AI Coursework – F322418

1. Data Pre-processing

1.1 Data cleansing

Note: *I used the pandas and numpy Python libraries for my data pre-processing. I reference pandas as ‘pd’ and numpy as ‘np’ throughout my code.*

Date Cleaning

The first step I took in terms of data pre-processing was cleaning the data. This process involves firstly detecting errors within the provided data set and then correcting or removing these errors.

The first thing I wanted to test was that all date rows were present in the data and that none had been missed out or incorrectly ordered. To do this I wrote the code that you see below:

```
def date_checker(df):
    date_column = 'Date'
    df[date_column] = pd.to_datetime(df[date_column], format='%d/%m/%Y',
errors='coerce')

    # calculate the difference between consecutive dates
    date_diff = df[date_column].diff()

    # find gaps larger than 1 day
    gaps = df[date_diff > pd.Timedelta(days=1)]

    if len(gaps) > 0:
        print("Found gaps in dates:")
        print(gaps)
    else:
        print("All dates are consecutive")
```

This code selects the ‘Date’ column from the Excel sheet we were provided (here passed in as a parameter called ‘df’, referring to a DataFrame in the pandas library) and then formats these dates as actual times that can be checked through. From here we are simply able to check if there are any gaps found in the 4 year time frame. The output was that ‘All dates are consecutive’ and so I continued with the next step of data cleansing.

Non-numerical Data Removal

Before removing spurious numerical values, I firstly removed all non-numerical values from the data set. After doing this, the next step becomes easier as I can simply use standard deviation across the dataset to find spurious numerical values and remove/correct them accordingly. Below is my implementation of this removal in Python:

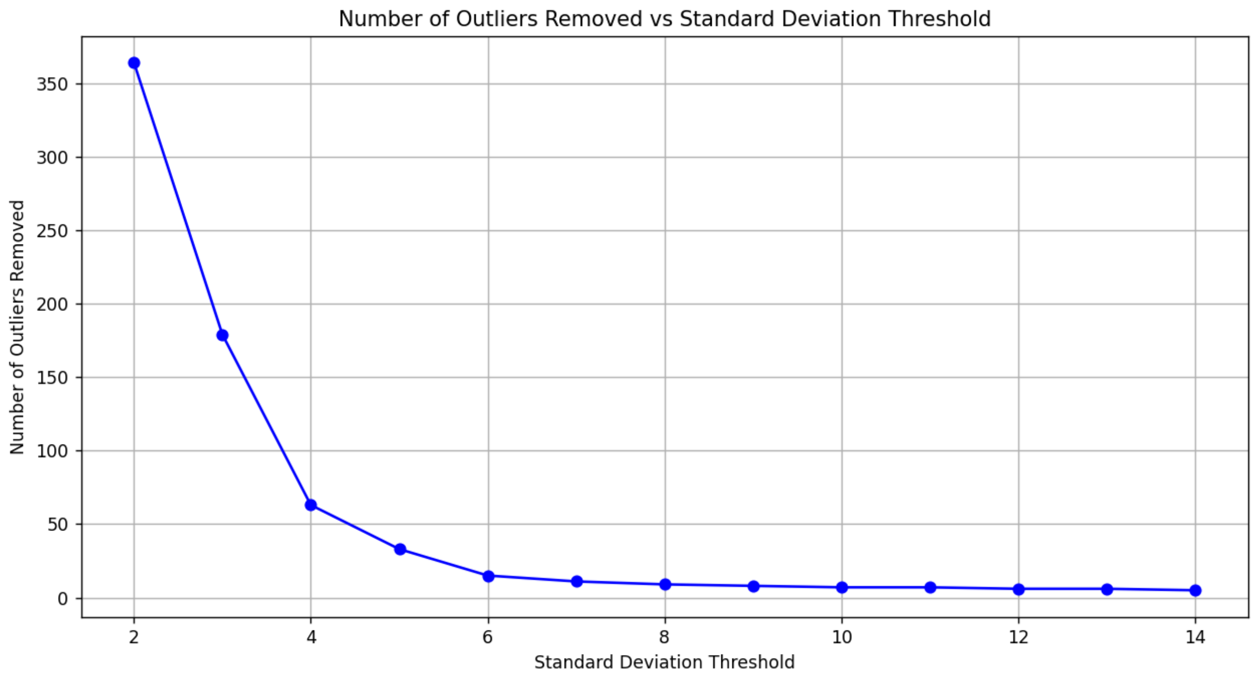
```
def validate_data(df):
    # check for non-numeric data
    non_numeric = df.apply(lambda x: ~pd.to_numeric(x,
errors='coerce').notna())
    non_numeric_rows = non_numeric.any(axis=1)

    if non_numeric_rows.any():
        print("Rows with non-numeric data:")
        print(df[non_numeric_rows])
        # removes the non-numeric rows from df
        df = df[~non_numeric_rows]
    else:
        print("No rows contain non-numeric data")

    return df
```

```
Rows with non-numeric data:
          Date  ...  Rainfall_mm_Snaizeholme
96   1993-04-07  ...                      3.2
789  1995-03-01  ...                     21.6
1134 1996-02-09  ...                     14.4
[3 rows x 9 columns]
```

From the information above I manually interpolated this data using the values from the rows either side of the non-numeric value. I did this simply by adding up both the values and dividing them by 2. Now mathematical operations can take place on this data set without errors occurring. As such from here I moved onto the removal of spurious numerical values.

Spurious Numerical Values Removal

For a model such as this, it didn't seem wise to remove a large number of data points as this may limit the performance of the model when it comes to predicting more extreme values that, whilst may be more uncommon, still occur frequently throughout the dataset. As such I tested a variety of different standard deviation points ranging from 2 to 14 and plotted the number of outliers that these deviations ended up identifying on a graph.

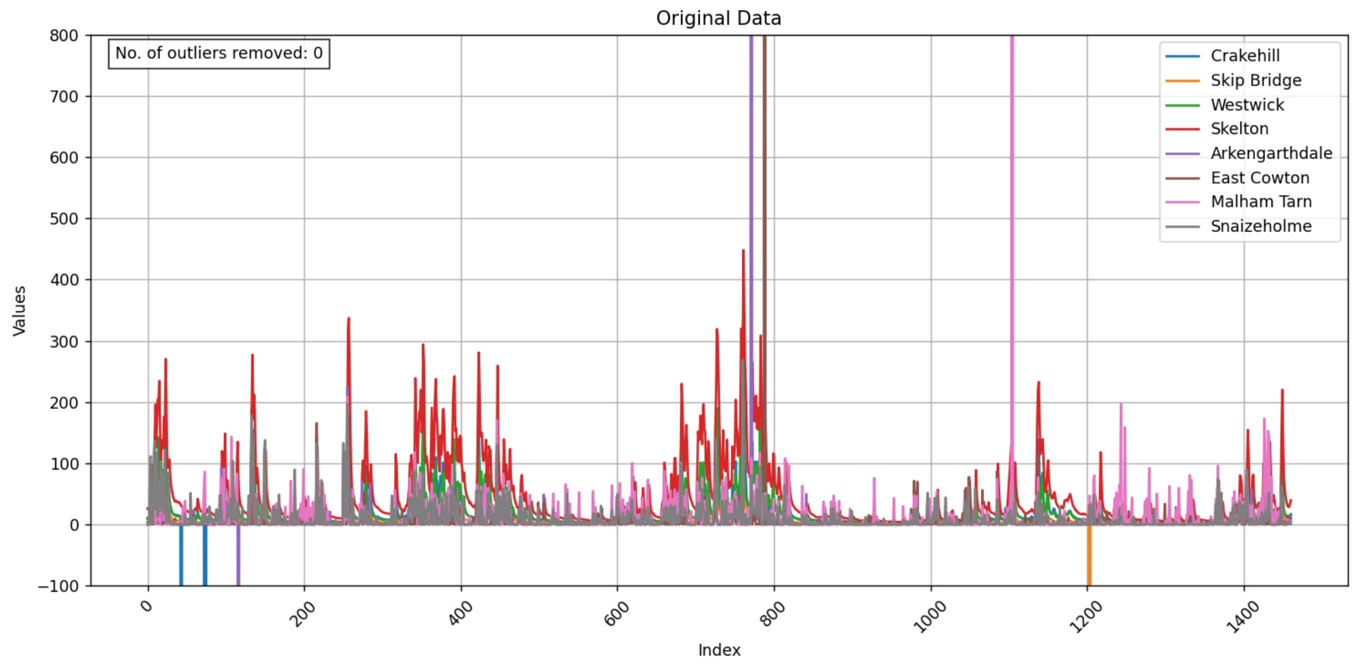


Here you can see this graph. From the information seen above I landed on a standard deviation of 6. This is because at a low standard deviation threshold such as 2, 3 and 4, quite a large number of data points are removed. This likely includes important data that could be used to train the model and without it the model may only be accurate at predicting the more commonly seen data points which is not ideal for a model that needs to accurately predict a variety of different values.

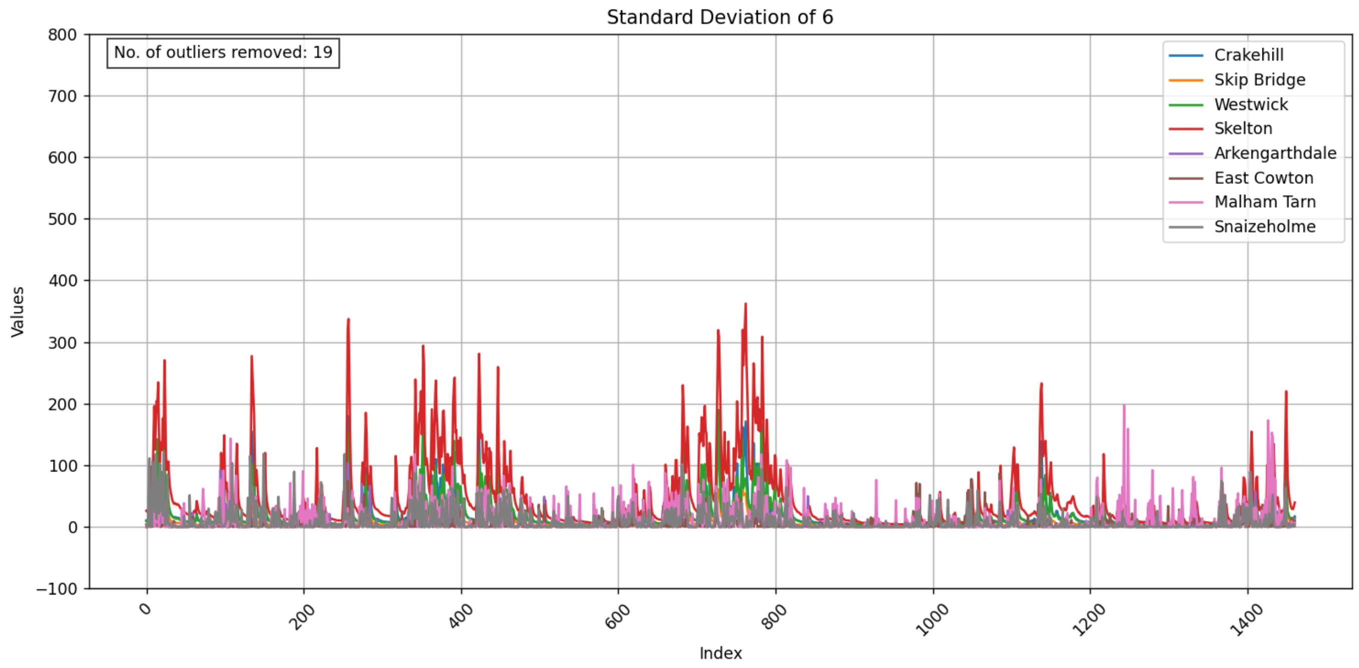
A standard deviation of 7 and onwards, the curve seems to flatten, whilst still slowly decreasing. This implies that with each drop down we are leaving a genuine outlier in the data set that, if not dealt with, when training may cause spurious outputs.

As such I decided on 6 as it struck a balance between keeping all important data points whilst ensuring spurious ones are removed. This also included removing data such as -999. Below you can see the dataset graphed after having removed outliers using a standard deviation of 6 vs the original data set.

Original:



Outliers removed:



As you can see extreme outliers aren’t present whilst important values remain in the dataset. Instead of actually removing the outliers however, I instead interpolated the data either side of each of these outliers and calculated a replacement value for them. This was done with the same method as done when interpolating for non-numerical data points. The code I used to do this is as follows:

```
def main():
    file_name = ("input.xlsx")
    threshold = float(input("Enter the threshold (in terms of standard
deviations): "))

    df = pd.read_excel(file_name)
    processed_df = df.copy()

    # loop through each numeric column.
    for col in processed_df.select_dtypes(include=[np.number]).columns:
        mean = processed_df[col].mean()
        std = processed_df[col].std()

        # identify outliers that exceed standard dev threshold
        outlier_mask = abs(processed_df[col] - mean) > (threshold * std)

        # replace outlier values with NaN.
        processed_df.loc[outlier_mask, col] = np.nan

        # interpolate missing values (linear interpolation using the
values on either side)
        processed_df[col] = processed_df[col].interpolate(method='linear')

    processed_df.to_excel("output.xlsx", index=False)

if __name__ == "__main__":
    main()
```

Note: *The reason I was able to interpolate across this data set during the removal of spurious data is because I am working with a time series data set.This process of interpolating data is known as imputation, whereby we are substituting values in rather than removing data points.*

1.2 Data Splitting

Initially I was considering a 60-20-20 split, however, considering the type of data set that we are dealing with (time series), I took the approach of splitting the data in terms of years. This is because there are 4 years within the data set and if we take a split of 50-25-25, the training data consists of 2 years and each the testing and validation takes 1 year of the dataset. This removes any issues regarding seasonality of the data and any fluctuations that may occur with the different seasons (e.g. less rainfall in summer). To do this I used the code below:

```python
# split the data into 50% training, 25% validation, 25% testing
n_total = len(df)
n_train = int(0.5 * n_total)
n_val = int(0.25 * n_total)
n_test = n_total - n_train - n_val # ensures no data is doubled up

# split the dataset sequentially
train_df = df.iloc[:n_train].copy()
val_df = df.iloc[n_train:n_train + n_val].copy()
test_df = df.iloc[n_train + n_val:].copy()
```

Training data should take up the majority of the data split as it is important to train your model with lots of data. Extending from this, testing and validation both require less, this is because it's only used to test the effectiveness of your model and evaluate its predictions. Therefore it's not necessary to use the bulk of the dataset. It's important we split these up since you can't test your model with the same data used to train your model as this would produce a biased output.

1.3 Selecting Predictors

Now that the data was cleaned, I started to attempt to find some meaningful correlations between the data such that my model can be effectively trained. To select these correlations, I used the Pearson correlation coefficient formula:

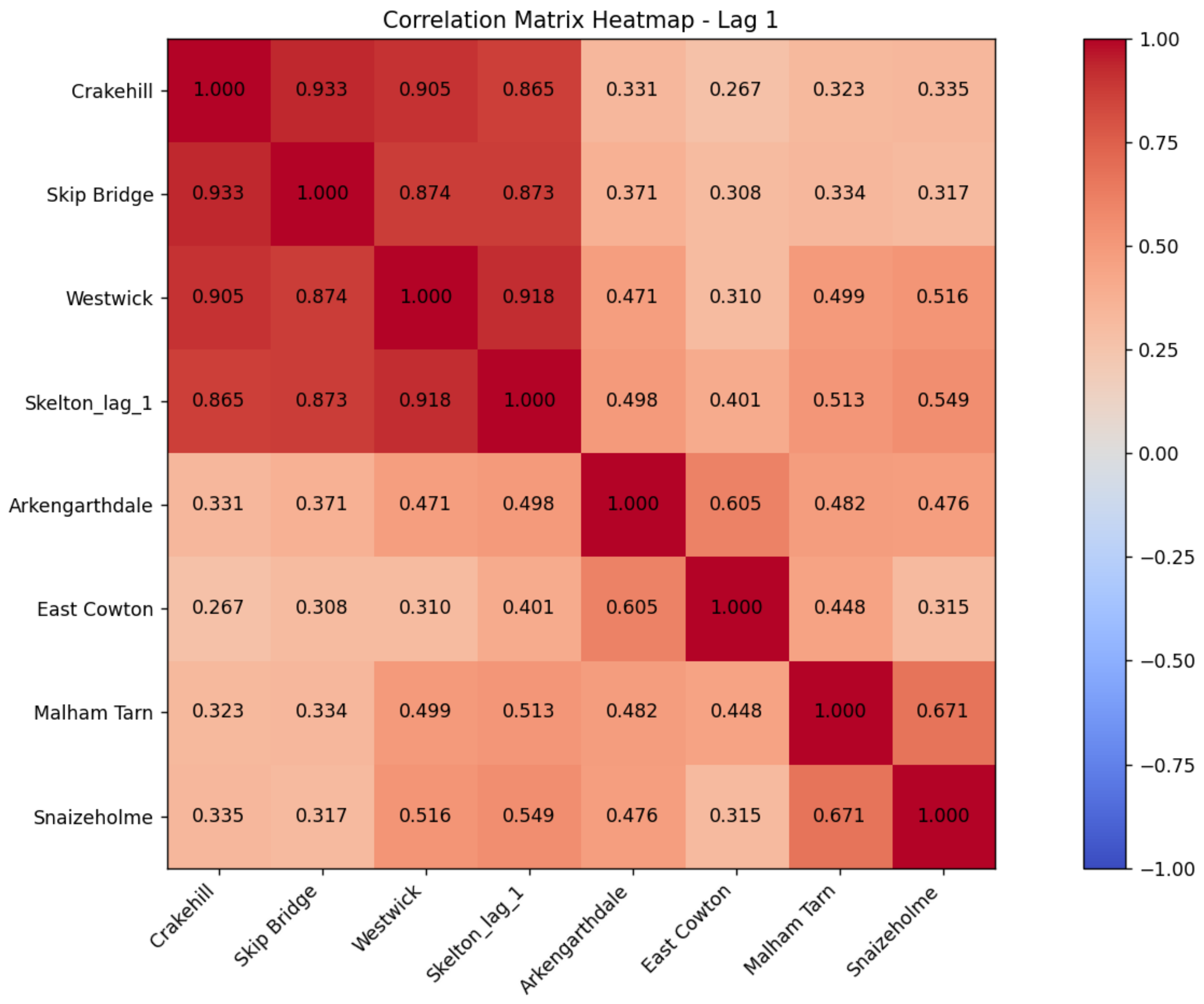
$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

It measures the strength and direction of the linear relationship between two variables *x* and *y*. Here a value of r = 1 indicates a perfect positive correlation, r = 0 suggests no linear correlation and r = -1 indicates a perfect negative correlation.

To actually apply this to my data set, I will simply run each of the columns as *x* values and have flow rate in Skelton by my *y* value. In this way, I can work out the correlation of each column and this will help me determine its importance as a predictor.

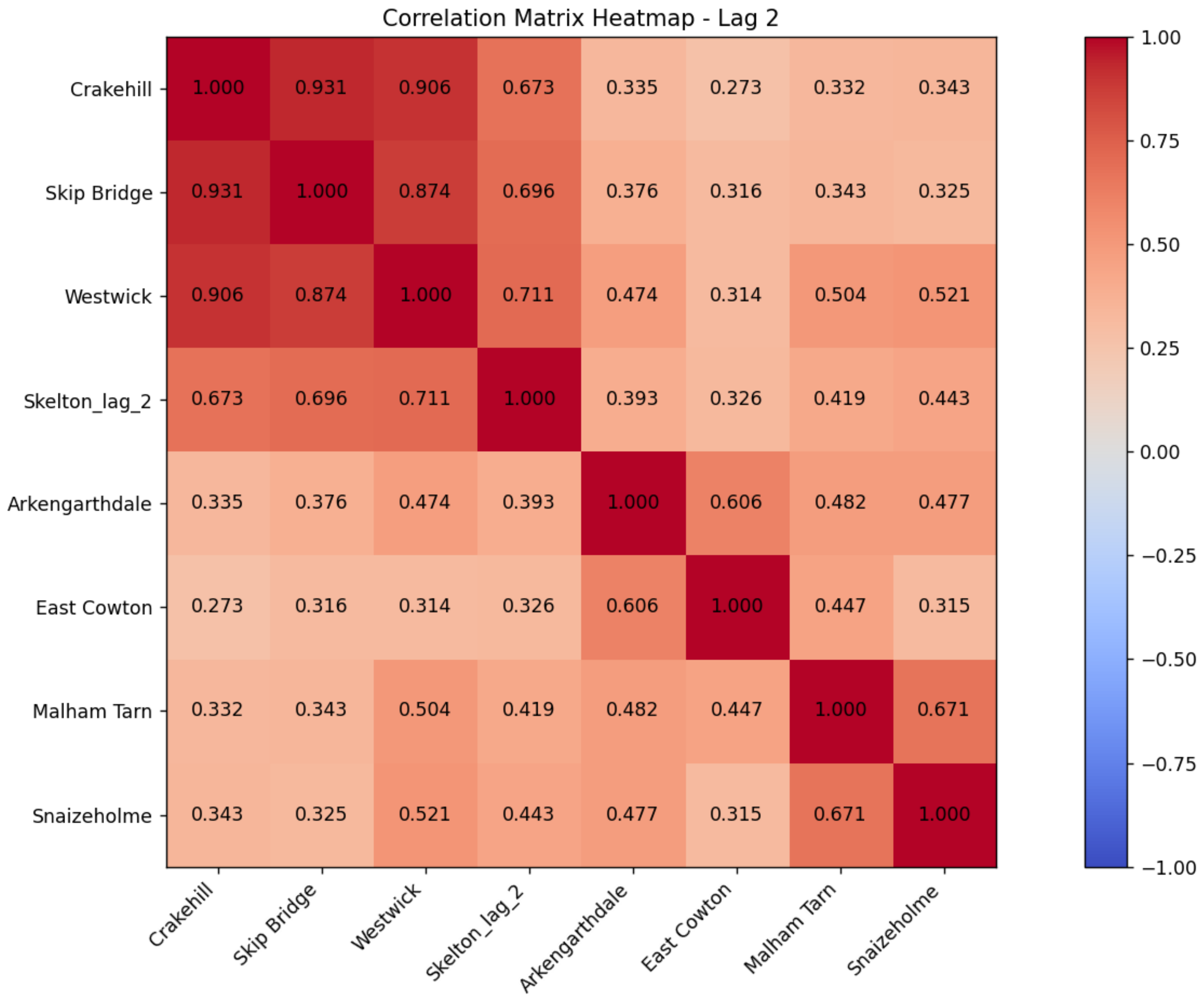
Since my data is time series and we are to work out the next day flow rate in Skelton, we are unable to use the current day as a predictor. This is because it would be data that would not be available until the next day, as such we must use the previous days as predictors. Therefore, we need to lag the data and find the best correlation for each of the columns, then we can test at different lag rates to determine the best set of predictors per column. For this I tested at lag rates of 1 - 3 on my training data and calculated the correlation coefficient for each of these.

<u>Lag 1:</u>



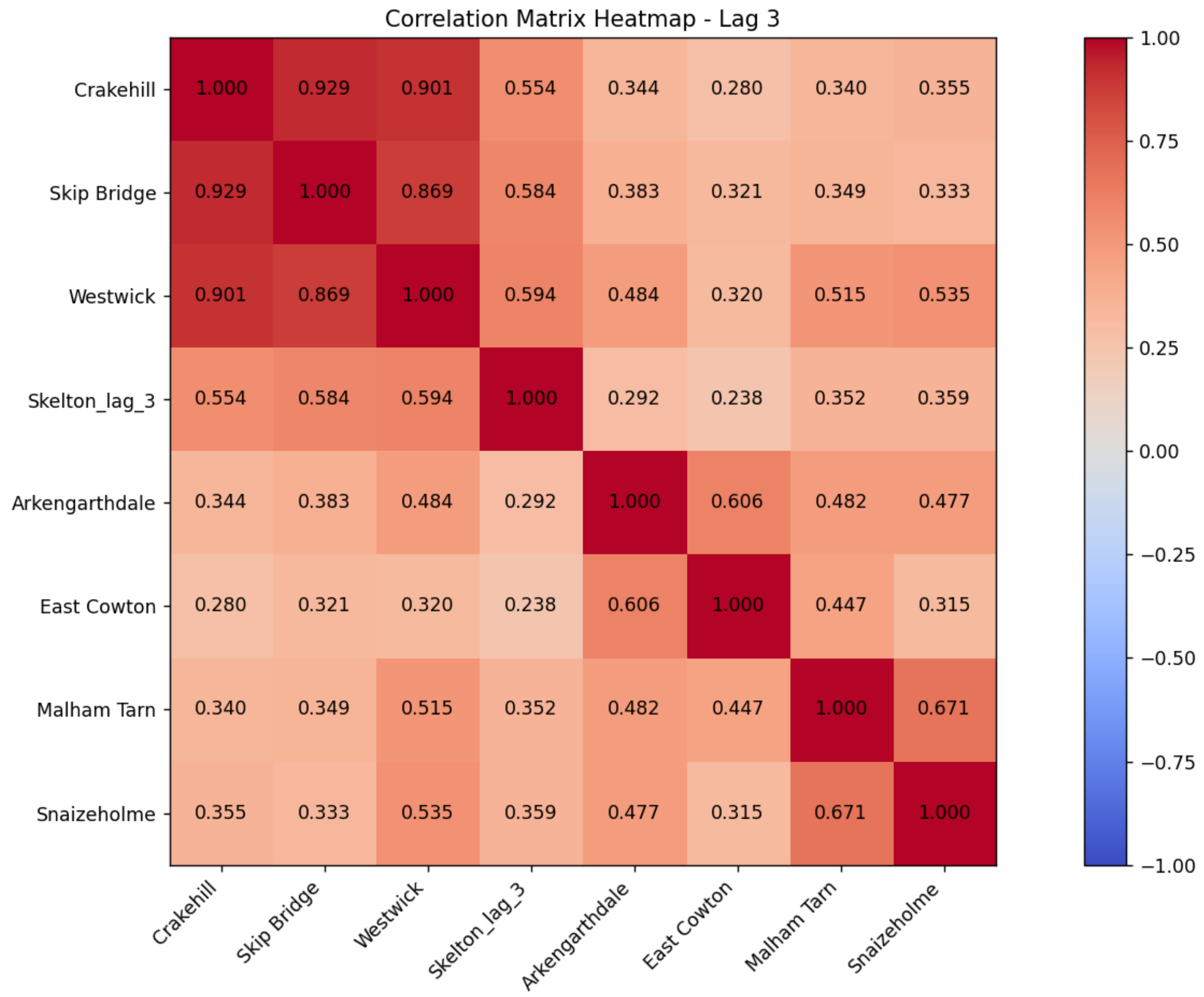
	Crakehill	Skip Bridge	Westwick	Skelton_lag_1	Arkengarthdale	East Cowton	Malham Tarn	Snaizeholme
Crakehill	1.000	0.933	0.905	0.865	0.331	0.267	0.323	0.335
Skip Bridge	0.933	1.000	0.874	0.873	0.371	0.308	0.334	0.317
Westwick	0.905	0.874	1.000	0.918	0.471	0.310	0.499	0.516
Skelton_lag_1	0.865	0.873	0.918	1.000	0.498	0.401	0.513	0.549
Arkengarthdale	0.331	0.371	0.471	0.498	1.000	0.605	0.482	0.476
East Cowton	0.267	0.308	0.310	0.401	0.605	1.000	0.448	0.315
Malham Tarn	0.323	0.334	0.499	0.513	0.482	0.448	1.000	0.671
Snaizeholme	0.335	0.317	0.516	0.549	0.476	0.315	0.671	1.000

Lag 2:



	Crakehill	Skip Bridge	Westwick	Skelton_lag_2	Arkengarthdale	East Cowton	Malham Tarn	Snaizeholme
Crakehill	1.000	0.931	0.906	0.673	0.335	0.273	0.332	0.343
Skip Bridge	0.931	1.000	0.874	0.696	0.376	0.316	0.343	0.325
Westwick	0.906	0.874	1.000	0.711	0.474	0.314	0.504	0.521
Skelton_lag_2	0.673	0.696	0.711	1.000	0.393	0.326	0.419	0.443
Arkengarthdale	0.335	0.376	0.474	0.393	1.000	0.606	0.482	0.477
East Cowton	0.273	0.316	0.314	0.326	0.606	1.000	0.447	0.315
Malham Tarn	0.332	0.343	0.504	0.419	0.482	0.447	1.000	0.671
Snaizeholme	0.343	0.325	0.521	0.443	0.477	0.315	0.671	1.000

<u>Lag 3:</u>



	Crakehill	Skip Bridge	Westwick	Skelton_lag_3	Arkengarthdale	East Cowton	Malham Tarn	Snaizeholme
Crakehill	1.000	0.929	0.901	0.554	0.344	0.280	0.340	0.355
Skip Bridge	0.929	1.000	0.869	0.584	0.383	0.321	0.349	0.333
Westwick	0.901	0.869	1.000	0.594	0.484	0.320	0.515	0.535
Skelton_lag_3	0.554	0.584	0.594	1.000	0.292	0.238	0.352	0.359
Arkengarthdale	0.344	0.383	0.484	0.292	1.000	0.606	0.482	0.477
East Cowton	0.280	0.321	0.320	0.238	0.606	1.000	0.447	0.315
Malham Tarn	0.340	0.349	0.515	0.352	0.482	0.447	1.000	0.671
Snaizeholme	0.355	0.333	0.535	0.359	0.477	0.315	0.671	1.000

As you can see from the table above, running the Pearson's coefficient on the Excel sheet provides me with this table output at the different lags. From this, we can see that the lag of 1 is a better predictor and that it only went downhill in terms of correlation as we increased the lags. As such I stuck to a lag of 1.

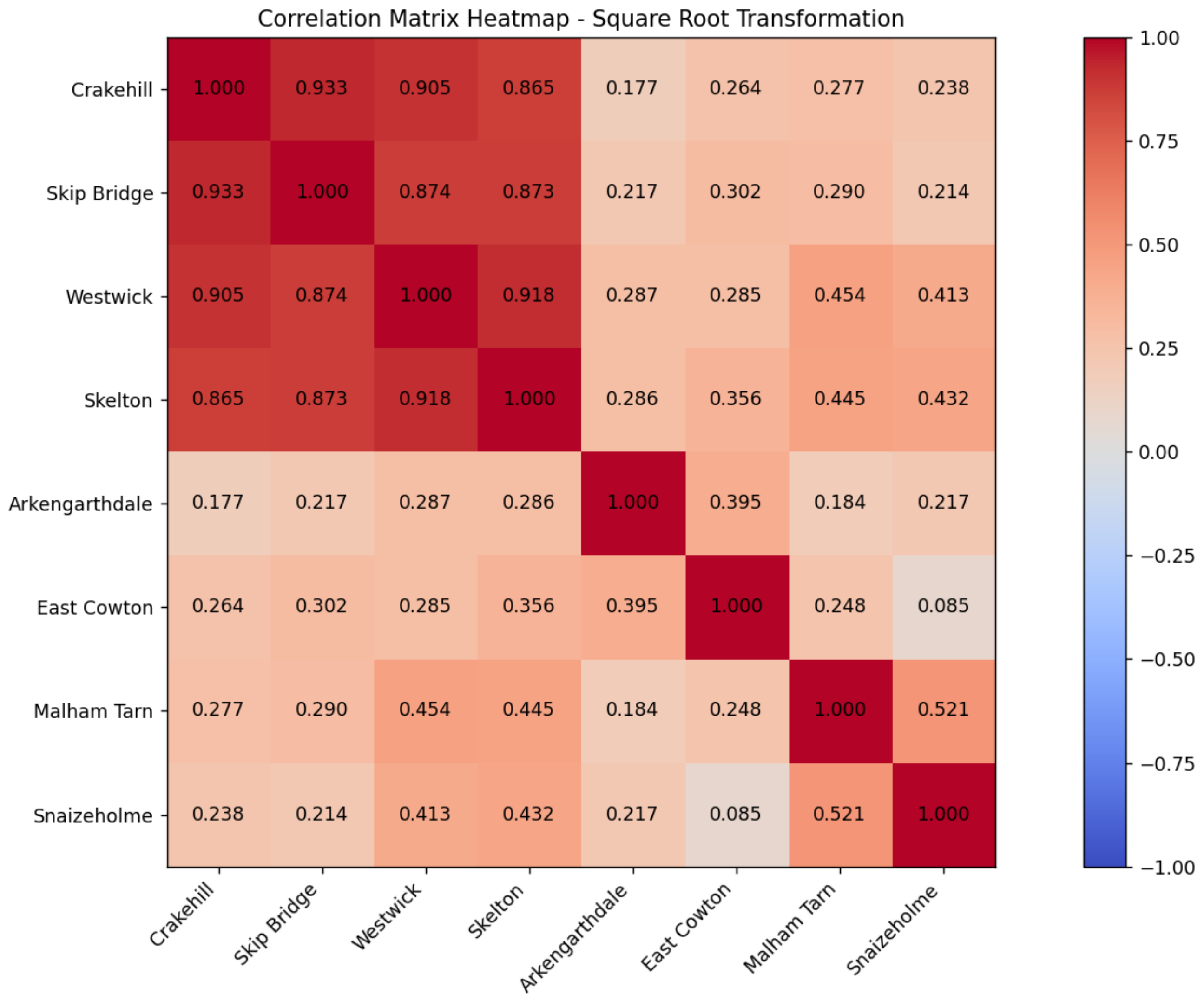
However, we may not want to simply remove this data as it could still offer some useful information for our model. Specifically the rainfall as rainfall in previous days is a good indicator of upcoming flow rate. To combat this, I implemented a weighted moving average of the last 3 days of rainfall.This was simply done straight in excel with the following command:

```
=SUMPRODUCT(A2:A4, {0.1; 0.3; 0.6})
```

I then dragged this and applied it to all the rainfall columns to ensure the data was consistent across all columns. In this way we don't skip over what could possibly be vital information in predicting future flow rate and as such the model becomes more accurate.

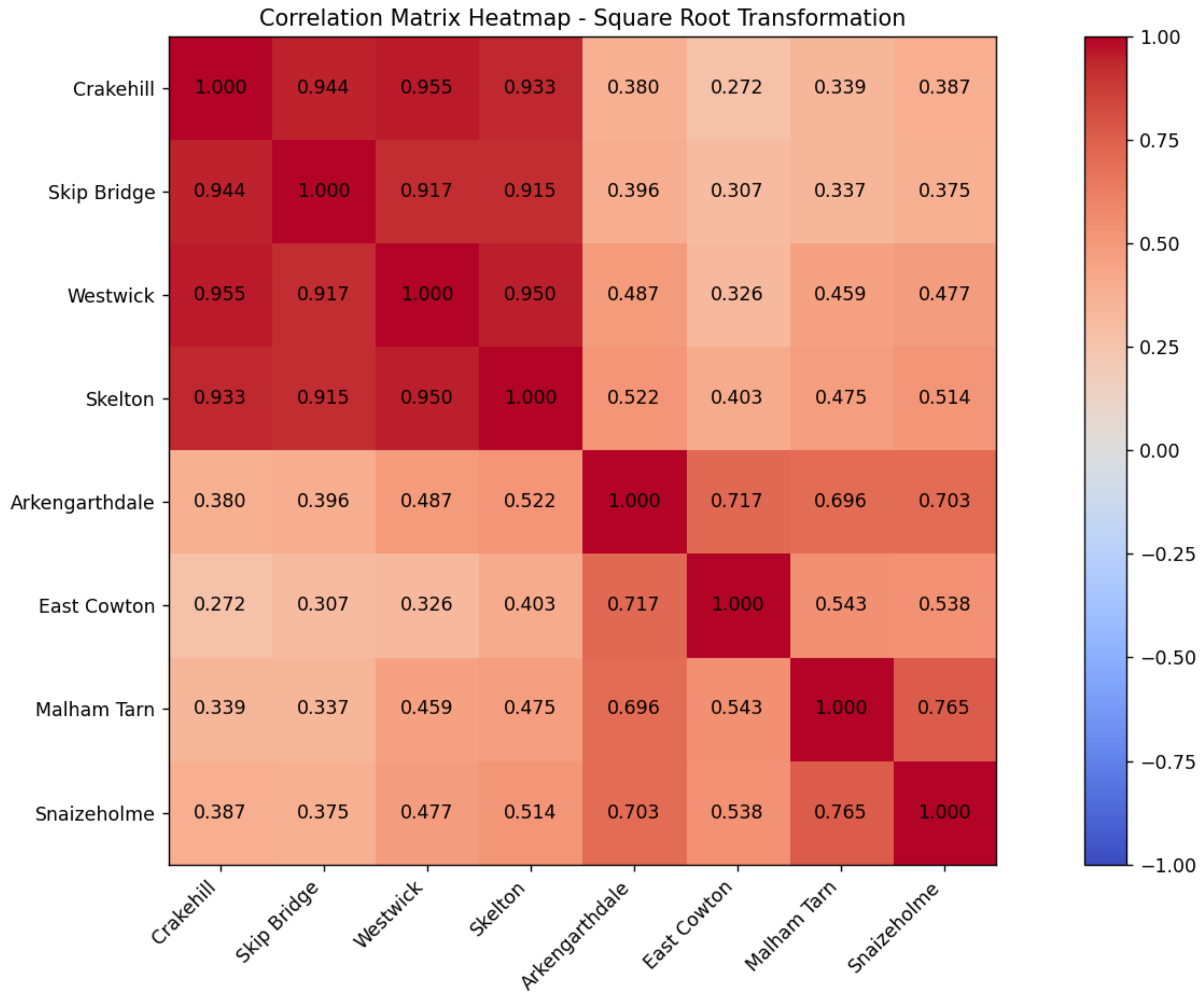
However, the predictors could be slightly improved upon in terms of correlation. As such I tested different transformations of each of the columns to find the best one. Below you can see the different transformations as correlation matrix heatmaps:

Square Transformation



The square transformation offers a lower correlation than that of the regular linear correlation. This implies increasing the polynomial value will only decrease correlation, as such I wanted to test some other approaches.

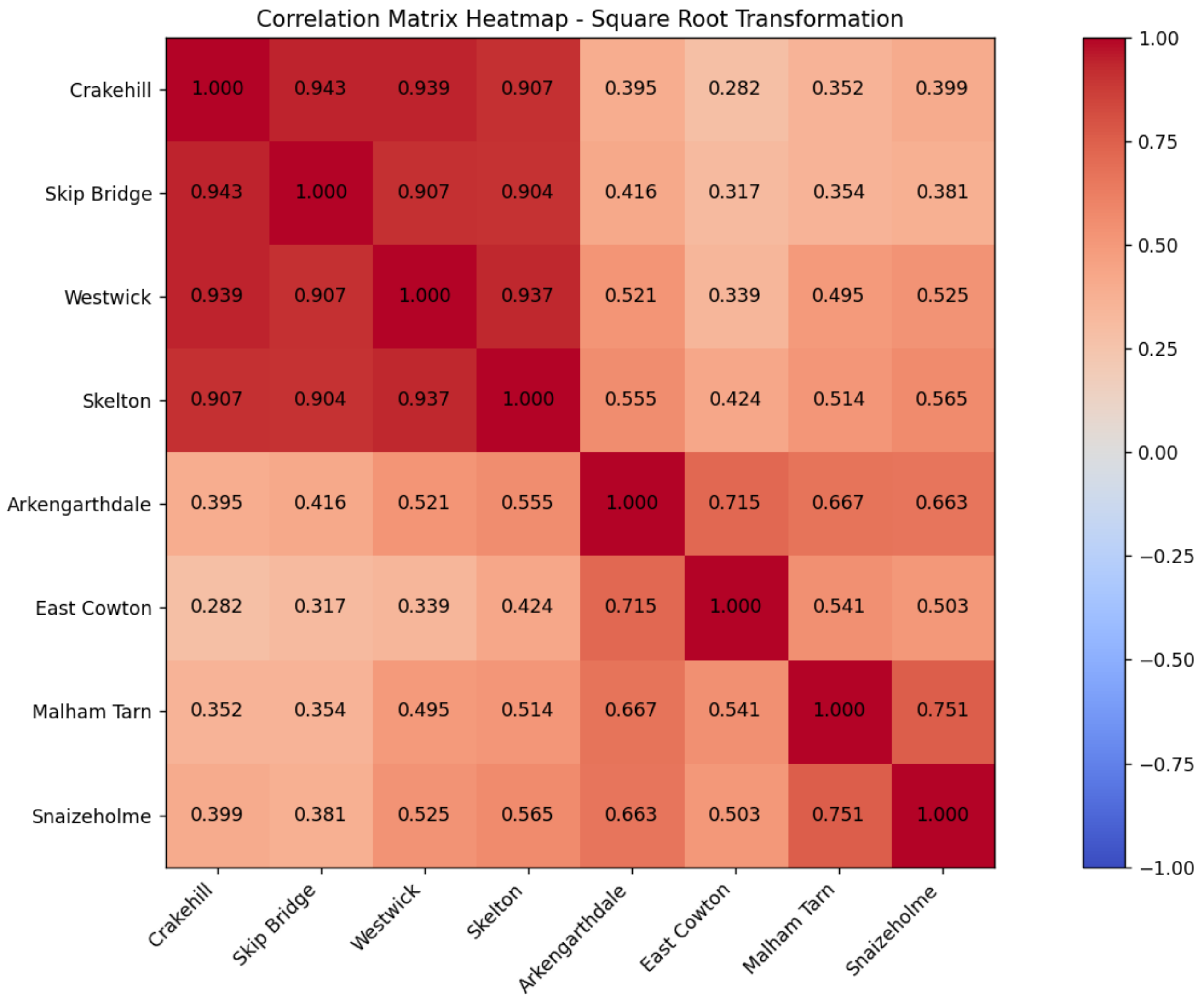
Log Transformation



	Crakehill	Skip Bridge	Westwick	Skelton	Arkengarthdale	East Cowton	Malham Tarn	Snaizeholme
Crakehill	1.000	0.944	0.955	0.933	0.380	0.272	0.339	0.387
Skip Bridge	0.944	1.000	0.917	0.915	0.396	0.307	0.337	0.375
Westwick	0.955	0.917	1.000	0.950	0.487	0.326	0.459	0.477
Skelton	0.933	0.915	0.950	1.000	0.522	0.403	0.475	0.514
Arkengarthdale	0.380	0.396	0.487	0.522	1.000	0.717	0.696	0.703
East Cowton	0.272	0.307	0.326	0.403	0.717	1.000	0.543	0.538
Malham Tarn	0.339	0.337	0.459	0.475	0.696	0.543	1.000	0.765
Snaizeholme	0.387	0.375	0.477	0.514	0.703	0.538	0.765	1.000

As you can see, logging the data bumps up the flow rates’ correlation a little into the 0.9’s however it drastically increases the rainfall correlation, bringing them to 0.5 in certain locations.

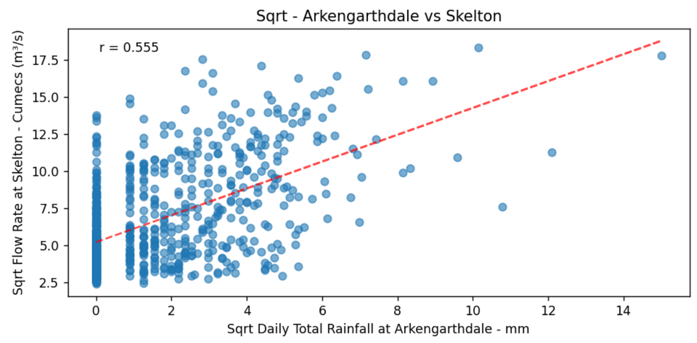
Square Root Transformation

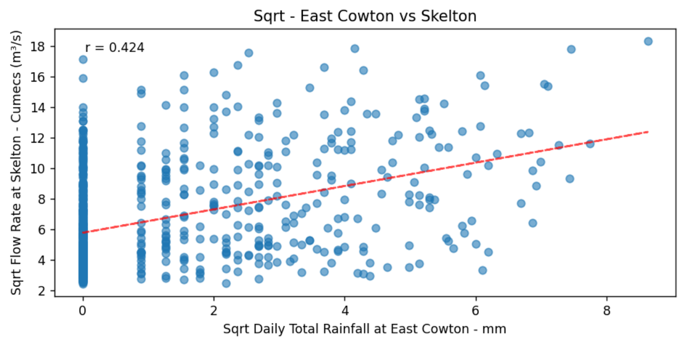


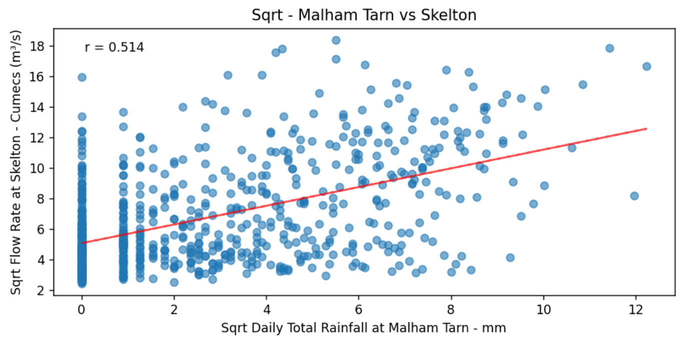
	Crakehill	Skip Bridge	Westwick	Skelton	Arkengarthdale	East Cowton	Malham Tarn	Snaizeholme
Crakehill	1.000	0.943	0.939	0.907	0.395	0.282	0.352	0.399
Skip Bridge	0.943	1.000	0.907	0.904	0.416	0.317	0.354	0.381
Westwick	0.939	0.907	1.000	0.937	0.521	0.339	0.495	0.525
Skelton	0.907	0.904	0.937	1.000	0.555	0.424	0.514	0.565
Arkengarthdale	0.395	0.416	0.521	0.555	1.000	0.715	0.667	0.663
East Cowton	0.282	0.317	0.339	0.424	0.715	1.000	0.541	0.503
Malham Tarn	0.352	0.354	0.495	0.514	0.667	0.541	1.000	0.751
Snaizeholme	0.399	0.381	0.525	0.565	0.663	0.503	0.751	1.000

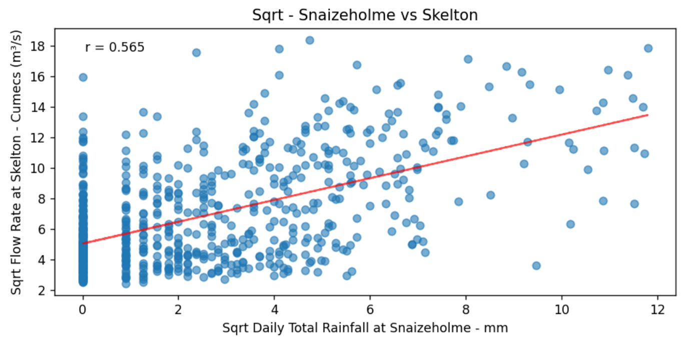
This square root transformation just increases correlation a slight bit more in the rainfall section however decreases correlation slightly in the flow rate. As such we could take an approach of square rooting the rainfall columns and logging the flow rate columns for optimal correlation coefficients for each column. Sadly this approach wouldn't work though as we are comparing this to a transformed predictand column as well, if we have different transformations then this would lead to erroneous and non correlated data. Therefore I decided to compromise and select square root as the transformation for all the data.

Square Rooted Rainfall

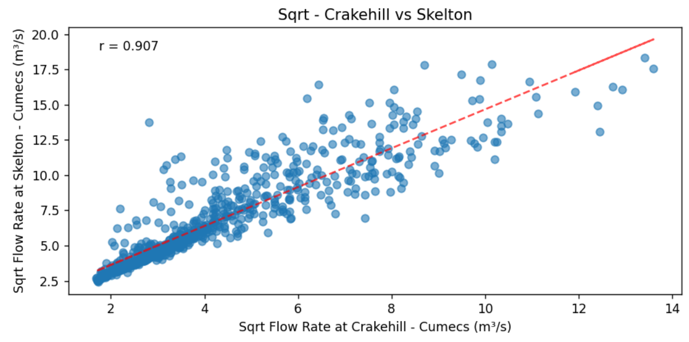


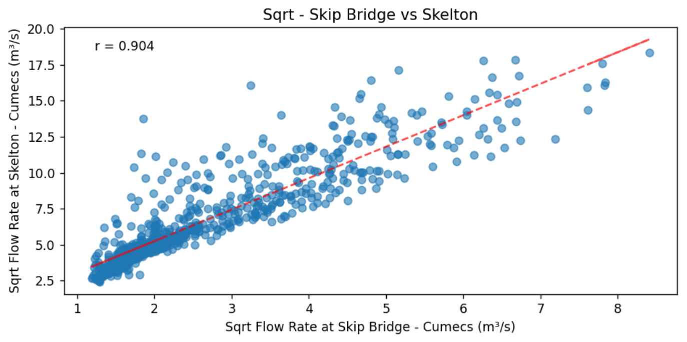


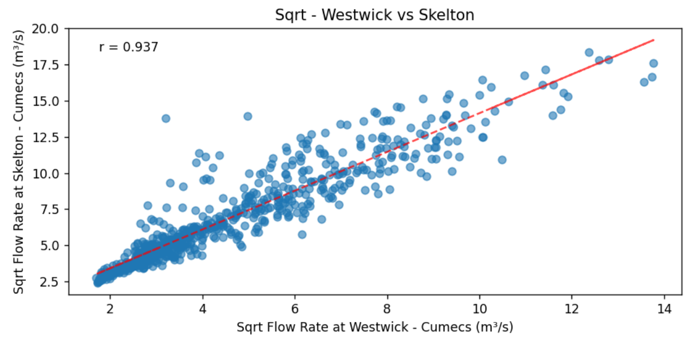




Square Rooted Flow Rate

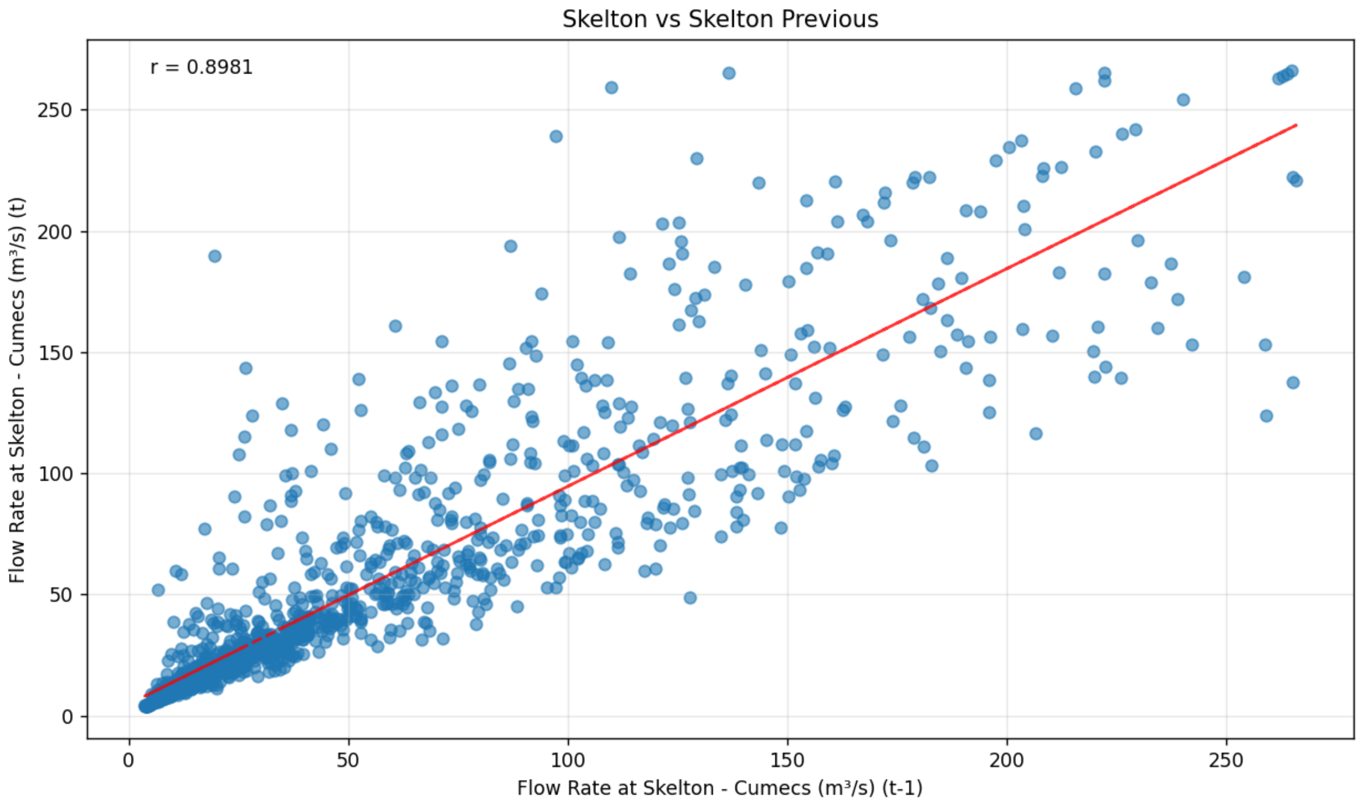


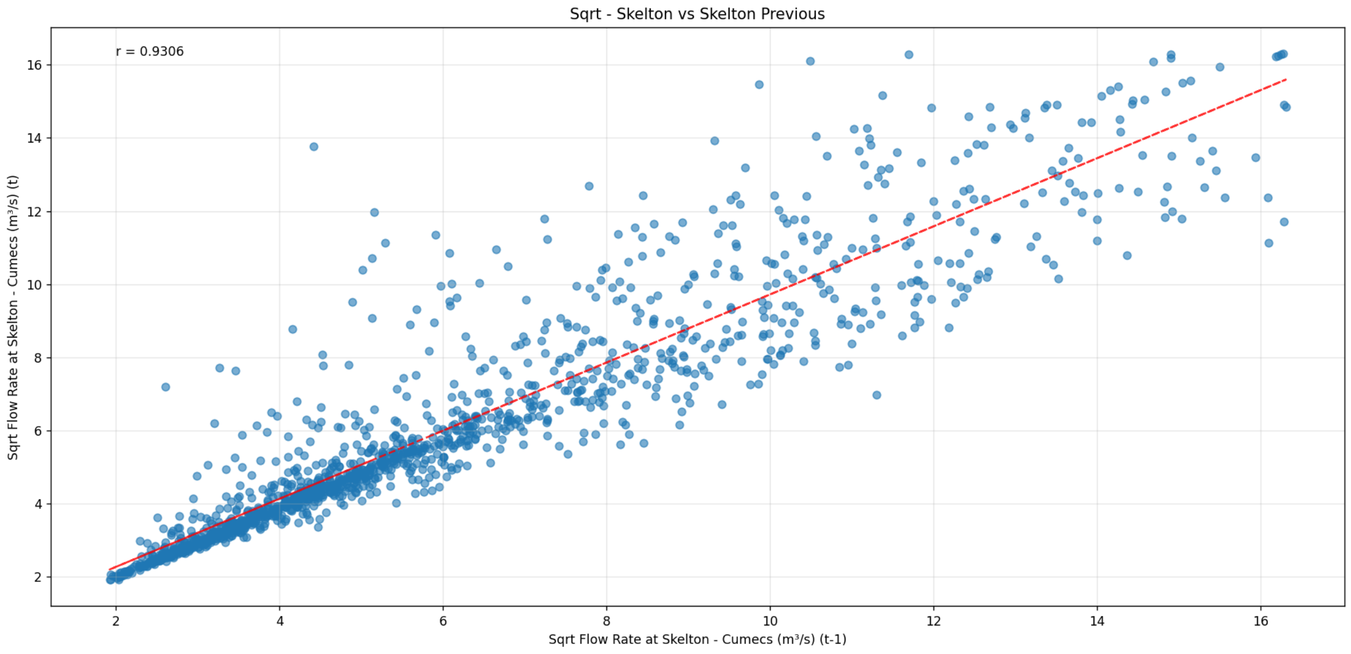




Skelton Previous Day

I also wanted to test using the previous day at Skelton as a predictor for the current day at Skelton, again testing with the square root transformation. This was just shy of the correlation achieved at Westwick.

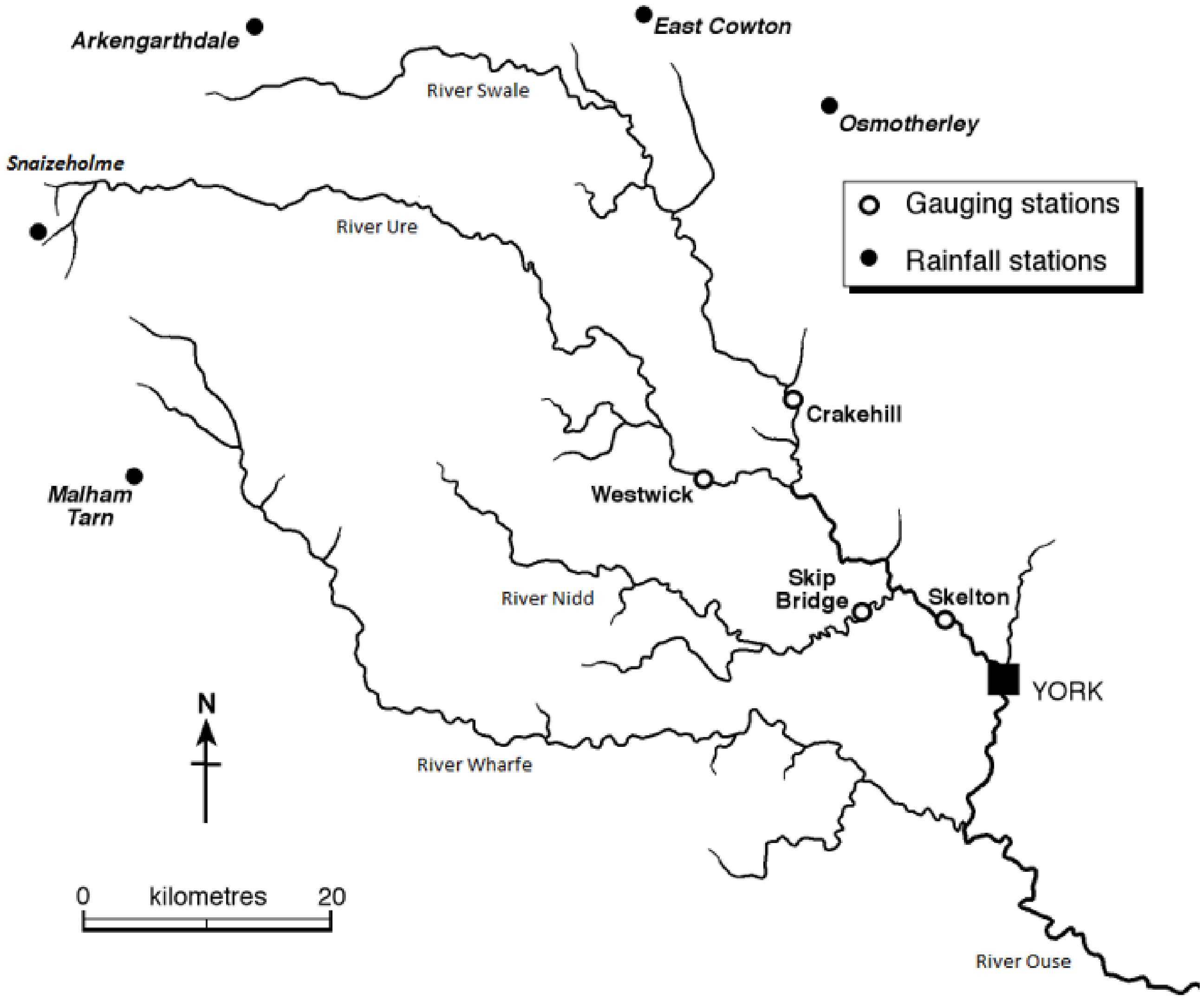




Removing Duplicate Predictors

It may be worth considering removing some predictors if they are too high. This is because repetitions of highly correlated data may lead to a weaker regression model and is simply unnecessary since one of the predictors may be enough. It can also be an issue if multiple are added since their inputs will dominate the other predictors if there are multiple saying the same thing and it won't consider the unique values that the rainfall has to offer because they aren't all duplicate data saying the same thing.

Since this is the case, it might be beneficial for us to remove some of the flow rates since they don't actually offer us up new information as the locations are all downstream from Skelton, as is seen in the image below:



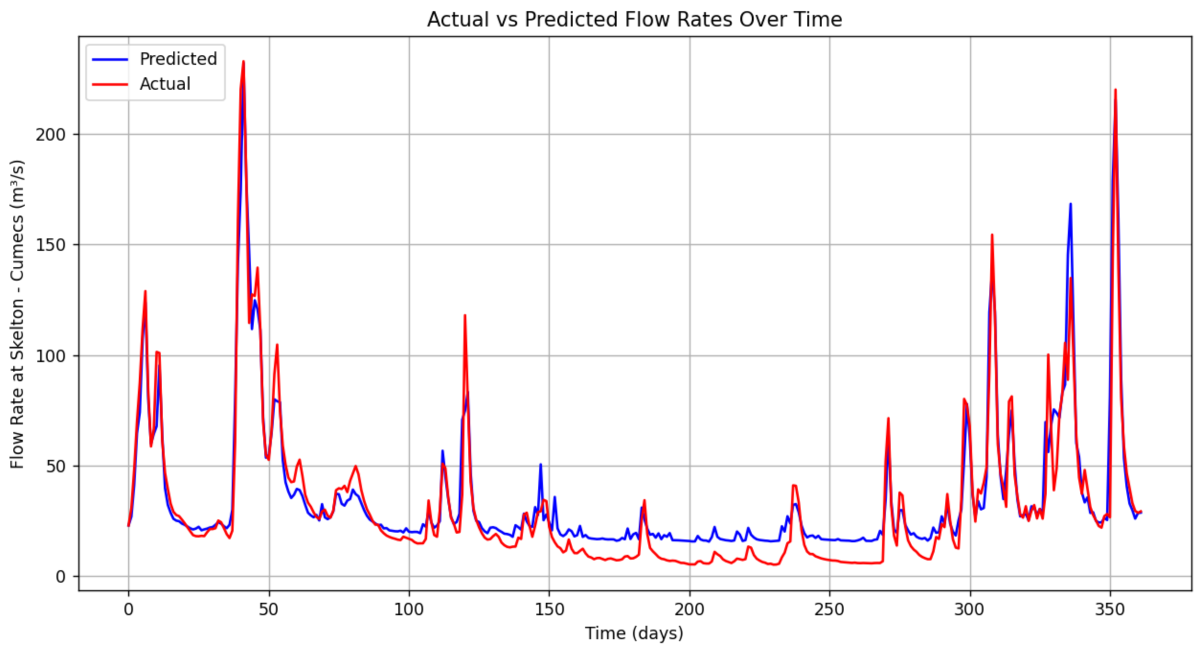
The rainfall locations on the other hand each offer unique sets of information and so we will keep that data within our model. Since we are removing data, we will remove the lowest correlated and keep the most correlated. As I have opted for a square root transformation, the highest correlated value is in the location of Westwick. Therefore I select this as the single column of flow rate that I will add in the model and remove the others since they don't provide any new information that Westwick doesn't offer. I did this at the very end of the model, during testing and implementation I found it easier to include all of these predictors.

1.4 Standardization

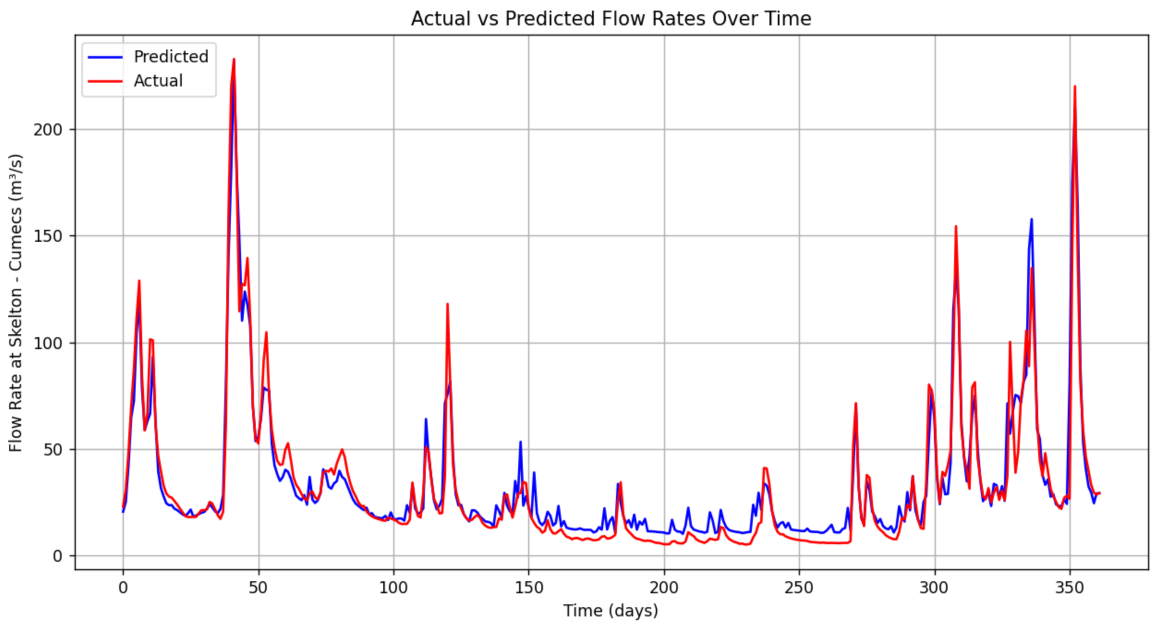
When it came to standardizing the data, I opted to wait and do this process within the program itself. I used the following formula to calculate using min max scaling:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

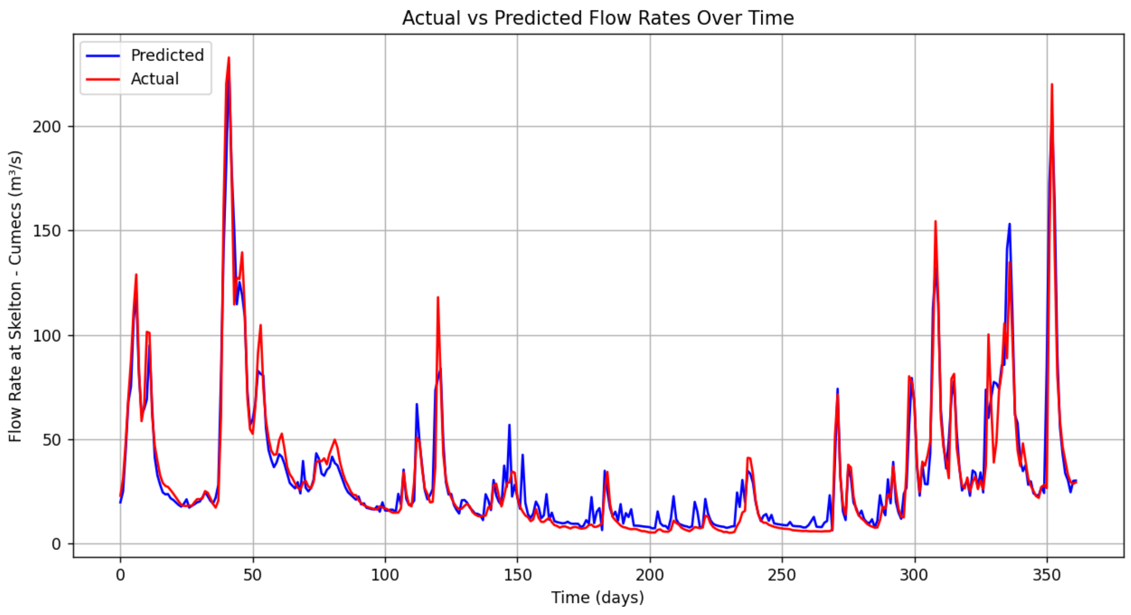
This enabled me to test and better review the output to give me a clearer insight into the benefits of standardizing in different ways. Initially, I decided to go for a range of [0, 1] for my values. This provided a graph that looked like this:



As you can see, the peaks are rather well followed and the general trend of the data appears quite well correlated. However, as you can see at the bottom, there is a large number of values that are predicted as being higher than they actually are. One way of fixing this is to change the range of standardization. As such, I tried using [0.1, 0.9]. Here is the graph that was produced with this:



Now as you can see this is a drastic improvement, however its still not as aligned as I would like it to be. As such I tried an even tighter range of [0.2, 0.8]. This provided an output like so:



As you can see, the graph is now far more aligned and well correlated. I decided to leave this range at [0.2, 0.8] because of this high correlation.

2. MLP Algorithm Implementation

Now that the data processing has been completed we are able to move onto the implementation of the backpropagation algorithm. I will firstly provide the entirety of the base algorithm below, it is thoroughly commented so hopefully shouldn't require much extra explanation. Then following this I will walk you through the additions and improvements made.

For this algorithm I made use of the Python programming language. I selected Python for a variety of reasons. One is that it is a language I am competent and comfortable working in, which made implementing the backpropagation algorithm far smoother. On top of this, Python has a number of useful libraries within it that assist with complex mathematics and data processing (these being numpy and pandas respectively). Numpy was really beneficial in matrix calculations when it came to working out the weights and biases.

As you will see below, my algorithm allows users to input their own datasets, learning parameter, epochs and node counts.

2.1 Base Backpropagation Algorithm

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# ask user for data location
training_input = input("Enter your training data file location: ")
training_data = pd.read_excel(training_input)
validation_input = input("Enter your validation data file location: ")
validation_data = pd.read_excel(validation_input)
test_input = input("Enter your test data file location: ")
test_data = pd.read_excel(test_input)

# ask user for learning parameter and epoch count
learning_parameter = float(input("Enter the learning parameter value: "))
num_epochs = int(input("Enter the number of epochs: "))

# number of neurons at each layer
input_size = int(input("Enter the input size for your network: "))
hidden_size = int(input("Enter the number of hidden nodes: "))
output_size = 1

# prepare data arrays
```

```python
X_train = training_data.iloc[:, :input_size].to_numpy()
Y_train = training_data.iloc[:, input_size].to_numpy()

X_val = validation_data.iloc[:, :input_size].to_numpy()
y_val = validation_data.iloc[:, input_size].to_numpy()

# set X_train, y_train min max
X_min, X_max = X_train.min(axis=0), X_train.max(axis=0)
y_min, y_max = y_train.min(), y_train.max()

def standardize(X, X_min, X_max):
    return 0.8*(X - X_min) / (X_max - X_min) + 0.2

def destandardize(y_norm, y_min, y_max):
    return ((y_norm-0.2)/0.8) * (y_max - y_min) + y_min

# standardize training data using min max scaling
X_train = standardize(X_train, X_min, X_max)
Y_train = standardize(y_train, y_min, y_max)

# standardize validation data using min max scaling
X_val = standardize(X_val, X_min, X_max)
y_val = standardize(y_val, y_min, y_max)

# initialize weights and biases to random variables using formula (-2/n,
2/n) as the range of random values
W1 = np.random.uniform(-2/input_size, 2/input_size, (hidden_size,
input_size))
b1 = np.random.uniform(-2/input_size, 2/input_size, (hidden_size, 1))
W2 = np.random.uniform(-2/hidden_size, 2/hidden_size, (output_size,
hidden_size))
b2 = np.random.uniform(-2/hidden_size, 2/hidden_size, (output_size, 1))

# function to calculate the weighted sum
def weighted_sum(W, x, b):
    return np.dot(W, x) + b

# function to calculate sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

```python
# function used to calculate predictions using weights
def predict(X_new, W1, b1, W2, b2):
    predictions = []
    for row in X_new:
        x = row.reshape(-1, 1)
        Z1 = np.dot(W1, x) + b1
        A1 = sigmoid(Z1)
        Z2 = np.dot(W2, A1) + b2
        A2 = sigmoid(Z2)
        predictions.append(A2.item())
    return np.array(predictions)

# function used to calculate mean squared error
def calculate_mse(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

# function used to save the best model (one with lowest MSE)
def save_best_model(W1, b1, W2, b2):
    return {
        'W1': W1.copy(),
        'b1': b1.copy(),
        'W2': W2.copy(),
        'b2': b2.copy(),
    }

# -----------------------------------
# -----------------------------------
# -----------------------------------
# MAIN BACKPROPAGATION IMPLEMENTATION
# -----------------------------------
# -----------------------------------
# -----------------------------------
for epoch in range(num_epochs):
    # loop over each training sample (gradient descent)
    for i in range(len(X_train)):
        # gets the first row in X_train and turns it into a column vector
        # this aids with numpy calculations
        x = X_train[i].reshape(input_size, 1)
        # sets ‘desired’ values to the C column
        C = y_train[i].reshape(1, 1)

        # forward pass
        S1 = weighted_sum(W1, x, b1)  # hidden layer weighted sum
        u0 = sigmoid(S1)              # hidden layer activation

        S2 = weighted_sum(W2, u0, b2) # output layer weighted sum
        u1 = sigmoid(S2)              # final prediction

        # backwards pass
        # delta for output layer
        delta_output = (C - u1) * (u1 * (1 - u1))
        # delta for hidden layer
        delta_hidden = np.dot(W2.T, delta_output) * (u0 * (1 - u0))

        # Update weights and biases
        W2 += learning_parameter * np.dot(delta_output, u0.T)
        b2 += learning_parameter * delta_output
        W1 += learning_parameter * np.dot(delta_hidden, x.T)
        b1 += learning_parameter * delta_hidden

test_data_predictors = test_data.iloc[:, :input_size].to_numpy()
test_data_predictand = test_data.iloc[:, input_size].to_numpy()

# standardize test data (min-max scaling)
test_data_predictors = standardize(test_data_predictors, X_min, X_max)

standardized_predictions = predict(test_data_predictors, W1, b1, W2, b2)

# destandardize predictions
standardized_predictions = standardized_predictions.flatten()
destandardized_predictions = destandardize(standardized_predictions,
y_min, y_max)
```

This is the base implementation of my multilayer perceptron making use of the backpropagation algorithm. I will give a brief overview of how this algorithm works. To start with, the program assigns some random small weights and biases to the cells in the form of a np array. These values range from [-2/n, 2/n] where n is the number of input nodes. Then there are some functions defined which get used later on.

We then enter our epoch loop. An epoch is pass of the data, as such we iterate through each row of the training data. With each of these rows we make what is called a forward pass. This is where we calculate the weighted sums and activations for each of the nodes. We do this by multiplying our weights by the training data and also adding the biases, this calculates the weighted sum. Then we apply a sigmoid function to this weighted sum and this gives us the activation value. This is conducted on both the input -> hidden node weights as well as the hidden -> output node weights.

Next we make a backwards pass through the data. Here we calculate the value of delta. We calculate delta output first as this value is used in calculating the delta value of the hidden nodes. We use the derivation of the sigmoid function and multiply this by the difference between our predictand and the activation. This is also conducted for the delta hidden. Lastly, we update the weights for this pass and then move onto the next row. We iterate through all the rows in the training data and once we have completed this we can say 1 epoch is complete. Then we run through this for x number of epochs.

2.2 Momentum Enhancement

Implementing momentum doesn't improve the actual model itself but the direction and speed at which it updates parameters to achieve the best solution. It works by initially setting up a momentum constant which is later used to multiply the weights by. You work out the change to the weight from the previous step and multiply this by the constant (typically set to 0.9 for momentum) and add this to the new weight calculation.

```
momentum_constant = 0.9

 ...

# update the weights according to delta values just calculated
W2_new = W2 + learning_parameter * np.dot(delta_output, A1.T)
b2 = b2 + learning_parameter * delta_output

W1_new = W1 + learning_parameter * np.dot(delta_hidden, x.T)
b1 = b1 + learning_parameter * delta_hidden

# momentum
W2_change = W2_new - W2
W1_change = W1_new - W1

W2 = W2_new + W2_change * momentum_constant
W1 = W1_new + W1_change * momentum_constant
```

So the momentum constant is defined at the start of the program and then the bit below that replaces the weight updates within the main loop of the original algorithm. Later on you will see the entire code with the full additions made.

2.3 Bold Driver Enhancement

The bold driver enhancement replaces the fixed learning parameter. It updates the parameter automatically depending on errors that it detects. Typically a learning parameter may be set to say 0.1 for example and this may be good in certain scenarios but can lead to training to oscillate. On the other hand, too small of a parameter may cause a network to become stuck at a local minima, whereby the model thinks its reached the bottom yet it in fact is still yet to be found. Automatically updating depending on MSE fixes this.

If the error has increased since the last time we checked then it could mean the learning rate was too large and therefore lowers it. On the other hand if the error rate has decrease that means we are heading in a direction we want to be and should drive forwards by increasing the learning parameter to try to achieve more of this. It is important to ensure that there are boundaries on this learning parameter though as otherwise it could spiral out of control and become unusable. Below is my implementation of the bold driver enhancement:

```
# bold driver
# updated every 100 to prevent 'snowballing'
if epoch % 100 == 0 and epoch != 0:
    prev_train_error = train_errors[epoch-100]
    if train_error < prev_train_error:
        learning_parameter *= 1.05
    else:
        learning_parameter *= 0.7
    learning_parameter = max(0.01, min(learning_parameter, 0.5))
```

This code goes at the end of each epoch so only occurs once each epoch. But you can make note that it doesn’t update this parameter every epoch, only every 100 since otherwise the system may ‘snowball’, whereby it just keeps heading in one direction.

2.4 Annealing

Annealing, an analogy to thermodynamics, is another way of updating the learning parameter over time. It cannot be used in conjunction with bold driver though as they both do opposite things. Annealing gradually decreases the learning parameter during training, with the end goal of fine-tuning this parameter to more accurately converge on the solution. Below is my implementation of annealing:

```
start_param = learning_parameter
end_param = float(input("Enter end parameter for annealing: "))

def annealing_func(x):
    return end_param + (start_param - end_param) * (1 - (1 / (1 +
np.exp(10 - ((20 * x) / num_epochs)))))

 ...

# annealing
learning_parameter = annealing_func(epoch)
```

Similarly to the bold driver implementation, this learning parameter update takes place at the end of each epoch, this one does update every single epoch however which differs from the bold driver code.

2.5 Weight Decay

Weight decay is a way of letting us avoid over-fitting. Overfitting is where a model learns too much from the training data and rather than learning a general underlying pattern, it learns noise and random fluctuations from the training data, meaning the weights are not applicable to other data sets. Weight decay works by penalizing large weights. This is because large weights can lead to high variance and larger changes in the output. As such you add this penalty term by updating delta output to include beta and omega. These are calculated as such:

```
def calculate_omega(W1, W2):
    n = W1.size + W2.size
    omega = (1 / (2 * n)) * (np.sum(W1**2) + np.sum(W2**2))
    return omega

...
# just before calculating delta_output
beta = 1 / (np.e * learning_parameter)
omega = calculate_omega(W1, W2)

# delta for output layer
delta_output = (predictand - A2 + (beta * omega)) * (A2 * (1 - A2))
```

3. Training and Network Selection

This section will cover how I selected appropriate learning parameters and hidden nodes through testing my neural network repeatedly.

Note: *For all of the MSE calculations you see in this section, the values are based on the normalised data so they may appear exceedingly low. This MSE is only useful whilst training to see if the model is actually learning and making progress but it is not the indicator I used in the evaluation of the final model.*

3.1 Learning Parameter Selection

Note: *In the following section you will see the use of the MSE on the validation data. This is not an accurate representation of the final model and is only useful to see if the model has actually learned and if it has learned more than other models. It can still be effectively used as a comparison for changing hyperparameters.*

Initially, I tested my network using some basic learning parameter values. I tested the following parameters to start with:

Note: *These were tested with hidden nodes set to 10 (default value that will be tested after)*

Learning Parameter	**MSE on validation data after 1000 epochs (average of 10 results each)**
0.001	0.008293
0.01	0.002932
0.1	0.002919

Since there is a low MSE at both 0.1 and 0.01 this leads me to want to take a different approach, one of annealing or bold driver as the results imply a fluctuating learning parameter would be beneficial for this dataset.

3.1.1 Annealing

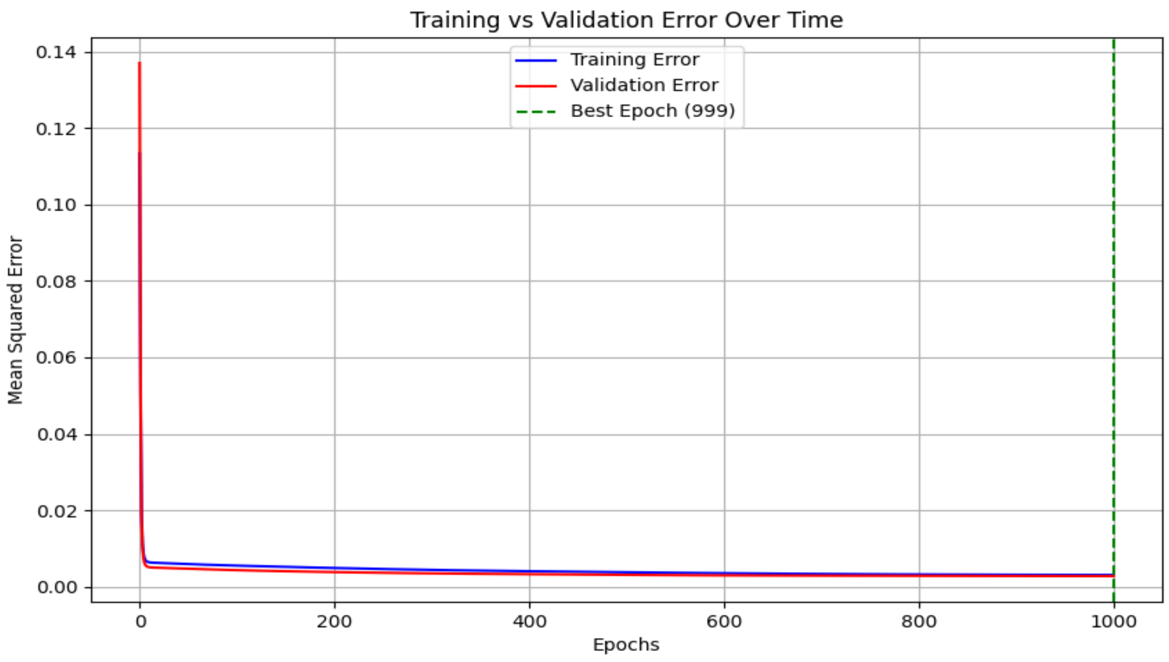
I firstly wanted to try out annealing. I attempted the following values:

Start Parameter	**End Parameter**	**MSE on validation data after 1000 epochs (average of 10 results each)**
0.1	0.001	0.002919
0.1	0.01	0.002953
0.5	0.001	0.002724
0.5	0.01	0.002736
0.5	0.1	0.002864
0.5	0.005	0.002753

I really struggled with this approach as it required such brute forcing and the results seemed to barely differ. As such I thought a better approach may be bold driver, a way of automatically updating the learning parameter meaning the inputs are out of the users hand.

3.1.2 Bold Driver

To start with my bold driver test I selected a value of 0.1 as it seemed wise for the program to work out what it should set this value to on its own. Upon doing this I achieved a similar result to the annealing approach (around 0.0028). However, there was one crucial difference, the best epoch that my algorithm detected on the validation data was the last one.

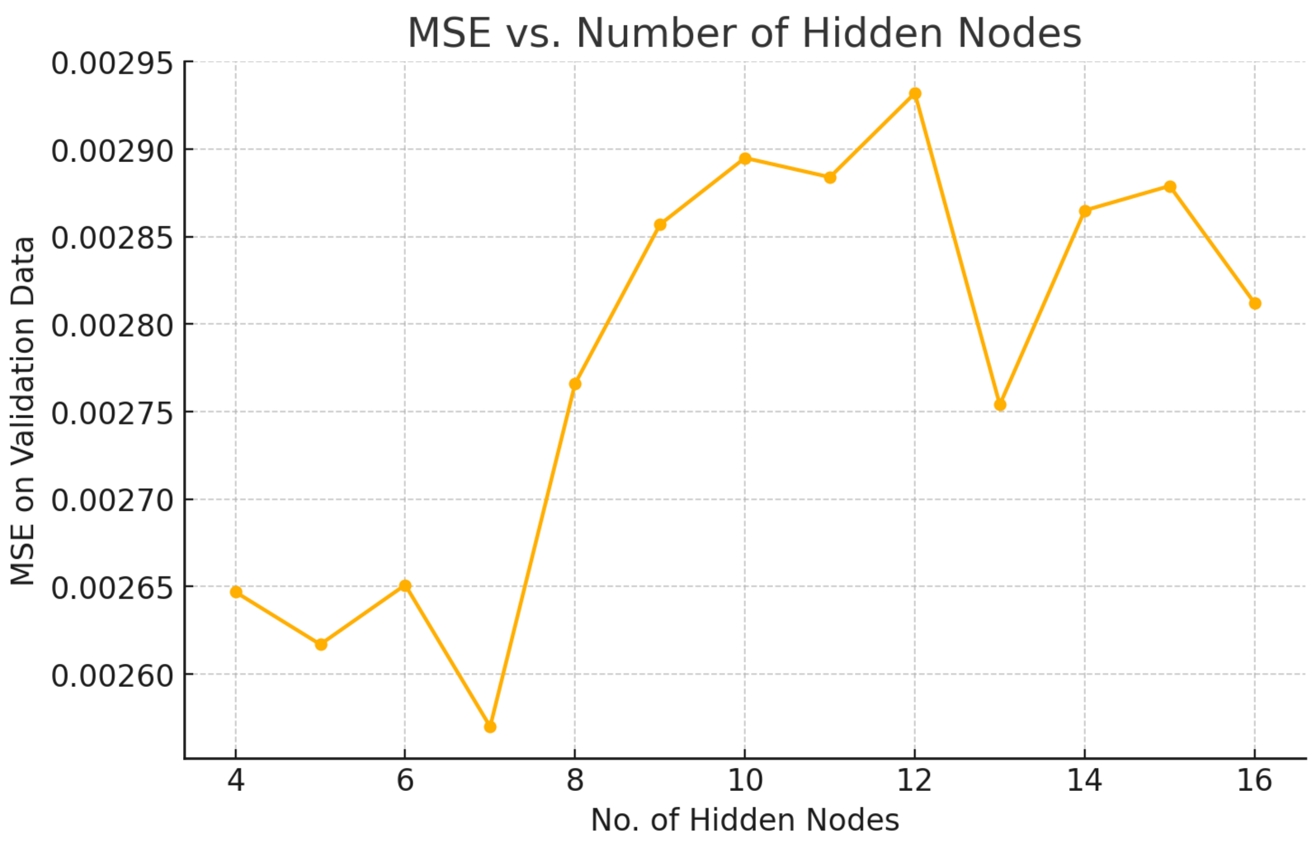


This was not the case for annealing in many cases but for most of the bold driver tests I did, the last epoch was the best epoch. As such I decided to increase the number of epochs when testing for bold driver. This proved very successful, providing an average MSE of 0.002577 across 10 tests. To ensure I was fair to the annealing approach I did test at the same number of epochs but to limited and inconsistent success. Due to this, and the fact that you cannot have both annealing and bold driver in the system, I decided to opt for the bold driver approach with a starting parameter of 0.1.

3.2 Hidden Nodes Selection

Another important factor for selecting the model is the number of hidden nodes. A typical range of values to choose from for a valid hidden node number would be from n/2 to 2n whereby n is the number of input nodes. As we are working with an input size of 8 (3 flow rate locations, Skelton previous days flow rate and 4 rainfall locations) we can test working from 4 to 16. Below is a table displaying the results of testing these:

No. of Hidden Nodes	**MSE on validation data after 1000 epochs (average of 10 results each)**
4	0.002647
5	0.002617
6	0.002651
7	0.002570
8	0.002766
9	0.002857
10	0.002895
11	0.002884
12	0.002932
13	0.002754
14	0.002865
15	0.002879
16	0.002812



3.3 Regularization

It’s very important that once we start reaching high numbers of epochs we prevent overfitting of the dataset. In order to prevent this, we can use MSE as our loss function on not only the training data but also the validation data after each epoch. In this way we are able to save the weights if the MSE is at an all time low for the validation data and then after training we can simply make use of this “best epoch” weights and biases in calculating the predictands for the testing data set. Whilst it is visible in the main code above, I will also display it here for ease of viewing:

```
# calculate and store validation error after each epoch
# prevents overfitting
validation_predictions = predict(X_val, W1, b1, W2, b2)
validation_error = calculate_mse(y_val, validation_predictions)
validation_errors.append(validation_error)

if validation_error < best_validation_error:
    best_validation_error = validation_error
    best_weights = save_best_model(W1, b1, W2, b2)
```

As you can see, the validation errors are added to an array and if the current validation error is lower than the best one found so far, it will save the new lowest validation error and the weights found at that epoch that achieved such good results. In this way we can avoid overfitting the weights to our training data.

Weight decay was also added as an enhancement which is another form of regularization but I will talk more on this in the next section, evaluation of models

3.4 Overview

Final Model: Multi-Layer Perceptron

Training Algorithm: Backpropagation

Network Architecture:

- **Input Layer:** 8 nodes
- **Hidden Layer:** 7 nodes
- **Output Layer:** 1 node

Learning Configuration:

- **Learning Rate:** 0.1
- **Adjustment Strategy:** Bold Driver (instead of Annealing)

From my testing in the above section, these values were optimal based on my findings. As such I will be using these values in the final model and testing of enhancements which you will be able to see in the following section all about evaluation of different models and comparisons.

4. Evaluation

4.1 Types of errors

For my evaluation and comparisons section here, I will use the following metrics to show the accuracy of my model:

- Root Mean Squared Error - RMSE
- Mean Squared Relative Error - MSRE
- Coefficient of Efficiency - CE
- R-Squared

In all of these cases, the following applies:

- Qi is the observed value
- Q̂i is the modelled value
- Q̄ is the mean of the observed values
- Q̃ is the mean of the modelled values
- n is the number of values we are comparing

4.1.1 RMSE

Here is the formula for calculating the RMSE:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}\left(\hat{Q}_i - Q_i\right)^2}{n}}$$

A value of 0 would be a perfect model.
Here is my code snippet used for calculating the RMSE:

```
def calculate_rmse(y_true, y_pred):
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)

    rmse = np.sqrt(np.mean((y_true - y_pred) ** 2))
    return rmse
```

4.1.2 MSRE

Here is the formula for calculating the MSRE:

$$MSRE = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{\hat{Q}_i - Q_i}{Q_i}\right)^2$$

A value of 0 would be a perfect model.
Here is my code snippet used for calculating the MSRE:

```
def calculate_msre(y_true, y_pred):
    # prevent divide by zero
    epsilon = 1e-8
    relative_errors = ((y_pred - y_true) / (y_true + epsilon)) ** 2
    msre = np.mean(relative_errors)
    return msre
```

Above, if the true value is 0 then adding epsilon (a really small value in this case) prevents divisions by this 0 value.

4.1.3 CE

Here is the formula for calculating the CE:

$$CE = 1 - \frac{\sum_{i=1}^{n}\left(\hat{Q}_i - Q_i\right)^2}{\sum_{i=1}^{n}\left(Q_i - \overline{Q}\right)^2}$$

A value of +1 would be a perfect model.
Here is my code snippet used for calculating CE:

```
def calculate_ce(y_true, y_pred):
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)

    numerator = np.sum((y_true - y_pred) ** 2)
    denominator = np.sum((y_true - np.mean(y_true)) ** 2)
    if denominator == 0:
        print("Cannot divide by 0")
        return

    ce = 1 - (numerator / denominator)
    return ce
```

4.1.4 R-Squared

Here is the formula for calculating the R-Squared error:

$$RSqr = \left(\frac{\sum_{i=1}^{n} (Q_i - \bar{Q})(\hat{Q}_i - \tilde{Q})}{\sqrt{\sum_{i=1}^{n} (Q_i - \bar{Q})^2 \sum_{i=1}^{n} (\hat{Q}_i - \tilde{Q})^2}} \right)^2$$

A value of +1 would be a perfect model.
Here is my code snippet used for calculating the R-Squared error:

```
def calculate_rsqr(y_true, y_pred):
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)

    numerator = np.sum((y_true - np.mean(y_true)) * (y_pred -
np.mean(y_pred)))
    denominator1 = np.sum((y_true - np.mean(y_true)) ** 2)
    denominator2 = np.sum((y_pred - np.mean(y_pred)) ** 2)

    if denominator1 == 0 or denominator2 == 0:
        print("Cannot divide by 0")
        return

    rsqr = (numerator ** 2) / (denominator1 * denominator2)
    return rsqr
```

4.2 Plotting the models

In this section I will simply be plotting the following:

- Linear regression (this will be my base model)
- Basic backpropagation (with no enhancements)
- Backpropagation with:
    - Momentum
    - Annealing
    - Bold Driver
    - Weight decay
    - All combined

In terms of how I set up each of the models, this will be based on the results from the previous section. Then in the section following this I will be displaying a table with all of the error rates from these models so we can see which one performs best.

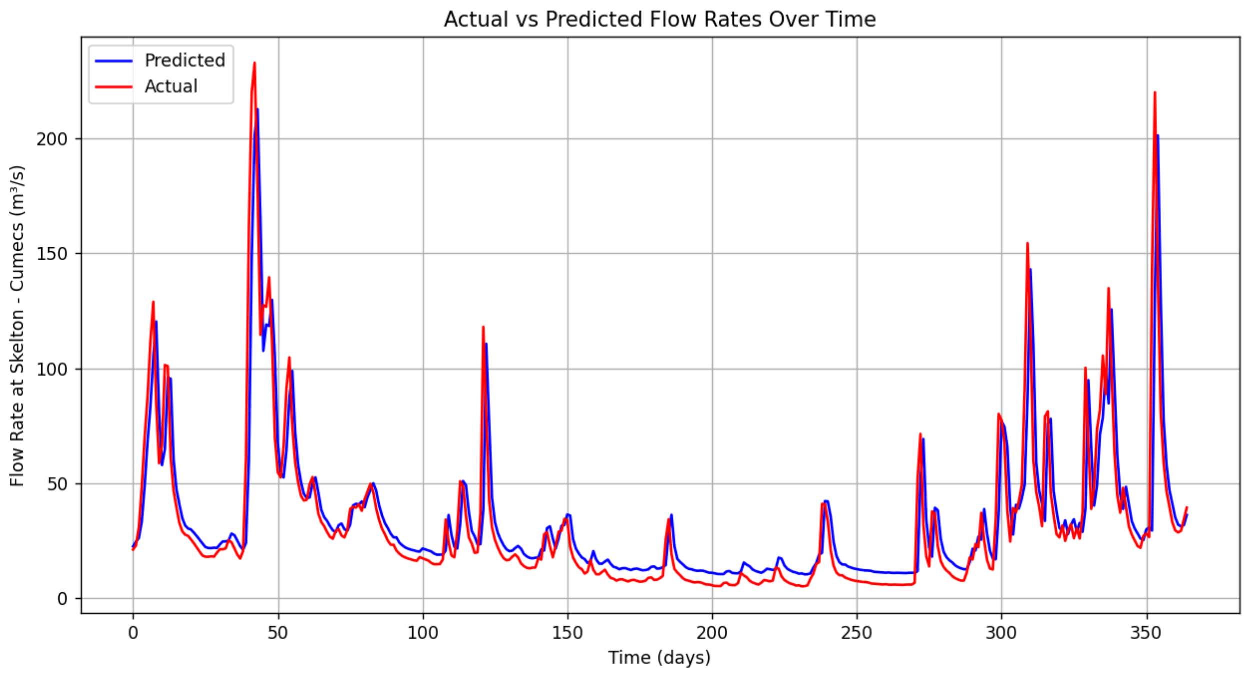
Note: *For all of the graphs you see, the test data is the final year from the data set. They are all line graphs as it is well suited for this time series data as opposed to a scatter graph.*

4.2.1 Linear regression

Firstly, here is the code for my linear regression base algorithm:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# -------------------------------
# -------------------------------
# Simple Linear Regression
# -------------------------------
# -------------------------------
def linear_regression(X, y):
    # convert to numpy arrays if they aren't already
    X = np.array(X)
    y = np.array(y)

    mean_x = np.mean(X)
    mean_y = np.mean(y)

    numerator = np.sum((X - mean_x) * (y - mean_y))
    denominator = np.sum((X - mean_x) ** 2)

    if denominator == 0:
        print("Error, can't divide by 0")
        return

    slope = numerator / denominator
    intercept = mean_y - slope * mean_x

    return intercept, slope

def predict_linear_regression(X, intercept, slope):
    return intercept + slope * X

def main():
    df = pd.read_csv('training_data.csv')

    # create the predictor, Skelton on previous day
    df['Skelton_previous'] = df['Skelton'].shift(1)

    # removes rows where Skelton_previous is NaN
    df.dropna(subset=['Skelton_previous'], inplace=True)

    X = df['Skelton_previous'].values
    y = df['Skelton'].values

    # split, 3/4 are training and 1/4 testing
    train_size = int(0.75 * len(X))
    X_train, X_test = X[:train_size], X[train_size:]
    y_train, y_test = y[:train_size], y[train_size:]

    intercept, slope = linear_regression(X_train, y_train)

    y_pred = predict_linear_regression(X_test, intercept, slope)

main()
```

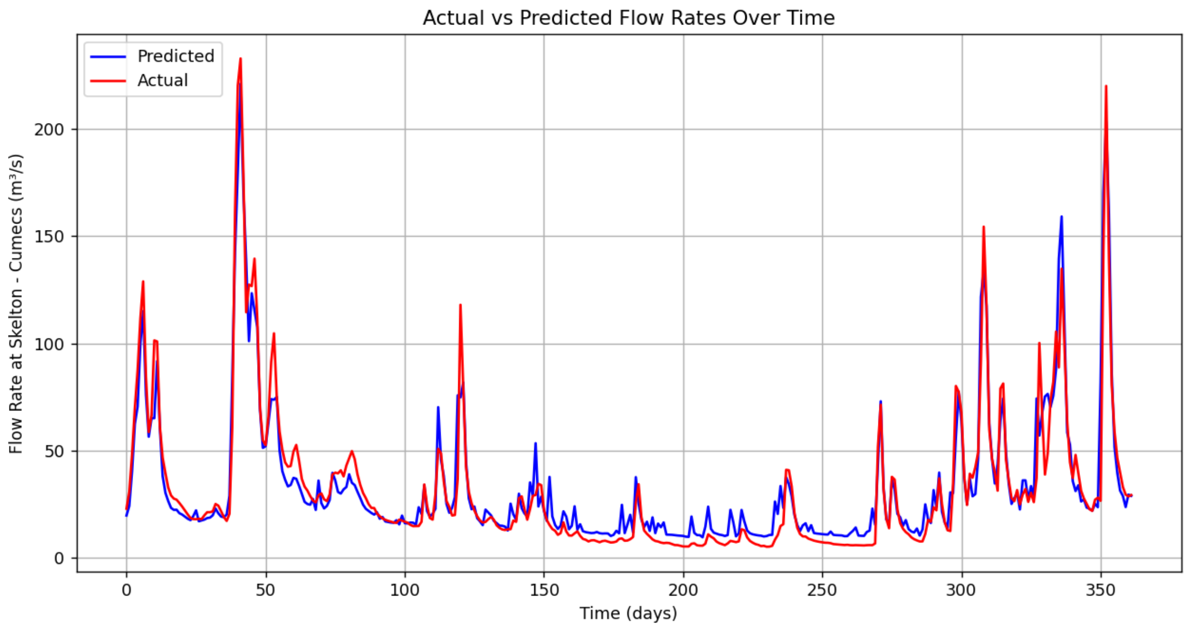
Below you can find the graph:



As you can see it follows the general trend of the data quite accurately. However one thing you may notice is that it is delayed by a little chunk of time. This is because the model needs time to work out the direction of the previous days before it can start accurately predicting. As such, whilst it may appear beneficial, in reality, it may only accurately predict days quite rarely. This is reflected in the error table at the end.

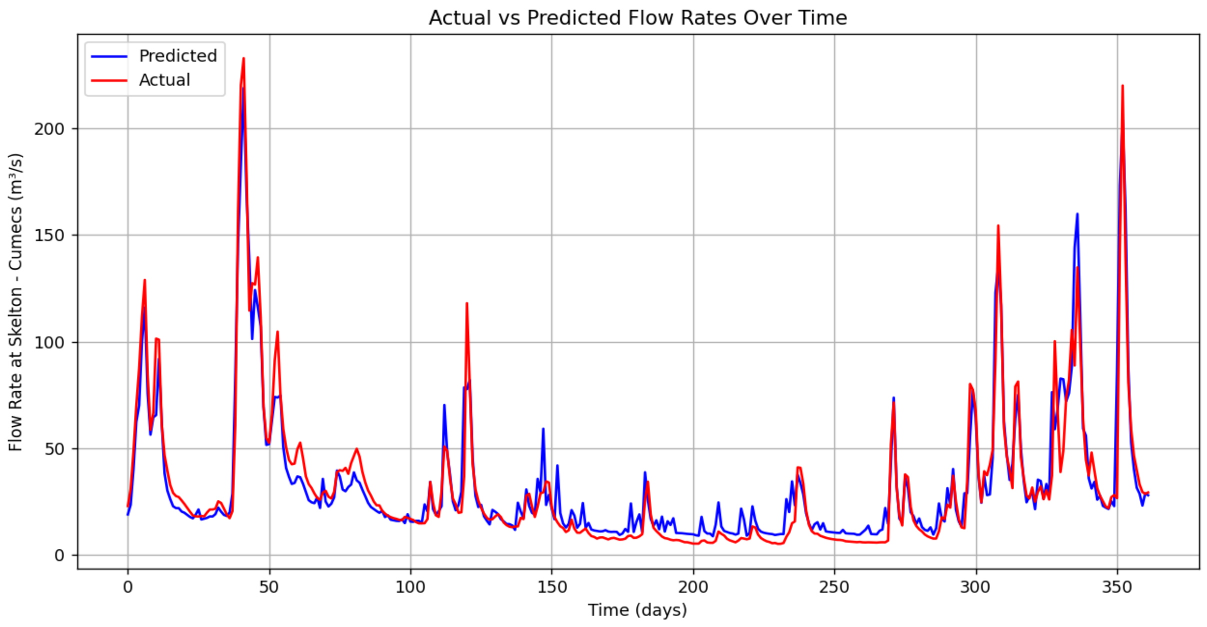
4.2.2 Basic Backpropagation

This is the base backpropagation implementation that you saw earlier, here is the graph after 5000 epochs using the optimal parameters identified in the previous section:



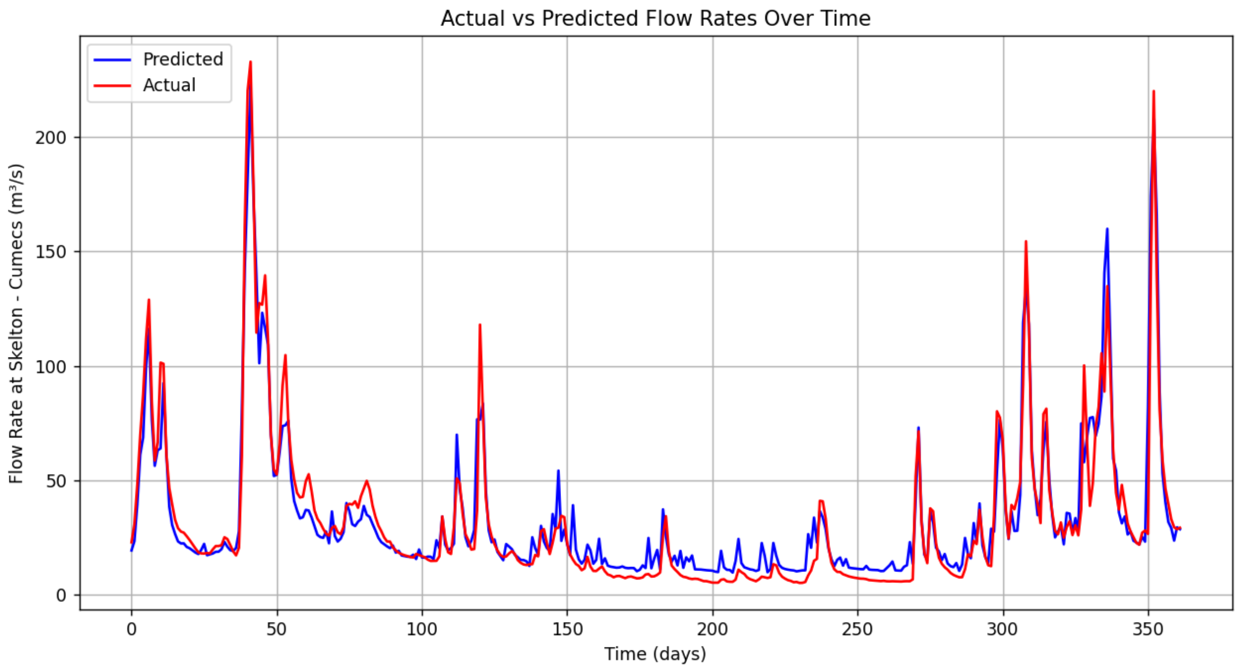
4.2.3 Backpropagation - Momentum

This is the base backpropagation implementation with the addition of momentum, here is the graph after 5000 epochs using the optimal parameters identified in the previous section:



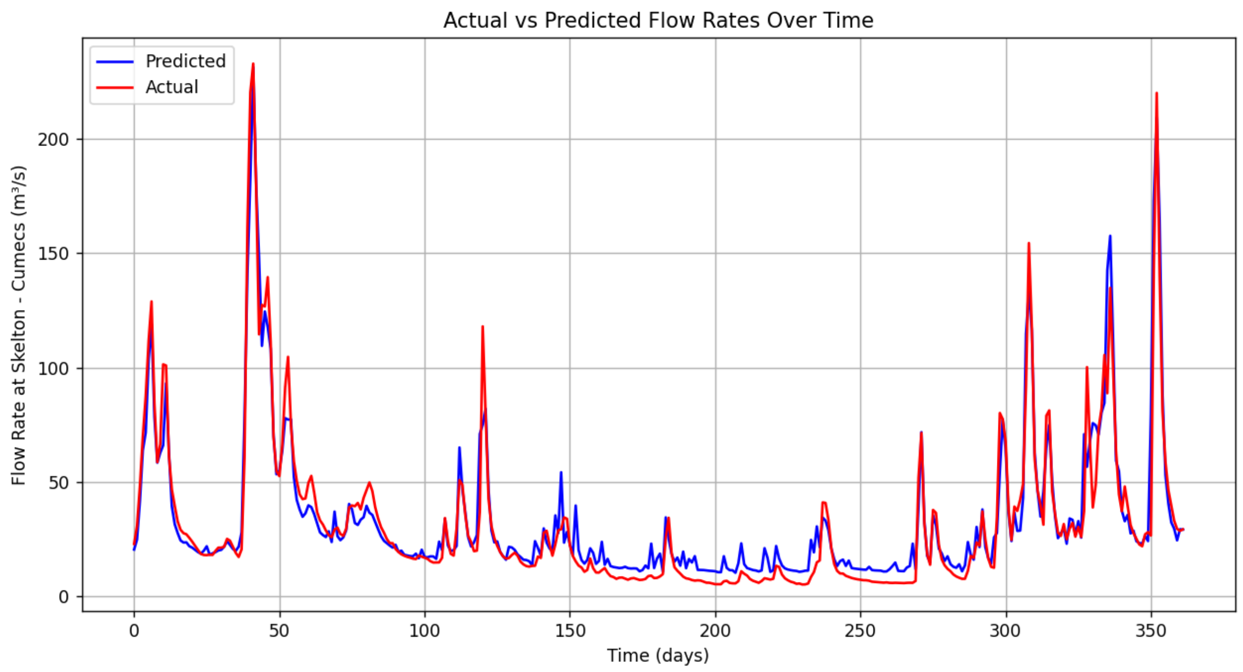
4.2.4 Backpropagation - Annealing

This is the base backpropagation implementation with the addition of annealing, here is the graph after 5000 epochs using the optimal parameters identified in the previous section:



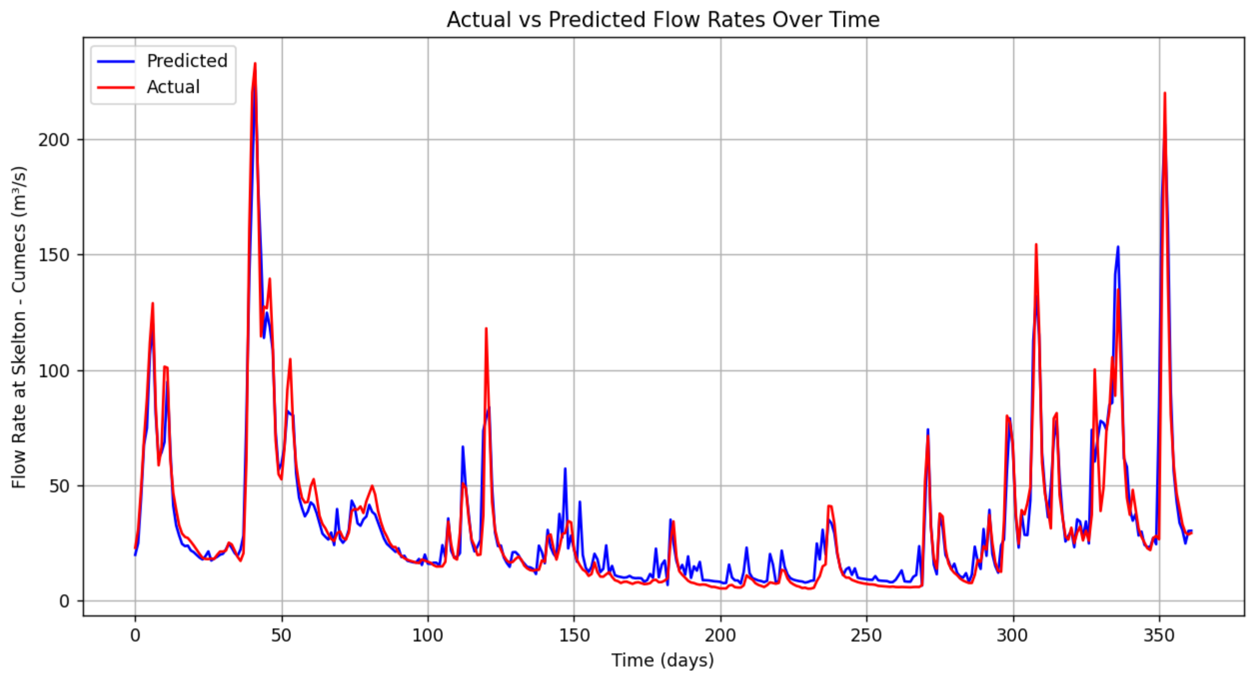
4.2.5 Backpropagation - Bold Driver

This is the base backpropagation implementation with the addition of bold driver, here is the graph after 5000 epochs using the optimal parameters identified in the previous section:



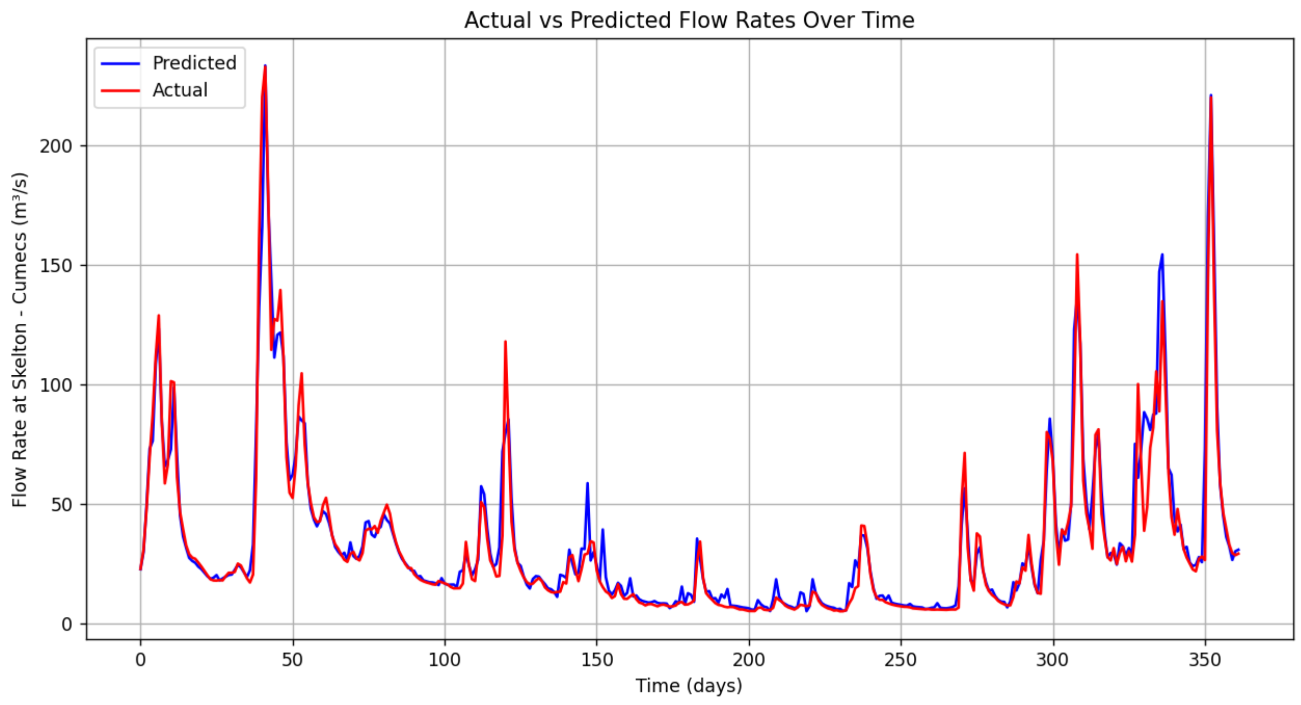
4.2.6 Backpropagation - Weight Decay

This is the base backpropagation implementation with the addition of weight decay, here is the graph after 5000 epochs using the optimal parameters identified in the previous section:



This plot differs most from the previous ones as you can see the gap at the bottom has been minimised, meaning the model is no longer overfitted on the training data.

4.2.7 Backpropagation - All Enhancements



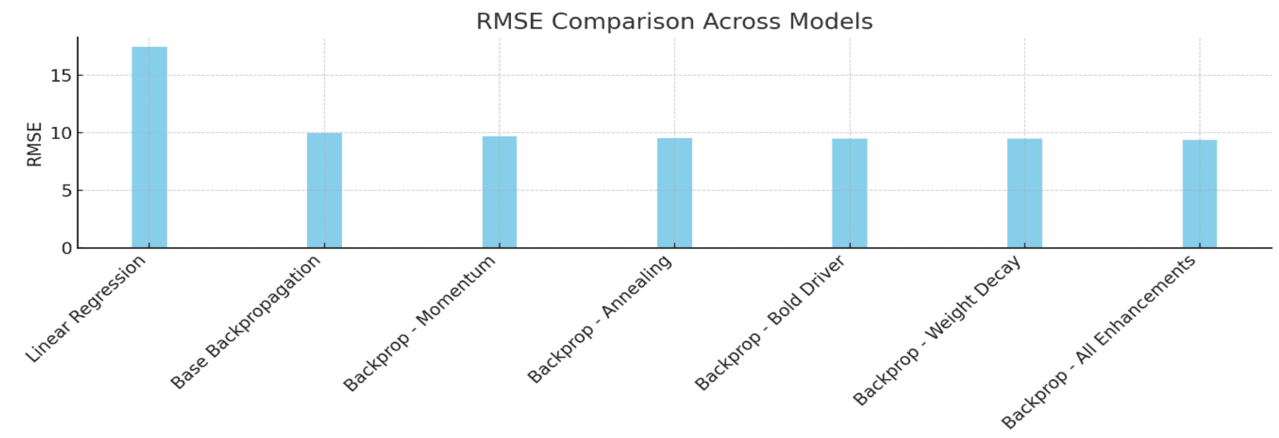
As you can see with all the enhancements (bold driver rather than annealing), the model produces a way cleaner result, with the peaks and valleys being followed very accurately throughout the plot.

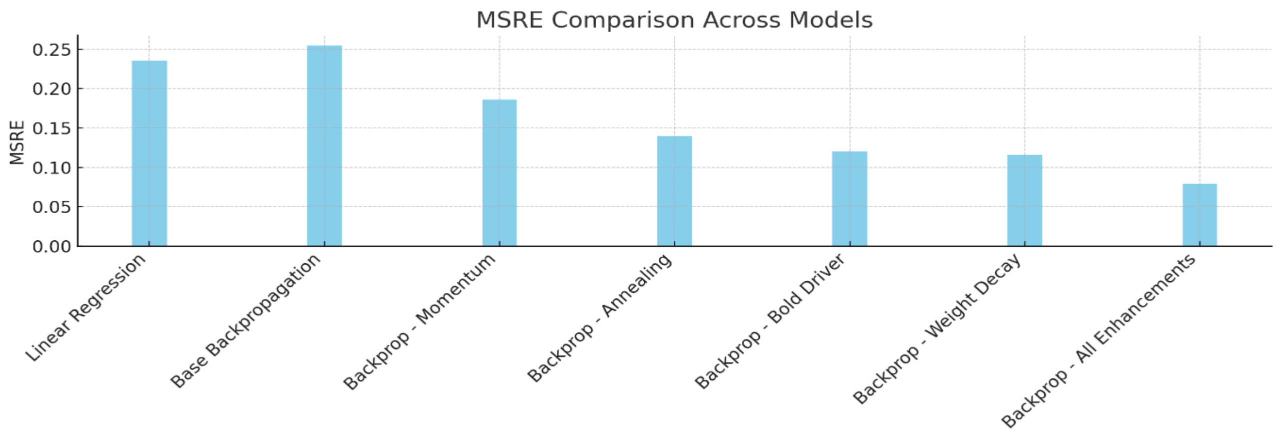
4.3 Error Table

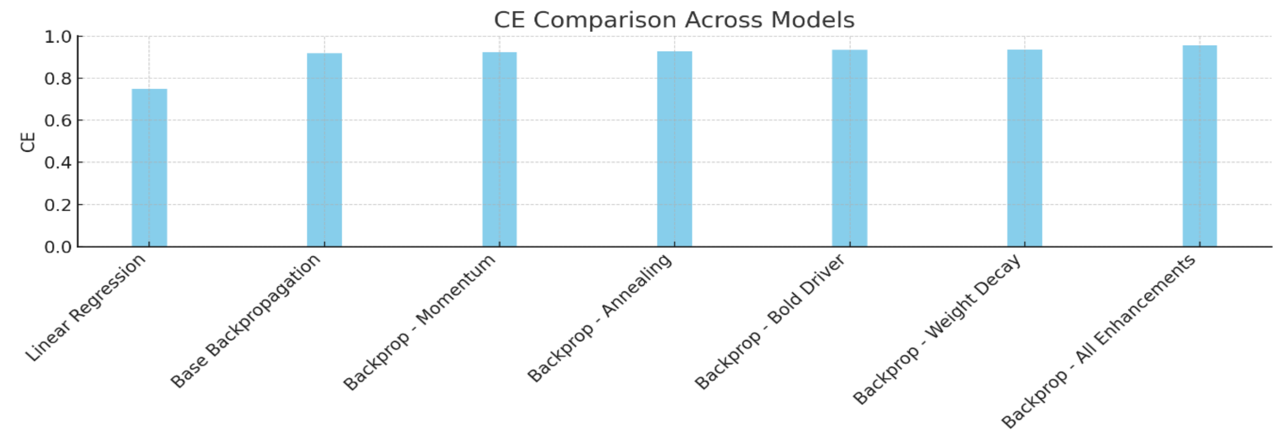
Below you can see an error table displaying each of the different error rates for the different models.

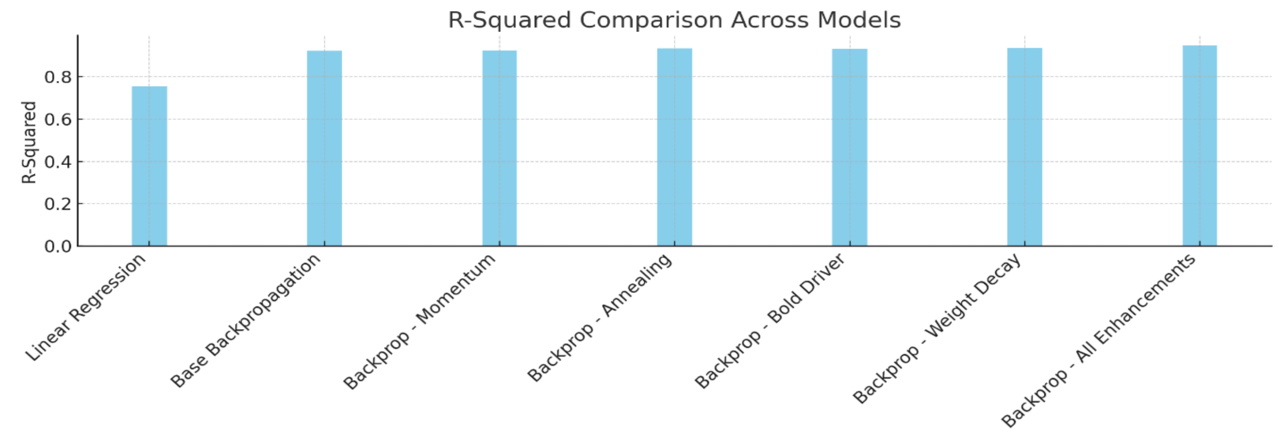
Model	**RMSE**	**MSRE**	**CE**	**R-Squared**
Linear Regression	17.438221	0.255558	0.749933	0.7540126
Base Backpropagation	9.928513	0.235024	0.919243	0.921898
Backpropagation - Momentum	9.689760	0.186129	0.923700	0.924692
Backpropagation - Annealing	9.509266	0.139375	0.929291	0.932869
Backpropagation - Bold Driver	9.437412	0.120153	0.935604	0.931953
Backpropagation - Weight Decay	9.484231	0.115987	0.938112	0.936745
Backpropagation - All Enhancements	9.342525	0.079472	0.955805	0.948371

Clearly as you can see from the data above, the linear regression model, whilst not awful, doesn’t stack up against the different iterations of the backpropagation algorithm, especially not the one that has all of the enhancements. Below is a clearer representation of this differences between the models.


RMSE Comparison Across Models
RMSE
0
5
10
15
Linear Regression
Base Backpropagation
Backprop - Momentum
Backprop - Annealing
Backprop - Bold Driver
Backprop - Weight Decay
Backprop - All Enhancements


MSRE Comparison Across Models
MSRE
0.00
0.05
0.10
0.15
0.20
0.25
Linear Regression
Base Backpropagation
Backprop - Momentum
Backprop - Annealing
Backprop - Bold Driver
Backprop - Weight Decay
Backprop - All Enhancements


CE Comparison Across Models
CE
0.0
0.2
0.4
0.6
0.8
1.0
Linear Regression
Base Backpropagation
Backprop - Momentum
Backprop - Annealing
Backprop - Bold Driver
Backprop - Weight Decay
Backprop - All Enhancements


R-Squared Comparison Across Models
R-Squared
0.0
0.2
0.4
0.6
0.8
Linear Regression
Base Backpropagation
Backprop - Momentum
Backprop - Annealing
Backprop - Bold Driver
Backprop - Weight Decay
Backprop - All Enhancements

The models improved each time after the addition of the enhancements, most notably at weight decay in terms of visually on the plot. You can see that the plot is no longer overfitted to the training data and the predicted values follow the actual values much more closely than other graphs. Bold driver just slightly edges out annealing and takes the win in my testing which is why I had that included instead of annealing (as you cannot have both within the model). Also momentum kept the model moving efficiently and prevented it getting stuck in local minima which led to a slightly better result than the original model.

4.4 Final Model

Below is my code which contains all enhancements:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


# ask user for data location
training_input = "train_data_lagged.xlsx"
training_data = pd.read_excel(training_input)
validation_input = "validation_data_lagged.xlsx"
validation_data = pd.read_excel(validation_input)
test_input = "test_data_lagged.xlsx"
test_data = pd.read_excel(test_input)


# ask user for learning parameter and epoch count
learning_parameter = float(input("Enter the learning parameter value: "))
start_param = learning_parameter
end_param = float(input("Enter end parameter for annealing: "))
num_epochs = int(input("Enter the number of epochs: "))
momentum_constant = 0.9


# number of neurons at each layer
input_size = int(input("Enter the input size for your network: "))
hidden_size = int(input("Enter the number of hidden nodes: "))
output_size = 1


# prepare data arrays
```

```
X_train = training_data.iloc[:, :input_size].to_numpy()
Y_train = training_data.iloc[:, input_size].to_numpy()

X_val = validation_data.iloc[:, :input_size].to_numpy()
y_val = validation_data.iloc[:, input_size].to_numpy()

# set X_train, y_train min max
X_min, X_max = X_train.min(axis=0), X_train.max(axis=0)
y_min, y_max = y_train.min(), y_train.max()

def standardize(X, X_min, X_max):
    return 0.8*(X - X_min) / (X_max - X_min) + 0.2

def destandardize(y_norm, y_min, y_max):
    return ((y_norm-0.2)/0.8) * (y_max - y_min) + y_min

# standardize training data using min max scaling
X_train = standardize(X_train, X_min, X_max)
Y_train = standardize(y_train, y_min, y_max)

# standardize validation data using min max scaling
X_val = standardize(X_val, X_min, X_max)
y_val = standardize(y_val, y_min, y_max)

# initialize weights and biases to random variables using formula (-2/n,
2/n) as the range of random values
W1 = np.random.uniform(-2/input_size, 2/input_size, (hidden_size,
input_size))
b1 = np.random.uniform(-2/input_size, 2/input_size, (hidden_size, 1))
W2 = np.random.uniform(-2/hidden_size, 2/hidden_size, (output_size,
hidden_size))
b2 = np.random.uniform(-2/hidden_size, 2/hidden_size, (output_size, 1))


# function to calculate the weighted sum
def weighted_sum(W, x, b):
    return np.dot(W, x) + b


# function to calculate sigmoid function
```

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


# function used to calculate predictions using weights
def predict(X_new, W1, b1, W2, b2):
    predictions = []
    for row in X_new:
        x = row.reshape(-1, 1)
        Z1 = np.dot(W1, x) + b1
        A1 = sigmoid(Z1)
        Z2 = np.dot(W2, A1) + b2
        A2 = sigmoid(Z2)
        predictions.append(A2.item())
    return np.array(predictions)

def calculate_omega(W1, W2):
    n = W1.size + W2.size
    omega = (1 / (2 * n)) * (np.sum(W1**2) + np.sum(W2**2))
    return omega

def annealing_func(x):
    return end_param + (start_param - end_param) * (1 - (1 / (1 +
np.exp(10 - ((20 * x) / num_epochs)))))

# function used to calculate mean squared error
def calculate_mse(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)


# function used to save the best model (one with lowest MSE)
def save_best_model(W1, b1, W2, b2):
    return {
        'W1': W1.copy(),
        'b1': b1.copy(),
        'W2': W2.copy(),
        'b2': b2.copy(),
    }
```

```python
# arrays to store error rates, useful for evaluating model
train_errors = []
validation_errors = []


best_validation_error = float('inf')
best_weights = None


# -----------------------------------
# -----------------------------------
# -----------------------------------
# MAIN BACKPROPAGATION IMPLEMENTATION
# -----------------------------------
# -----------------------------------
# -----------------------------------
for epoch in range(num_epochs):
    # loop over each training sample (gradient descent)
    for i in range(len(X_train)):
        # gets the first row in X_train and turns it into a column vector
        # this aids with numpy calculations
        x = X_train[i].reshape(input_size, 1)
        # sets ‘desired’ values to the C column
        C = y_train[i].reshape(1, 1)

        # forward pass
        S1 = weighted_sum(W1, x, b1)  # hidden layer weighted sum
        u0 = sigmoid(S1)              # hidden layer activation

        S2 = weighted_sum(W2, u0, b2) # output layer weighted sum
        u1 = sigmoid(S2)              # final prediction


        # weight decay
        beta = 1 / (np.e * learning_parameter)
        omega = calculate_omega(W1, W2)

        # backwards pass
        # delta for output layer
        delta_output = (C - u1 + (beta * omega)) * (u1 * (1 - u1))
        # delta for hidden layer
        delta_hidden = np.dot(W2.T, delta_output) * (u0 * (1 - u0))

        # update the weights according to delta values just calculated
        W2_new = W2 + learning_parameter * np.dot(delta_output, A1.T)
        b2 = b2 + learning_parameter * delta_output

        W1_new = W1 + learning_parameter * np.dot(delta_hidden, x.T)
        b1 = b1 + learning_parameter * delta_hidden

        # momentum
        W2_change = W2_new - W2
        W1_change = W1_new - W1

        W2 = W2_new + W2_change * momentum_constant
        W1 = W1_new + W1_change * momentum_constant


    # bold driver
    # updated every 100 to prevent 'snowballing'
    if epoch % 100 == 0 and epoch != 0:
        prev_train_error = train_errors[epoch-100]
        if train_error < prev_train_error:
            learning_parameter *= 1.05
        else:
            learning_parameter *= 0.7
        learning_parameter = max(0.01, min(learning_parameter, 0.5))

    '''
    # annealing (if thats what you choose)
    learning_parameter = annealing_func(epoch)
    '''

    # calculate and store training error after each epoch
    train_predictions = predict(X_train, W1, b1, W2, b2)
    train_error = calculate_mse(y_train, train_predictions)
    train_errors.append(train_error)

    # calculate and store validation error after each epoch
    # prevents overfitting
```

```
    validation_predictions = predict(X_val, W1, b1, W2, b2)
    validation_error = calculate_mse(y_val, validation_predictions)
    validation_errors.append(validation_error)


    if validation_error < best_validation_error:
        best_validation_error = validation_error
        best_weights = save_best_model(W1, b1, W2, b2)


test_data_predictors = test_data.iloc[:, :input_size].to_numpy()
test_data_predictand = test_data.iloc[:, input_size].to_numpy()

# standardize test data (min-max scaling)
test_data_predictors = standardize(test_data_predictors, X_min, X_max)

standardized_predictions = predict(test_data_predictors, W1, b1, W2, b2)

# destandardize predictions
standardized_predictions = standardized_predictions.flatten()
destandardized_predictions = destandardize(standardized_predictions,
y_min, y_max)
```