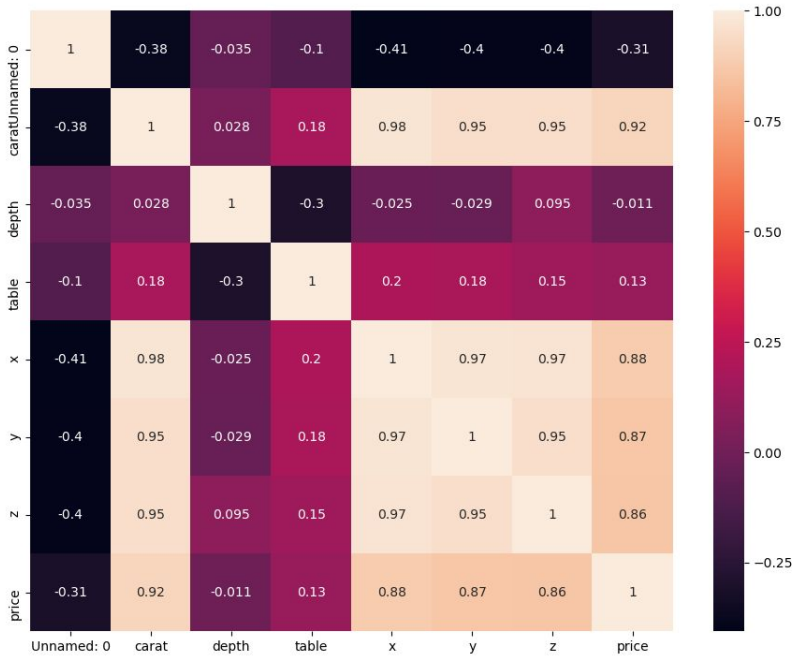
COMP 304 - Assignment 4
Liam Royal - 300368747

Part 1: Performance metrics in Regression [35 marks]

```
print(data.groupby("price").mean())
```

Numbers show that there is a correlation between price and carat, and between price and the dimension attributes (x, y, z) suggesting, the higher the carat, and the bigger the diamond, the higher the price, this correlation makes sense according to past experience, and my business understanding.

I used the seaborn library to do some exploratory data analysis on correlations between attributes:



The seaborn heatmap shows the indexing attribute has negative correlations with every attribute this makes sense, as it is only used for indexing, thus I will remove it to ensure it does not negatively affect the regression model.

```
data = data.drop(data.columns[0], 1)
```

Noticing attributes cut, color, and clarity all contain string values, I have decided to assign equivalent integer values so that these values can contribute towards the regression model and will not just be ignored.

I used:

```
def cutInt(v):
    if v =="Ideal": return 2
    if v =="Premium": return 1
    if v =="Good": return 4
    if v =="Very Good": return 3
    if v =="Fair": return 5

data['cut'] = data['cut'].apply(cutInt)
```

For each corresponding attribute to transform the string values to integer.

Parameters:

K-neighbours regression: On this algorithm I set the weight parameter to use distance, oppose to the default uniform weights based upon neighbourhood. This meant the closest neighbours were identified as opposed to grouping to the closest neighbour hood. This slightly decreased the mean absolute error.

```
KNeighborsRegressor(weights='distance')
```

Gradient Boosting regression: I set the number of iterations to 500, and the validation fraction to 0.3 this enables the algorithm to stop training early if the validation score is not improving upon training, I did this to prevent overfitting

```
GradientBoostingRegressor(n_iter_no_change=500,
validation_fraction=0.3)
```

MLPRegression: I set early stopping to true, once again to prevent overfitting as this can be a common weakness of neural networks. I also changed the max number of iterations to 800 from the default of 200 as the algorithm was reaching 200 iterations without the optimization converging, thus increasing this would allow convergence.

```
MLPRegressor(early_stopping=True, max_iter=800)
```

Regression Algorithms:

Results:

Algorithm	Linear	K-neighbours	Ridge	Decision Tree	Random Forest
MAE	856.67	461.86	857.22	378.02	376.79
MSE	2006437.22	835292.60	2006712.70	611059.90	604267.22
RMSE	1416.49	913.94	1416.58	781.70	777.35
R^2E	0.87	0.95	0.87	0.96	0.96
Execution Time	0.01	0.49	0.01	0.15	0.15

Algorithm	Gradient Boosting	SGD	SVR	Linear SVR	MLP
MAE	421.67	15535536.50	2772.53	1082.59	672.09
MSE	694272.56	359868863078700.19	17977151.92	3511650.93	1478353.46
RMSE	777.35	18970209.88	4239.95	1873.94	1215.88
R^2E	0.96	-22600473.96	-0.13	0.78	0.91
Execution Time	0.83	0.67	82.54	0.17	120.83

Conclusion:

MAE shows us the average error for each value in comparison to the accurate value, thus can be an effective measurement of how accurate the model is. Interestingly both tree family algorithms Random forest and decision tree, have a very similar MAE, this makes sense as Random forest uses decision trees to predict. We can see a relative relationship between MAE and RMSE across all the algorithms, this shows that a higher RMSE is also likely to result in a higher MAE, meaning the higher the RMSE it is likely the algorithm will perform worse due to a higher variance of values. The Linear Regression algorithm and Ridge algorithm also performed very similarly putting up very similar numbers across the board. Ridge and Linear are also very

similar algorithms much like decision tree and random forest, with Ridge regression being a slight extension upon the linear regression model. From the findings in the table above we can conclude that the performance of the regression algorithm boils down to its prediction method, and through the evaluation metrics we can see similar algorithms with similar prediction methods performing similarly.

Part 2: Performance metrics in classification [35 marks]

Preprocessing:

To begin the preprocessing of the data I applied labels to all attributes so it was easier to identify attributes in analysis. I used 'A1 - A14' to identify the attributes and used 'Class' to identify the column of class labels

Code:

```
trainData.columns = ["A1", "A2", "A3", "A4", "A5", "A6", "A7", "A8",
"A9","A10", "A11", "A12", "A13", "A14", "Class"]
testData.columns = ["A1", "A2", "A3", "A4", "A5", "A6", "A7", "A8",
"A9","A10", "A11", "A12", "A13", "A14", "Class"]
```

Now that I had a way to identify each attribute I used:

```
for column in trainData.columns:
    print(column, ":" , trainData[column].unique(), "\n")
```

To observe the unique values for each attribute. To further analyze the data set I created the correlation heat map to observe the correlations between the attributes and the class. Prior to doing this it was necessary to transform all string values to a corresponding integer value to enable the correlation function to work.

To convert attributes of integer values I applied a series of transformations on each attribute.

For example:

```
def transA10(str):
    if str == " Female":
        return 1
    if str == " Male":
        return 0
    else:
```

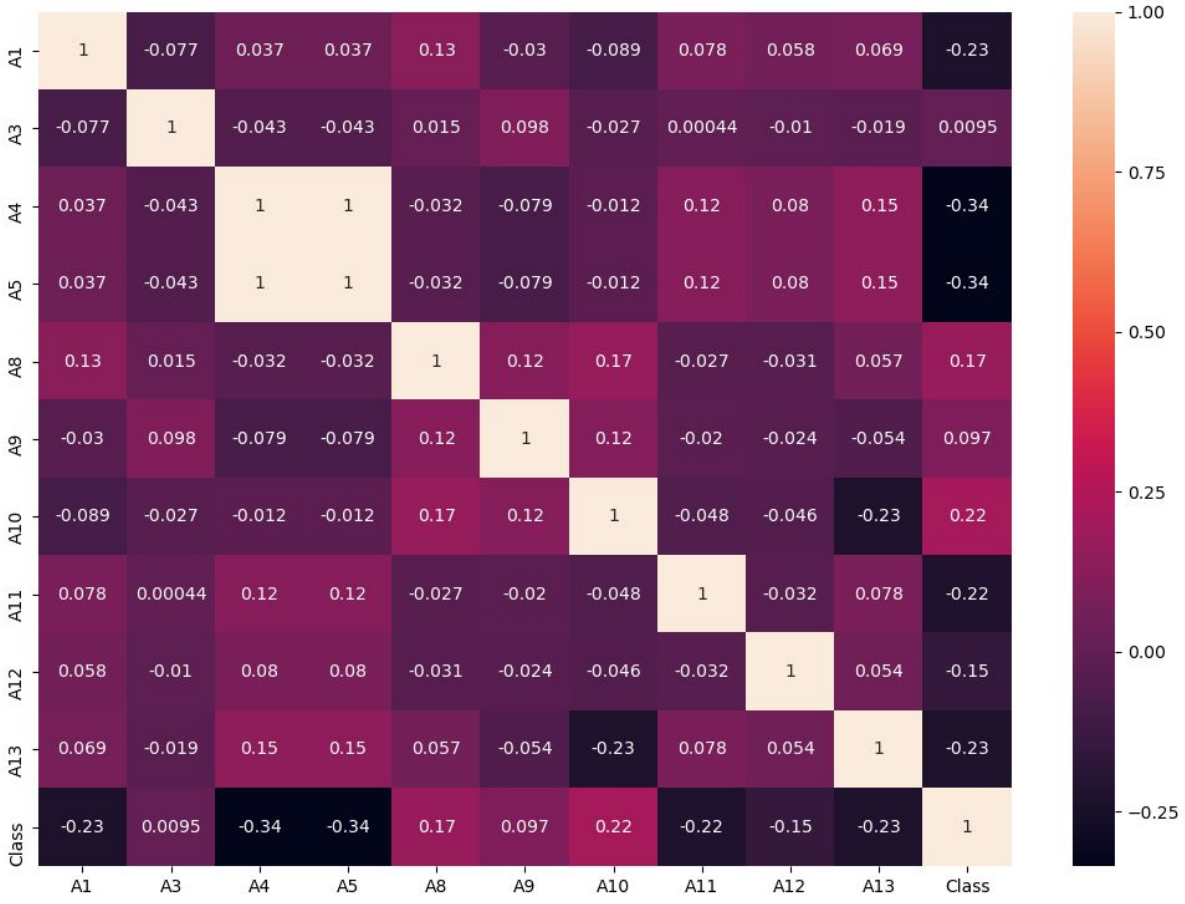
```
        return "?"
```

Then:

```
trainData['A10'] = trainData['A10'].apply(transA10)
testData['A10'] = testData['A10'].apply(transA10)
```

Once the transformations had been applied I was able to produce a heatmap using the seaborn library that visualized correlations between attributes, allowing me to determine which attributes were important in classifying an instance.

Correlation heatmap:



The heat map shows a very weak correlation between ‘A3’, the ‘A9’ attribute and the class label, thus I will drop these attributes

Code:

```
trainData = trainData.drop(trainData.columns[2],1)
trainData = trainData.drop(trainData.columns[7],1)
testData = testData.drop(testData.columns[2],1)
testData = testData.drop(testData.columns[7],1)
```

The data was still scattered with unwanted NaN values due to missing values, thus in the final preprocessing step I used the following code snippet to replace these values with mean values.

```
trainData.fillna(trainData.mean(), inplace = True)
testData.fillna(testData.mean(), inplace = True)
```

Results:

Algorithm	KNeighbours	Naive Bayes	SVC	Decision Tree	Random Forest
Classification Accuracy	0.84	0.80	0.80	0.82	0.84
Precision	0.83	0.79	0.79	0.82	0.84
Recall	0.62	0.34	0.27	0.60	0.59
F1 score	0.71	0.47	0.40	0.69	0.69
AUC	0.76	0.64	0.62	0.74	0.76

Algorithm	AdaBoost	Gradient Boosting	Linear discriminant analysis	MLP	Logistic Regression
Classification Accuracy	0.86	0.87	0.81	0.84	0.82
Precision	0.85	0.86	0.79	0.83	0.81
Recall	0.59	0.59	0.38	0.53	0.36

F1 score	0.70	0.70	0.51	0.65	0.50
AUC	0.77	0.77	0.66	0.73	0.66

Is classification accuracy the best performance metric to evaluate a classifier? And why?

No, MMC (Matthews Correlation Coefficient) seems to be the best metric to evaluate the performance of a classifier, as it takes into account all cells of the confusion matrix, also because the range of values lie between -1 and 1 it is very easy to interpret, with a score of 1 being a perfect model, along with being easily interpretable it is robust to changes in the prediction goal (R1). Not only is the MMC more accurate, but the classification accuracy is not always truly accurate, for example if the data set is significantly imbalanced, for example 95 of class A and 5 of class B, the model may correctly classify all instances of A and incorrectly classify all instances of B. This would result in 95% classification accuracy, which would lead one to assume it is a good model, but this metric is misleading in this case as it is a good model for classifying instances of A but very poor at classifying instances of B but this is masked by the class imbalance.

Best performing algorithms:

The two best performing algorithms were the 'AdaBoost' algorithm, and the 'GradientBoosting' algorithm both putting up very similar numbers:

Adaboost:

Algorithm	AdaBoost	Gradient Boosting
Classification Accuracy	0.86	0.87
Precision	0.85	0.86
Recall	0.59	0.59
F1 score	0.70	0.70
AUC	0.77	0.77

The are similar algorithms as they are both boosting algorithms, meaning they both iteratively build upon a strong learner using a number of weaker learners. Thus having the same fundamental system they have managed to produce very similar results.

References:

Joseph, J. J. (2016). *The Best Metric to Measure Accuracy of Classification Models*. Retrieved from: https://www.kdnuggets.com/2016/12/best-metric-measure-accuracy-classification-models.html