ZZSC5836 2: Linear Regression Modelling

Amanjit Gill

May 21, 2023

1 Introduction

In order to avoid the invasive and time-consuming process usually required in order to determine the age of an abalone, alternatives have been proposed that involve the use of machine learning with data that are relatively non-invasive to collect, such as length and weight.

This analysis proposes to assess the suitability of one such method, linear regression. It employs a well-known dataset (refer to the sources at the end of Section 2) upon which many similar attempts have been made. The bulk of the computations have been completed by using the `NumPy` Python package to implement the gradient descent numerical method, with some assistance from `scikit-learn` to verify correctness.

2 Data Processing

2.1 Cleaning

The features in the dataset are entirely numerical with the exception of `sex`. In order to allow its inclusion as an input to a linear regression model, it must be converted as shown below. The Python code that performed this conversion is also given.

- M (Male) = 0
- F (Female) = 1
- I (Infant) = 2

Listing 1: Cleaning the dataset.

```
import pandas as pd
abalone = pd.read\_csv("abalone.data", header=None)
abalone.iloc[:,0] = abalone.replace("M", 0).replace("F",
    1).replace("I", 2)
```

2.2 Correlation Map

The number of rings found on an abalone gives an indication of its age. Therefore, this feature is the target variable - that is, the variable to be predicted by the others. In order to understand which features are correlated to one another, and which features are highly correlated to the target variable, a correlation map has been developed and is shown in Figure 1. The corresponding code is also given.

Listing 2: Calling the `corr_map` utility function.

```
var_names = ["sex", "len", "dia", "hei", "w_who", "w_shu", "w_vis",
    "w_she", "rings"]

corr_map(abalone, var_names, "images/part1_q2.png")
```

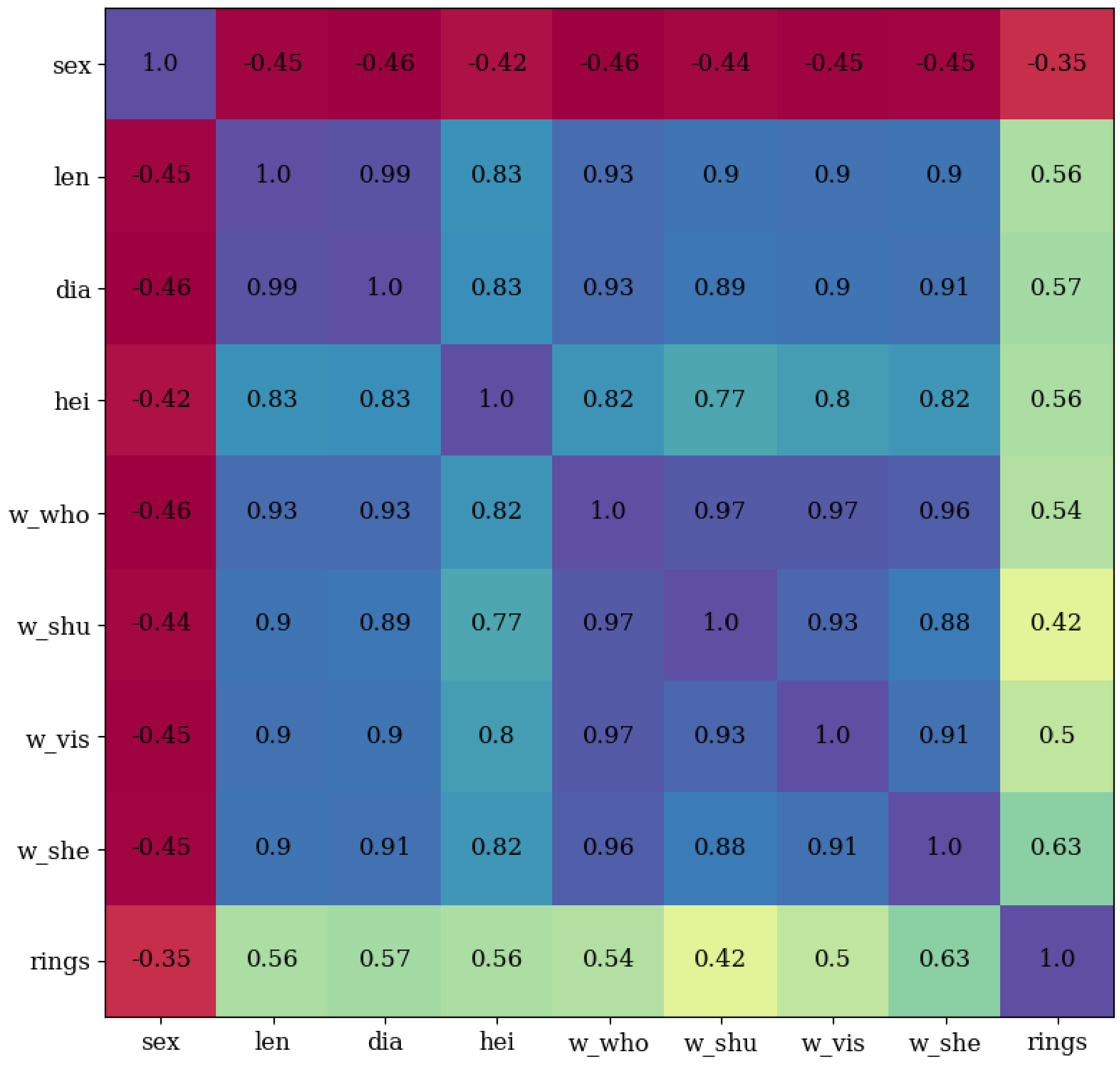

Figure 1: Correlation map.

Listing 3: The `corr_map` utility function.

```python
import matplotlib.pyplot as plt
from numpy import corrcoef

def corr_map(data, var_names, filename):

    n_vars = len(var_names)
    corr = corrcoef(data.T)

    fig, ax = plt.subplots(figsize=(8,8))
    ax.imshow(corr, cmap="Spectral") # type: ignore

    ax.set_xticks(range(0, n_vars), labels=var_names) # type: ignore
    ax.set_yticks(range(0, n_vars), labels=var_names) # type: ignore
```

```
for i in range(0, n_vars):
    for j in range(0, n_vars):
        ax.text( # type: ignore
            j, i,
            round(corr[i,j], 2),
            ha = "center", va = "center",
            size = 12
        )

fig.tight_layout() # type: ignore
plt.savefig(filename)
```

The correlation map shows that the features that have the highest correlation with `rings` are `shell weight` and `diameter`.

The correlation coefficients between these features and `rings` may not be sufficiently high to yield an accurate model. In addition, `diameter` and `shell weight` are very highly correlated to each other (0.91), rendering them unlikely to be suitable for multiple linear regression.

2.3 Scatter Plots

In order to confirm the observations made with regard to the correlations between the input features and `rings`, it may be useful to examine these relationships as scatter plots, as shown in Figures 2 and 3.

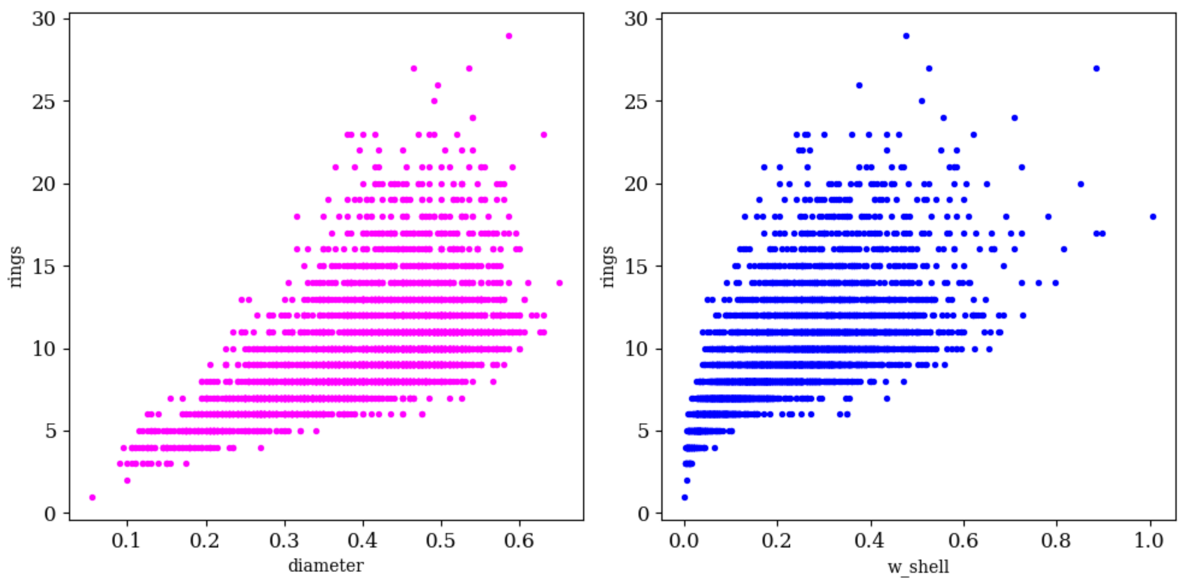

Figure 2: Scatter plots for the two chosen input features.

Listing 4: Calling the scatter plot utility functions.

```
diameter = abalone[:,2]
w_shell = abalone[:,7]
```

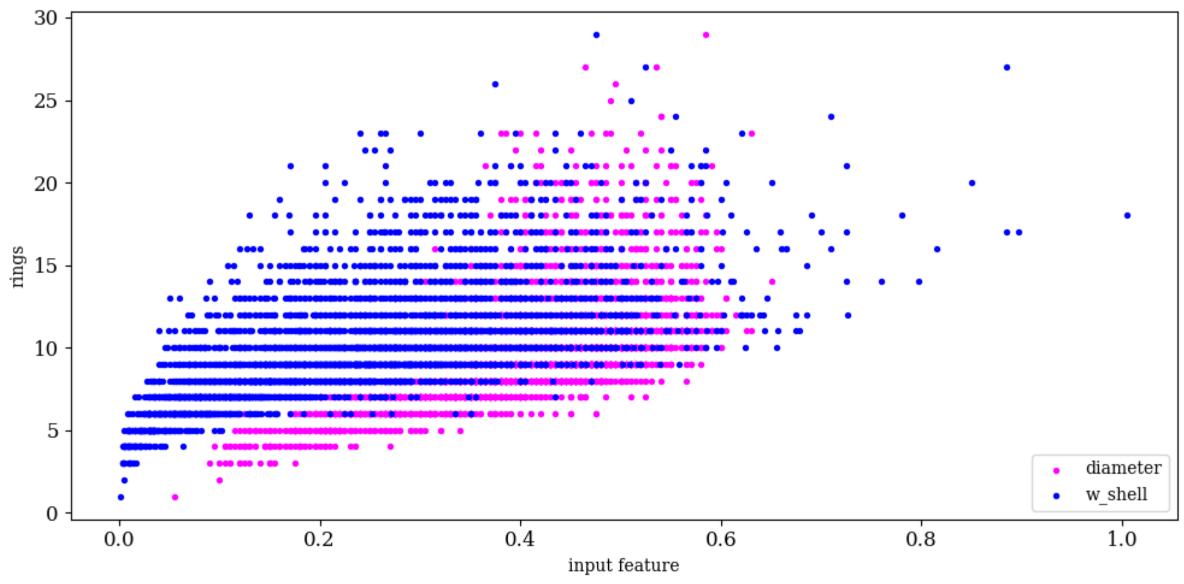

Figure 3: Scatter plot showing the close relationship between shell weight and diameter.

```
rings = abalone[:,8]

scatter_subplots(
    [diameter, w_shell], [rings, rings],
    x_names=["diameter", "w_shell"], y_names=["rings", "rings"],
    filename="images/part1_q3a.png"
    )

scatter_superimposed(
    [diameter, w_shell], rings,
    x_names=["diameter", "w_shell"], y_name="rings",
    filename="images/part1_q3b.png"
    )
```

Listing 5: Utility function to create Figure 2.

```
def scatter_subplots(x_data, y_data, x_names, y_names, filename):

    fig = plt.figure(figsize=(10,5))

    n_plots = len(x_names)
    ax_list = []

    for i in range(n_plots):
        ax_list.append(fig.add_subplot(1, n_plots, i+1))
        ax_list[i].scatter(x_data[i], y_data[i], color=colors[i], s=8)
        plt.xlabel(x_names[i])
        plt.ylabel(y_names[i])

    plt.tight_layout()
    plt.savefig(filename)
```

Listing 6: Utility function to create Figure 3.

```
def scatter_superimposed(x_data, y_data, x_names, y_name, filename):

    fig, ax = plt.subplots(figsize=(10,5))
    n_series = len(x_names)

    for i in range(n_series):
        ax.scatter(x_data[i], y_data, color=colors[i], s=8) # type:
            ignore

    plt.xlabel("input feature")
    plt.ylabel(y_name)
    plt.legend(labels=x_names, loc="lower right")
    plt.tight_layout()
    plt.savefig(filename)
```

Figure 2 appears to confirm that the two features that are most highly correlated with `rings` exhibit an association with the target variable that is not favourable to a reliable model. In addition, each of these two features forms a ”funnel” shape, indicating the presence of heteroscedasticity. This is a contraindication for linear regression.

Figure 3 places the two scatter plots on a shared set of axes. Here, it is apparent that there is a high level of correspondence between `diameter` and `shell weight`, indicating potential collinearity - another contraindication for linear regression.

2.4 Histograms

Another visualisation worth consideration is a histogram, as this considers the distribution of a variable on its own, without regard for its relationships with others. The code for creating multiple histograms in a single figure is given below.

Listing 7: Utility function for plotting histograms.

```
def plot_histograms(col_data, var_names, filename):

    fig = plt.figure(figsize=(10,3))
    n_hists = len(var_names)
    ax_list = []

    for i in range(n_hists):
        ax_list.append(fig.add_subplot(1, n_hists, i+1))
        ax_list[i].hist(col_data[i], color=colors[i])
        plt.tick_params(axis="y", right=False, labelright=False)
        plt.xlabel(var_names[i])

    plt.tight_layout()
    plt.savefig(filename)
```

Listing 8: Calling the histogram utility function.

```
plot_histograms(
    [diameter, w_shell, rings],
    ["diameter", "w_shell", "rings"],
    "images/part1_q4.png"
    )
```

Creating a histogram for each of the chosen input features, as well as for the target variable, yields another concerning observation. As Figure 4 shows, the target variable, `rings`, is positively skewed. This means that it does not meet one of the assumptions of linear regression, which is that the response should follow a Gaussian distribution. It is possible that performing a transformation on `rings` would yield a more suitable distribution; this has not been done for the present analysis, but should be considered by analysts wishing to continue the study of abalone in future.

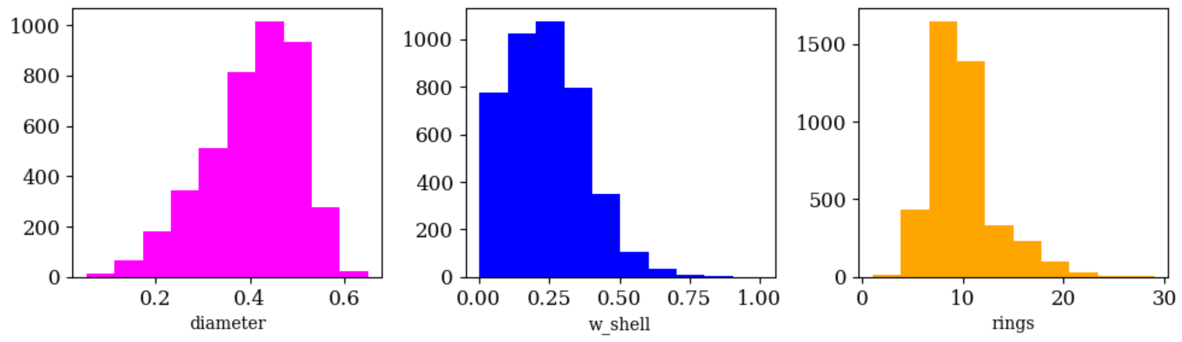

Figure 4: Histograms showing skewed distributions.

2.5 Split Dataset

In order to determine that a model is generalisable to novel data, the dataset has been split into two smaller sets - one for training, and one for testing. The code required for this process is given below. Not only has the dataset been split (with 60% of the records assigned to training), it has also been shuffled, to mitigate the risk that data points that are adjacent in the dataset are also adjacent numerically i.e. similar in magnitude. Clusters of values that are similar in size can make it more difficult for a numerical algorithm to converge upon a solution.

The utility function also allows the analyst to choose a seed to ensure that experiments can be repeated with the same split.

Listing 9: Calling the `split_random` utility function.

```
X = abalone[:,:8]
y = abalone[:,8]

X_train, X_test, y_train, y_test = split_random(
    X, y, train_fraction=0.6, seed=1)
```

Listing 10: The split_random utility function.

```
def split_random(
    X: np.ndarray,
    y: np.ndarray,
    train_fraction: float,
    seed: int):

    return train_test_split(X, y, train_size=train_fraction,
        random_state=seed)
```

2.6 Sources For This Section

```
-- data cleaning

https://archive.ics.uci.edu/ml/datasets/abalone

https://pandas.pydata.org/pandas-docs/stable/reference/
    api/pandas.DataFrame.replace.html

-- correlation map

https://numpy.org/doc/stable/reference/
    generated/numpy.corrcoef.html

https://matplotlib.org/stable/tutorials/colors/colormaps.html

-- scatter plot

https://matplotlib.org/stable/api/_as_gen/
    matplotlib.pyplot.subplots.html

https://matplotlib.org/stable/gallery/subplots_axes_and_figures/
    subplots_demo.html

https://stackoverflow.com/questions/4270301/
    multiple-datasets-on-the-same-scatter-plot

https://matplotlib.org/stable/gallery/color/
    named_colors.html

https://stackoverflow.com/questions/45294833/
    on-setting-fontsizes-for-matplotlib-pyplot-text-elements

https://matplotlib.org/stable/tutorials/introductory/
    customizing.html

https://stackoverflow.com/questions/12998430/
```

```
    how-to-remove-xticks-from-a-plot

-- histogram

https://matplotlib.org/stable/api/_as_gen/
    matplotlib.pyplot.hist.html

-- split data

https://scikit-learn.org/stable/modules/generated/
    sklearn.model_selection.train_test_split.html

https://edstem.org/au/courses/11802/lessons/34144/slides/238769
```

3 Modelling

3.1 Linear Regression by Gradient Descent

Gradient descent is an iterative algorithm that allows the minimum stationary point of a function to be approximated numerically. It is most useful for functions that are too complex to be solved analytically; however, it is instructive to apply it to a relatively simple problem like linear regression, as it is easier to understand its behaviour as it approaches convergence.

Below is an implementation of gradient descent for multiple linear regression. This code is a direct implementation of the mathematical formulae given in the listed sources (see the end of this section), although the sample code was referred to for comparison during debugging.

Unlike the implementations often seen for simple linear regression (that is, regression involving only one input variable), this instance is reliant on a ”design matrix” whose lefthand column is 1. This accounts for the intercept parameter, eliminating the need to handle the intercept and the coefficients separately.

The `cost_derivatives` function computes the derivative of the entity that is being minimised - the error in the estimate of the target variable.

Listing 11: The `cost_derivatives` function.

```python
def cost_derivatives(
    X_design: np.ndarray,
    y_preds: np.ndarray,
    y_actuals: np.ndarray
    ) -> np.ndarray:

    n = len(X_design)

    return (2/n) * X_design.T.dot(y_preds - y_actuals)
```

The new_weights function takes the computed error, and uses this to update the model parameters such that the cost derivative should decrease at the next step. The learning rate tempers the size of these steps, ensuring that larger steps are taken on a steep descent (to reach the minimum more quickly) and smaller steps are taken near convergence (so as to avoid stepping over the minimum).

Listing 12: The new_weights function.

```python
def new_weights(
    w_curr: np.ndarray,
    cost_derivs: np.ndarray,
    learning_rate: float
    ) -> np.ndarray:

    return w_curr - learning_rate*cost_derivs
```

This implementation offers two stopping conditions; either the difference between the previous parameter values and the current parameter values is so small that convergence is assumed, or the maximum allowed number of iterations has been reached. This guards against scenarios where convergence is unlikely to occur within a reasonable timeframe, or where convergence has been prevented by an unsuitable learning rate.

Listing 13: High-level function that calls the others.

```python
def lr_grad_descent(
    X: np.ndarray,
    y: np.ndarray,
    learning_rate: float,
    threshold: float,
    max_iters: int
    ) -> np.ndarray:

    # convert X into design matrix
    X_design = np.concatenate((np.ones((len(X), 1)), X), axis=1)

    # initialise weights
    w_curr = np.random.rand(X_design.shape[1])

    # iterate until either max iterations or convergence
    for i in range(max_iters):

        y_preds = X_design.dot(w_curr.T)

        cost_derivs = cost_derivatives(X_design, y_preds, y)
        w_next = new_weights(w_curr, cost_derivs, learning_rate)

        if np.max(abs(w_next - w_curr)) < threshold:
            break

        w_curr = w_next
    return w_curr
```

Listing 14: Calling the gradient descent function.

```
learning_rate = 0.1
threshold = 0.00001
max_iters = 100000

weights = lr_grad_descent(
    X_train, y_train, learning_rate, threshold, max_iters)

print_gd_result(weights, "Part 2, Q1")
```

This numerical method, having been applied to all available input features in the training set, produces the following result (refer to the appendix for implementations of all the printing functions in `model_utils.py`). The performance metrics have been computed by running the fitted model on the reserved test data.

Listing 15: Making predictions using the fitted model.

```
def predict_gd(weights: np.ndarray, X: np.ndarray) -> np.ndarray:
    intercept = weights[0]
    coeffs = weights[1:len(weights)]

    return intercept + coeffs.dot(X.T)
```

Listing 16: Utility functions for assessing performance.

```
def calculate_mse(y_actual: np.ndarray, y_pred: np.ndarray) -> float:
    return np.mean((y_actual - y_pred) ** 2) # type: ignore

def calculate_rmse(y_actual: np.ndarray, y_pred: np.ndarray) -> float:
    return sqrt(calculate_mse(y_actual, y_pred))

def calculate_rsq(y_actual: np.ndarray, y_pred: np.ndarray) -> float:
    numerator = np.sum((y_actual - y_pred) ** 2)
    denominator = np.sum((y_actual - np.mean(y_actual)) ** 2)
    return 1 - numerator/denominator
```

```
Gradient Descent Results
--------------------------
Intercept:          3.4629
Coeffs:            -0.4263
                   -2.5137
                   11.7303
                   22.8788
                    8.4959
```

```
                 -19.5082
                  -9.5777
                   7.8598

Regression Metrics
------------------
RMSE:        2.2703
R-squared:   0.4993
```

In order to confirm that this result is correct, the regression problem has also been solved using the `scikit-learn` Python library. Because this study is focused on gradient descent, and not on the use of that library, the source code is not reproduced here; however, it can be found in the appendix, in the `model.py` source file.

The result from `scikit-learn` is given below.

```
Sklearn Model Results
---------------------
Intercept:     3.4608
Coeffs:       -0.4256
              -2.7516
              11.9867
              23.0855
               8.5018
             -19.5046
               -9.594
                7.816

Compare GD and Sklearn
----------------------
Model:      GD      SK
RMSE:   2.2703  2.2717
Rsq     0.4993  0.4987
```

One can observe that there is good agreement between the numerical method and the library function. If there were no limitations on hardware, then the stoppage conditions could have been adjusted to yield a result even closer to that of `scikit-learn`. However, given the hardware constraints, this is an acceptable approximation.

Figure 5 confirms the earlier concerns about the suitability of this dataset for linear regression, and is consistent with the poor coefficient of determination shown in the above results. While the input features have some impact on the number of rings, they account for just under 50% of the variability in this target variable. This means that collectively, they cannot be relied upon to make predictions about abalone age.

In addition, the righthand plot in Figure 5 shows non-random spread of residuals, indicating effects (possibly non-linear) that have not been accounted for by the model.

The code to produce Figure 5 is given below.

```python
    def plot_regression_results(y_actual, y_pred, filename):
```

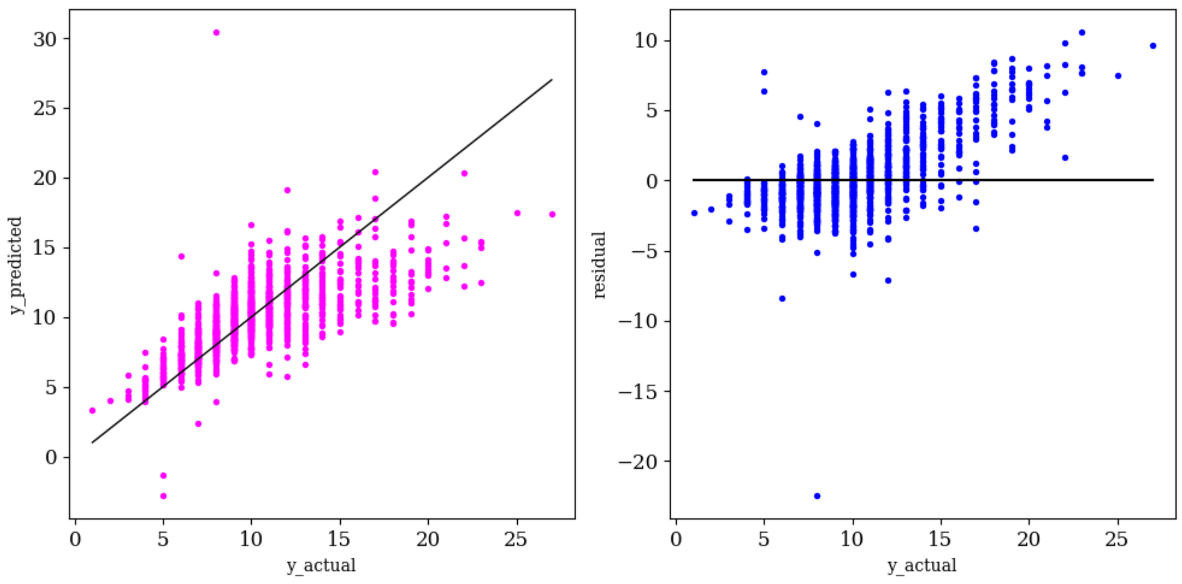

Figure 5: Results of linear regression with all input features.

```
resids = y_actual - y_pred
x_min = min(y_actual)
x_max = max(y_actual)

fig = plt.figure(figsize=(10,5))
ax1 = fig.add_subplot(1, 2, 1)
ax2 = fig.add_subplot(1, 2, 2)

ax1.scatter(y_actual, y_pred, color=colors[0], s=8)
ax1.plot([x_min, x_max], [x_min, x_max], linewidth=1,
    color="black")
ax1.set_xlabel("y_actual")
ax1.set_ylabel("y_predicted")

ax2.scatter(y_actual, resids, color=colors[1], s=8)
ax2.plot([x_min, x_max], [0, 0], color="black")
ax2.set_xlabel("y_actual")
ax2.set_ylabel("residual")

plt.tight_layout()
plt.savefig(filename)
```

3.2 Normalisation

Normalisation can aid the gradient descent algorithm to converge upon a solution by ensuring that all of the input variables are scaled similarly. Shown here is a simple method for normalising variables.

Listing 17: Utility function to normalise data.

```
def normalise(X: np.ndarray) -> np.ndarray:
    X_min = X.min(axis=0)
    X_max = X.max(axis=0)

    return (X - X_min) / (X_max - X_min)
```

Having normalised all of the input features, the gradient descent model has been re-run using the same code as previously. While it does converge slightly faster than for the non-normalised dataset, it presents no improvement to model performance, as shown by the output here.

```
Gradient Descent Results
------------------------
Intercept:        3.9469
Coeffs:          -0.8536
                 -2.0408
                  7.2066
                 25.3491
                 23.2497
                -28.6263
                 -7.0566
                  8.2568

Regression Metrics
------------------
RMSE:       2.2679
R-squared:  0.5003
```

The reason for the near-identical performance metrics is that normalisation does not change how good (or poor) a model is; it merely scales it, which affects the magnitude of the coefficients, but not the suitability of the model design. Figure 6 shows that the outcome after normalisation is very similar to the non-normalisation result in Figure 5.

3.3 Best Features

In a final attempt at producing acceptable performance, the gradient descent implementation has been run on just the two features that were mentioned earlier as having the highest correlation to `rings` - `diameter` and `shell weight`.

Other than the code used to siphon these features from the dataset, the operation remains the same.

Listing 18: Running the model on only two features.

```
X_train_best = X_train[:,(2,7)]
X_test_best = X_test[:,(2,7)]

weights_best = lr_grad_descent(
```

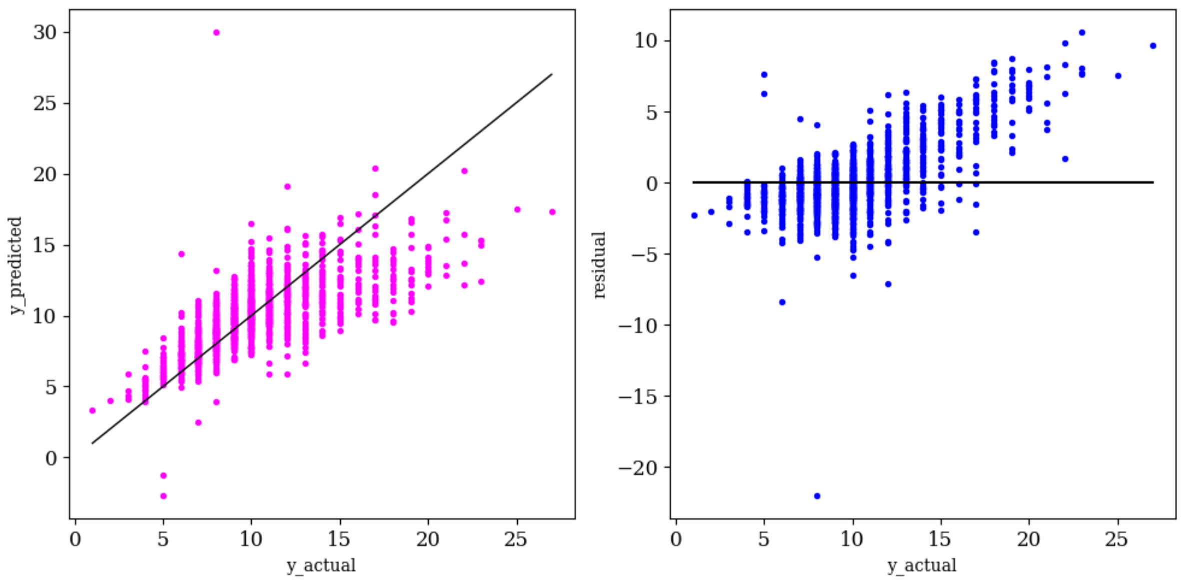

Figure 6: Results after normalisation.

```
            X_train_best, y_train, learning_rate, threshold, max_iters)
```

Limiting the model to the two ”best” input features does not yield an improvement at all; rather, there is degradation in the coefficient of determination. This may be because the omitted features do contribute to the variability in the model. It might also be because the two chosen features, as shown in Figure 3, overlap each other. This means it is possible that one of these features is redundant - therefore, one might achieve similar performance by using only one of them. Figure 7 visualises the significant degradation of performance - a very clear non-linear pattern is visible in the residual plot.

```
Gradient Descent Results
------------------------
Intercept:        6.1204
Coeffs:            1.145
                 13.8736

Compare GD and Sklearn
----------------------
Model:       GD       SK
RMSE:     2.531    2.531
Rsq      0.3777   0.3777
```

One noteworthy improvement is in the convergence time; it appears that choosing fewer features yields a much faster outcome - in this analysis, convergence occurred almost instantaneously.

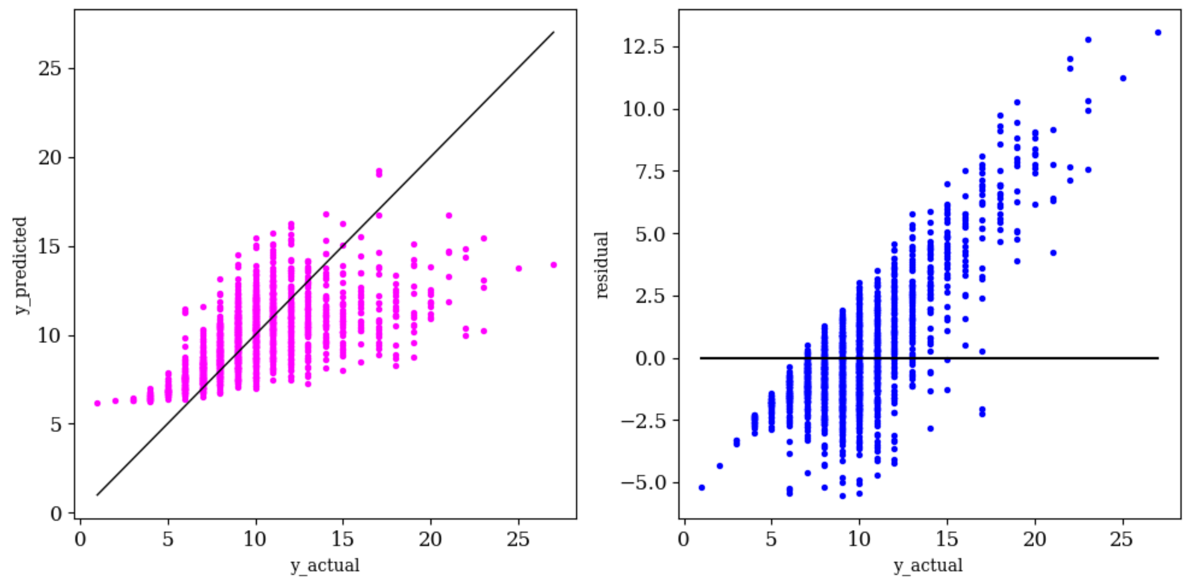

Figure 7: Degraded performance after omitting most features.

3.4 Variability in Performance

All three of the above model designs have been run thirty times in order to ascertain how much variability there may be in the performance metrics (RMSE and R-squared). This is intended to give an indication of how dependent the reported performance is upon the randomly selected data i.e. how robust the model parameters are, to variability in the data.

The code used for this process is inelegant due to time constraints. It also contains a large amount of printing code; therefore, it has been omitted from this report but is available in full in `model.py` and `model_utils.py`, located in the appendix.

The output is as shown here.

```
Experiment Results     RMSE MEAN
--------------------------------
Model                Train    Test
All features        2.1854  2.2319
All - norm          2.1854  2.2319
Two features        2.5029    2.52

Experiment Results      RSQ MEAN
--------------------------------
Model                Train    Test
All features        0.5389   0.522
All - norm          0.5389   0.522
Two features        0.3952   0.391

Experiment Results      RMSE STD
--------------------------------
Model                Train    Test
All features        0.0337  0.0534
All - norm       0.0337   0.0534
Two features     0.0394   0.0597

Experiment Results       RSQ STD
--------------------------------
Model             Train     Test
All features     0.0122   0.0235
All - norm       0.0122   0.0235
Two features     0.0128   0.0192
```

One may observe that the central tendency (mean) of both performance measures is fairly uniform between training and testing sets, regardless of which approach of the three has been used. Unsurpisingly, the performance for normalised and non-normalised data is identical. However, the standard deviation of both metrics changes markedly when novel (test) data are used. This indicates that the fitted model parameters may not be robust to variability in novel data, because the testing set yields greater spread in the performance metrics (albeit around a similar mean).

3.5 Sources For This Section

```
-- multiple linear regression with gradient descent

https://medium.com/nerd-for-tech/multiple-linear-regression-
    and-gradient-descent-using-python-b931a2d8fb24

-- append to array

https://numpy.org/doc/stable/reference/
    generated/numpy.append.html

-- create array of zeros

https://numpy.org/doc/stable/reference/generated/numpy.zeros.html

-- create array of ones

https://numpy.org/doc/stable/reference/generated/numpy.ones.html

-- type hinting (especially for numpy)

https://docs.python.org/3/library/typing.html

-- align printed strings

https://stackoverflow.com/questions/829667/
    str-format-how-to-left-justify

-- sklearn
```

```
https://scikit-learn.org/stable/modules/classes.html

https://edstem.org/au/courses/11802/lessons/34144/slides/238764

-- residuals and residual plot

from Regression Analysis notes on Ed (ZZSC5806)
```

4 Conclusion

It has been established that the features available in the abalone dataset are not sufficient to reliably predict the age of an abalone. In addition, the efficacy of gradient descent as a mechanism for linear regression has been shown, although there appears to be some sensitivity to the number of features in use.