COMP 309 — Machine Learning Tools and Techniques
Assignment 4: Performance Metrics and Optimisation

Name: Karu Skipper
ID: 300417869

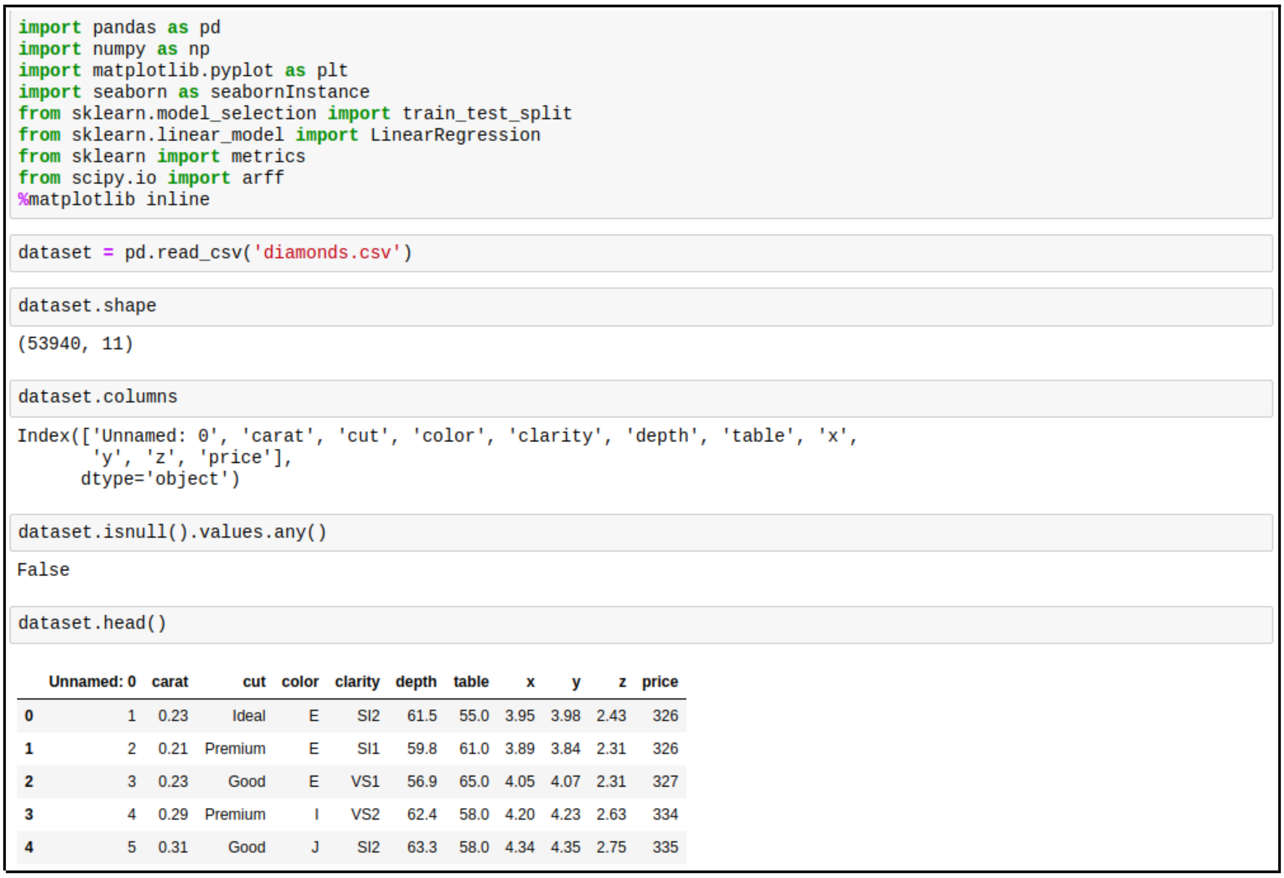
2.1 Part 1: Performance Metrics in Regression [35 marks]

For this section I will be performing regression techniques on the diamonds.csv dataset located in *part1/Part 1 – regression/diamonds.csv* . I will be using the sci-kit learn tool to perform the regression techniques on this dataset. When running the pipeline, it is preferred to use the Pipeline-Tensor-Flow.py on google colab as some algorithms take about 20 minutes locally.

Loading the dataset:
The first step is to import the required sci-kit learn and basic Python libraries that will help us when we apply our regression techniques. Pandas is developed for fast, efficient & easy practical real-world data manipulation and data analysis. Using pandas, we can load in the diamonds.csv file with simply one line “dataset = pd.read.csv(‘diamonds.csv’).

Initial Data Analysis:
I have attatched an image below that helps us analyse some of the initial data. From quickly applying some functions to our dataset, we see there are eleven different attributes to each diamond instance. We can also see there is an *‘Unnamed’* attribute, which labels the number of the listed instance. I will remove this as it doesn’t help our investigation and more importantly, won’t help our regression techniques.

Screenshot of data_analytics.py, this gives us a preview of the information in our dataset



```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as seabornInstance
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn import metrics
from scipy.io import arff
%matplotlib inline
```

```
dataset = pd.read_csv('diamonds.csv')
```

```
dataset.shape
```

```
(53940, 11)
```

```
dataset.columns
```

```
Index(['Unnamed: 0', 'carat', 'cut', 'color', 'clarity', 'depth', 'table', 'x',
       'y', 'z', 'price'],
      dtype='object')
```

```
dataset.isnull().values.any()
```

```
False
```

```
dataset.head()
```

	Unnamed: 0	carat	cut	color	clarity	depth	table	x	y	z	price
0	1	0.23	Ideal	E	SI2	61.5	55.0	3.95	3.98	2.43	326
1	2	0.21	Premium	E	SI1	59.8	61.0	3.89	3.84	2.31	326
2	3	0.23	Good	E	VS1	56.9	65.0	4.05	4.07	2.31	327
3	4	0.29	Premium	I	VS2	62.4	58.0	4.20	4.23	2.63	334
4	5	0.31	Good	J	SI2	63.3	58.0	4.34	4.35	2.75	335

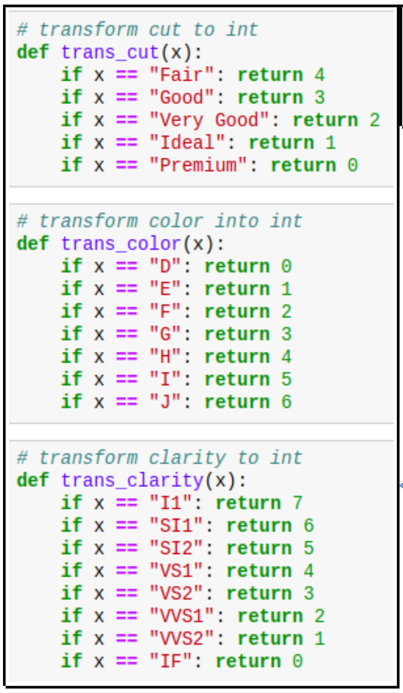
Attributes

- Carat: Carat weight of the Diamond.
- Cut: Describe cut quality of the diamond.
- Color: Color of the Diamond
- Clarity: Diamond Clarity refers to the absence of the Inclusions and Blemishes
- Depth: The Height of a Diamond, measured from the Culet to the table, divided by its average Girdle Diameter.
- Table: The Width of the Diamond's Table expressed as a Percentage of its Average
- X: Length of the Diamond in mm.
- Y: Width of the Diamond in mm.
- Z: Height of the Diamond in mm.
- Price: The Price of the Diamond.

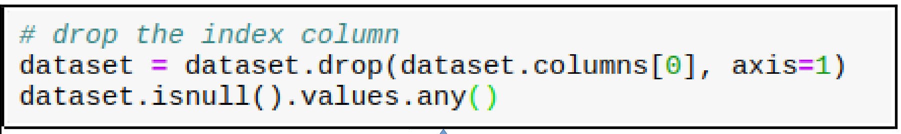
Description of attributes

- Cut in increasing order Fair, Good, Very Good, Premium, Ideal.
- With D being the best and J the worst.
- (In order from Best to Worst, FL = flawless, I3= level 3 inclusions) FL, IF, VVS1, VVS2, VS1, VS2, SI1, SI2, I1, I2, I3
- Qualitative Features (Categorical): Cut, Color, Clarity.
- Quantitative Features (Numerical): Carat, Depth, Table, Price, X, Y, Z.
- Price is the Target Variable.

Preprocess Data:
Drop the index from the dataset using "data = data.drop(data.colums[0], axis=1).
Cut, Color and Clarity all have string values, we need to change this these to numerical values this is demonstrated in the screenshot below.

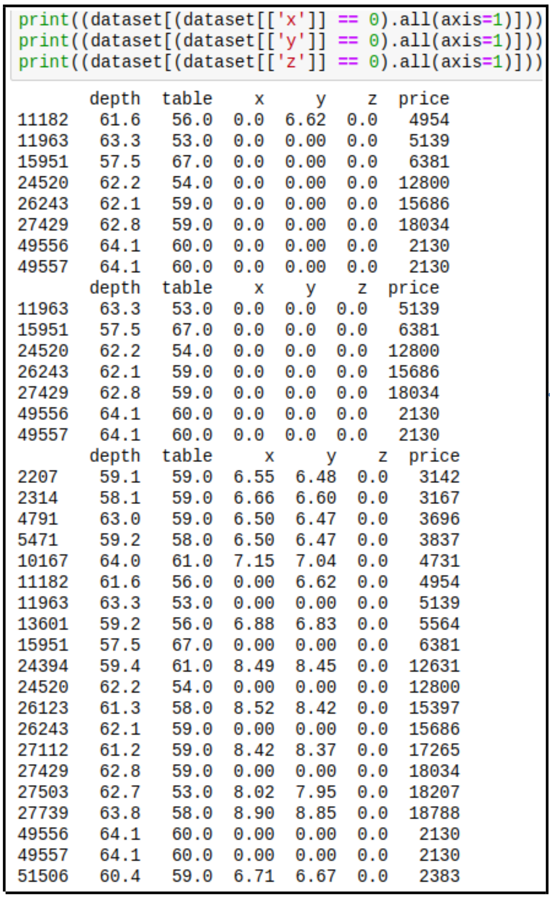


```
# transform cut to int
def trans_cut(x):
    if x == "Fair": return 4
    if x == "Good": return 3
    if x == "Very Good": return 2
    if x == "Ideal": return 1
    if x == "Premium": return 0

# transform color into int
def trans_color(x):
    if x == "D": return 0
    if x == "E": return 1
    if x == "F": return 2
    if x == "G": return 3
    if x == "H": return 4
    if x == "I": return 5
    if x == "J": return 6

# transform clarity to int
def trans_clarity(x):
    if x == "I1": return 7
    if x == "SI1": return 6
    if x == "SI2": return 5
    if x == "VS1": return 4
    if x == "VS2": return 3
    if x == "VVS1": return 2
    if x == "VVS2": return 1
    if x == "IF": return 0
```



```
# drop the index column
dataset = dataset.drop(dataset.columns[0], axis=1)
dataset.isnull().values.any()
```

To begin our pre-processing, we need to drop the first attribute, then run a null value scan to see if there are any missing values in the dataset. This resulted in '*False*,' so we can continue our investigation and convert each result in a hierachical numerical structure. (As pictured on the left). This converts each string result for each attribute in the dataset. We are applying this pre-processing tehcnique for our regression algorithms later. From doing this, we should see if there are any correlations between our attributes a lot easier. This will be described in the later steps.



```
print((dataset[(dataset[['x']] == 0).all(axis=1)]))
print((dataset[(dataset[['y']] == 0).all(axis=1)]))
print((dataset[(dataset[['z']] == 0).all(axis=1)]))

       depth  table    x     y    z  price
11182   61.6   56.0  0.0  6.62  0.0   4954
11963   63.3   53.0  0.0  0.00  0.0   5139
15951   57.5   67.0  0.0  0.00  0.0   6381
24520   62.2   54.0  0.0  0.00  0.0  12800
26243   62.1   59.0  0.0  0.00  0.0  15686
27429   62.8   59.0  0.0  0.00  0.0  18034
49556   64.1   60.0  0.0  0.00  0.0   2130
49557   64.1   60.0  0.0  0.00  0.0   2130
       depth  table    x    y    z  price
11963   63.3   53.0  0.0  0.0  0.0   5139
15951   57.5   67.0  0.0  0.0  0.0   6381
24520   62.2   54.0  0.0  0.0  0.0  12800
26243   62.1   59.0  0.0  0.0  0.0  15686
27429   62.8   59.0  0.0  0.0  0.0  18034
49556   64.1   60.0  0.0  0.0  0.0   2130
49557   64.1   60.0  0.0  0.0  0.0   2130
       depth  table     x     y    z  price
2207    59.1   59.0  6.55  6.48  0.0   3142
2314    58.1   59.0  6.66  6.60  0.0   3167
4791    63.0   59.0  6.50  6.47  0.0   3696
5471    59.2   58.0  6.50  6.47  0.0   3837
10167   64.0   61.0  7.15  7.04  0.0   4731
11182   61.6   56.0  0.00  6.62  0.0   4954
11963   63.3   53.0  0.00  0.00  0.0   5139
13601   59.2   56.0  6.88  6.83  0.0   5564
15951   57.5   67.0  0.00  0.00  0.0   6381
24394   59.4   61.0  8.49  8.45  0.0  12631
24520   62.2   54.0  0.00  0.00  0.0  12800
26123   61.3   58.0  8.52  8.42  0.0  15397
26243   62.1   59.0  0.00  0.00  0.0  15686
27112   61.2   59.0  8.42  8.37  0.0  17265
27429   62.8   59.0  0.00  0.00  0.0  18034
27503   62.7   53.0  8.02  7.95  0.0  18207
27739   63.8   58.0  8.90  8.85  0.0  18788
49556   64.1   60.0  0.00  0.00  0.0   2130
49557   64.1   60.0  0.00  0.00  0.0   2130
51506   60.4   59.0  6.71  6.67  0.0   2383
```

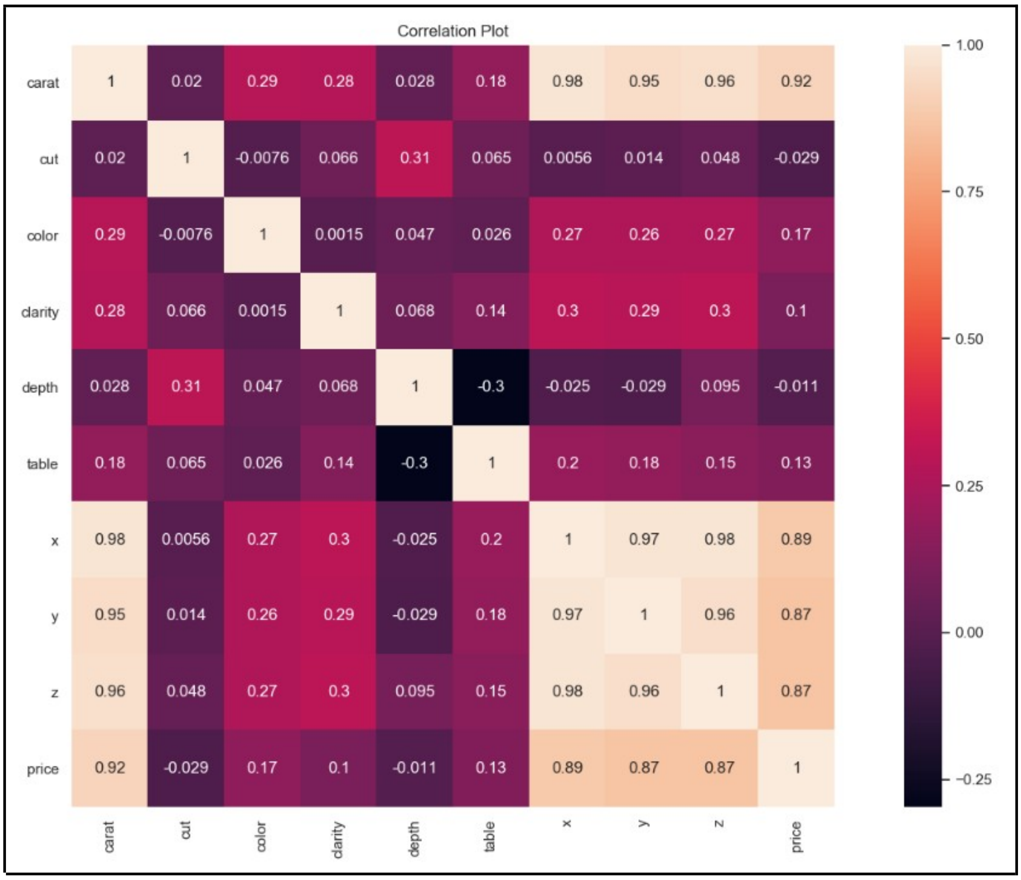
Image of zero checking (data cleasing), this can be found in the data_analytics.py file. But is also applied in my pipeline.py.

Just because there are no ‘*null*’ values present in the dataset, doesn’t mean we shouldn’t check attributes for innacurate values. For example, I have individually checked each x, y, z attribute for the number ‘0.’ This is because you physically cant have a diamond with a dimension of 0. For presentation purposes, I have printed every occurance of this in the dataset. The code I used to drop all 0 occurances is:

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

This will give us a far more accurate result when applying our regression algorithms.

Exploratory Data Analysis:
plt.figure(figsize=(15, 10))
sns.heatmap(dataset.corr(), annot=True)

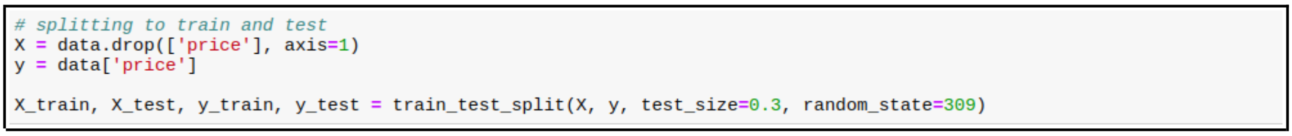


Now let's look at the correlation graph using seaborn heat correlation map:

- We can see x, y, z and carat have extremely high correlations with the price of diamonds.
- Depth has one of the lowest correlations, when comparing to price, we could assume that the depth of a diamond does not affect the price of a diamond, soley because of the close to ‘0’ correlation presented in the heatmap.
- We can also see that table and depth have a strong negative correlation with each other, we may remove this later depending on the performance of our machine learning techniques.

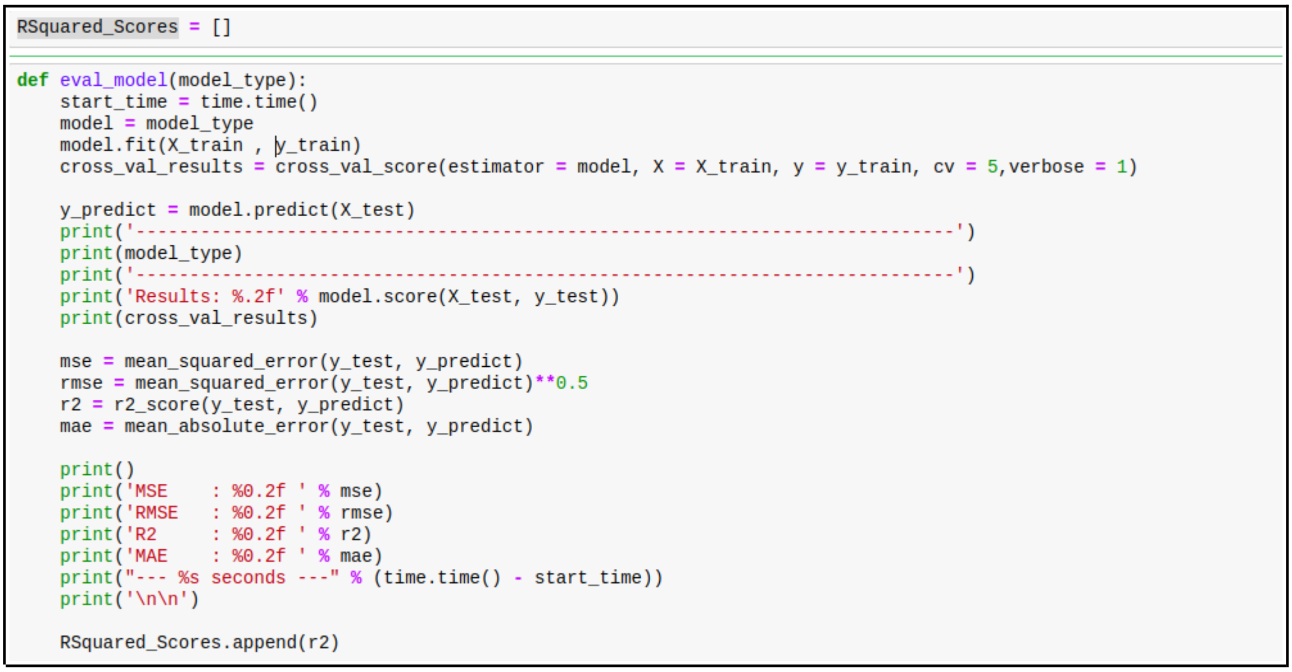
Build classification (or regression) models using the training data

Before we apply our regression algorithm, we must split the data in to the required training and test set. As required by the handout, I have applied the random_state = 0 and the test size to be 30% of the dataset. This is done through a simple function ‘train_test_split(X, y, test_size=0.3, random_state=309)’



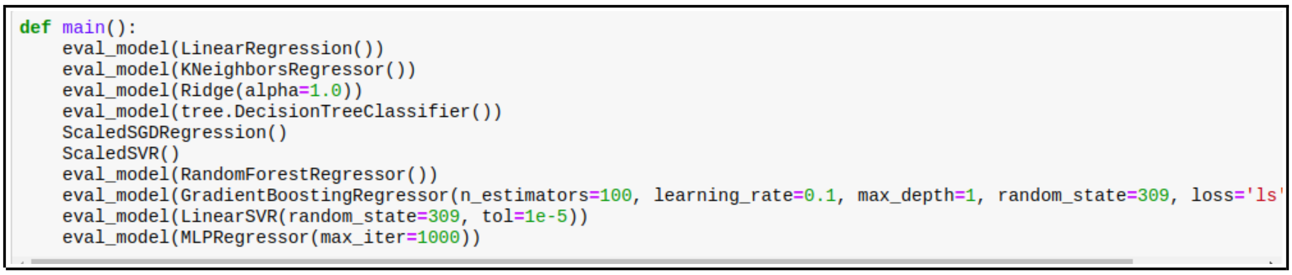
```
# splitting to train and test
X = data.drop(['price'], axis=1)
y = data['price']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=309)
```

We can create an evaluation model by combining common steps, therefor I have created the below function to act on our parsed model_type.



```
RSquared_Scores = []

def eval_model(model_type):
    start_time = time.time()
    model = model_type
    model.fit(X_train , y_train)
    cross_val_results = cross_val_score(estimator = model, X = X_train, y = y_train, cv = 5,verbose = 1)

    y_predict = model.predict(X_test)
    print('--------------------------------------------------------------------------------')
    print(model_type)
    print('--------------------------------------------------------------------------------')
    print('Results: %.2f' % model.score(X_test, y_test))
    print(cross_val_results)

    mse = mean_squared_error(y_test, y_predict)
    rmse = mean_squared_error(y_test, y_predict)**0.5
    r2 = r2_score(y_test, y_predict)
    mae = mean_absolute_error(y_test, y_predict)

    print()
    print('MSE    : %0.2f ' % mse)
    print('RMSE   : %0.2f ' % rmse)
    print('R2     : %0.2f ' % r2)
    print('MAE    : %0.2f ' % mae)
    print("--- %s seconds ---" % (time.time() - start_time))
    print('\n\n')

    RSquared_Scores.append(r2)
```

This applies to most models but not all, bringing us to our main() function which will execute all 10 algorithms when run.



```
def main():
    eval_model(LinearRegression())
    eval_model(KNeighborsRegressor())
    eval_model(Ridge(alpha=1.0))
    eval_model(tree.DecisionTreeClassifier())
    ScaledSGDRegression()
    ScaledSVR()
    eval_model(RandomForestRegressor())
    eval_model(GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, max_depth=1, random_state=309, loss='ls'
    eval_model(LinearSVR(random_state=309, tol=1e-5))
    eval_model(MLPRegressor(max_iter=1000))
```

As you can see both *scaledSVR* and *scaledSGDRegression* require extra work and cannot be passed through the eval_model. Thus, I have created seperate functions for these algorithms to be run along side the rest.

Assess model on the test data.

Model	LinearRegression	Ridge	DecisionTree	SVR	RandomForest
SCORE	0.90	0.90	0.97	0.9416	0.98
MSE	1631602.42	1632206.75	547945.21	949286.03	320793.45
MAE	839.32	839.73	363.80	507.52	285.01
RMSE	1277.34	1277.58	740.23	974.31	566.39
R2	0.90	0.90	0.97	0.94	0.98

Model	GradientBoosting	LinearSVR	MLPRegressor	K-NN	MLP
SCORE	0.91	0.80	0.95	0.95	0.93
MSE	1427945.21	3274041.47	858964.08	752865.31	1128628.63
MAE	699.97	1059.18	541.45	467.84	672.95
RMSE	1194.97	1809.43	926.80	867.68	1062.37
R2	0.91	0.80	0.95	0.95	0.93

Part 2: Performance Metrics in Classification [35 marks]

Step 1. Load Data

There are two methods of approach for loading the data, due to me using google-colab and jupyter notebook. The reason for this is the google TPU runs 4x faster when running the pipelines.

Loading the data via juypter-notebook:
“data_train = pd.read_csv('adultTrain.csv')” &
“dataTest = pd.read_csv('adultTest.csv')”

Loading the data via google-colab:
“drive.mount('/content/gdrive')”
“train_data = pd.read_csv('/content/gdrive/My Drive/Colab Notebooks/adultTrain.csv')” &
“test_data = pd.read_csv('/content/gdrive/My Drive/Colab Notebooks/adultTest.csv')”

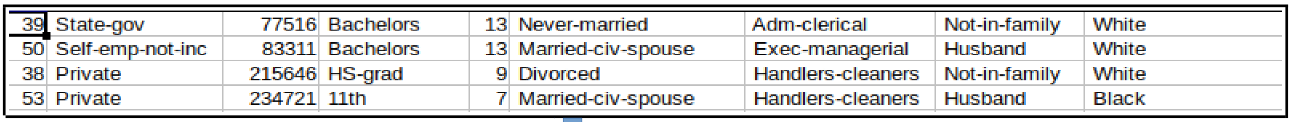
Step 2. Initial Data Analysis

- Age: continuous.
- Work-class: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked.
- Fnlwgt: continuous.
- Education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool.
- Education-num: continuous.
- Marital-status: Married-civ-spouse, Divorced, Never-married, Separated, Widowed,
- Married-spouse-absent, Married-AF-spouse.
- Occupation: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing,
- Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces.
- Relationship: Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried.
- Race: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black.
- Sex: Female, Male.
- Capital-gain: continuous.
- Capital-loss: continuous.
- Hours-per-week: continuous.

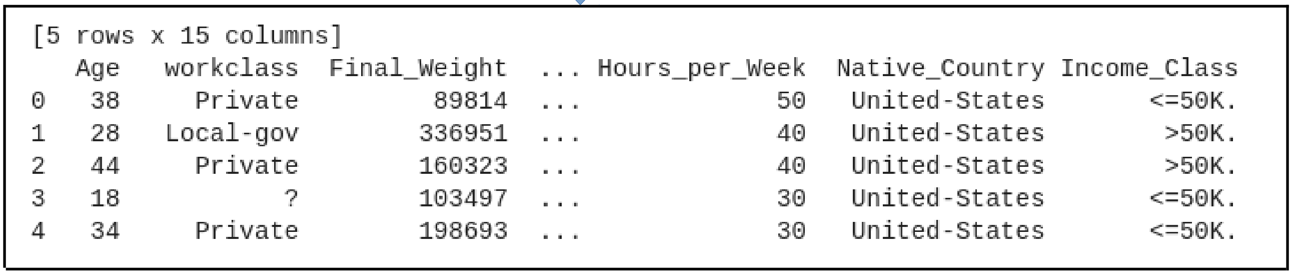
- Native-country: United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad&Tobago, Peru, Hong, Holland-Netherlands.
- Income-Class - >50K, <=50K. “Extraction was done by Barry Becker from the 1994 Census database. A set of reasonably clean records were extracted using the following conditions: ((AAGE>16) && (AGI>100) && (AFNLWGT>1)&& (HRSWK>0))”

Step 3. Preprocess Data

Firstly, we notice both the test and train data files do not have column titles so let's start by assigning a title to each column using dataset.columns command and the column titles discovered in part step 2.

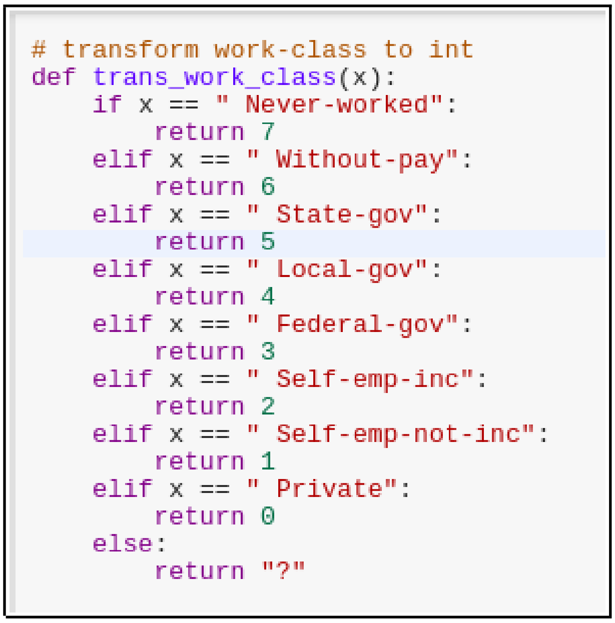


39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White
50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White
38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White
53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black

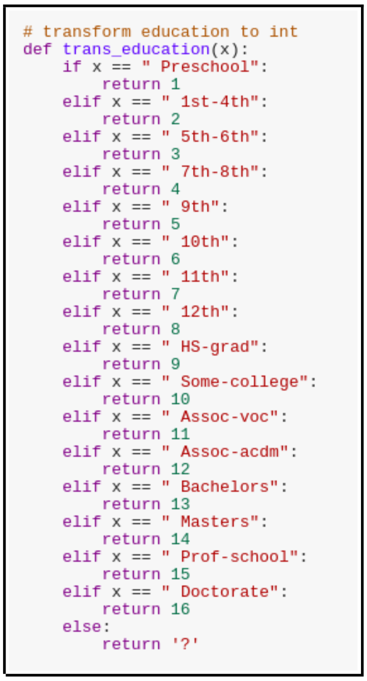


```
[5 rows x 15 columns]
   Age   workclass  Final_Weight  ... Hours_per_Week  Native_Country Income_Class
0   38     Private         89814  ...             50   United-States        <=50K.
1   28   Local-gov        336951  ...             40   United-States         >50K.
2   44     Private        160323  ...             40   United-States         >50K.
3   18           ?        103497  ...             30   United-States        <=50K.
4   34     Private        198693  ...             30   United-States        <=50K.
```

Now also looking more closely at the description of the data in part 2, we see that there are string values we need to convert to numeric values as the decision trees implemented in scikit-learn uses only numerical features and these features are interpreted always as continuous numeric variables. Here are some of the transformations made:



```
# transform work-class to int
def trans_work_class(x):
    if x == " Never-worked":
        return 7
    elif x == " Without-pay":
        return 6
    elif x == " State-gov":
        return 5
    elif x == " Local-gov":
        return 4
    elif x == " Federal-gov":
        return 3
    elif x == " Self-emp-inc":
        return 2
    elif x == " Self-emp-not-inc":
        return 1
    elif x == " Private":
        return 0
    else:
        return "?"
```

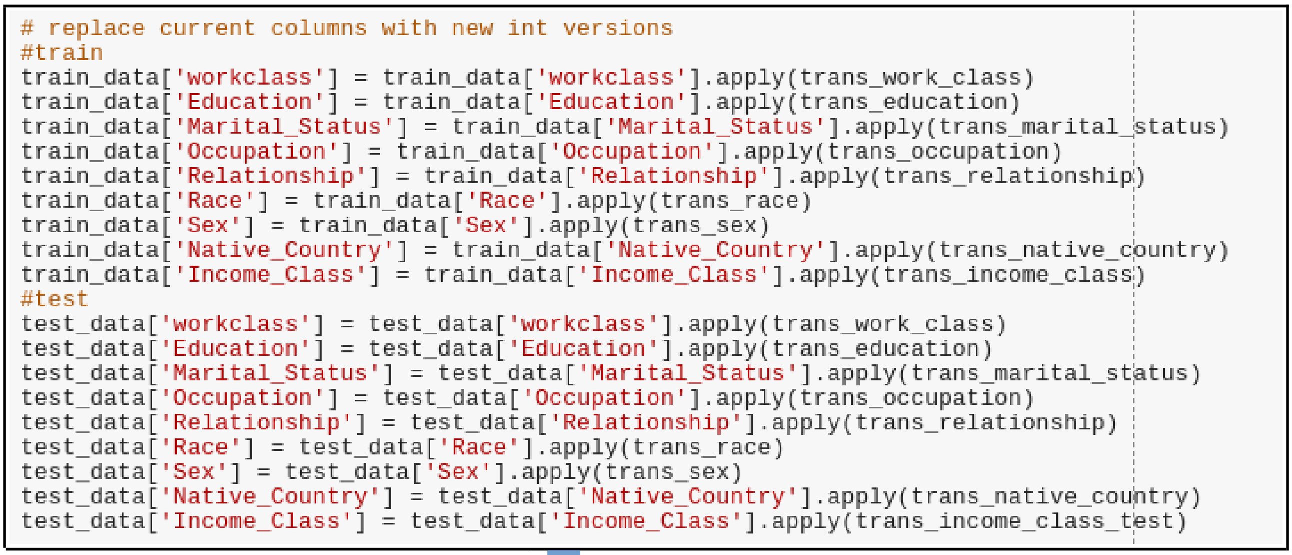


```
# transform education to int
def trans_education(x):
    if x == " Preschool":
        return 1
    elif x == " 1st-4th":
        return 2
    elif x == " 5th-6th":
        return 3
    elif x == " 7th-8th":
        return 4
    elif x == " 9th":
        return 5
    elif x == " 10th":
        return 6
    elif x == " 11th":
        return 7
    elif x == " 12th":
        return 8
    elif x == " HS-grad":
        return 9
    elif x == " Some-college":
        return 10
    elif x == " Assoc-voc":
        return 11
    elif x == " Assoc-acdm":
        return 12
    elif x == " Bachelors":
        return 13
    elif x == " Masters":
        return 14
    elif x == " Prof-school":
        return 15
    elif x == " Doctorate":
        return 16
    else:
        return '?'
```

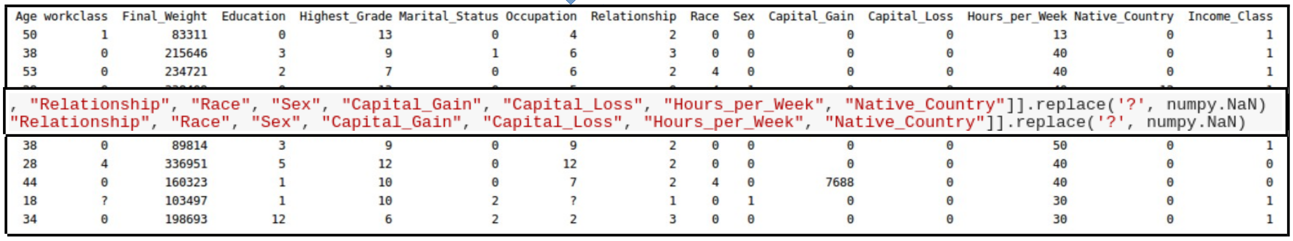
We repeat this process for the other following:

- Marital status
- Education
- Occupation
- Race
- Income class
- Native country

Then calling on the functions we implemented above, we replace all string values within both the train and test set with numeric values respectively as seen below.



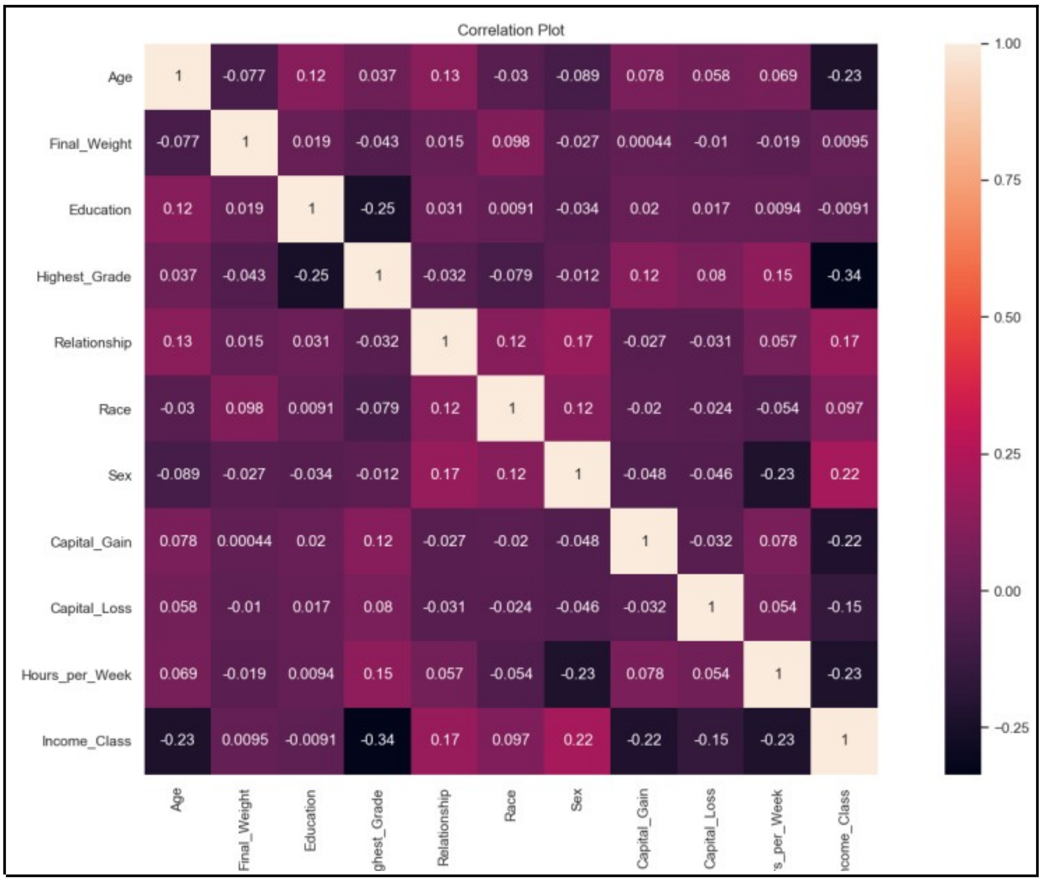
```
# replace current columns with new int versions
#train
train_data['workclass'] = train_data['workclass'].apply(trans_work_class)
train_data['Education'] = train_data['Education'].apply(trans_education)
train_data['Marital_Status'] = train_data['Marital_Status'].apply(trans_marital_status)
train_data['Occupation'] = train_data['Occupation'].apply(trans_occupation)
train_data['Relationship'] = train_data['Relationship'].apply(trans_relationship)
train_data['Race'] = train_data['Race'].apply(trans_race)
train_data['Sex'] = train_data['Sex'].apply(trans_sex)
train_data['Native_Country'] = train_data['Native_Country'].apply(trans_native_country)
train_data['Income_Class'] = train_data['Income_Class'].apply(trans_income_class)
#test
test_data['workclass'] = test_data['workclass'].apply(trans_work_class)
test_data['Education'] = test_data['Education'].apply(trans_education)
test_data['Marital_Status'] = test_data['Marital_Status'].apply(trans_marital_status)
test_data['Occupation'] = test_data['Occupation'].apply(trans_occupation)
test_data['Relationship'] = test_data['Relationship'].apply(trans_relationship)
test_data['Race'] = test_data['Race'].apply(trans_race)
test_data['Sex'] = test_data['Sex'].apply(trans_sex)
test_data['Native_Country'] = test_data['Native_Country'].apply(trans_native_country)
test_data['Income_Class'] = test_data['Income_Class'].apply(trans_income_class_test)
```



```
Age workclass  Final_Weight  Education  Highest_Grade Marital_Status Occupation  Relationship  Race  Sex  Capital_Gain  Capital_Loss  Hours_per_Week Native_Country  Income_Class
 50         1         83311          0             13              0          4             2     0    0             0             0              13              0             1
 38         0        215646          3              9              1          6             3     0    0             0             0              40              0             1
 53         0        234721          2              7              0          6             2     4    0             0             0              40              0             1
 38         0         89814          3              9              0          9             2     0    0             0             0              50              0             1
 28         4        336951          5             12              0         12             2     0    0             0             0              40              0             0
 44         0        160323          1             10              0          7             2     4    0          7688             0              40              0             0
 18         ?        103497          1             10              2          ?             1     0    1             0             0              30              0             1
 34         0        198693         12              6              2          2             3     0    0             0             0              30              0             1
```

```
, "Relationship", "Race", "Sex", "Capital_Gain", "Capital_Loss", "Hours_per_Week", "Native_Country"]].replace('?', numpy.NaN)
"Relationship", "Race", "Sex", "Capital_Gain", "Capital_Loss", "Hours_per_Week", "Native_Country"]].replace('?', numpy.NaN)
```

This is the result of applying the transformed data after this technique. Once this is complete we can identify ‘?’ values and replace them with a numpy.NaN value.



Looking at the correlation graph above we can see the both final_weight and education have very low correlation with the class attribute, we can go ahead and remove these two using “data.drop”

Results:

Model	K-NN	GaussianNB	SVM	DecisionTree	RandomForest
ACCURACY	0.85	0.82	0.83	0.82	0.81
PRECISION	0.84	0.81	0.82	0.82	0.84
RECALL	0.91	0.93	0.95	0.87	1.00
F1	0.88	0.87	0.88	0.85	0.91
AUC	0.77	0.71	0.70	0.76	0.59

Model	AdaBoost	GradientBoost	LinearDiscriminant	MLP	Logistic
ACCURACY	0.86	0.87	0.83	0.84	0.84
PRECISION	0.85	0.86	0.82	0.83	0.83
RECALL	0.94	0.95	0.94	0.95	0.94
F1	0.89	0.90	0.87	0.89	0.88
AUC	0.77	0.78	0.71	0.72	0.74

The best performance metric

The accuracy output of a model is not the best metric to evaluate the classifiers performance. The best way to way to evaluate the performance of a classifier is the confusion matrix. This is a combination of accuracy, recall and F1-score. We must use the four outputs to generate our confusion matrix to draw the best conclusions.

The best two algorithms.
The two best classifiers are Adaboost and GradientBoosting, these our perform the other classifiers and are very similar in results. We can see that both algorithms are using ‘boosting,’ which is a good way to construct a robust model. AdaBoost is adaptive in the sense that weak learners are tweaked in favour of instances misclassified by previous classifiers. Adaboost is sensitive to noisy data and outloers. In some problems in can be less susceptible to the overfitting problem that other learning algorithms. Gradient boosting is used for both regression and classification problems. This algorithm also allows for the optimization of arbitrary differentiable loss functions. This algorithm constructs a new base learner which needs to be maximally correlated with a negative gradient.

Adaboost

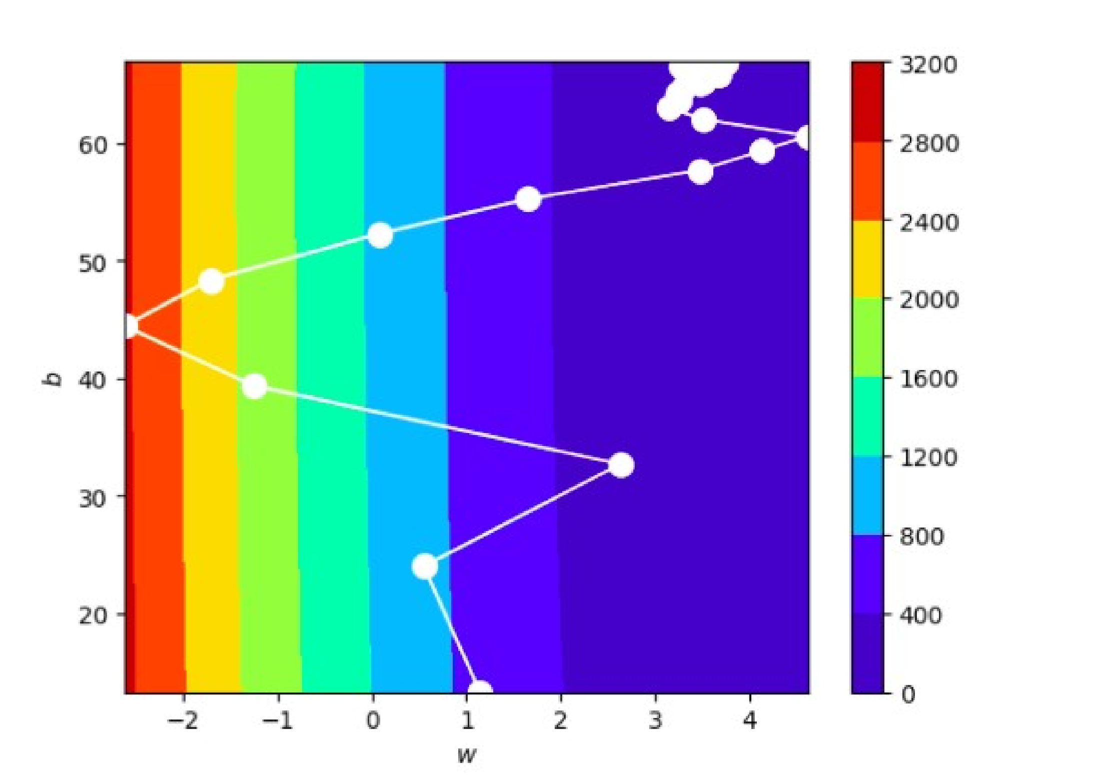
- Accuracy : 0.86
- Precision : 0.85
- Recall : 0.94
- F1 : 0.89
- AUC : 0.77

GradientBoosting

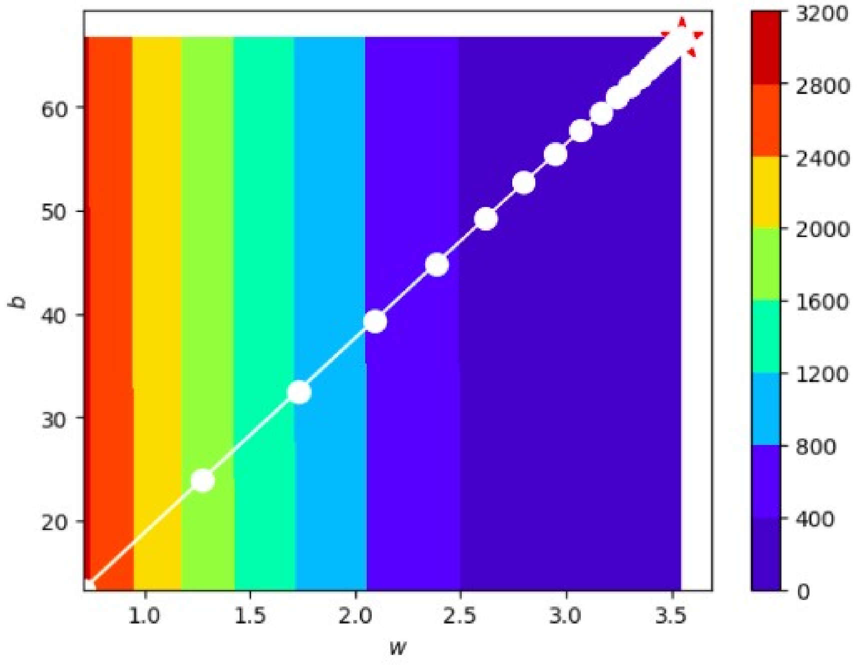
- Accuracy : 0.87
- Precision : 0.86
- Recall : 0.95
- F1 : 0.90
- AUC : 0.78

Part 3: Optimisation Methods [30 marks]

OptimisationMethod.py found at:
/part3/Part 3 – optimisation/template/Template/LinearRegression/src/
I have had to re-locate the utilities folder as the current release of this program is broken and doesn’t work. It would benefit everyone if this was fixed in the next release of the template code.



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.

	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.

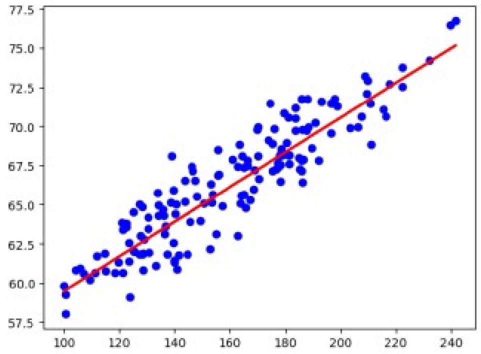


fig :PSO+MAE

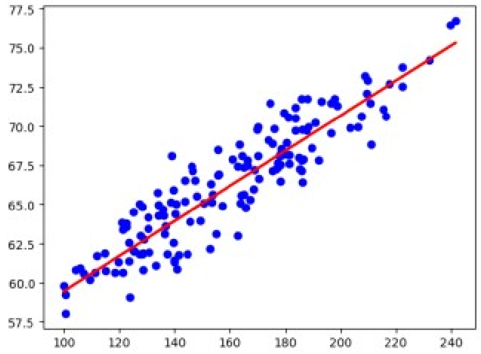


fig :PSO+MSE

	BGD + MSE	MiniBGD + MSE	PSO+MSE
Execution time	0.008	0.015	0.360
Execution speed	Fast	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.

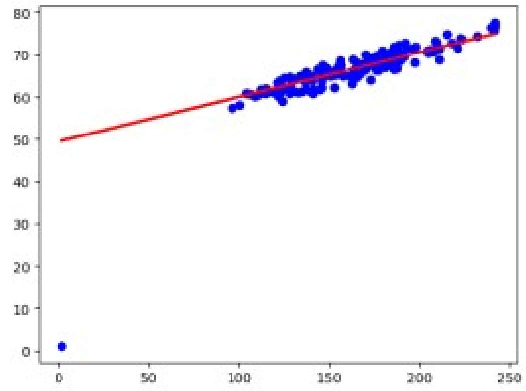


fig :PSO+MAE

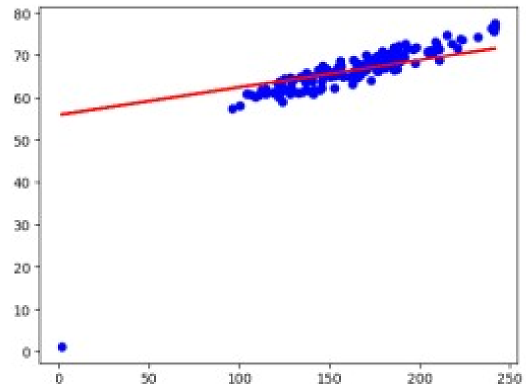


fig: PSO+MSE

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.