are the exact same as RandomSearch).

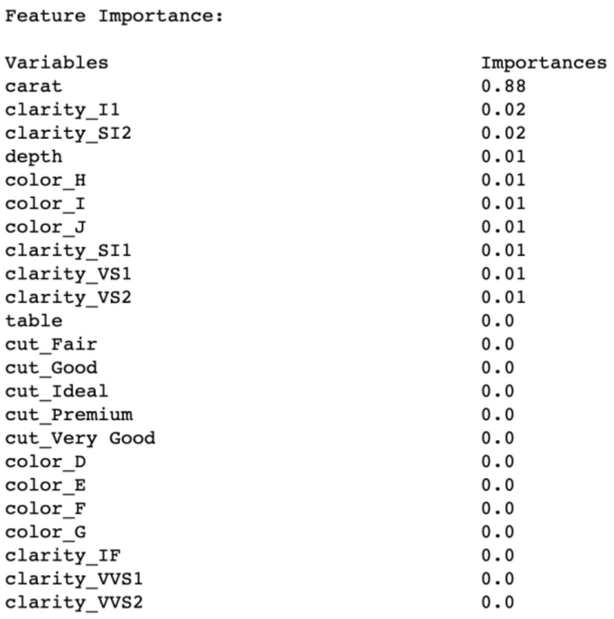
	Base Model	**Grid Search**
R-Squared	0.9822	0.9834
RMSE	536.78	518.15
Accuracy	91.62%	91.87%

From the above table we can immediately say that GridSearch was the superior model in this case as it maximised R-Squared, minimised RMSE and increased its accuracy score.

xvi. Understanding our model

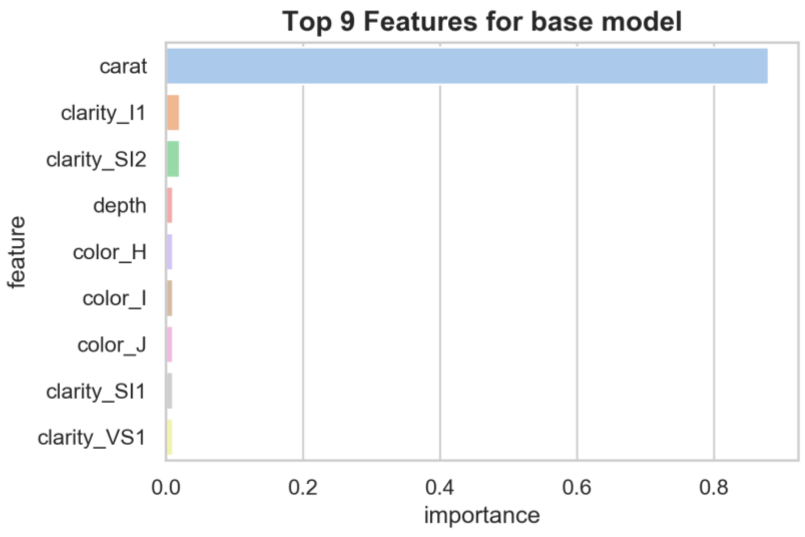
Random forest models are seen as “black box” models. This is mainly due to the fact that a forest consists of a large number of deep trees – each tree is then trained using bagged data from a random selection of features, therefore gaining a full understanding of the process by studying each tree is completely infeasible. Furthermore, let’s say even if we wanted to examine a single tree – the only way of analysing a tree is if it has a small depth and a low number of features. This fact does not seem to truly reflect the kind of datasets that are dealt with on a daily basis.

One way to understand how our model is interacting with variables is to compute feature importances. A Feature importance graph can tell us which variables are important in predicting the price of a diamond. Below we can observe a feature importance graph created from the GridSearch random forest algorithm.



```
Feature Importance:

Variables                                Importances
carat                                    0.88
clarity_I1                               0.02
clarity_SI2                              0.02
depth                                    0.01
color_H                                  0.01
color_I                                  0.01
color_J                                  0.01
clarity_SI1                              0.01
clarity_VS1                              0.01
clarity_VS2                              0.01
table                                    0.0
cut_Fair                                 0.0
cut_Good                                 0.0
cut_Ideal                                0.0
cut_Premium                              0.0
cut_Very Good                            0.0
color_D                                  0.0
color_E                                  0.0
color_F                                  0.0
color_G                                  0.0
clarity_IF                               0.0
clarity_VVS1                             0.0
clarity_VVS2                             0.0
```



We can immediately see, from both the graph and the feature importances table, that carat, clarity, color and depth account for how the random forest used variables to predict the price. The only colours that the Random Forest Regressor used were colours H, I and J. Intuitively, this aligns with our preliminary data analysis as we saw that these specific colours had the highest prices in the dataset. As expected too, the regression model used the majority of the mid-tier quality diamonds in terms of clarity. Since the majority of diamonds in the dataset had clarity qualities of this range, it makes sense that the model prioritised these.

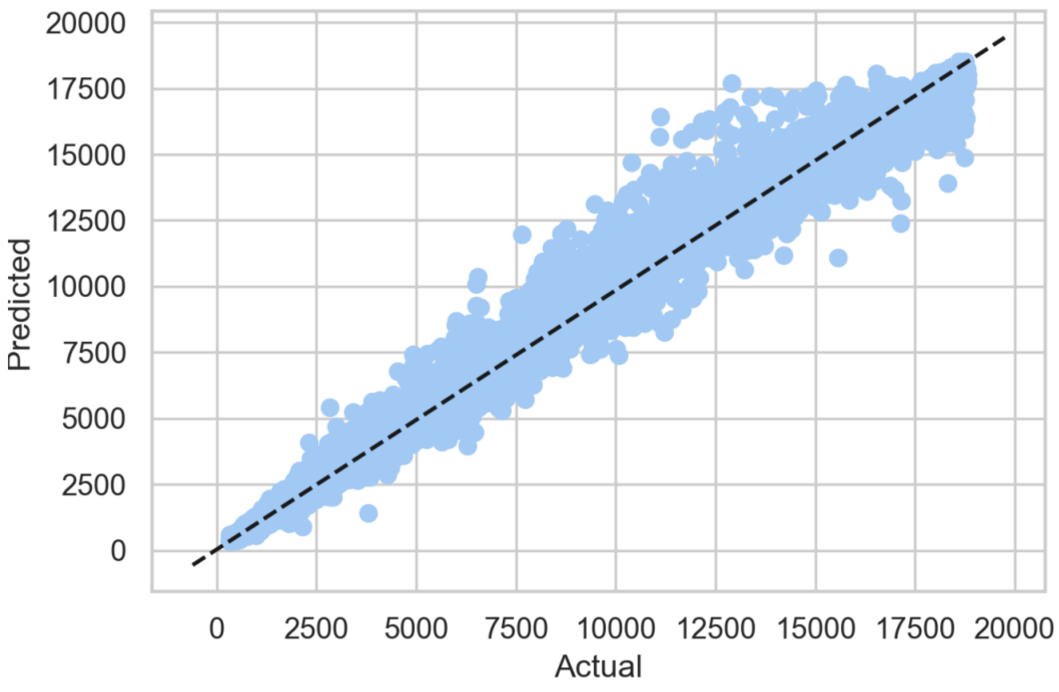
Carat accounts for 88% of feature importance, this is a lot higher than all the other variables. On one hand, this seems reasonable as in our preliminary analysis of the data, the majority of variables were highly related with carat, thus determining price.

xvii. Partial Dependency Plots

A partial dependency plot shows how two or more variables affect the predicted outcome of a machine learning outcome. Once created, a partial dependency plot can illustrate whether the relationship between them is linear, monotonic, or more complex. Having tried to plot partial dependency plots with the diamond data, I had no luck as unfortunately Scikit-Learn currently only lets you create this plot from a gradient boosting algorithm. But from our other analysis, one can estimate the kind of plots you would expect to see.

Compared to R, Python has great machine learning capabilities, however I do not think the visualization capabilities are on the same level as R's.

xviii. How well did my model perform? – Visualizations



x.

x.

i.

xxii.

Here we can observe that the predicted values from our Random Forest model falls In line with the Actual model, giving us confidence that are model works. However, the fact still remains that there is plenty of scope to explore other avenues that may very well impact our current model (good or bad). For example, one possibility of approaching this interesting topic is by not excluding any variables at all and seeing how that impacts our results. Another way is to perhaps merge sub categories with each that have extremely similar statistics, another way could be creating derived variables from our current variables. At this stage, it is clear that you can go down many routes to try and achieve the best model – but one must always keep in mind that they are trying to achieve that "fine balance" which is extremely hard to achieve sometimes. All in all, it is why some people say Data analysis is an art, and not a skill.

Python Code

```
#!/usr/bin/env python

# coding: utf-8


# In[ ]:



import pandas as pd

import numpy as np

import math

import os

import matplotlib.pyplot as plt

import seaborn as sns

sns.set()



# In[ ]:



import warnings

warnings.filterwarnings(action = "ignore")
```

```
# In[ ]:



#reading in CSV file

diamonds = pd.read_csv("/Users/gabrielogundipe/Documents/Data Analytics/diamonds.csv")



# In[ ]:



diamonds.describe()



# In[ ]:



#dropping first column as it is just describing the index

diamonds = diamonds.drop("nid", axis = 1)

diamonds.head()



# In[ ]:



diamonds = diamonds[(diamonds[['x','y','z']] != 0).all(axis=1)]
```

```
diamonds.describe()



# In[ ]:



summary = round(diamonds.describe().transpose(),2)

print(summary)



# In[ ]:



sns.set(style='whitegrid', palette='pastel')
sns.distplot(diamonds['price'], kde=False)
plt.title('Price Distribution', fontsize=19, fontweight='bold')
plt.ylabel("Frequency")
#plt.savefig('price.png', dpi=220)
plt.figure(figsize=(20,20))




# In[ ]:
```

```
corr_matrix = diamonds.corr()


plt.subplots(figsize = (10,8))

sns.heatmap(corr_matrix, annot = True, cmap = "Blues", )

#plt.savefig('price.png', dpi=250)

plt.show()




# In[ ]:




g = sns.FacetGrid(diamonds, col="cut", palette='pastel', col_wrap = 3)

g = g.map(plt.hist, "price", bins=30)

g.set_ylabels("Frequency")

#plt.savefig('Cut distribution against price', dpi=220)


#f, axes = plt.subplots(3, 2, figsize=(7, 7), sharex=True)

#sns.distplot( diamonds[diamonds.cut == 'Ideal'] ,kde = False, color="skyblue")#, ax=axes[0, 0])

#sns.distplot( diamonds["sepal_width"] , color="olive", ax=axes[0, 1])

#sns.distplot( diamonds["petal_length"] , color="gold", ax=axes[1, 0])

#sns.distplot( diamonds["petal_width"] , color="teal", ax=axes[1, 1])
```

```
# In[ ]:



sns.lmplot(y="price", x="carat", data=diamonds, height=6, palette = 'pastel', line_kws={'color': 'red'})

plt.ylim(0,20000)

plt.xlim(0, 3)

plt.title('Linear fit, price vs carat')


plt.savefig('Linear fit, carat vs price.png', dpi=220)



# In[ ]:



boxplot_cut_price = sns.boxplot(x="cut", y="price", data=diamonds)

plt.title("Boxplot of Diamond price vs Cut")

plt.savefig('boxplot cut.png', dpi=220)



# In[ ]:



sns.lmplot(y="table", x="price", data=diamonds, height=6, palette = 'pastel', fit_reg=False)

#sns.lmplot(y="depth", x="price", data=diamonds, height=6, palette = 'pastel', ax=axes[1])
```

```
plt.title("Table vs Price")

#plt.savefig('table vs price.png', dpi=220)



# In[ ]:




#sns.lmplot(y="table", x="price", data=diamonds, height=6, palette = 'pastel', fit_reg=False)

sns.lmplot(y="depth", x="price", data=diamonds, height=6, palette = 'pastel', fit_reg=False)


plt.title("Depth vs Price")

plt.savefig('depth vs price.png', dpi=220)



# In[ ]:



sns.lmplot(data=diamonds,y='price', height=5,x='carat',  hue='cut', fit_reg=False, markers =
["d","p","s","v","o"])

plt.savefig('scatter cut vs carat vs price.png', dpi=220)



# In[ ]:
```

```
sns.set(style='whitegrid', palette='pastel')

sns.distplot(diamonds['carat'], kde=False, bins = 60, rug=True,

rug_kws={"color": "b", "alpha":0.3, "linewidth": 1, "height":0.01 }

)


plt.title('Carat distribution', fontsize=19, fontweight='bold')

plt.ylabel("Frequency")

plt.savefig('carat distribution.png', dpi = 200)

plt.figure(figsize=(20,20))



# In[ ]:



boxplot_color_price = sns.boxplot(x="color", y="price", order = ['D', 'E', 'F', 'G', 'H', 'I',
'J'],data=diamonds)

plt.title("Boxplot of diamond color vs price")

plt.savefig('boxplot clarity price.png', dpi = 200)



# In[ ]:
```

```
boxplot_clarity_price = sns.factorplot(x='clarity', y='price',kind='box', data=diamonds, aspect=2, order=['I1', 'SI2', 'SI1', 'VS2','VS1', 'VVS2', 'VVS1', 'IF'] )

plt.title("Boxplot of Diamond Clarity vs Price")

plt.savefig('boxplot clarity price.png', dpi = 200)



# In[ ]:



sns.countplot(x='clarity',data=diamonds,order=['I1', 'SI2', 'SI1', 'VS2','VS1', 'VVS2', 'VVS1', 'IF'])

plt.ylabel("Frequency")

plt.title("Frequency of Diamond Clarity")

plt.savefig('clarity count.png', dpi = 200)

#come back to this later and try and stack them



# In[ ]:



sns.lmplot(data=diamonds,y='price', height=5,x='carat',  hue='color', fit_reg=False, markers = ["d","p","s","v","o", "p","h"])

plt.savefig('scatter color vs carat vs price.png', dpi=220)



# In[ ]:
```

```
sns.countplot(x='clarity',data=diamonds, order = ['D','E','F', 'G', 'H', 'I', 'J'])

plt.ylabel("Frequency")

plt.title("Frequency of Diamond Color")

plt.savefig('Color count.png', dpi = 200)



# In[ ]:



sns.lmplot(data=diamonds,y='price', height=5,x='carat',  hue='clarity', fit_reg=False, markers =
["d","p","s","v","o", "p","h", "*"])

plt.savefig('scatter clarity vs carat vs price.png', dpi=220)



# In[ ]:



diamonds.cut.count()



# In[ ]:



corr_matrix = diamonds[['x','y','z', 'table', 'depth','price','carat']].corr()
```

```
plt.subplots(figsize = (10,8))

# Generate a mask for the upper triangle

mask = np.zeros_like(corr_matrix, dtype=np.bool)

mask[np.triu_indices_from(mask)] = True


# Set up the matplotlib figure




# Generate a custom diverging colormap

cmap = sns.diverging_palette(220, 10, as_cmap=True)


sns.heatmap(corr_matrix, mask=mask, cmap=cmap, vmax=1, center=0,annot = True,
cbar_kws={"shrink": .5})

plt.savefig('dimensional diagnoal.png', dpi=220)


#exclude = ['carat']

#df.loc[:, df.columns.difference(exclude)].hist()




# In[ ]:




diamonds.describe()
```

```
diamond_data = diamonds.drop(['x',"y","z"], axis=1)

diamond_data.head()



# In[ ]:



from sklearn.model_selection import train_test_split

from sklearn.model_selection import cross_val_score, KFold

from sklearn.metrics import mean_squared_error, r2_score

from sklearn.preprocessing import StandardScaler, LabelEncoder



# In[ ]:



#label_cut = LabelEncoder()

#label_color = LabelEncoder()

#label_clarity = LabelEncoder()



#diamond_data['cut'] = label_cut.fit_transform(diamond_data['cut'])

#diamond_data['color'] = label_color.fit_transform(diamond_data['color'])

#diamond_data['clarity'] = label_clarity.fit_transform(diamond_data['clarity'])
```

```
# In[ ]:



X = diamond_data.drop(['price'], axis = 1)

X = pd.get_dummies(data=X)#encode categeorical instances

variable_list = list(X.columns)

X = np.array(X)



# In[ ]:



y = diamond_data['price']



# In[ ]:




X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2, random_state=0)



# In[ ]:
```

```
def evaluate(model, X_test, y_test):
    predictions = model.predict(X_test)
    errors = abs(predictions - y_test)
    mape = 100 * np.mean(errors / y_test)
    accuracy = 100 - mape
    rmse = np.sqrt(mean_squared_error(y_test, predictions))
    r2 = r2_score(y_test, predictions)

    print('\nModel Performance: ')
    print('\nAccuracy (1-MAPE) = {:0.2f}%'.format(accuracy))
    print('RMSE = {:0.2f}'.format(rmse))
    print('R^2 Score = {:0.4f}'.format(r2))

def rmse_evaluate(model, X_test, y_test):
    predictions = model.predict(X_test)
    rmse = np.sqrt(mean_squared_error(y_test, predictions))
    print('\nModel Performance: ')
    print(rmse)



# In[ ]:
```

```
# Import the RF Regression Model

from sklearn.ensemble import RandomForestRegressor

# Fitting our base RandomForest model using the default parameters

base_model = RandomForestRegressor(random_state = 99, n_jobs=-1)

base_model.fit(X_train, y_train)

evaluate(base_model, X_test, y_test)




# In[ ]:




print(base_model.oob_score_)

#print oob score , not too disimilar from R^2 score


from pprint import pprint

pprint(base_model.get_params())


from collections import OrderedDict

ensemble_clfs = [

    ("RandomForestClassifier, max_features='sqrt'",

        RandomForestRegressor(

                              oob_score=True,

                            max_features="sqrt",

                                warm_start=True,

                                random_state=99,
                              n_jobs=-1))
          ]


def oob_error_plot(ensemble_clfs, min_estimators, max_estimators):
    # Map a classifier name to a list of (<n_estimators>, <error rate>) pairs
    error_rate = OrderedDict((label, []) for label, _ in ensemble_clfs)
    for label, clf in ensemble_clfs:
        for i in range(min_estimators, max_estimators + 1):
            clf.set_params(n_estimators=i)
            clf.fit(X_train, y_train)



            # Record the OOB error for each `n_estimators=i`
            oob_error = 1 - clf.oob_score_
            error_rate[label].append((i, oob_error))



    # Generate the "OOB error rate" vs. "n_estimators" plot.
    for label, clf_err in error_rate.items():
            xs, ys = zip(*clf_err)
            plt.plot(xs, ys, label=label)


    plt.title('Number of Trees vs OOB Error Rate', fontsize=20, fontweight='bold')
    plt.xlim(min_estimators, max_estimators)
    plt.xlabel("number of trees")
    plt.ylabel("OOB error rate")

    plt.legend(loc="upper right")

    plt.savefig('OOB rate.png', dpi=230)

    plt.show()


oob_error_plot(ensemble_clfs, min_estimators=15, max_estimators=350)

plt.savefig('OOB rate.png', dpi=230)

#flattens at around 220 trees




#---------------------------------------------------------------------------

#--------------------------- Hyperparameter tuning -----------------------------

#----------------------------------------------------------------------------


#Using Random Search with cross validation

from sklearn.model_selection import RandomizedSearchCV


n_estimators = [180,190,200,210,220,230] #number of trees based on OOB error rate

max_features = ['sqrt', None] #Number of features to consider at every split

max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]

max_depth.append(None)


#Create random grid of parameters to sample from

random_grid = {'n_estimators': n_estimators,

    'max_features': max_features,
    'max_depth': max_depth,

    'n_jobs': [-1], # this enables parallel processing

    'oob_score': [True], # to use

    'random_state': [99]}


from pprint import pprint

print('\nGrid of Parameters to be randomly sampled from: \n')

pprint(random_grid)




# In[ ]:




# Finding the best performing combination of parameters by fitting it to our training set

RS_classifier = RandomizedSearchCV(estimator=RandomForestRegressor(),

param_distributions=random_grid, n_iter=80, cv=5, verbose=2, random_state=99, n_jobs=-1)


RS_classifier.fit(X_train, y_train)




evaluate(RS_classifier, X_test, y_test)




# In[ ]:
```

```
RS_classifier.best_params_

print(RS_classifier.oob_score_)



# In[ ]:



#Using GridSearch

from sklearn.model_selection import GridSearchCV

gridsearch_grid = {

    'n_estimators': [170, 180, 190],

    'max_features': ['sqrt', None],

    'max_depth': [80,90,100,110],

    'n_jobs': [-1],

    'oob_score' : [True],

    'random_state': [99]

}


print('\nGrid of Parameters to feed into model: \n')

pprint(gridsearch_grid)


GS_classifier = GridSearchCV(estimator=RandomForestRegressor(), param_grid=gridsearch_grid,
cv=5,

verbose=2, n_jobs=-1)
```

```
GS_classifier.fit(X_train, y_train)

GS_classifier.best_params_


evaluate(GS_classifier, X_test, y_test)




# In[ ]:




GS_classifier.best_params_




pprint(base_classifier.get_params())




base_classifier.best_params_






max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]

max_depth


plt.scatter(y, GS_classifier.predict(X))

plt.xlabel("Actual")
plt.ylabel("Predicted")

x_lim = plt.xlim()

y_lim = plt.ylim()

plt.plot(x_lim, y_lim, "k--")

plt.savefig('actual vs predicted.png', dpi=240)

plt.show()




# In[ ]:




GS_classifier.predict(X)






diamond_data.head()





def get_feature_importances(classifier):

    importances = list(classifier.feature_importances_)

    feature_importances = [(feature, round(importance, 2)) for feature, importance in
zip(variable_list,importances)]

    feature_importances = sorted(feature_importances, key = lambda x: x[1], reverse = True)

    print('\nFeature Importance: \n')

    print('{:<40}'.format('Variables'),'Importances')
    [print('{:40} {}'.format(*pair)) for pair in feature_importances]

    return pd.DataFrame(feature_importances, columns=['feature', 'importance'])



# In[ ]:



def plot_feature_importances(feature_importance, num_features=10, label=''):

    ax = sns.barplot(x='importance', y='feature', data=feature_importance.loc[0:num_features-1,])

    ax.set_title('Top {} Features for {}'.format(num_features, label), fontsize=14, fontweight='bold')

    plt.tight_layout()

    plt.savefig('importances.png', dpi = 250)

    plt.show()



# In[ ]:



plot_feature_importances(get_feature_importances(GS_classifier.best_estimator_),
num_features=9, label='base model')



# In[ ]:
```