Comp309 Assignment 4

Vincent Yu

September 2018

1 Part 1

In this assignment, it requires ten different regression techniques.

1.1 Tuning Parameters

1.1.1 Multi-layer Regression

In this technique there are lots parameters we can play with, I changed two parameters, the first one is "early_stopping", the default setting was false, it was changed to True. Early_stopping is one of the most popular and efficient techniques. This could stop the model construction early, to avoid overfitting. But this technique doesn't work well for small dataset.
The other parameter I changed is learning_rate_init : default setting is 0.001 (constant). It controls the step-size in updating the weights. It was changed to 0.01. It increase the step-size in updating the weights. By changing this two parameters, the time of constructing the model highly reduced and the MSE(mean squared error) dropped significantly.

1.1.2 gradient Boosting regression

For this technique, the default setting of the maximum step was 3, and it was set to 10. The MSE value dropped significantly. The maximum depth limits the number of nodes in the tree. The parameter n_estimators was changed to 200 from 100(default). This parameter controls the number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting so a large number usually results in better performance.

1.2 Preprocessing before regression

1.2.1 Replace categorical string level with numeric value

The diamond dataset has three categorical features, which needs to be converted to numeric value. The categorical feature was converted by using discrete numbers to much different levels.

```
def convert_categorical_value_to_numeric(df):
    df['cut'] = df['cut'].map({'Ideal':5, 'Premium':4, 'Very Good':3,'Good': 2,'Fair':
    df['clarity'] = df['clarity'].map({'IF':8,'VVS1':7,'VVS2':6,'VS1':5,'VS2':4,'SI1':
    df['color'] =df['color'].map({'J':7, 'I': 6, 'H':5, 'G':4, 'F':3,'E':2,'D':1})
    df.drop(['Count'],axis=1)#drop the index number
    print(df.head())
    return df;
```

```
df.drop(['Count'],axis=1)#drop the index number
```

The correspondence between these numbers and categories stems from the relationship between diamond quality and these categories. The relationship is shown below. (fig 1.1)

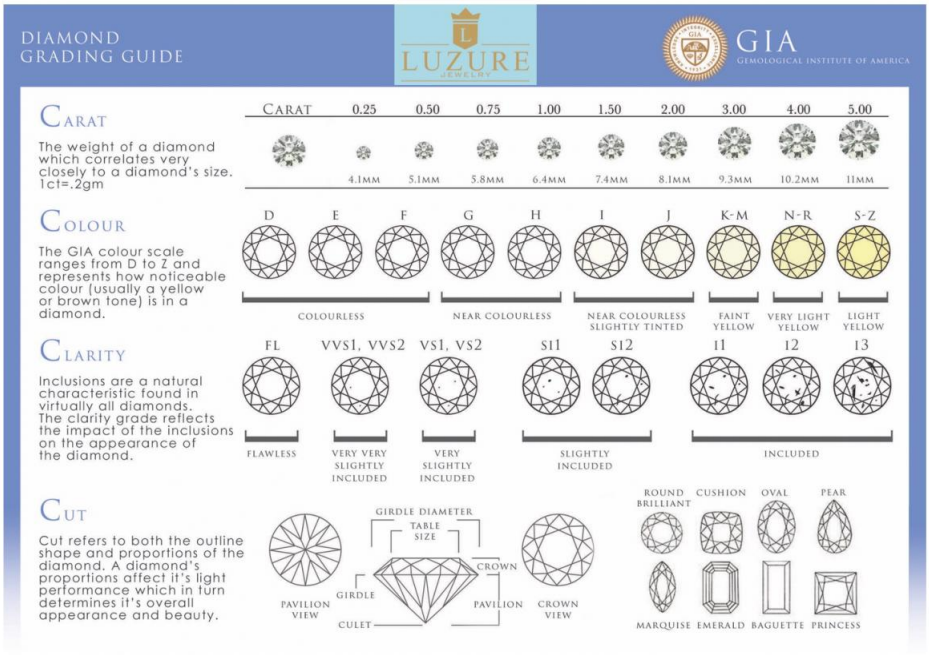


fig 1.1

Take clarity as an example, the reason I didn't use encoder.transform, is because there is a risk which python could randomly convert the categorical value into different number without following the relationship between quality of diamond and clarity level. Thus I defined a number for different level of clarity, which follow the relationship.

1.2.2 Standardize

The features in the dataset have different units, so it was wise to standardize the dataset. Here is the code for the way I did standardize:

```
    train_data_copy = train_data.copy()
train_be4_standard = train_data_copy.drop(['price'],axis=1)
test_be4_standard = test_data.copy()
test_be4_standard = test_be4_standard.drop(['price'],axis=1)
train_mean = train_be4_standard.mean()
train_std = train_be4_standard.std()
train_standarded = (train_be4_standard-train_mean) / train_std
test_standarded = (test_be4_standard - train_mean) /train_std
test_standarded['price'] = test_data['price']
train_standarded['price'] = train_data['price']
```

A copy of data was made before drop the class label. So it can be added back after standardize. The reason of dropping the class label because we desire to explore the price of diamond which is the class label. Thus we should keep the price of diamond unchanged. If it wasn't dropped but also did standardize, the price would be varied from -1 to 1, which could the ruin the dataset.(it makes no sense then) After standardizing all the features then the class label was added back for both training set and test set.

1.2.3 drop the index of the dataset

```
df.drop(['Count'],axis=1)#drop the index number
```

This part was easy, this line of code drop the index of each instance, because it is useless for regression and it also may ruin the model, because if it was included in the model it will actually affect the class label which is the price of the diamond.

1.3 result for 10 different regression techniques

Algorithm	MSE	RMSE	R-Squared	MAE	Execution time
Linear Regression	1537312.97	1239.88	0.91	810.49	0.016
K-neighbors regression	426118.85	652.77	0.97	311.80	0.083
Ridge Regression	1537313.14	1239.88	0.91	810.61	0.001
Decision Tree	2926.85	54.10	0.99	3.53	0.24
Random Forest	2174.52	46.63	0.99	3.85	2.17
Gradient Boosting	1946.57	44.12	0.99	123.05	3.17
SGD regression	1558618.83	1248.44	0.90	806.91	0.029
Support vector	8089927.79	2844.27	0.50	1328.47	79.30
Linear SVR	2673741.67	1635.15	0.83	858.08	0.07
Multi-layer perceptron	101229.62	318.16	0.99	181.84	1.324045

fig1.2

As the table is shown above, it illustrates the result of 10 different regression.

1.4 Comparison and conclusions

From fig1.2, the best model should be the one constructed by using Gradient boosting. It has the highest R squared value and lowest MSE . R squared value is a good measurement of how good or bad a model is. R squared value for an ideal model is one.

$$R^2 = 1 - \frac{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y})^2} = 1 - \frac{MSE}{\frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y})^2}$$

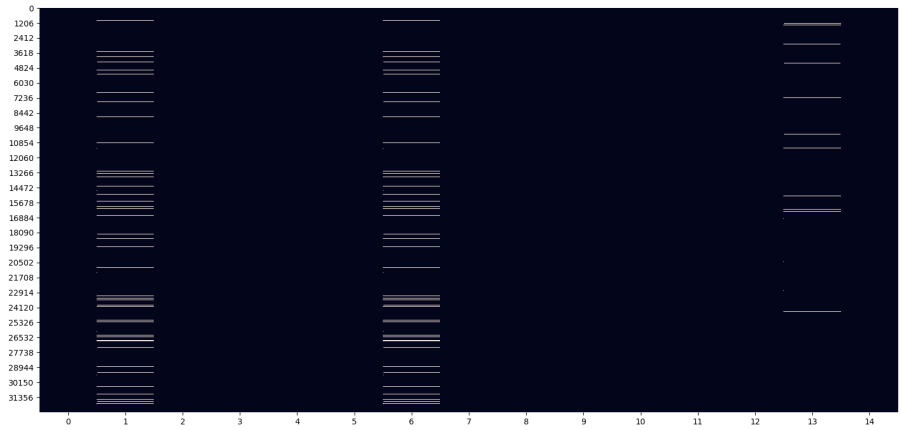
fig 1.3

As the formula is shown above, we can see there is a relationship between MSE and R squared. So optimizing R squared value can also optimize the MSE value. The execution time of Gradient boosting model construction is acceptable small.
For selecting best model, we should first check R-squared value (if the model can provide), and then we need to check the MSE and RMSE (RMSE is the square root of MSE). The best model should also can be constructed fast. So so in some cases the best model is selected under compromise.

2 Part 2

2.1 reprocessing

Both the training set and the test set included missing values, as the graph is shown below:



So the first thing was done is convert "?" into "Nan" by using the code is shown below :

```
train_df = train_df.replace(" ?",value= np.NaN)
test_df = test_df.replace(" ?",value= np.NaN)
```

The missing value was fixed by replace the missing value with the most common value in each features. The code is shown below:

```
train_df=train_df.apply(lambda x:x.fillna(x.value_counts().index[0]))
test_df = test_df.apply(lambda x: x.fillna(x.value_counts().index[0]))
```

This dataset has different type of features. Some of the categorical feature represent different value by using some String. The string values was converted into numbers by using the code is shown below:

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
train_df[1] = le.fit_transform(train_df[1])
train_df[3] = le.fit_transform(train_df[3])
train_df[5] = le.fit_transform(train_df[5])
.
.
.
test_df[9] = le.fit_transform(test_df[9])
test_df[13] = le.fit_transform(test_df[13])
test_df[14] = le.fit_transform(test_df[14])
```

2.2 result for ten different classification

Techniques	KNN	Naive-Bayes	SVM	Decision Tree	Random Forest	AdaBoost	Gradient Boosting	Linear discriminant	Multi-layer perceptron	Logistic Regression
Accuracy	0.77	0.79	0.76	0.80	0.84	0.85	0.86	0.81	0.79	0.79
Precision	0.54	0.63	0.57	0.59	0.72	0.76	0.79	0.68	0.66	0.65
Recall	0.32	0.30	0.002	0.60	0.56	0.58	0.59	0.40	0.27	0.29
F1-score	0.40	0.41	0.005	0.59	0.63	0.66	0.67	0.51	0.38	0.40
AUC	0.61	0.62	0.50	0.73	0.75	0.76	0.77	0.67	0.61	0.62

2.3

The accuracy is not the best metric to evaluate a classifier. Because this measurement is useless for a extremely imbalanced class. For example we say the 99% of the dataset are female, and the classifier predict the label of all the data is female, although the accuracy is 99% but the classifier is useless. Thus measuring the performance of a classifier only consider the accuracy is not a good idea. Actually the best way to evaluate the performance of a classifier should be the confusion matrix. The combination of accuracy, precision, recall and F1-score. The formula for calculating these performance factors is shown below:

Measure	Formula
Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$
Misclassification rate (1 – Accuracy)	$\frac{FP+FN}{TP+TN+FP+FN}$
Sensitivity (or Recall)	$\frac{TP}{TP+FN}$
Specificity	$\frac{TN}{TN+FP}$
Precision (or Positive Predictive Value)	$\frac{TP}{TP+FP}$

As the graph shown above, all these parameters are related to the confusion matrix. Thus the best way to evaluate the performance of the classifier is to generate the confusion matrix and find all these four parameters.

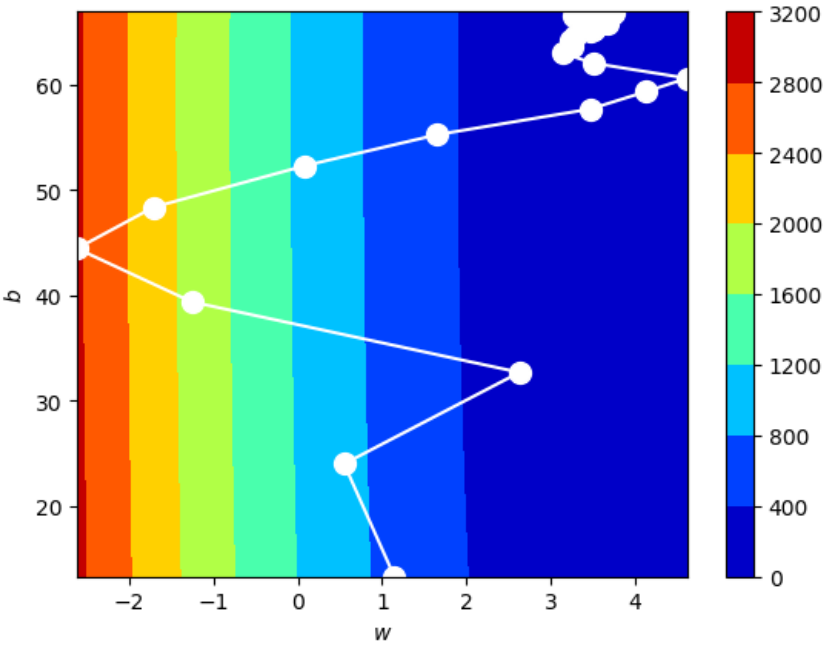
2.3.1 Best two classifier

The best two classifiers are AdaBoost and Gradient Boosting, the performance of these two are very similar. The reason of these two classifiers have the best performance is they all using "Boosting", which is widely. "Boosting" is a good way to help construct a robust model. The term 'Boosting' refers to a family of algorithms which converts weak learner to strong learners.
Gradient boosting classifier, this algorithm is to construct a new base learners which need to be maximally correlated with negative gradient of the loss function. (Which is to cancel the error to make the model more accurate and more robust).
AdaBoost, it fits a sequence of weak learners on different weighted training data. The algorithm assign the same weight to each observation in the first iteration. If the prediction is incorrect, then it gives higher weight to the observation which have been predicted correctly. And the iterative process keeps looping until a limit is reached in the number of models or accuracy. This could also explained why set the estimator numbers to a higher value can lead a better performance for this classifier.

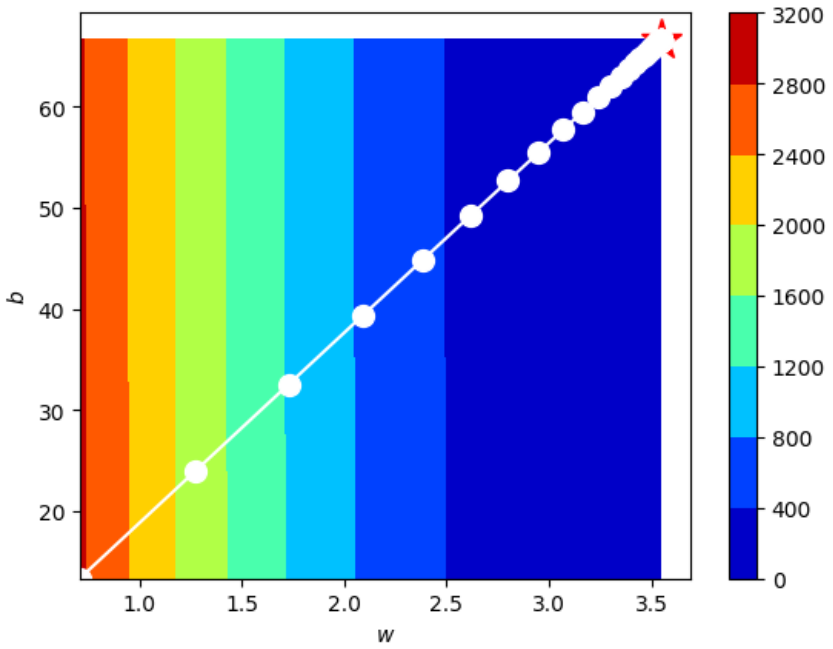
3 Part 3

3.1 1

3.1.1 a



MBGD+MSE



BGD+MSE

The path of gradient descent of BGD+MSE is stable and it is a straight line which go to the local optimal point.
The path of gradient descent of MBGD+MSE is highly unstable, and the path has high variance.
BDG computes the gradient over the entire dataset, averaging over potentially a vast amount of information. Thus BGD is a straight line.
Mini-batch gradient descent is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model coefficients.

3.1.2 b

	BGD+MSE	MiniBGD+MSE	PSO+MSE	PSO+MAE
MSE	2.41	2.62	2.41	2.43
R-Squared	0.83	0.82	0.84	0.83
MAE	1.28	1.32	1.28	1.28

From the table is shown above, the model optimized by PSO+MSE has the highest R-squared value. It also has the smallest MSE and MAE value, which means it is the best model among these four models.The MiniBGD+MSE has the poorest performance. It has the smallerst R-squared value and highest MSE and MAE value.

3.1.3 c

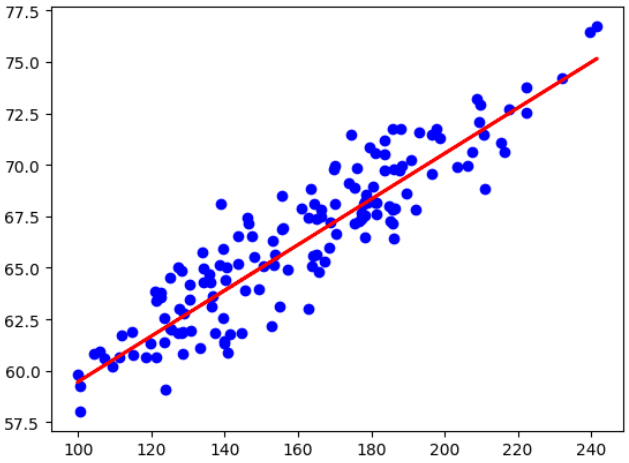


fig :PSO+MAE

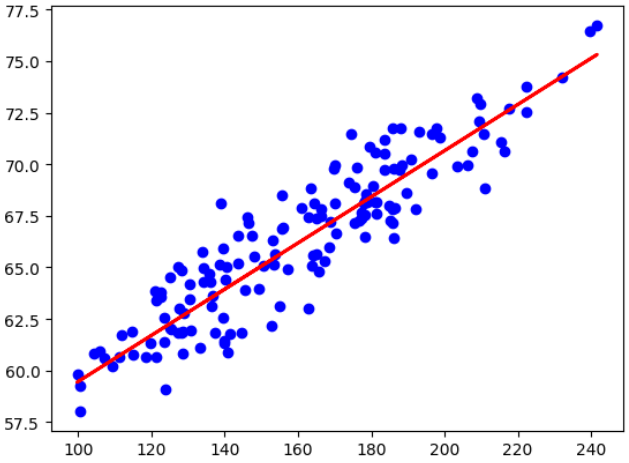


fig :PSO+MSE

3.1.4 d

	BGD+MSE	MiniBGD+MSE	PSO+MSE
Execution time	0.008	0.015	0.360
Execution Speed	Fastest	Medium	Slowest

The BGD is the fastest, because it only use all the sample in each iteration. From the result, MiniBGD is little bit slower than BGD, but it may caused by the dataset is not very large and the other reason maybe the minimum batch size is not very appropriate.Normally for a very large dataset, miniBGD should be faster than constant BGD. Because for constant BGD it only deal with a single sample, but mini-BGD can deal with b samples(b=batch size). With a good vectorization and a appropriate batch size mini-BGD can be very fast.PSO is the slowest because it has a low convergence rate in the iterative process.

3.2

3.2.1 a

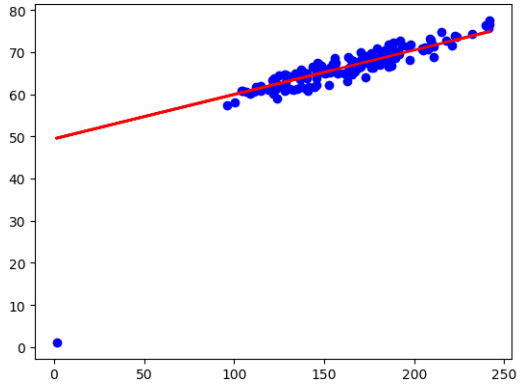


fig :PSO+MAE

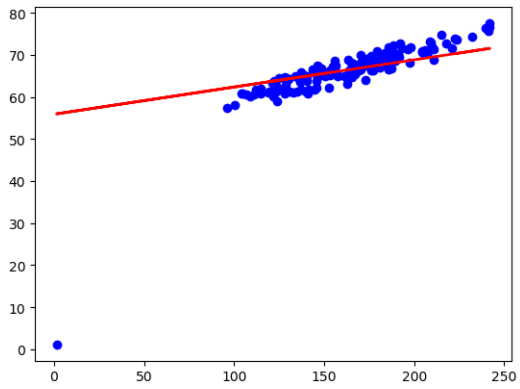


fig: PSO+MSE

3.2.2 b

Comparing to the graphs are shown in 3.1.c, the PSO+MAE is less sensitive to the outlier. Because it does not square the errors in the calculation.

3.2.3 c

We can use mini-BGD to optimize MAE. Mean Absolute Error (MAE) is the average vertical distance between each point and the identity line.
Batch gradient descent refers to calculating the derivative from all training data before calculating an update. But actually MAE can not be well optimized by derivative.