COMP309 Assignment 4 Report

Part One:

Parameter Tuning:

Most of the parameter tuning I did was to avoid warnings and convergence problems.

Random Forest Regressor: n_estimators = 100.

This was changed, as mentioned above, to avoid a FutureWarning message when running my code. This variable changes the number of trees in the forest, this is how many separate decision trees the data is split into before applying the regression algorithm

Support Vector Regression: gamma = 'scale'

Again this was a variable that was changed to avoid a warning. With gamma set to scale, the algorithm uses 1 / (n_features * X.var()) as the gamma value. The Gamma value in a Support Vector Machine controls how to measure the similarity of points, and helps control overfitting of new points.

Multi-Layer Perceptron Regression: early_stopping = True, learning_rate_init = 0.5

These values helped fix problems in which the MLP was not converging. *early_stopping* controls whether the MLP will stop early if a validation score, based on a sample of the training data, is not improving. *learning_rate_init* effects the step-size used when increasing the weights. Both of these variables are only used with the default solver.

Preprocessing:

```
   Unnamed: 0  carat      cut color clarity  ...  table     x     y     z  price
0           1   0.23    Ideal     E     SI2  ...   55.0  3.95  3.98  2.43    326
1           2   0.21  Premium     E     SI1  ...   61.0  3.89  3.84  2.31    326

[2 rows x 11 columns]
       Unnamed: 0  carat      cut color clarity  ...  table     x     y     z  price
53938       53939   0.86  Premium     H     SI2  ...   58.0  6.15  6.12  3.74   2757
53939       53940   0.75    Ideal     D     SI2  ...   55.0  5.83  5.87  3.64   2757

[2 rows x 11 columns]
Unnamed: 0      int64
carat         float64
cut            object
color          object
clarity        object
depth         float64
table         float64
x             float64
y             float64
z             float64
price           int64
dtype: object
Missing values: False
```

These are the results of my preprocessing methods. From this I got an example of what my dataset looks like, mostly noting that there is an indexing row which does not provide any information that is actually part of the data.

In addition to this I printed out the data types of each column, with this information you can see that columns ‘cut’, ‘color’ and ‘clarity’ are not integers or floats, but instead ‘object’ this indicates that the data here is some form of categorical variable and will need to be converted into a numeric one, as scikit learn machine learning algorithms only understand numeric inputs.
One final check performed is to see if there are any missing values within the dataset, in this case there are not, so nothing needed to be done.

```
# create label encoder object to be used to convert categories
# may use a different encoder, depending on later results
le = preprocessing.LabelEncoder()
# convert categories into numeric
data['cut'] = le.fit_transform(data['cut'])
data['color'] = le.fit_transform(data['color'])
data['clarity'] = le.fit_transform(data['clarity'])
# remove index column
data.drop(data.columns[0], axis=1, inplace=True)
```

This is my preprocessing code, I use a LabelEncoder to convert the categorical variables into numeric ones, and then the indexing column is dropped.

I also attempted to standardize/normalize/scale the data in an attempt to get better results from the algorithms, however any improvements I noticed were negligible in relation to the values of the results and so I decided to skip scaling. I have left my scaling code as a comment in my file, it uses seaborn to plot and this include is also commented out so I can run the code from the command prompt on a lab workstation rather than via the IDE.

Results:

Algorithm	MSE	RMSE	R^2	MAE	Execution Time
Linear Regression	1896505.75	1377.14	0.88	858.61	0.01
K-Neighbours Regression	907854.64	952.81	0.94	499.23	0.49
Ridge Regression	1896460.59	1377.12	0.88	859.02	0.00
Decision Tree	541403.42	735.80	0.97	362.04	0.17
Random Forest	306255.36	553.40	0.98	272.77	9.35
Gradient Boosting	477092.60	690.72	0.97	374.84	1.08
SGD Regression	3845108818011745.00	62008941.44	-237050121.08	47648587.63	1.13
Support Vector Regression	18242410.61	4271.11	-0.12	2781.57	88.03
Linear SVR	3595759.36	1896.25	0.78	1096.94	0.18
Multi-Layer Perceptron Regression	1136112.89	1065.89	0.93	578.26	5.98

Compare and Conclude:

Clearly something has gone terribly wrong with SGD Regression, and will be ignored in the rest of this conclusion. Looking at the values in the table, I shall also be ignoring the MSE values in favour of RMSE, as it is simply an easier number to look at and compare, and because it is just the square root of the MSE, the comparisons made using RMSE are equivalent to conclusions made on the MSE, even though they represent different things.

Looking at the Mean Absolute Error, which is an absolute measure of fit, and essentially represents how far off from the actual true value that our prediction can be expected to be. The MAE is displayed in the same units as the desired label, in this case it is the price in dollars (I think). From the results in the table, Random Forest, Decision Tree and Gradient Boosting have the lowest MAE, and thus the predictions made by these models are expected to be closer to what the actual value is.

Looking at the Root Mean Squared Error provides us with the measure of the square root of the average of the squared difference between the model's predictions and the true values, this method is more sensitive to outliers. Again, Random Forest, Decision Tree and Gradient Boosting come ahead of the rest of the models when looking at RMSE.

Looking at R-Squared we again see Random Forest, Decision Tree and Gradient Boosting ahead of the pack. This value shows how much the independent variables explain the variance in the model. The closer the R-Squared value is to 1 the better the model is performing, this value doesn't need to be compared to other models as enough can be inferred from how close the value is to 1.

To evaluate which model is performing the best, we can first look at the R-Squared value, followed by both the RMSE and MAE, and in this case the Random Forest is the best performing on the dataset, and Support Vector Regression is performing the worst.

Part Two:
Preprocessing:

```
   0                 1       2           3   ...  11  12              13      14
0  39         State-gov   77516   Bachelors  ...   0  40   United-States   <=50K
1  50   Self-emp-not-inc   83311   Bachelors  ...   0  13   United-States   <=50K
2  38           Private  215646     HS-grad  ...   0  40   United-States   <=50K
3  53           Private  234721        11th  ...   0  40   United-States   <=50K
4  28           Private  338409   Bachelors  ...   0  40            Cuba   <=50K

[5 rows x 15 columns]
   0          1       2             3   ...  11  12              13       14
0  25    Private  226802          11th  ...   0  40   United-States   <=50K.
1  38    Private   89814       HS-grad  ...   0  50   United-States   <=50K.
2  28  Local-gov  336951    Assoc-acdm  ...   0  40   United-States    >50K.
3  44    Private  160323  Some-college  ...   0  40   United-States    >50K.
4  18        NaN  103497  Some-college  ...   0  30   United-States   <=50K.
```

The first thing I do in my preprocessing method is to print the head of both data files, in this we can see that there is at least one missing value, represented by NaN. This value was initially a '?' in the data, and as such I added *na_values=' ?'* into the *read_csv()* arguments for both datasets, including the whitespace next to ? because the *read_csv()* method only removes the ',' leaving any whitespace that was included in the file in it's .csv form

```
pd.read_csv("adult.data", na_values=" ?", header=None),\
pd.read_csv("adult.test", skiprows=[0], na_values=" ?", header=None)
```

In addition during the loading of the file I have also had to state that neither of the files have associated headers for column names, as well as the 'adult.test' file supplied also has "|1x3 cross validator" as the only entry in the first row, hence the argument given *skiprows[0].*

```
0      int64
1     object
2      int64
3     object
4      int64
5     object
6     object
7     object
8     object
9     object
10     int64
11     int64
12     int64
13    object
14    object
dtype: object
```

```
0      int64
1     object
2      int64
3     object
4      int64
5     object
6     object
7     object
8     object
9     object
10     int64
11     int64
12     int64
13    object
14    object
dtype: object
```

As with part one, I then printed the data types of each column out to identify if any variables need to be converted from categorical into numeric data types. I did this to each data set, just in case. As we can see, variables 1, 3, 5, 6, 7, 8, 9, 13 and 14 are all 'object' type and need conversion.

```
convert_to_numeric(d):
# convert columns 1, 3, 5-9, 13-14, from categorical to numeric
le = LabelEncoder()
for i in range(d.shape[1]):
    if i in [1, 3, 5, 6, 7, 8, 9, 13, 14]:
        d[i] = le.fit_transform(d[i])
```

For this I created a helper method to pass each data set into, and use the LabelEncoder again to do the conversion.

The final thing I checked during exploratory analysis was if there were any missing values in the provided data sets, even though I already knew there were from looking at the data head. Handling these were done in two ways.

```
d.dropna(axis=0, thresh=(d.shape[1]/2), inplace=True)
```

Firstly, I used the *dropna()* method to drop any row *axis=0* that had more than the threshold number of missing values, in this case I used half of the variable count *thresh=(d.shape[1]/2).* The additional argument *inplace=True* was to update the current DataFrame, which helps avoid unnecessary object creation and assignment.

```
d.fillna(d.mode().iloc[0], inplace=True)
```

Secondly, I used the *fillna()* method to replace any missing values in each column with the most frequently appearing value in each column *mode().*

Results:

Algorithm	Accuracy	Precision	Recall	F1-score	AUC
KNN	0.78	0.54	0.32	0.40	0.62
Naive-Bayes	0.80	0.64	0.31	0.41	0.63
SVM	0.80	0.96	0.15	0.27	0.58
Decision Tree	0.81	0.59	0.61	0.60	0.74
Random Forest	0.86	0.74	0.60	0.66	0.77
AdaBoost	0.86	0.76	0.59	0.67	0.77
Gradient Boosting	0.87	0.79	0.59	0.68	0.77
Linear Discriminant	0.82	0.69	0.41	0.51	0.68
Multi-Layer Perceptron	0.80	0.73	0.28	0.40	0.62
Logistic Regression	0.80	0.66	0.30	0.41	0.62

The Accuracy measure is not a good metric to solely evaluate the performance of a given classifier, this is because the accuracy is heavily influenced by unbalanced target variables, for example if the split is Y:95/N:5 and if the classifier predicts all the cases to be Y, then its accuracy would be 95% even though the classifier is terrible at classifying an N.

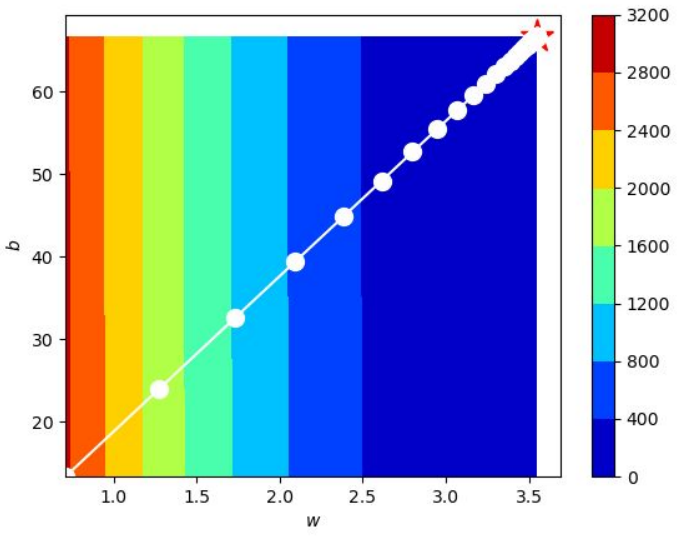
Two Best Classifiers:

Best at Accuracy: Gradient Boost and a tie between AdaBoost and Random Forest
Best at Precision: SVM and Gradient Boosting
Best at Recall: Decision Tree and Random Forest
Best at F1-score: AdaBoost and Gradient Boost
Best at AUC: 3-way tie between Random Forest, AdaBoost and Gradient Boost.

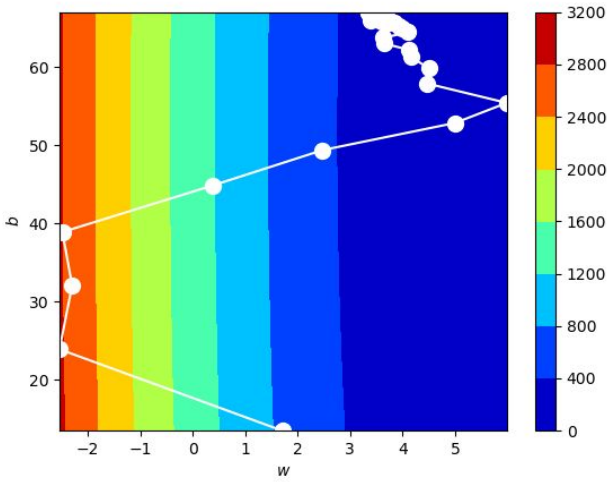
Based on these results the Two best overall would be Gradient Boost and AdaBoost/Random Forest, two of these share the similarity of being ‘boosting’ algorithms. The ‘boosting’ family of algorithms convert weak learners into strong learners, each does it in a different way but the overall result is that the chances of accurate classification increases, and the true positives and true negatives increase, which are the basis of many of the above metrics. Because they both come from the same family of algorithms, if one performs well on a dataset, the other should also.

Part Three:

BGD + MSE



MiniBGD + MSE

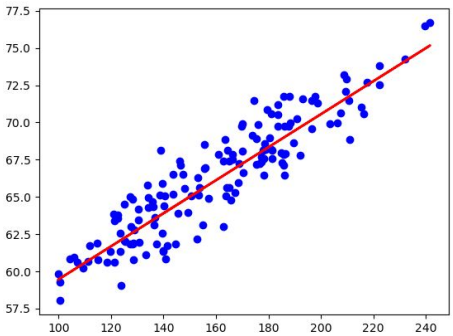


BGE + MSE is a linear decent to the local optima, compared to MiniBGE + MSE in which the path has a high variance due to the ‘online’ adjustment that MiniBGE does because of the splitting the training data into subsets and updating the model, the erratic nature of the plot is due to over-adjustments in this process.

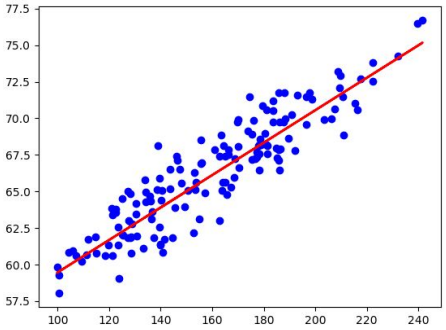
	MSE	R-Squared	MAE
BGD + MSE	2.42	0.84	1.28
MiniBGD + MSE	2.51	0.83	1.29
PSO + MSE	2.41	0.84	1.28
PSO + MAE	2.43	0.84	1.28

The table shows that all the R-Squared values are either identical or very similar, even if due to rounding to 2 decimal places, because of this we have to look at the other metrics for performance analysis. PSO + MSE has the lowest MSE and MAE out of all. MiniBGD + MSE has the poorest performance due to having the highest MSE and MAE

PSO + MAE on Test Set



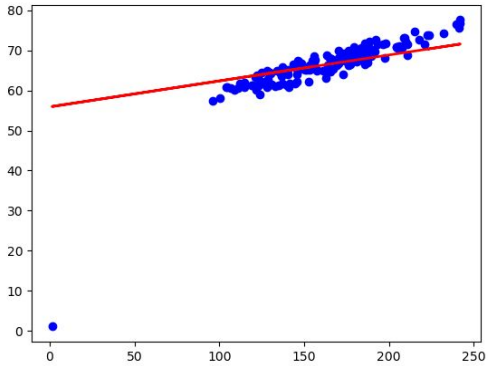
PSO + MSE on Test Set



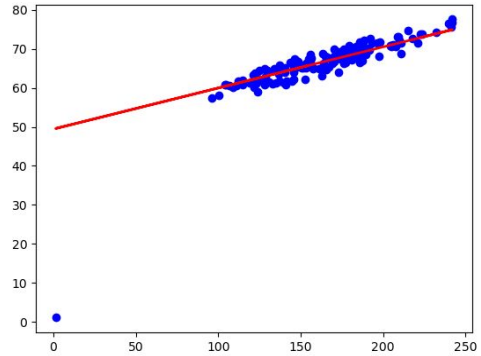
	BGD + MSE	MiniBGD + MSE	PSO + MSE
Execution Time	0.005	0.009	0.307

Because of the size of the dataset, BGD outperforms MiniBGD, whereas on a larger dataset one would expect MiniBGD to be faster, because of how BGD takes the whole dataset as a single sample. PSO is slow due to low convergence rates.

PSO + MSE on Test Set with Outliers



PSO + MAE on Test Set with Outliers



These graphs show that PSO+MSE is affected by the outlier, this is due to the process of MSE squaring the error, thus it is emphasized by the process.