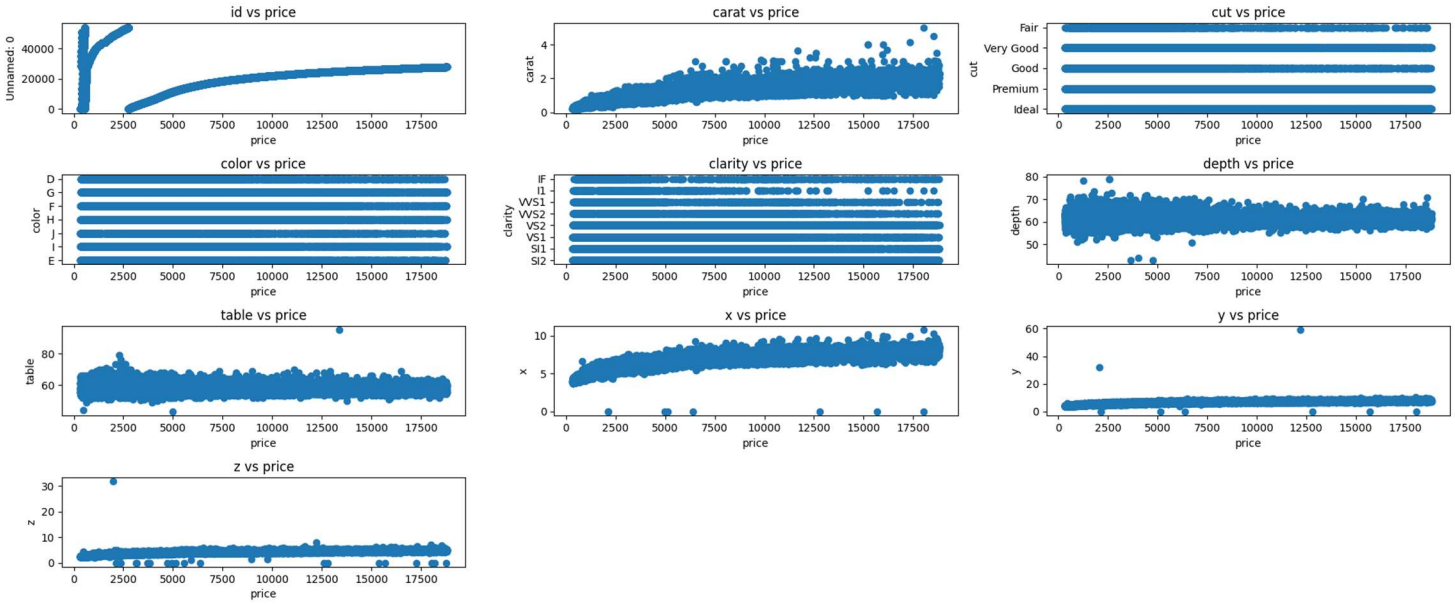
COMP 309 — Machine Learning Tools and Techniques

Assignment 4: Performance Metrics and Optimisation

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

Initial data analysis

The first step in my initial data analysis was to compare the relationships between each variable and the price variable. I first did this using a function to draw scatterplots between the feature variable s and price. The output is shown below, but because the dataset contains 53,940 entries it made the graphs hard to read, and there was no discernible trend for most of the scatterplots, although there does seem to be a mild positive correlation between x and price, and carat and price.



Next, I decided to do some pre-processing of the data. The first thing I did here was I checked whether there were any missing values, of which there were none in the dataset. I then determined the type of each feature. ‘cut, ‘color’ and ‘clarity’ are all string values, which is not great for performing linear regression. After some more research into these variables, I determined that they were ordinal variables, so I converted them into a number representing their place in the category’s ordering, for instance, ‘ideal’ is the best cut, so it was assigned the number 1. Note that I did this by hardcoding a number to represent each string. I later changed to an approach that created binary variables for each string in each category, as this performed better, get_dummies from the pandas library.

Following this, I then decided to generate a correlation matrix between the features and the target variable, to try to get a clearer picture of what was going on. This is shown below but note that the unnamed column is 'id', which should not affect on the price if assigned randomly. Notably, in these correlations, carat, x, y, and z have extremely high correlations to price, yet the depth, which is simply a calculated value based on the x, y, and z values has virtually no correlation with the price.

```
Correlation to price
Unnamed: 0   -0.306873
carat         0.921591
cut           0.053491
color         0.172511
depth        -0.010647
table         0.127134
x             0.884435
y             0.865421
z             0.861249
price         1.000000
```

After this, I then used the data_preprocess function defined in simple_linear_regression.py to split the dataset into test and training parts, split outputs and inputs, standardize the train and test sets, and add an intercept dummy for computation convenience.

Finally, I ran each algorithm on the data, in turn, calculating all required metrics, which are displayed in the table below, highlighting the best and worst results in each column in red and green respectively. Note that all values have been rounded to two decimal places as per the guidelines. Therefore, even though I will talk about there being a three-way tie for the best R-Squared score, when you analyse with 4 decimal places, decision tree is the closest to the perfect 1.00 out of the three. I found that the first try I had at MLPRegression, there weren't enough iterations for it to converge, so I increased the iterations from 500 to 5000, but this made it take a very long time as a result.

Algorithm	Metrics				
	MSE	RMSE	R-Squared	MAE	Execution Time (seconds)
LinearRegression	1317557.23	1147.85	0.92	744.02	0.03
KNeighborsRegressor	658790.44	811.66	0.96	382.93	5.03
Ridge	1317594.86	1147.87	0.92	744.08	0.01
DecisionTreeRegressor	3438.95	58.64	1.00	4.04	0.26
RandomForestRegressor	1824032.44	1350.57	0.89	901.43	2.37
GradientBoostingRegressor	59889.28	244.72	1.00	136.79	4.95
SGDRegressor	1321474.71	1149.55	0.92	752.14	0.15
SVR	11126054.17	3335.57	0.31	1763.70	106.18
LinearSVR	2527626.00	1589.85	0.84	821.74	0.07
MLPRegressor	9242.70	96.14	1.00	53.44	1456.32

The first thing to notice about the table is that in every category except one, DecisionTreeRegressor is the best regression algorithm to use for this dataset. And similarly, in every category bar one, Support Vector regression is by far the worst algorithm to use on this dataset. Although multilayer perceptron achieved the joint best R Squared value in the dataset, it took just over 24 minutes to complete, compared to decision tree which also achieved an R Squared value of 1.00, but in 0.26 seconds, approximately 5500 times faster, and GradientBoostingRegression, getting 1.00 in 4.95 seconds, roughly 300 times faster.

In terms of Mean Squared Error, the best algorithm was DecisionTreeRegressor, followed closely by MLPRegressor. The third algorithm was relatively close, being GradientBoostingRegressor. The rest of the algorithms all had quite high mean squared errors, but interestingly LinearRegression and Ridge came out with very similar mean squared errors

Looking at Root Mean Squared Error, since it is heavily related to Mean Squared Error, there was no change in the rankings from mean squared error, but the values of each algorithm are a lot closer together

Next, for R-Squared error once again DecisionTree came out on top, but this time tied with Multi-Layer Perceptron and GradientBoostingRegression when rounded to 2 decimal places. Interestingly, LinearRegression, Ridge, and SGDRegressor all tied at 0.92, and surprisingly SVR was well below all other numbers at 0.31

For Mean Absolute Error, DecisionTreeRegressor was well below all the rest, as was MLPRegressor. Most algorithms returned a MAE in the high hundreds, between 700-900, although GradientBoostingRegressor only scored 137.

In conclusion, I think that looking at the data, the best regression technique for the dataset would be DecisionTreeRegressor, then MLPRegressor (Although this did take an extraordinarily long time to process), then GradientBoostingRegressor, and in fourth I would place KNearestNeighbours. I would say that SVR was clearly the worst technique for the dataset, followed by RandomForestRegressor, then Ridge and LinearRegression.

2.2 Part 2: Performance Metrics in Classification [35 marks]

Initial Data Analysis

This part of the assignment focuses on the adult.data and adult.test files. For my initial analysis, I located the source repository on the UCI Machine Learning Repository and read into the background and context of the data to help me better understand it. I then used Excel to open both files to make it clearer to see what was happening, and then used python code to import the headers for each file when I loaded them into python. These attribute names were sourced from the UCI website.

When looking through the data, I noticed several columns had missing values, or simply question marks which I treated like missing values. After further investigation, I noticed that every column which had missing data was categorical. After some deliberation, I decided to use imputation to fill these missing values, by simply taking the most common answer in that column. I know this method

of imputation has its flaws as it introduces bias and reduces variability, but since in my opinion the amount of missing data is low, I don't believe this will negatively affect the final results from the algorithms.

After this, I separated both the training and the test sets into the dependent and independent variables. I noticed that the test set had each line finishing with a full stop, so I had to code a line to remove this, as it was making the variable different to the ones in the training set, that is ' >50K' and ' >50K.'. This caused problems when I tried to predict, as the model had never trained on ' >50K.' so didn't know what to do. Removing the full stop fixed this error

Next, I converted each categorical variable into several binary variables, although I ran into a problem with this, as in the adult.test file, there was no one with the native-country 'Holand-Netherlands', which lead to there being more columns in the training set than the testing set, which caused an error. To rectify this, I combined the two datasets and then created the binary variables, before separating them again.

Another major problem was that, to begin with, RandomForestClassifier performed by far the worst out of all of them. It is worth noting that all classifiers underwent the same method of fitting, predicting, and analysing so this was not a fault of the code which was doing the calculations. This also generated a warning in the python console when I ran it. It appears that Precision was zero, and since precision is the sum of true and false positives, it means that there were exactly 0 true positives and 0 false positives. This then means that RandomForestClassifier simply classified everything as negative, which in this context means the person's income is less than or equal to 50K. I searched online but was unable to come up with a valid explanation as to why this classifier did this, and all the other nine ran fine on the same code. I finally realised whilst writing the report that the reason it failed was that the maximum depth of the tree had been initialised to 2, which is very little. I raised it to 20 and the algorithm then ran fine

One of the last problems I encountered with this section was the fact that the function 'metrics.roc_auc_score' required numeric input, so I created a separate set of data series for this by taking the predicted and correct incomes and changing ' >50K' to 1 and ' <=50K' to 0.

Results

Algorithm	Metrics					
	Classification Accuracy	Precision	Recall	F1-score	AUC	Execution Time (seconds)
KNeighborsClassifier	0.78	0.55	0.32	0.40	0.62	5.22
GaussianNB	0.80	0.64	0.30	0.41	0.63	0.57
SVC	0.78	0.99	0.09	0.16	0.54	169.41
DecisionTreeClassifier	0.80	0.84	0.20	0.33	0.60	0.72
RandomForestClassifier	0.80	0.99	0.16	0.27	0.58	5.18
AdaBoostClassifier	0.81	0.98	0.22	0.36	0.61	10.73
GradientBoostingClassifier	0.81	0.99	0.18	0.31	0.59	8.43
LinearDiscriminantAnalysis	0.77	0.95	0.02	0.05	0.51	1.49
MLPClassifier	0.78	0.55	0.32	0.41	0.62	34.23
LogisticRegression	0.80	0.69	0.26	0.38	0.61	0.71

The table above details my results for all 10 algorithms across the 6 metrics. I have colour coded this table to make it easier to read. Since the last question requires me to find the best two in each category, I have coloured the best score in the category (at two decimal places) dark green, and if it

wasn't a tie then I have shaded in a lighter green the second place for the category. Similarly, I have done the same thing for the worst results per category, but with dark red and orange, again shading in the ties.

One thing that struck me immediately about these results was that for several of the categories all algorithms returned similar results. For classification accuracy, there was a range of only 0.04 or 4% of the data, and similarly, AUC was quite closely grouped too, having a range of only 0.12. I also noticed that most algorithms returned high precisions and low recalls, which makes sense as precision and recall tend to be inversely related.

To answer the question of whether accuracy is the best performance metric, I personally think it depends on the context. Since precision and accuracy are inversely related, we need to decide whether it is more important that we detect every possible positive case, or whether we want to be sure that only true positives are found to be positive. For instance, in a hospital setting, we are much more likely to want to detect every possible case when testing for something like cancer. Therefore, we allow a higher rate of false positives as it is much better to tell someone they have cancer when they don't than to tell someone they don't have cancer when they do. In this scenario, we are more likely to want to use recall as the metric to ensure minimal false negatives occur. There are other scenarios where low recall and high precision are preferred, such as when we don't want a false positive, for instance, we might have an algorithm to determine whether someone is old enough to buy alcohol at our store. It is critical that we don't allow false positives, i.e. a person underage to buy, as this is breaking the law, so we would rather turn away a legal customer than permit an illegal one.

Alternatively, we could use the F1 score if our situation requires a more even balance of precision and recall. The F1 score is simply the harmonic mean of precision and recall and therefore is a balanced metric to use. We also have the AUC metric, Area Under the Curve. The curve is the Receiver Operating Characteristic curve. AUC is in general a measure of the average diagnostic accuracy of the test. Finally, we have classification accuracy, which gives us a general overview of how many instances were correctly classified. If we don't care about false negatives and positives, and just want to correctly classify as many instances as possible, this is a great metric to use for that scenario.

I will now look at the two best algorithms according to the main four performance metrics: Precision, Recall, F1-score, and AUC, to conclude my part 2.2 analysis, by referencing the table shown at the start of the results section.

In terms of precision, at the second decimal place, three algorithms have tied for the highest precision, these being SVC, RandomForestClassifier, and GradientBoostingClassifier.

In terms of recall, the best two algorithms are KNeighboursClassifier and MLPClassifier

For F1-score, the top two algorithms are GaussianNB and MLPClassifier

Finally, for AUC the top algorithm is GaussianNB, and two algorithms are tied for second, KNeighboursClassifier and MLPClassifier.

This analysis shows quite a range of different algorithms coming first over the categories, only three algorithms did not place in the top two for any of these categories. This is in part due to what was discussed further up, that precision and recall are inversely correlated, so the algorithms that performed very highly on that metric scored low on recall, and vice versa, the two best-performing algorithms on the recall metric were the two worst-performing algorithms on the precision metric.

Also, it is worth noting that even though SVC was one of the best-performing in terms of precision, it was one of the worst-performing in every other category. Although there appear to be differences between the metrics, both the F1-score and the AUC metric both have the same top-performing algorithms, suggesting that these two metrics might have more commonality than other metrics.

2.3 Part 3: Optimisation Methods [30 marks]

In this section, I implemented the mini-batch gradient descent optimiser and then optimised the four algorithms for linear regression. The algorithms were Batch Gradient Descent + Mean Squared Error (BGD+MSE), Mini-Batch Gradient Descent + Mean Squared Error (MiniBGD+MSE), Particle Swarm Optimisation + Mean Squared Error (PSO+MSE), and finally Particle Swarm Optimisation + Mean Absolute Error (PSO+MAE). I have collected a series of graphs for each one and also collected a few performance metrics for each option. I will display the graphs for each algorithm here, and then discuss them further on in the piece. I will refer to each figure by its number assigned in its title.

Fig 1: Graphs for BGD_MSE

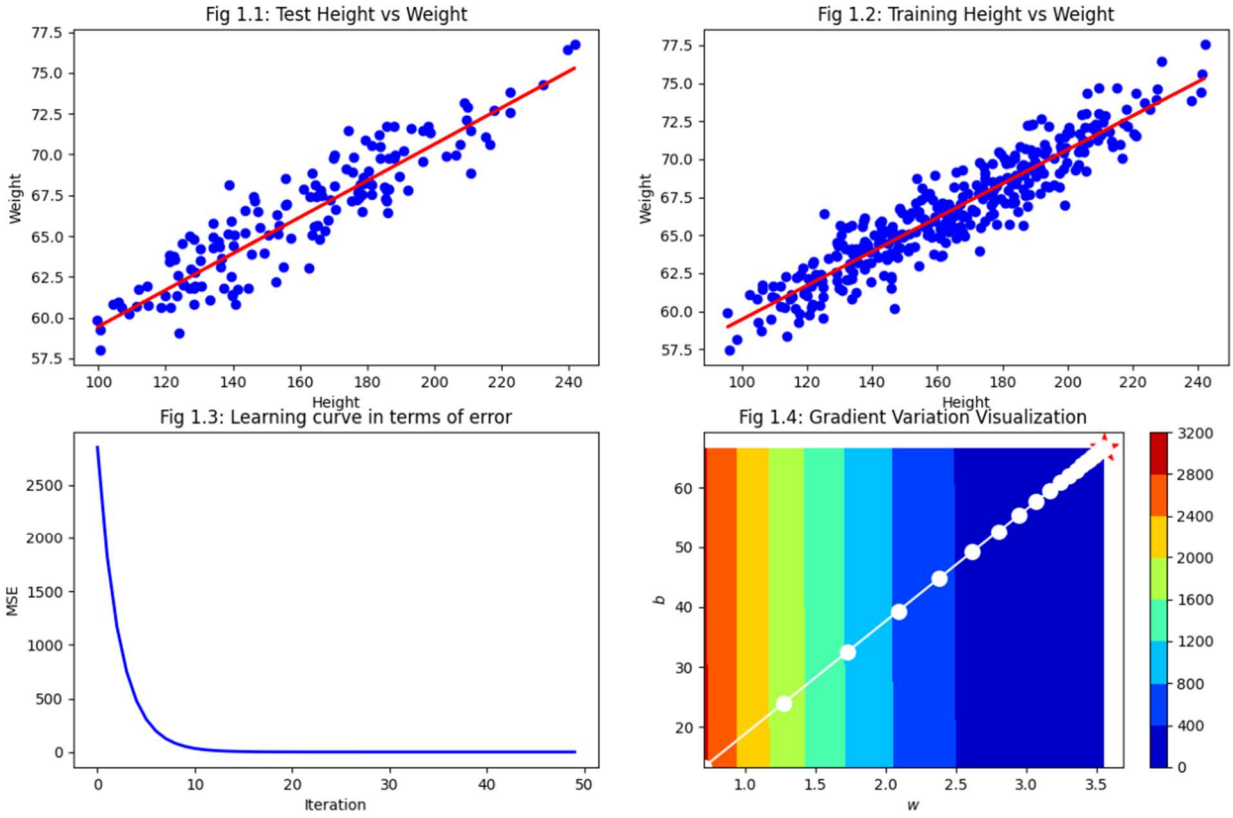


Fig 2: Graphs for MiniBGD_MSE

Fig 2.1: Test Height vs Weight

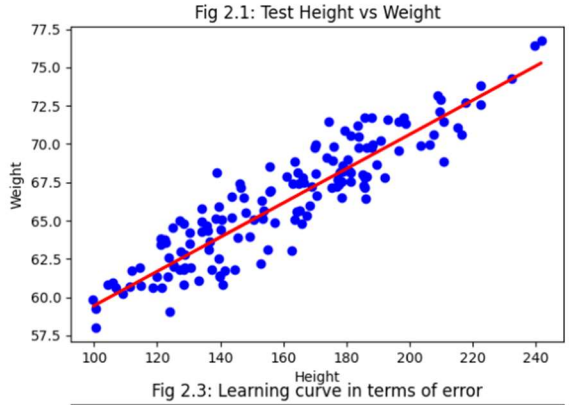


Fig 2.2: Training Height vs Weight

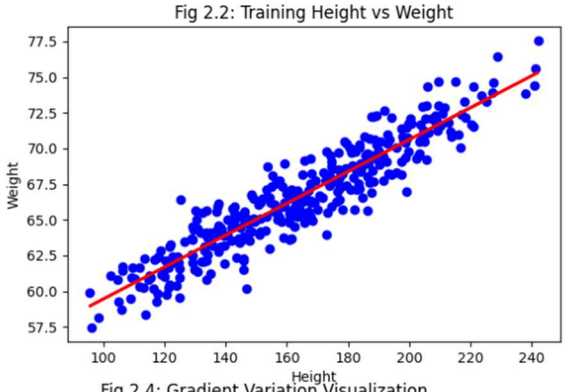


Fig 2.3: Learning curve in terms of error

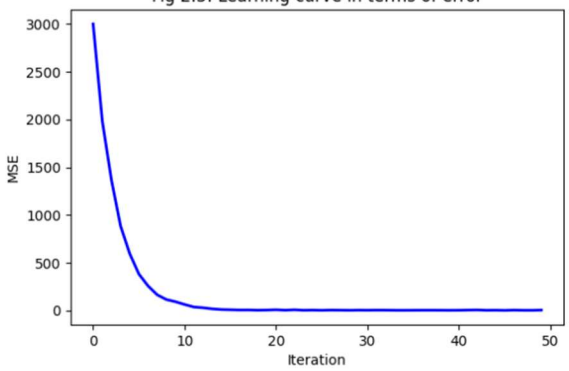


Fig 2.4: Gradient Variation Visualization

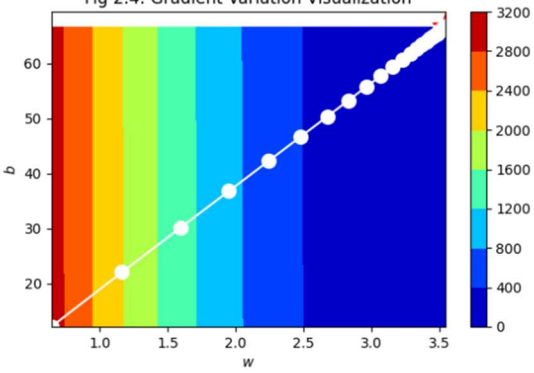


Fig 3: Graphs for PSO_MSE

Fig 3.1: Test Height vs Weight

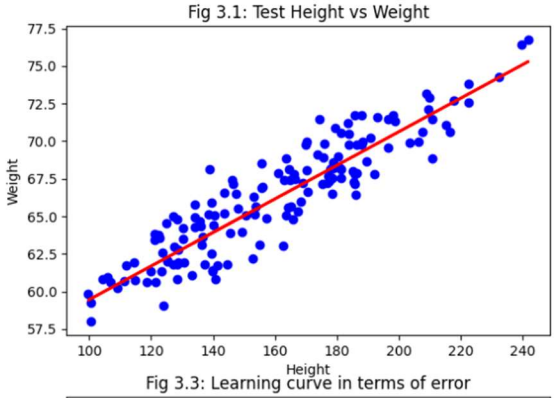


Fig 3.2: Training Height vs Weight

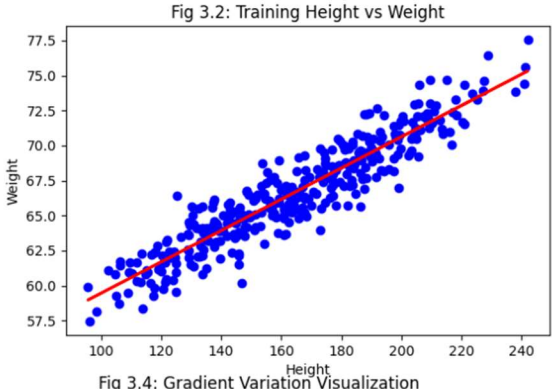


Fig 3.3: Learning curve in terms of error

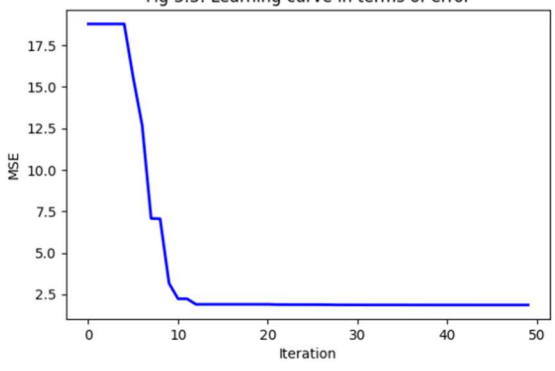


Fig 3.4: Gradient Variation Visualization

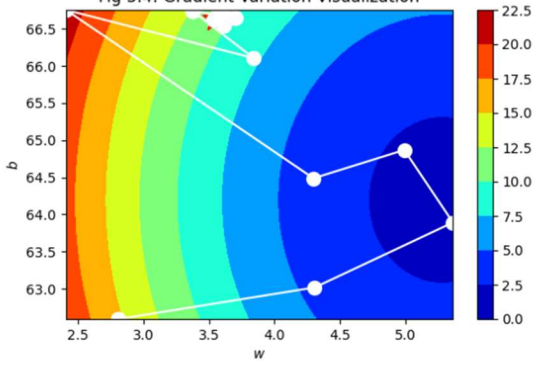
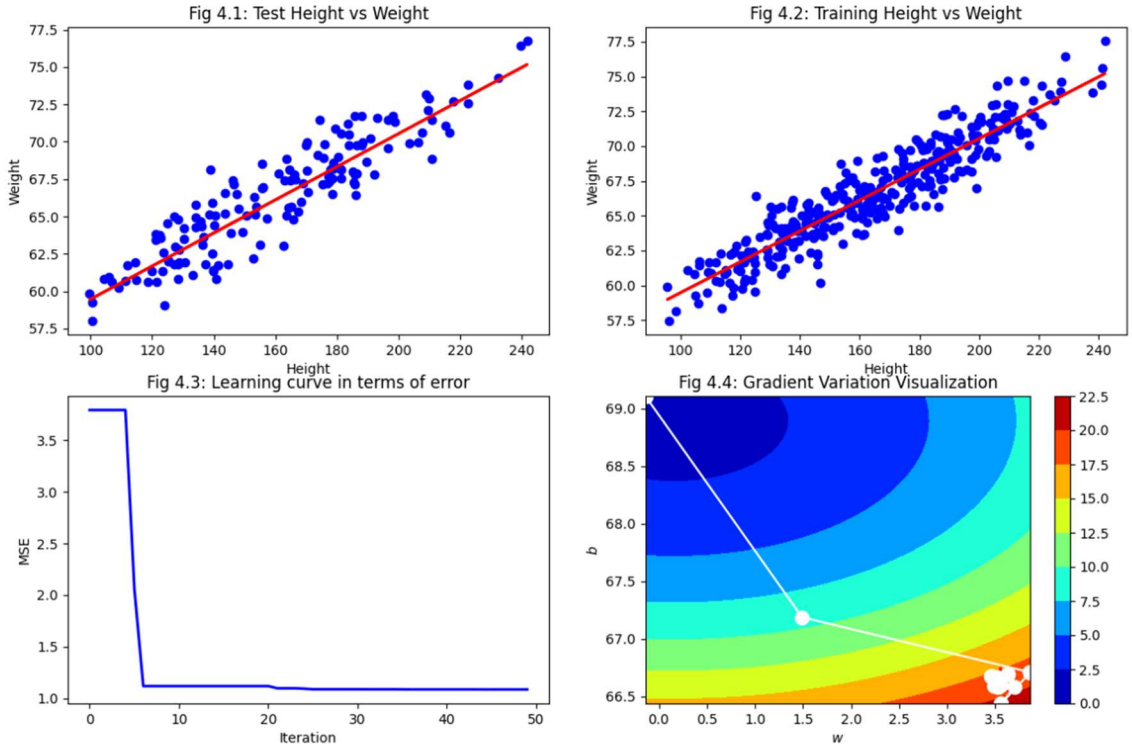


Fig 4: Graphs for PSO_MAE



Question 1a

The two plots that are referenced in this question are Fig 1.4 and 2.4. From my analysis, I cannot see a visual difference between the gradient descent curves for Batch Gradient Descent and Mini-batch Gradient Descent. I originally thought that the MiniBGD gradient descent curve would be less of a straight line than the BGD gradient descent curve, and have a slight 'zig-zag' type pattern to it, but for this dataset that does not appear to be the case. I was originally confused by this, as I couldn't understand what went wrong with my function, however, I believe my function is correct and the graphs are accurate. I checked this with a tutor to confirm. This leaves me to conclude that because the differences in the results are very small, as I will discuss next, there simply was not enough difference to make a meaningful change to the gradient descent curve.

Question 1b

The performance metrics are displayed in the table directly below this paragraph. Once again, I have highlighted in green the best performing algorithm(s) and in red the worst-performing algorithm. Note that although the assignment asked for 2 decimal places, I have taken the liberty of extending it to 3 so that a clearer difference between the algorithms emerges, as they all performed very similarly. I will discuss them underneath the table below.

Algorithm	Metrics				
	R-Squared	MSE	RMSE	MAE	Execution Time (seconds)
BGD+MSE	0.836	2.416	1.554	1.280	0.002
MiniBGD+MSE	0.836	2.417	1.555	1.280	0.036
PSO+MSE	0.836	2.415	1.554	1.280	0.356
PSO+MAE	0.835	2.429	1.559	1.284	0.259

Compared to the other two tables I have discussed, this one is less insightful. This is because all of the algorithms performed similarly across all four metrics, however, there are some things worth noting. Firstly, even though it was by a small margin, the worst algorithm according to all four performance metrics was PSO+MAE. Although it is worth noting that this was not the slowest algorithm, which was PSO+MSE. The best algorithm overall I would say was BGD+MSE, which had the best score for three out of four metrics, and was the fastest algorithm by far. They all came out with a very similar R-Squared, which was identical at the two decimal place margin, as was MAE. MSE was the most different of all the four metrics, simply because squaring numbers exacerbates the difference between them.

Question 1c

See figures 3.1 and 4.1

Question 1d

The fastest algorithm was BGD+MSE, and this was 18X faster than the next fastest algorithm, MiniBGD. The third fastest algorithm was PSO+MAE, with the slowest algorithm being PSO+MSE. These results are in line with the shape of the gradient descent curves produced in this question, as the straightest gradient descent curve produced the fastest time and vice versa for the slowest time. The PSO algorithms were significantly slower than the gradient descent algorithms, as they focus on multiple particles at once which is more computationally intensive than gradient descent methods.

Question 2

The following figures are for PSO+MSE and PSO+MAE, but with the dataset being part2outliers.csv, as opposed to the original graphs which were based on part2.csv.

Question 2a

The generated figures for this question are fig 5.1 for PSO+MSE with outliers, and fig 6.1 for PSO+MAE with outliers.

Fig 5: Graphs for PSO_MSE

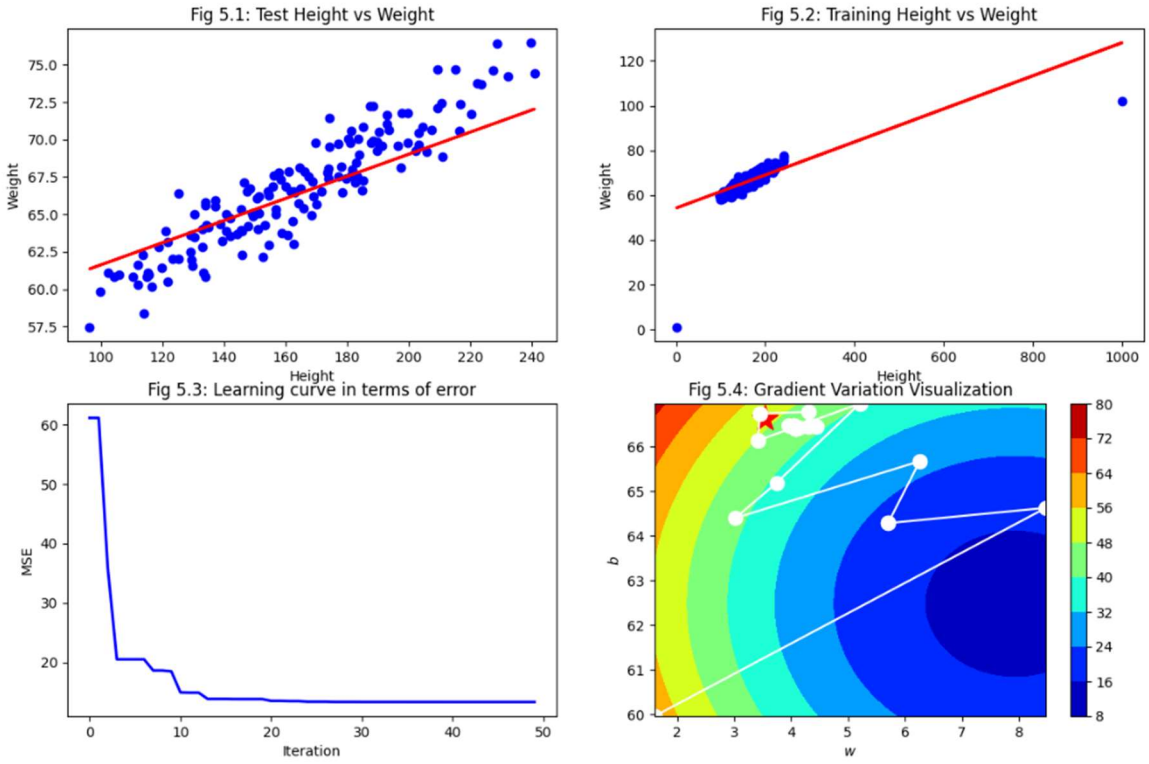
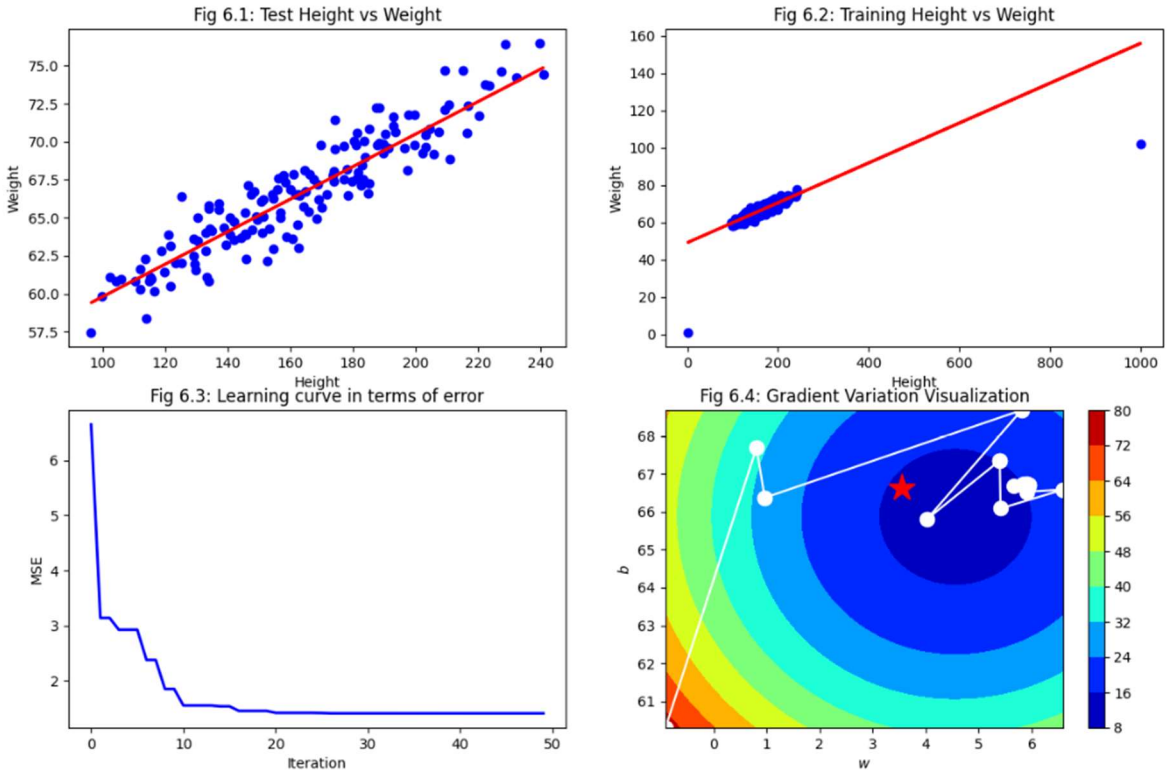


Fig 6: Graphs for PSO_MAE



Question 2b

The figures which are in response to this question are fig 3.1 for PSO+MSE without outliers, fig 4.1 for PSO+MAE without outliers, fig 5.1 for PSO+MSE with outliers, and fig 6.1 for PSO+MAE with outliers. Immediately just looking at fig 5.1 and 6.1 we can see a difference, without even looking at the original plots. In fig 5.1, the gradient is significantly flatter than in fig 6.1, and both of them are flatter than either of the original plots. This tells me straight away that PSO+MSE is more sensitive to outliers as there has been a bigger shift in the gradient than in PSO+MAE.

The reason that the gradient shift is indicative of sensitivity is that each outlier will shift the regression line due it being far away from all other points by its nature. Given that the two outliers in the dataset lie well below the rest of the points, as evidenced by fig 5.2 and fig 6.2, a model less sensitive to outliers will shift the regression line by less than a model more sensitive to it. Also, it is worth noting that MSE involves squaring values which exponentially increases the weight of outliers as opposed to MAE which doesn't

To summarise, PSO+MAE is less sensitive to outliers than PSO+MSE.

Question 2c

I believe that we should not use batch gradient descent and mini-batch gradient descent to optimise mean absolute error. This is because the cost function for gradient descent methods needs to be differentiable, and mean average error is not fully differentiable at a point, so using MAE for the loss function is unacceptable for this optimiser.