Introduction - F325508

CONTENTS

Introduction - F325508

INTRODUCTION

I used Python for this project because I've worked with it before during a data science course. It was familiar and comfortable for me, especially since I've already used some of the data-cleaning techniques needed for this coursework.

To make my project work, I used several libraries:

- **Pandas**: This helped me read Excel files, create data frames, convert dates to datetime, and make sure numeric data was in the right format.
- **Numpy**: I used this to create arrays, calculate dot products with matrices, and handle the activation and error functions in my MLP.
- **Matplotlib**: This library was great for plotting graphs, which helped me see how my data cleaning and MLP improvements were progressing visually.
- **Seaborn**: I used it to create heatmaps so I could easily spot correlations in the data.
- **Scipy.interpolate**: This made it easy to fill in gaps in my data using interpolation.
- **Scipy.stats**: I relied on this to calculate the correlation between my model's predictions and actual results.

I built my MLP (Multi-Layer Perceptron) class using object-oriented programming. It includes methods for things like:

- Calculating error metrics (e.g., Root Mean Square Error, Mean Square Error, Absolute Error) and their derivatives.
- Activation functions (Sigmoid, Tanh, ReLU) and their derivatives.
- The forward pass, backward pass, weight updates, training, and handling the weight loss function.

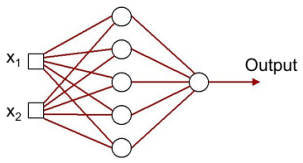
I used matrices to store and update weights throughout the training process, making the calculations more efficient.

PLAN - F325508

PLAN

STEP 1. Gather data - Ensure sufficient data are available for a meaningful study in terms of both quantity and quality

STEP 2. Select predictand(s) (what are we trying to predict?)



STEP 3. Data preprocessing (Stage 1)

3.1 Data cleansing - Remove significant underlying upward or downward trends (eg, using first differences or remove regression line), seasonal components, missing data, errors, skewness.

3.2 Predictors - Identify the most significant predictors for the chosen predictand. If necessary, reduce the number of predictors by means of principal components. Determine suitable lag times for each predictor and calculate appropriate moving averages for the predictors when applicable. **Feature Construction and Feature Encoding**.

STEP 4. ANN Selection

4.1 Network type - Select the most appropriate network type for the application.

4.2 Training algorithm - Select a suitable training algorithm to modify weights and biases and determine network architecture. Choose appropriate values for learning parameters (momentum and learning rate).

STEP 5. Data preprocessing (Stage 2)

5.1 Data standardization - According to the algorithm chosen, standardize data to the ranges [0,1], [-1,1], [0.1,0.9], etc., or normalize the data.

5.2 Data sets - Create cross validation data sets by splitting the data into appropriate calibration, testing and validation sets. With a large data set this is relatively easy as the data can be split into three representative sets. However, for smaller data sets, k-fold cross training should be used

STEP 6. Network Training

6.1 Architecture - Specify the number of hidden layers and number of nodes in these layers. This may be unnecessary if a pruning algorithm, or cascade correlation is used. Begin with one hidden layer – probably enough.

6.2 Training - Train a number of networks using the calibration and validation data. Terminate the training process when results from the validation data indicate over-fitting to the calibration set.

STEP 7. Evaluation - Select error measures that are appropriate to the model output and purpose. Compare results with those derived from alternative model configurations and alternative methods. Use Test set for evaluation.

STEP 1 AND 2 - F325508

STEP 1 AND 2

Thankfully you provided us with data in the excel spreadsheet, so I just used that and it was definitely sufficient data due to it summing up to 4 years' worth of rainfall and river flow.

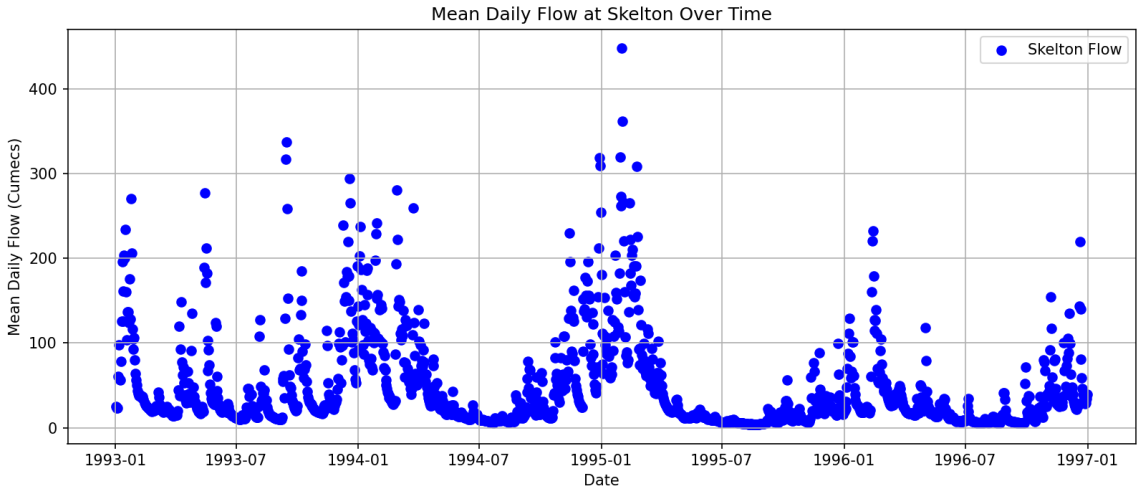
We also know the predictand will be Skelton tomorrow.

STEP 3 DATA PREPROCESSING (STAGE 1)

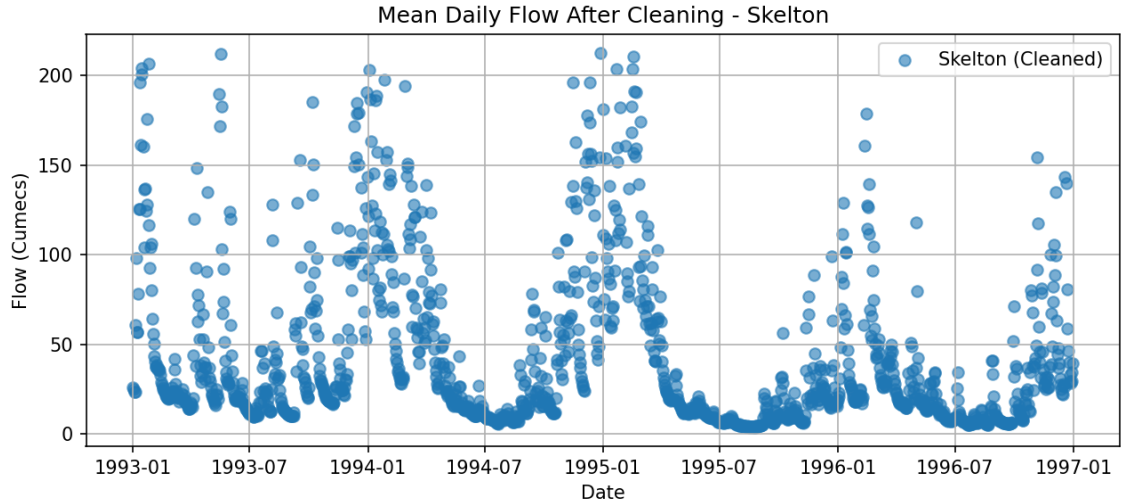
STEP 3.1 – DATA PREPROCESSING

First, I plotted each of the mean daily river flow at all the rivers to see the data we are working on. This helps if I can visually see any outliers and so after cleansing the data I can then compare with the data before cleaning to see how many outliers were removed.

Below you can I have plotted all the data points from the mean daily flow at Skelton over the period of the data given. This is the data before any data cleansing has taken place.



Below this you can see my data points after cleaning, I used 3 standard deviations and then on top of that also Interquartile Range to ensure that any data points lost would be definite outliers so my data cleansing would be sufficient. All data that my data cleansing deemed to be outliers were set to nan but not deleted so I can use interpolation to fill those data points back in. I also made sure to set any data that is "-999" or "a" to nan too just to make sure my data will be accurate.



STEP 3 DATA PREPROCESSING (STAGE 1) - F325508

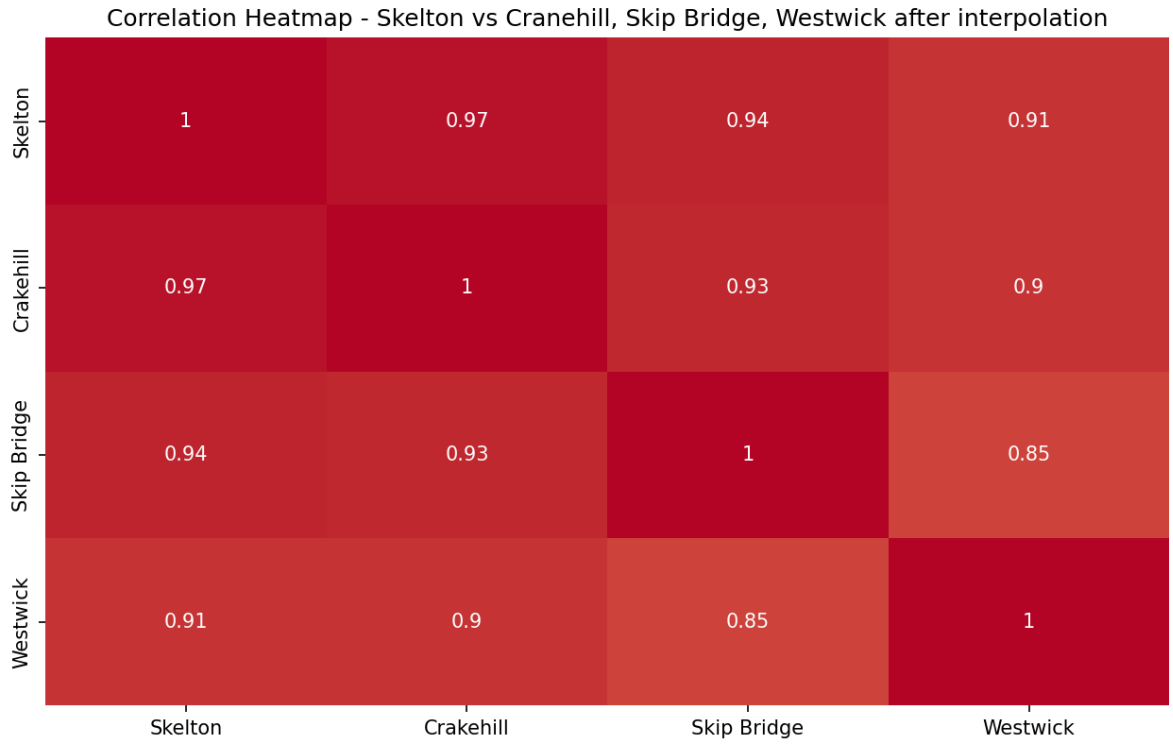
During the winter there will be significant more rainfall and a couple of its extreme points were set to nan but still over 90% of my data remained so my MLP would have enough data to train with. Those data points that have been set to nan should not change my results negatively, so I think its fine to exclude those points.

My code for data cleansing is below:

```
def remove_outliers(values, mean_value, standard_deviation):
    cleaned_values = values.copy()


    # Convert to numpy array if not already
    cleaned_values = np.array(cleaned_values, dtype=float)


    # Relace -999 or 'a' with NaN
    cleaned_values[(cleaned_values == -999) | (cleaned_values == "a")] = np.nan

    # Replace values beyond 4 standard deviations from the mean with NaN
    cleaned_values[(cleaned_values < (mean_value - 3 * standard_deviation)) |
                   (cleaned_values > (mean_value + 3 * standard_deviation))] = np.nan

    cleaned_values = pd.Series(cleaned_values)
    Q1 = cleaned_values.quantile(0.25)
    Q3 = cleaned_values.quantile(0.75)
    IQR = Q3 - Q1
    # Define the lower and upper bounds for outliers
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Filter out the outliers
    cleaned_values[(cleaned_values <= lower_bound) & (cleaned_values >= upper_bound)] = np.nan

    return cleaned_values
```

I did this for the remainder of the rivers and then I found the correlation between Skelton and the other rivers to ensure that there is still high correlation after cleaning which would mean I have not removed too many values. This is the image below.

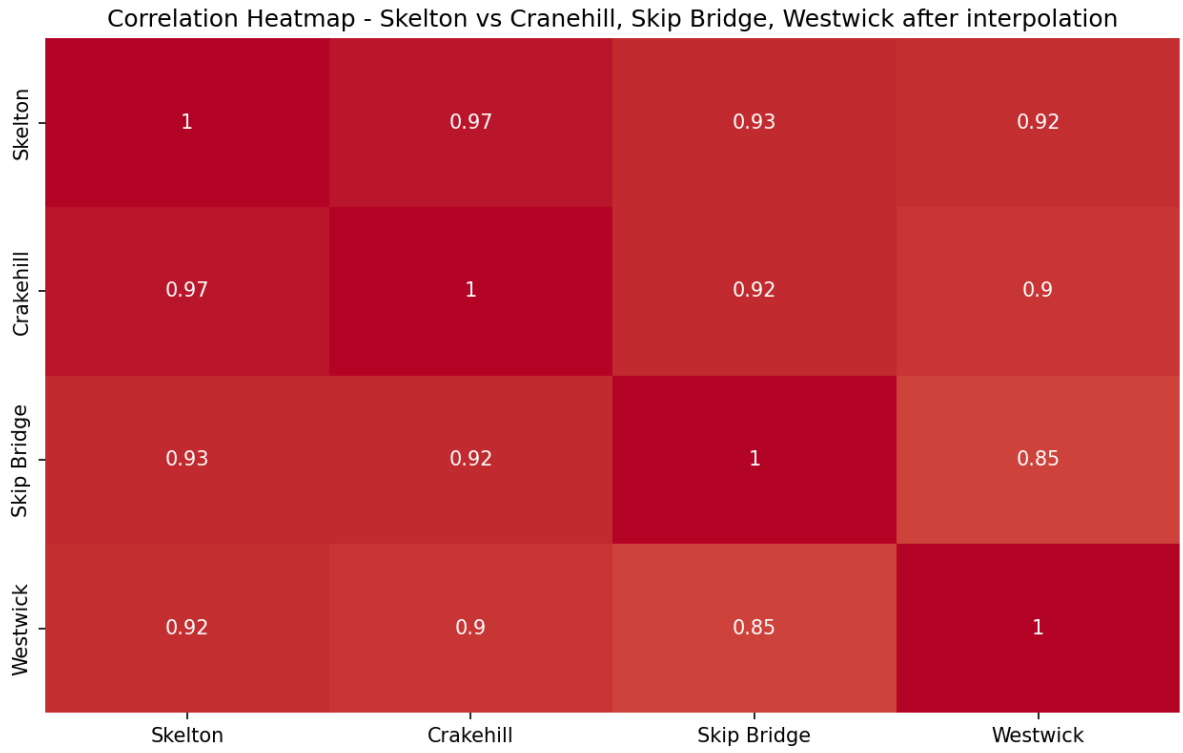
STEP 3 DATA PREPROCESSING (STAGE 1) - F325508



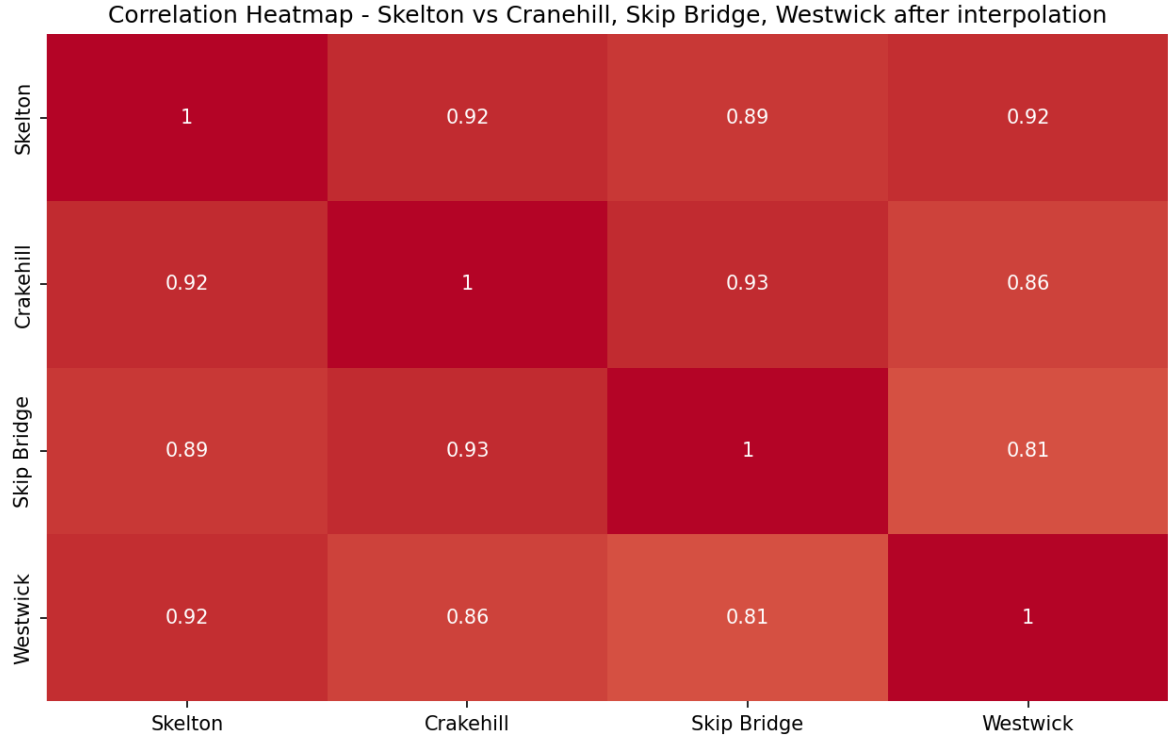
This image is with 4 Standard Deviations and interquartile range. As you can see there is very good correlation between Skelton and the other rivers are very high which is very good. However, unfortunately the correlation between the rivers (Crakehill, Skipbridge, and Westwick) to each other is also very high. This may mean if they were to be some of the predictors to the model, they may have a higher influence on the predictions, which would influence bias which is not what we want.

Below I also tested out the correlation between Skelton and the other rivers when we only use 3 Standard deviations not 4 which would mean less data but still above the 90% of the database remaining so it will be fine. As you can see there is slightly less correlation between the other rivers to themselves (Crakehill, Skipbridge, and Westwick) so I decided to use 3 Standard deviations as this would mean there is a better chance of less outlier data points.

STEP 3 DATA PREPROCESSING (STAGE 1) - F325508



In the image below I found out the correlation after using pchip interpolation to interpolate the nan values so my dataset would be full. This will also fill in any missing data that was in the database. I was going to use the built in interpolate function, but I wanted to be more accurate. Unfortunately, this reduced the correlation between Skelton and the other rivers but the correlation is still reasonably high so I feel this is a good data set. Also the correlation between the other rivers (Crakehill, Skipbridge, and Westwick) has decreased which means that it will be better for them being the predictors.



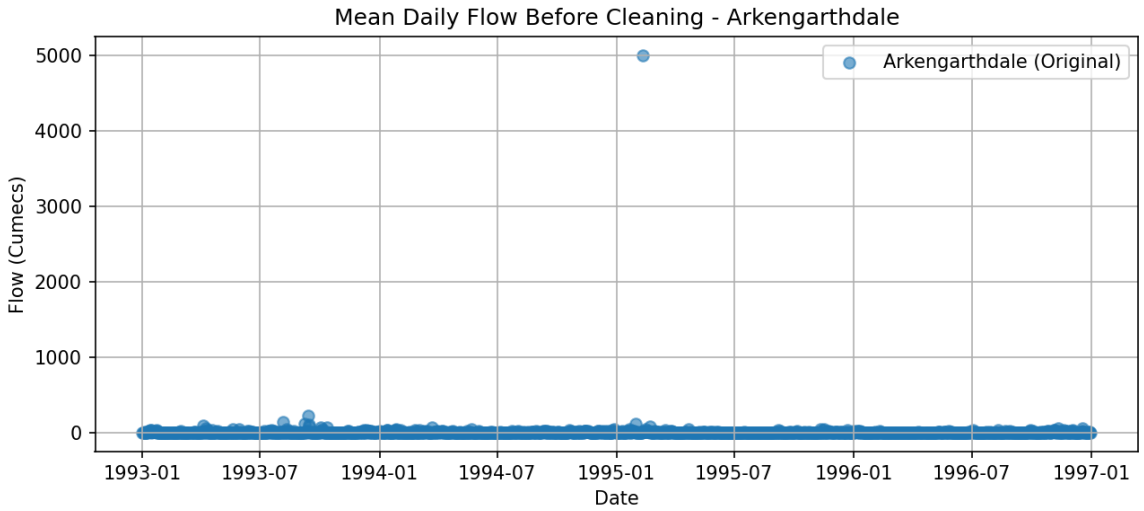
STEP 3 DATA PREPROCESSING (STAGE 1) - F325508

Below is a screenshot of some of a couple rows of my database which I saved to a file to ensure there is no missing data:

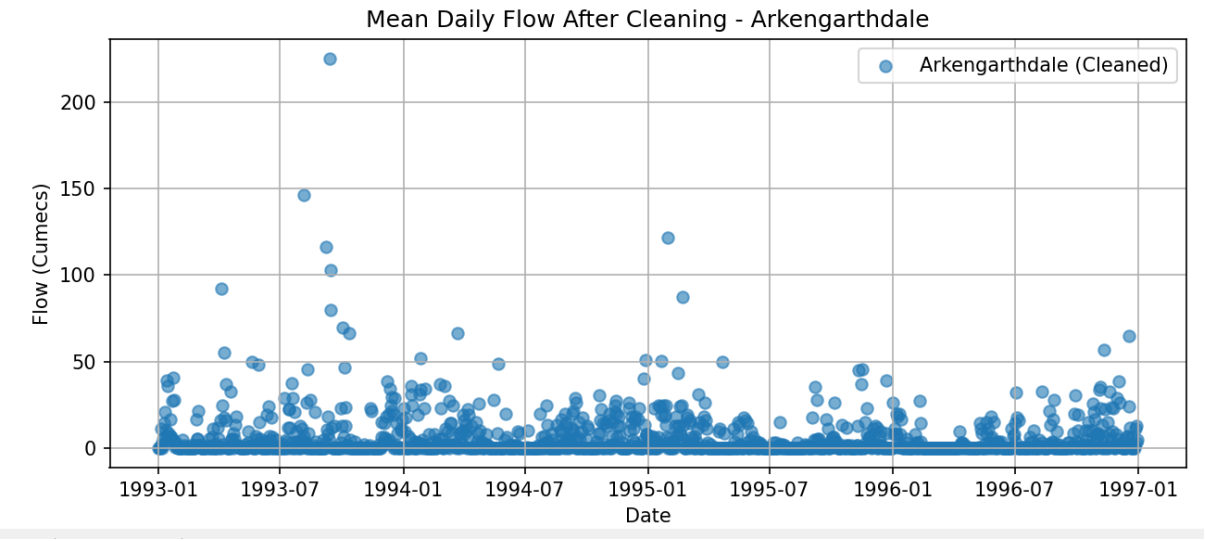
Date	Crakehill	Skip Bridge	Westwick	Skelton
1993-01-01	10.4	4.393	9.291	26.1
1993-01-02	10.175	4.316	8.956	24.86
1993-01-03	9.937	4.252	8.657	23.6
1993-01-04	9.805	4.28	8.474	23.47
1993-01-05	13.78	6.172	20.827	60.7
1993-01-06	19.318	7.672	27.276	98.01
1993-01-07	21.778	8.454	30.827	56.99
1993-01-08	24.925	9.18	36.242	56.66
1993-01-09	27.225	8.228	36.63	78.1
1993-01-10	31.95	8.987	51.513	125.7
1993-01-11	43.5	13.564	63.323	195.9
1993-01-12	46.0	16.926	66.055	125.4
1993-01-13	53.2	24.023	63.023	161.5
1993-01-14	57.725	27.818	54.616	204.0
1993-01-15	65.0	29.868	55.035	200.6
1993-01-16	81.75	34.311	64.266	186.846
1993-01-17	80.225	30.431	66.511	160.1
1993-01-18	71.475	27.114	62.104	104.1
1993-01-19	62.775	22.788	64.067	136.4
1993-01-20	50.025	17.814	62.254	137.1
1993-01-21	46.125	17.23	70.101	124.3
1993-01-22	54.25	20.058	78.018	175.9
1993-01-23	49.65	21.725	83.569	128.1
1993-01-24	68.65	31.546	87.666	167.3
1993-01-25	75.5	35.605	81.127	206.5
1993-01-26	67.7	33.609	73.371	116.4
1993-01-27	65.65	31.856	57.98	92.52
1993-01-28	43.725	21.902	51.758	104.3
1993-01-29	34.7	17.744	47.094	106.0
1993-01-30	32.3	16.914	44.178	80.07
1993-01-31	29.525	15.841	40.855	64.14
1993-02-01	25.5	13.982	30.903	56.13

All interpolated data is rounded to 3 decimal places to ensure some level of accuracy.

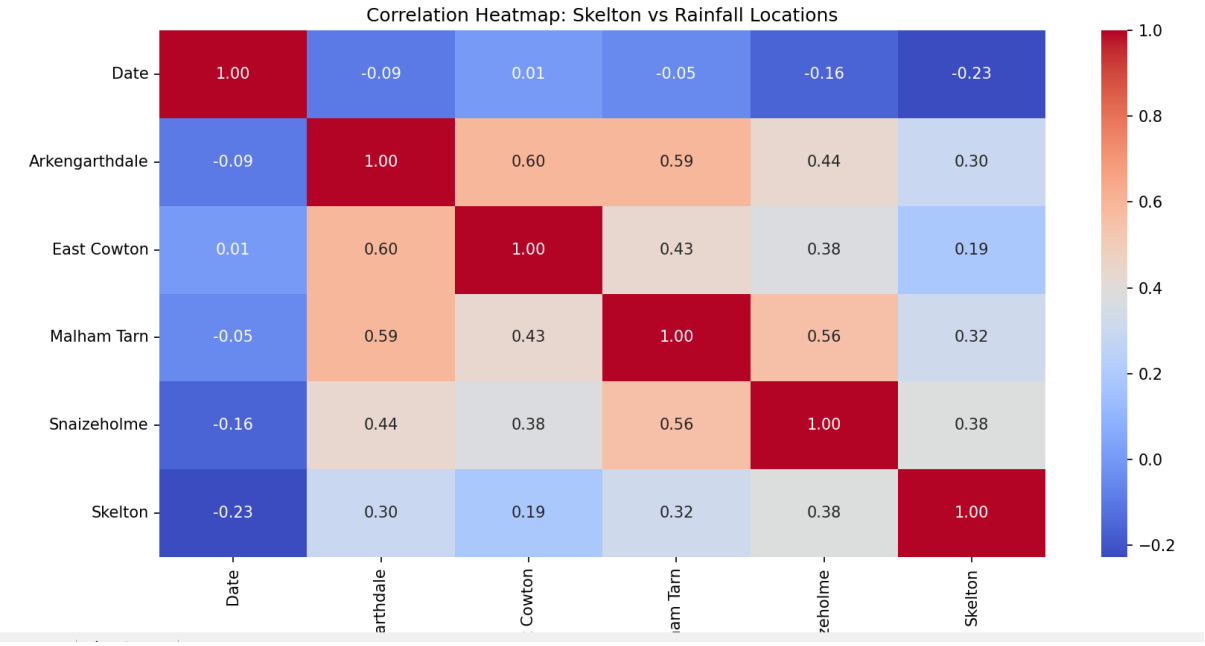
I then did the same method for the mean daily rainfalls. As you can see below Arkengarthdale has an extreme data point of 5000 which is cleaned in the next graph below.



STEP 3 DATA PREPROCESSING (STAGE 1) - F325508



I was thinking of removing the values over 100 but then I decided not too as there is not that many so they will not have much of an impact as 6 days out of 2 years will not affect much. Also there could have been a storm around then which causes the rain to have a massive increase.



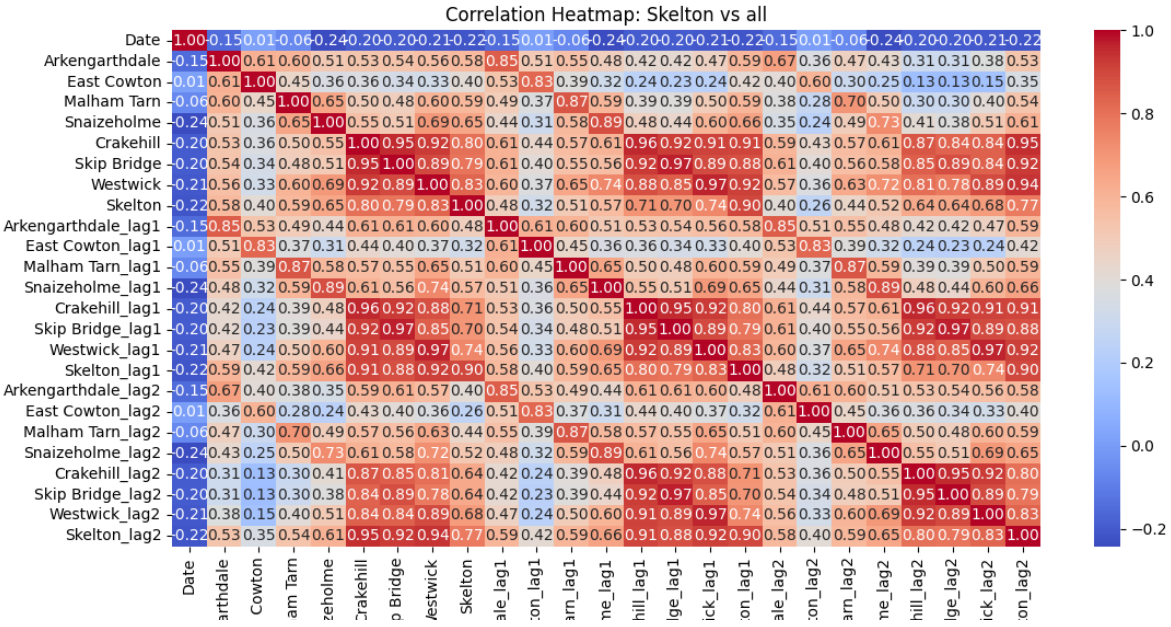
Above is the correlation between Skelton and the other rainfall river sources. As you can see unfortunately the correlation is quite low.

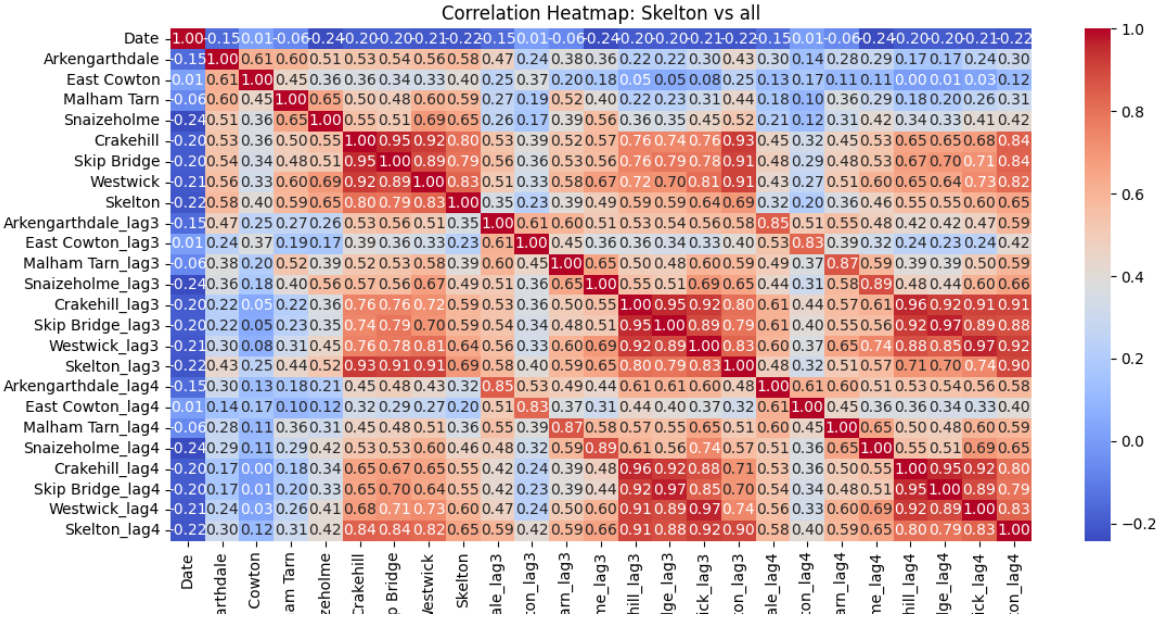
STEP 3 DATA PREPROCESSING (STAGE 1) - F325508

STEP 3.2 PREDICTORS

So, I decided not to use the date as a predictor due to its correlation being too low with so that was the first predictor I excluded.

I started off by lagging all the predictors by up to 7 days and producing different graphs with graphs of lagging 2 days at a time. Here below you can see Skelton Tomorrow as Skelton and me lagging the predictors for 1 and 2 days, then 3 and 4 days, as 5+ days the correlation is too low in my opinion to be used. These graphs are using moving averages of 4 days.

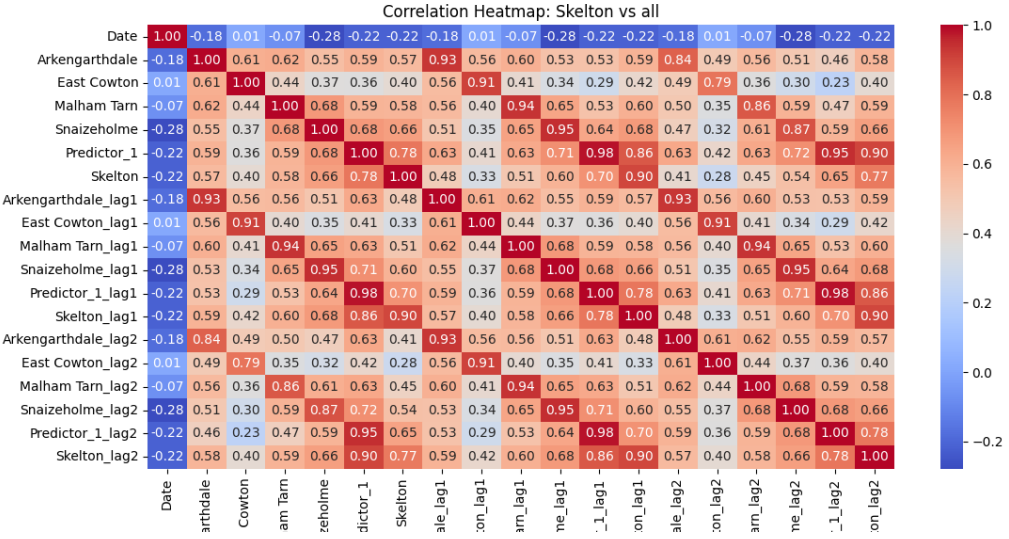




STEP 3 DATA PREPROCESSING (STAGE 1) - F325508

As you can see the more you lag the predictors the worse their correlation with Skelton is. Also, the one that is the highest correlation to Skelton Tomorrow is actually Skelton (Skelton lagged by 1 on my graph), so I will definitely be using that for one of my predictors. I will also be using Skelton lagged by 1 day (Skelton lagged by 2 on my graph) as one of my predictors that correlation with Skelton is still high.

I then combined the rivers together to make predictor 1. Then I applied a moving day average of 7 to see if it was any better as you can see in the graph below.



Unfortunately, a moving day average of 7 actually decreased the correlation between predictor 1 and Skelton tomorrow so I decided against using a 7-day average and just used the 4 day average instead.

The code I used below is to lag all my predictors by up to 7 days by after the first couple days I realised there is no need to use the data as its correlation was too low:

```
# Lagging Rainfall Data by 1 to 7 Days
df_lagged_data = df_cleaned.copy()
'''
for lag in range(1, 7):
    for col in ["Arkengarthdale", "East Cowton", "Malham Tarn", "Snaizeholme", "Predictor_1", "Skelton"]:
        df_lagged_data[f"{col}_lag{lag}"] = df_cleaned[col].shift(lag)

df_rainfall = df_rainfall.round(3)
df_lagged_data = df_lagged_data.round(3)
df_lagged_data.to_csv('Lagged_Rainfall_Data.txt', sep='\t', index=False)
'''
```

I also tried a moving day average of almost a month (28 days) which decreased correlation, so then I stuck with a moving average of 4 days. I then changed my predictor 1 to be the mean of all the rivers flow and called it Average River flow as then the outliers of certain rivers would have less of an effect, allowing my predictor to be sufficient.

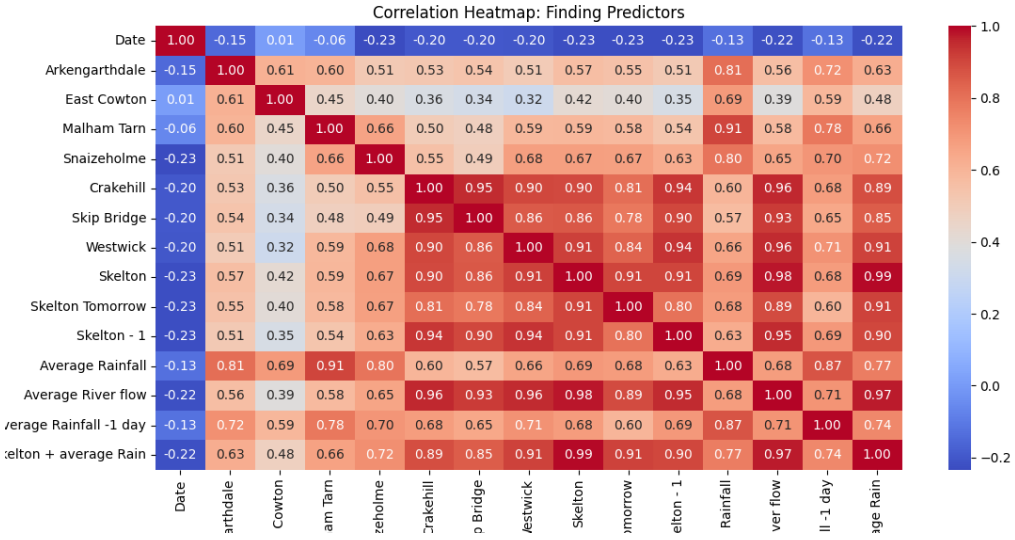
I did the same for the rainfall and found that its correlation was quite good too.

MY FINAL PREDICTORS

Below is what I decided will be my predictors, and Skelton Tomorrow is my predictand. With all the predictors below I used a moving average of 4 days.

STEP 3 DATA PREPROCESSING (STAGE 1) - F325508

I am going to use Skelton as one of my predictors as it has a very good correlation with Skelton Tomorrow (0.91) as should hopefully produce good results. I will also use Skelton -1, which is Skelton lagged by 1 day and this still also has a really good correlation (0.8). I decided on using Average rainfall which take a mean of all the rainfall sources (Arkengarthdale, East Cowton, Malham Tarn, and Snaizeholme) and has a decent correlation with Skelton Tomorrow of 0.68. I then did the same but with the river flow ('Crakehill', 'Skip Bridge', 'Westwick', 'Skelton') and had a really good correlation of 0.89 to Skelton Tomorrow. As this has such a good correlation I decided to use Average River flow again but this time lagged by 1 day and its correlation was still quite good (0.60). Then finally I used Skelton + the average rain as my final predictor.



As you can see some of the predictors have very high correlations with each other but then I realised with my data set being so large that hopefully will not affect my final MLP values that much or skew my results that badly. The correlation between Skelton Tomorrow and the rest of my predictors are quite high apart from Average Rainfall – 1 but even then, 0.60 will not terribly affect my results as Skelton + Average rain is 0.91 so hopefully will influence my results better.

This is a screenshot of the first couple lines of my final predictors I wrote to a text file. I needed to put all my new columns that I made to 3 decimal places, but you can see all the different culumns with no data missing.

Date	Arkengarthdale	East Cowton	Malham Tarn	Snaizeholme	Crakehill	Skip Bridge	Westwick	Skelton	Skelton Tomorrow
	Skelton - 1	Average Rainfall	Average	River flow	Average Rainfall -1 day	Skelton	+ average Rain		
1993-01-01	0.0	0.0	0.0	4.0	10.4	4.393	9.291	26.1	24.86	1.0	12.546	27.1	
1993-01-02	0.0	0.0	0.4	2.0	10.175	4.316	8.956	24.86	23.6	26.1	0.6	12.07675	1.0	25.46
1993-01-03	0.0	0.0	0.533	1.333	9.937	4.252	8.657	23.6	23.47	24.86	0.4665	11.6115	0.6	24.0665
1993-01-04	0.6	6.2	0.6	16.4	9.805	4.28	8.474	23.47	60.7	23.6	5.949999999999999	11.507249999999999	0.4665	29.419999999999998
1993-01-05	3.4	7.6	9.0	23.2	13.78	6.172	20.827	60.7	98.01	23.47	10.8	25.36975	5.949999999999999	71.5
1993-01-06	3.4	7.6	9.2	23.4	19.318	7.672	27.276	98.01	56.99	60.7	10.899999999999999	38.069	10.8	108.91
1993-01-07	4.8	8.6	13.4	32.4	21.778	8.454	30.827	56.99	56.66	98.01	14.799999999999999	29.51225	10.899999999999999	71.79
1993-01-08	4.6	2.4	13.6	17.6	24.925	9.18	36.242	56.66	78.1	56.99	9.55	31.75175	14.799999999999999	66.21
1993-01-09	5.4	1.2	19.0	12.059	27.225	8.228	36.63	78.1	125.7	56.66	9.41475	37.54575	9.55	87.51474999999999
1993-01-10	10.6	1.8	37.6	17.2	31.95	8.987	51.513	125.7	195.9	78.1	16.8	54.537499999999994	9.41475	142.5
1993-01-11	11.8	4.8	36.2	15.2	43.5	13.564	63.323	195.9	125.4	125.7	17.0	79.07175000000001	16.8	212.9
1993-01-12	13.2	5.8	36.0	20.6	46.0	16.926	66.055	125.4	161.5	195.9	18.9	63.59525	17.0	144.3
1993-01-13	19.4	7.8	42.9	22.654	53.2	24.023	63.023	161.5	204.0	125.4	23.188499999999998	75.4365	18.9	184.6885
1993-01-14	15.6	7.6	25.9	20.514	57.725	27.818	54.616	204.0	200.6	161.5	17.403499999999998	86.03975	23.188499999999998	221.4035
1993-01-15	22.0	6.6	40.9	31.514	65.0	29.868	55.035	200.6	186.846	204.0	25.2535	87.62575	17.403499999999998	225.8535
1993-01-16	22.4	6.2	42.5	35.514	81.75	34.311	64.266	186.846	160.1	200.6	26.6535	91.79325	25.2535	213.4995
1993-01-17	14.2	4.2	21.8	36.6	80.225	30.431	66.511	160.1	104.1	186.846	19.200000000000003	84.31674999999998	26.6535	179.3
1993-01-18	17.0	4.4	30.8	39.111	71.475	27.114	62.104	104.1	136.4	160.1	22.82775	66.19825	19.200000000000003	126.92775
1993-01-19	9.8	1.4	21.4	27.4	62.775	22.788	64.067	136.4	137.1	104.1	15.0	71.5075	22.82775	151.4
1993-01-20	9.2	0.8	24.4	24.0	50.025	17.814	62.254	137.1	124.3	136.4	14.6	66.79825	15.0	151.7
1993-01-21	14.4	1.6	37.2	24.2	46.125	17.23	70.101	124.3	175.9	137.1	19.35	64.43900000000001	14.6	143.65
1993-01-22	11.4	1.6	27.8	23.089	54.25	20.058	78.018	175.9	128.1	124.3	15.972249999999999	82.0565	19.35	191.87225
1993-01-23	19.8	6.2	34.8	22.8	49.65	21.725	83.569	128.1	167.3	175.9	20.9	70.761	15.972249999999999	149.0
1993-01-24	25.2	10.4	31.8	23.6	68.65	31.546	87.666	167.3	206.5	128.1	22.75	88.79050000000001	20.9	190.05
1993-01-25	19.0	9.4	19.0	18.6	75.5	35.605	81.127	206.5	116.4	167.3	16.5	99.68299999999999	22.75	223.0
1993-01-26	18.0	8.8	17.4	16.4	67.7	33.609	73.371	116.4	92.52	206.5	15.15	72.77000000000001	16.5	131.55

STEP 4 ANN SELECTION

I chose the back propagation algorithm.

STEP 5 DATA PREPROCESSING (STAGE 2)

STEP 5.1 DATA STANDARDISATION

In Step 6 where I implemented the algorithm you will be able to see after testing out several activation functions I came to using the Sigmoid function. During this I then used Min-Max standardisation which worked quite well as it is more suited to the sigmoid function.

The code is below:

```
# Min-Max Standardization
x = 0.8*(x - x_min) / (x_max - x_min)+0.1
y = 0.8*(y - y_min) / (y_max - y_min)+0.1
```

When I was using other activation functions for example the tanh activation function, I used the mean and standard deviation to standardise my data.

Code below:

```
# Normalize features (Standardization: mean = 0, std = 1)
#x = (x - x.mean()) / X.std()
#y = (y - y.mean()) / y.std()
```

STEP 5.2 DATA SPLIT

I split the data based on the years. The test data should always be large in comparison to the test and validation data size. This is why I decided to use the first 2 years of data as my training data. Although year 3 had more outliers compared to year 1 and 2, I still think I cleaned my data well, so I decided to use year 3 as my testing data. This left me with the final year as my validation data.

So, my split is therefore 50%-20%-20% which is very reasonable and should correlate quite well.

The code for splitting my data is below:

STEP 5 DATA PREPROCESSING (STAGE 2) - F325508

```
# Split data into training (50%), testing (25%), validation (25%)

train_size = int(0.5 * len(x))    # 50% Training
test_size = int(0.25 * len(x))    # 25% Test
val_size = len(x) - (train_size + test_size)  # 25% Validation

x_train, x_test, x_val = np.split(x, [train_size, train_size + test_size])
y_train, y_test, y_val = np.split(y, [train_size, train_size + test_size])
```

STEP 6 NEWTORK TRAINING

IMPLEMENTATION OF MY MLP:

As I said previously, I implemented my MLP in Object Orientated form and have the different methods inside the MLP class. My first method is the initialise method as shown below:

```
def __init__(self,x,y,input_size,output_size, hidden_size, learning_rate, epochs):
    self.x = x
    self.y = y
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.output_size = output_size
    # Weight initialization with uniform distribution
    self.weights_input_hidden = np.random.uniform(-2/input_size, 2/input_size, (input_size, hidden_size))
    self.weights_hidden_output = np.random.uniform(-2/hidden_size, 2/hidden_size, (hidden_size, output_size))

    # Bias initialization to go between 1 and 0 for sigmoid activation
    self.bias_hidden = np.random.uniform(-1, 1, (1, hidden_size))
    self.bias_output = np.random.uniform(-1, 1, (1, output_size))

    self.learning_rate = learning_rate
    self.epochs = epochs
    self.loss_history = []
```

In this I take the x which is my predictor values and my y which is my predictand values. This also takes in the input size which is the number of predictors I currently have which is 6. This also takes in the output size which is the number of predictands. This for my model is 1, which is Skelton Tomorrow. The hidden size will be the number of hidden nodes in the hidden layer. I only got my model working with 1 hidden layer which I think it still definitely enough as my model has good results. The learning rate is the learning rate I provide, and the epochs is the number of epochs I put my model to train through.

This also uses random.uniform (-2/number_input_node, 2/number_input_nodes), to create the weight matrix for the input to the hidden layer and the other matrix for the hidden layer to the output layer. Using random values allows it to change and potentially create better models when I run my code every time.

I created my bias to be in the range from -1 to 1 to help with sigmoid activation but then when I used tanh activation function, I used Xavier initialisation.

STEP 6 NEWTORK TRAINING - F325508

```
'''----------------------ACTIVATION FUNCTIONS----------------------'''
#Sigmoid function
def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))

#Sigmoid derivative function
def sigmoid_derivative(self, node_sum):
    activation = self.sigmoid(node_sum)
    return activation * (1 - activation)

#Tanh function
def tanh(self, x):
    return np.tanh(x)

#Tanh derivative function
def tanh_derivative(self, x):
    return 1 - np.tanh(x) ** 2

#Relu function
def relu(self, x):
    return np.maximum(0, x)

#Relu derivative function
def relu_derivative(self, x):
    return np.where(x <= 0, 0,
                    np.where(x > 0, 1, 0))
```

This is how I defined my activation functions which applies the function to the node sums.

STEP 6 NEWTORK TRAINING - F325508

```
'''----------------------Error FUNCTIONS----------------------'''
#Mean Absolute error
def absolute_error(self, y, y_pred):
    return np.mean(np.abs(y - y_pred))

#Mean Absolute error derivative
def absolute_error_derivative(self, y, output):
    return np.where(output > y, 1, -1)


#Mean squared error
def mean_squared_error(self, y, y_pred):
    return np.mean((y - y_pred) ** 2)

#Mean squared error derivative
def mean_squared_error_derivative(self, y, y_pred):
    return 2 * (y - y_pred) / len(y)


#Root Mean squared error
def root_mean_squared_error(self, y, y_pred):
    return np.sqrt(np.mean((y - y_pred) ** 2))

#Root Mean squared error derivative
def root_mean_squared_error_derivative(self, y, y_pred):
    return 2 * (y - y_pred) / len(y)
```

This is how I defined my error functions to find the error between the actual y (predictand) values and my predicted y values.

```
def forward(self, x):
    #Input to hidden layer

    hidden_layer_sum = np.dot(x, self.weights_input_hidden) + self.bias_hidden
    hidden_layer_activation = self.sigmoid(hidden_layer_sum)

    #Hidden to output layer
    output_layer_sum = np.dot(hidden_layer_activation, self.weights_hidden_output) + self.bias_output
    output_layer_activation = self.sigmoid(output_layer_sum)

    return output_layer_activation, output_layer_sum, hidden_layer_activation, hidden_layer_sum
```

This is my forward pass. This takes in my predictors, and then does the dot product of x (predictors) and the weight matrix for the input to hidden layer and then adds the hidden bias to find the hidden layer sum. I then apply the sigmoid function to the hidden layer sum to find the hidden layer activation.

I also do the dot product of the hidden layer activation with the weight matrix for the hidden to output layer and then add the output bias which equates to the output later sum. Then I apply the sigmoid function to the output layer sum which equates to the output layer activation.

This then returns the output later activation, output layer sum, hidden layer activation and hidden layer sum for use later in the program.

```
def backward(self, y, output_layer_activation, output_layer_sum, hidden_layer_sum):
# Output node calculation
    output_error = -self.absolute_error_derivative(y, output_layer_activation)
    output_delta = output_error * self.sigmoid_derivative(output_layer_sum)


    # Hidden node calculation
    hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
    hidden_delta = hidden_error * self.sigmoid_derivative(hidden_layer_sum)
    return output_delta, hidden_delta
```

This is my backward propagation. This takes in the actual predictand value, the output layer activation, the output layer sum, the hidden layer sum.

To calculate the output error, I negate the absolute mean error derivative of y and the output layer activation. This means that the error will always be negative for consistency. To calculate the output delta, I multiplied the output error by the sigmoid derivative applied to the output layer sum.

Then to calculate the hidden error I did the dot product of the output delta and the matrix for the hidden to output layer transposed. This means the dimensions will be correct for multiplication. To find the hidden delta I multiplied the hidden error by the sigmoid derivative function applied to the hidden layer same.

This function then returns the output delta and the hidden delta for later use.

```
'''----------------------UPDATE WEIGHTS AND BIASES----------------------'''
def update_weights(self, x, output_delta, hidden_delta,hidden_layer_activation):
    #update weights and biases

    x = x.reshape(1, -1)

    self.weights_hidden_output += np.dot(hidden_layer_activation.T,output_delta) * self.learning_rate
    self.weights_input_hidden += np.dot(x.T, hidden_delta) * self.learning_rate

    self.bias_output += output_delta * self.learning_rate
    self.bias_hidden += hidden_delta * self.learning_rate
```

This is my function to update my weights and biases, which takes in the predictors (x), the output delta, the hidden delta and the hidden layer activation.

I use the reshape function on my predictors to force them to have the correction dimensions for matrix calculation.

To update the weight matrix for hidden to output layer we need to add itself to the dot product of the hidden layer activation transposed (for correct dimensions for matrix calculation) and the output delta) multiplied by the learning rate.

Then to update the weight matrix for the input layer to the hidden layer we need to add itself to the dot product of the predictors transposed by the learning rate. This means we have successfully calculated the weight matrices.

STEP 6 NEWTORK TRAINING - F325508

Then we need to update the bias output and bias input. To do this, for the bias output we add itself to the output delta multiplied by the learning rate and to find the bias for the hidden layers we do the same principle except the hidden delta instead of the output delta. This should be clear from my coding hopefully.

```
'''----------------------TRAINING FUNCTION----------------------'''
def train(self, x, y):
    self.loss_history = []  # Store loss for analysis

    for epoch in range(self.epochs):  # Loop through epochs
        total_loss = 0  # Track total loss for this epoch

        for i in range(len(x)):  # Iterate over each training sample
            predictors = x[i].reshape(1, -1)  # Ensure correct shape (row vector)
            target = y[i].reshape(1, -1)  # Ensure correct shape (row vector)


            # Forward pass
            output, output_sum, hidden_activation, hidden_sum = self.forward(predictors)

            # Compute loss and accumulate it
            loss = self.absolute_error(target, output).sum()

            total_loss += loss

            # Backward pass (compute gradients)
            output_delta, hidden_delta = self.backward(target, output,output_sum, hidden_sum)

            # Update weights and biases
            self.update_weights(predictors, output_delta, hidden_delta, hidden_activation)
```

```
    # Store and display average loss for the epoch
    avg_loss = total_loss / len(x)

    # Denormalize average loss
    denormalised_loss = denormalise(avg_loss, y_min, y_max)
    print(f"Epoch: {epoch+1}, Denormalised Loss: {denormalised_loss}")

    # Append denormalized loss to history
    self.loss_history.append(denormalised_loss)

return self.loss_history  # Return loss history for analysis
```

My training function takes in the predictors and the predictand. The function initialises a list for loss history. Then for every epoch it will go through every data point in my predictors and reshape my predictors and predictands for matrix calculations. The function the calls the forward pass for my predictors and then will calculate loss. After this the function calls the backward propagation and then

updates the matrices weights and biases. Then finds the average loss and denormalises it so then they can plot the loss vs epoch graph later.

```
# Split the data into predictors and target variable
x = df_cleaned.drop(columns=["Skelton Tomorrow"])
y = df_cleaned["Skelton Tomorrow"]
```

This is how my code splits my data into predictors and predictands after dropping the irrelevant columns below, with Westwick too:

```
# Drop the Date column and unnecessary columns
drop_cols = ["Date", "Arkengarthdale", "East Cowton", "Malham Tarn", "Snaizeholme", "Crakehill", "Skip Bridge",
df_cleaned = df_cleaned.drop(columns=[col for col in drop_cols if col in df_cleaned.columns])
```

I have also used the code below to get my weights and loss history for plotting the error loss graph:

```
def get_weights(self):
    return self.weights_input_hidden, self.weights_hidden_output

def get_loss_history(self):
    return self.loss_history
```

After I implemented my MLP I then did the following to find the most effective number of epochs, learning rate, activation function and number of hidden nodes:

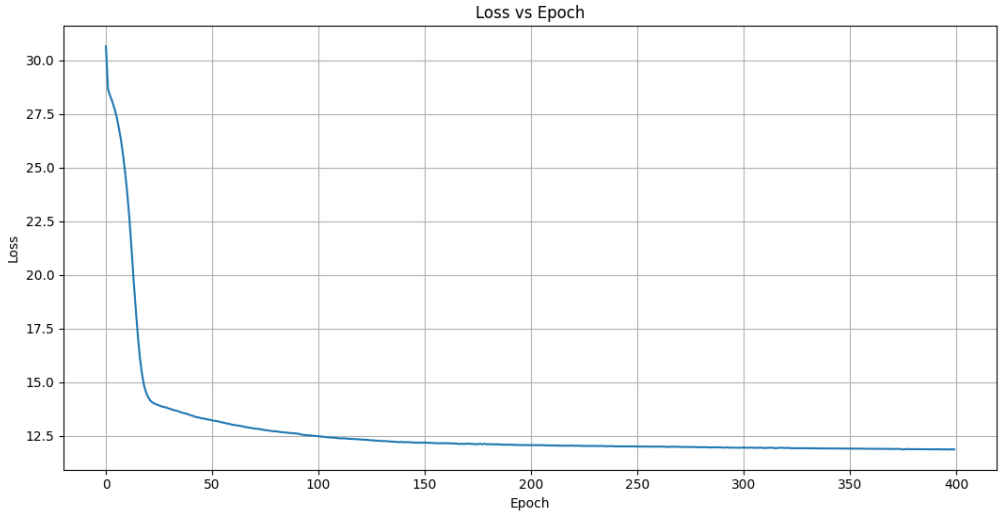
I first tried the sigmoid function from 200 epochs to 1000 epochs with a learning rate of 0.1, then I did the same with a learning rate of 0.01. I found my best results were 0.01 and epochs of 400. Then I went through every combination of error functions (mean absolute error, mean square error and root mean square error) with every activation function (Sigmoid, Relu and Tanh) with 3, 6 and 12 hidden nodes each time. When I used the tanh transfer function, I standardized my data using the mean and standard deviation instead of min max standardisation to output better results.

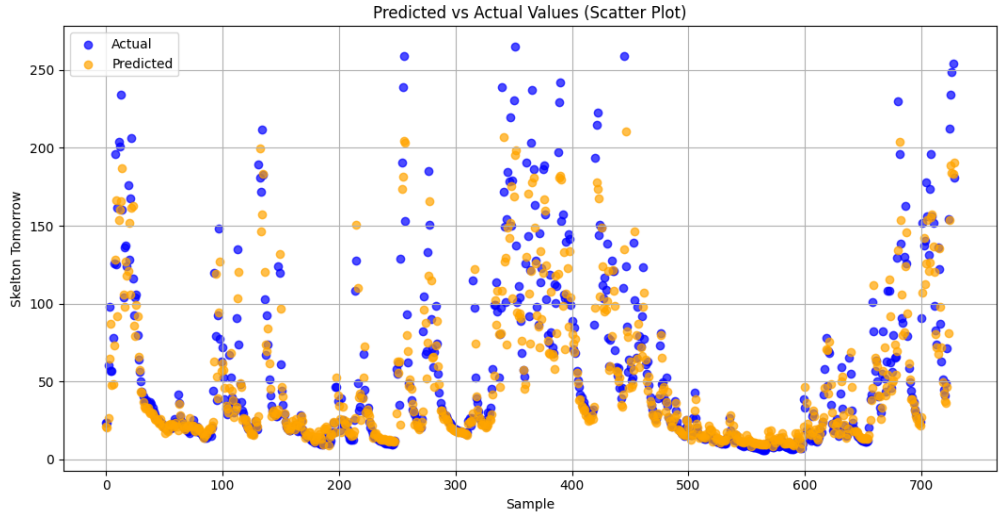
When I initiated the bias, I changed it to (-1,1) to help output the sigmoid function better which I found improved my results.

For my MLP algorithm I found that a learning rate of 0.01 and 400 epochs with 6 hidden nodes for the sigmoid function and absolute mean differentiation worked the best.

My results are below.

STEP 6 NEWTORK TRAINING - F325508

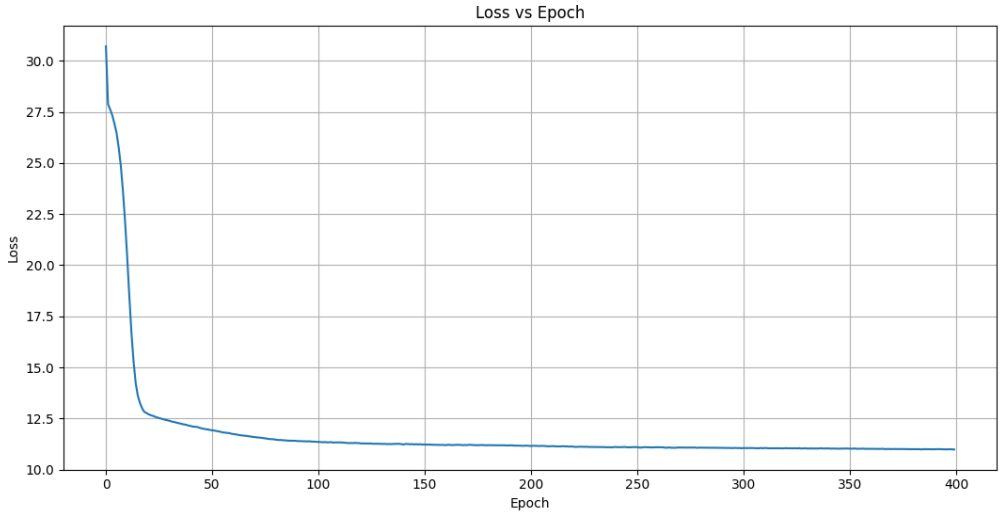


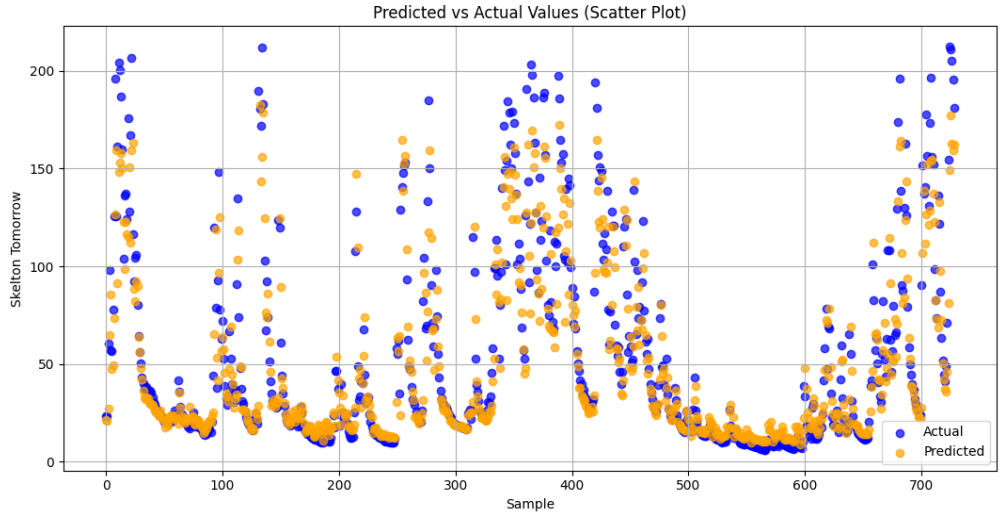


Above, I then standardised my error loss before plotting the error loss graph to figure out my average error was about 11.85 which makes sense due to at the very start my error loss is quite high, maybe due to my cleaning, but then near the middle and the end the error loss become very low and so very accurate.

Then after this I tried 3 standard deviations in the cleaning and my results are below to see if there was much difference. It is on average slightly better nearer the end and roughly the same nearer the start, so I decided to go through with 3 standard deviations as my results were more accurate. The average went down to roughly 11 which is definitely better than 4 standard deviations due to less anomalies in the predictors. With 3 standard deviations there is still a lot of data to play with (over 90%) so my results should and will still be fair based on the number of points in the data set. My graph is below.

STEP 6 NEWTORK TRAINING - F325508



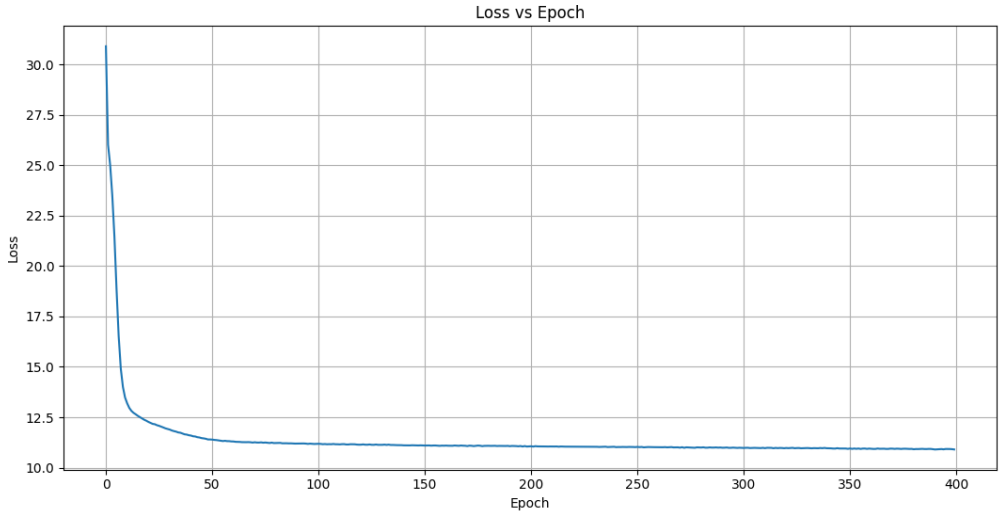


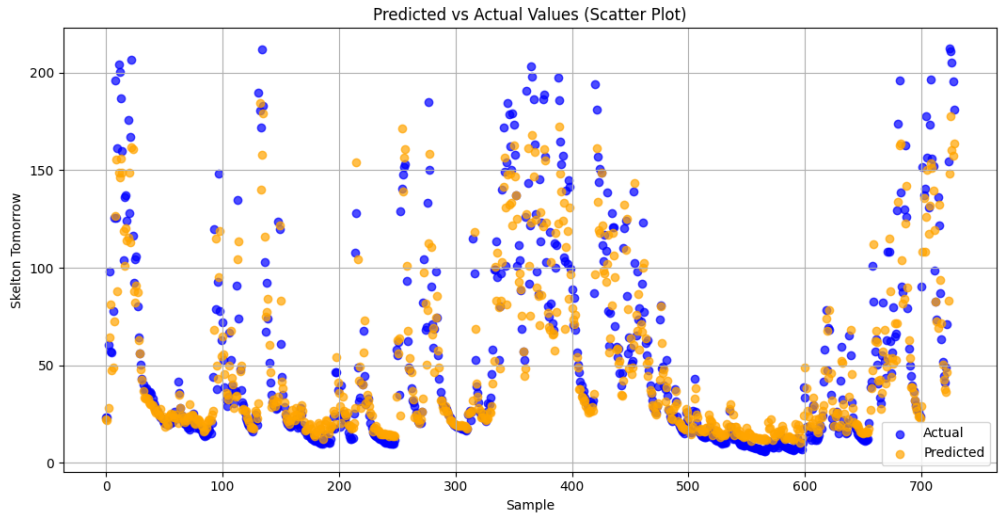
STEP 6 NEWTORK TRAINING - F325508

TRAINING AND IMPROVEMENTS

MOMENTUM

After implementing momentum my mean absolute error went from about 11 to 10.9 as you can see below, but I don't believe there is much difference.





As you can see this helps with predicting the earlier values but meant the middle values became worse. I made this code using a new train function which contained a new weight update function which you can see below.

STEP 6 NEWTORK TRAINING - F325508

```
'''-----------------UPDATE WEIGHTS AND BIASES WITH MOMENTUM----------------------'''
def update_weights_with_momentum(self, x, output_delta, hidden_delta, hidden_layer_activation):
    x = x.reshape(1, -1)

    # Initialise previous weight changes if they don't exist
    if not hasattr(self, 'prev_weight_change_hidden_output'):
        self.prev_weight_change_hidden_output = np.zeros_like(self.weights_hidden_output)
        self.prev_weight_change_input_hidden = np.zeros_like(self.weights_input_hidden)
        self.prev_bias_change_output = np.zeros_like(self.bias_output)
        self.prev_bias_change_hidden = np.zeros_like(self.bias_hidden)

    # Momentum parameter
    momentum = 0.9  # Feel free to adjust this value
```

```
# Momentum parameter
momentum = 0.9  # Feel free to adjust this value

# Compute the current weight changes (gradients scaled by the learning rate)
weight_change_hidden_output = np.dot(hidden_layer_activation.T, output_delta) * self.learning_rate
weight_change_input_hidden = np.dot(x.T, hidden_delta) * self.learning_rate
bias_change_output = output_delta * self.learning_rate
bias_change_hidden = hidden_delta * self.learning_rate

# Apply momentum: combine the current weight changes with 0.9 * previous changes
self.weights_hidden_output += weight_change_hidden_output + momentum * self.prev_weight_change_hidden_output
self.weights_input_hidden += weight_change_input_hidden + momentum * self.prev_weight_change_input_hidden
self.bias_output += bias_change_output + momentum * self.prev_bias_change_output
self.bias_hidden += bias_change_hidden + momentum * self.prev_bias_change_hidden
```

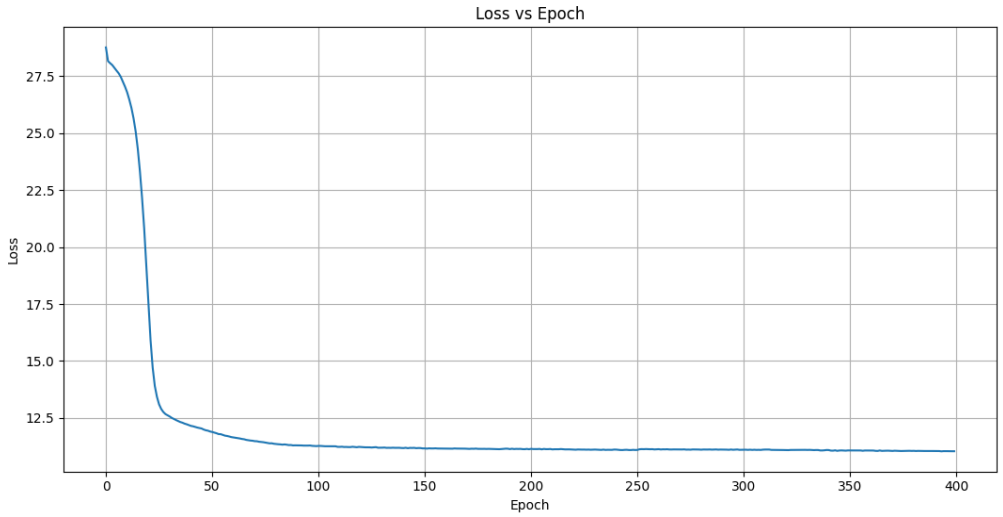
```
 # Save the current weight changes as the previous changes for the next iteration
 self.prev_weight_change_hidden_output = weight_change_hidden_output
 self.prev_weight_change_input_hidden = weight_change_input_hidden
 self.prev_bias_change_output = bias_change_output
 self.prev_bias_change_hidden = bias_change_hidden
```

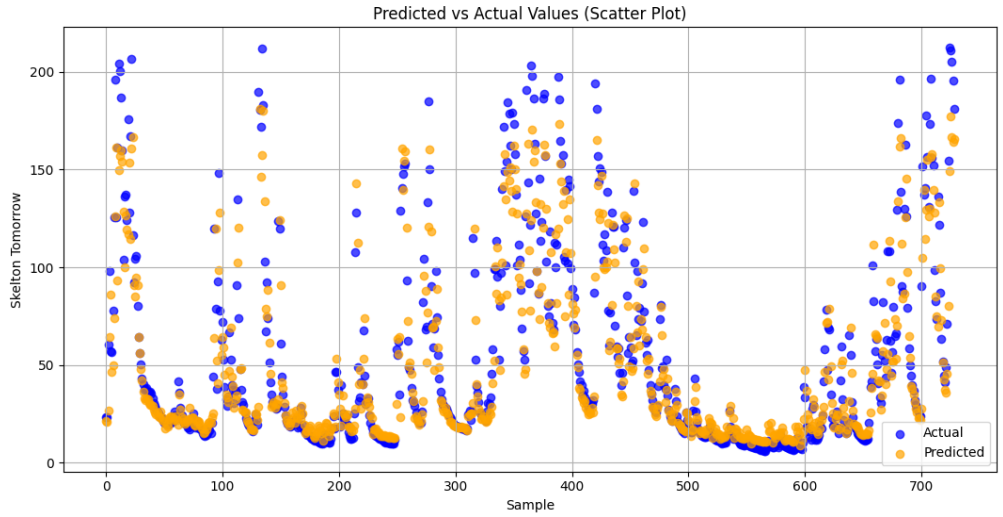
In this code I still reshape the predictors. Then I use hasattr to initialise previous weight changes as for the first epoch they will not exist. Then I set the momentum value to 0.9 and computer the current weight changes like normal. Then I apply the momentum formula to the weight matrices and the biases. Finally, I save the weight changes are the previous changes for the next iteration.

STEP 6 NEWTORK TRAINING - F325508

BOLD DRIVER

After implementing bold driver by itself for every epoch you can see it gets caught on a single point every time but is still nevertheless accurate and around 11 again.





As you can see from above the graph is fairly similar.

STEP 6 NEWTORK TRAINING - F325508

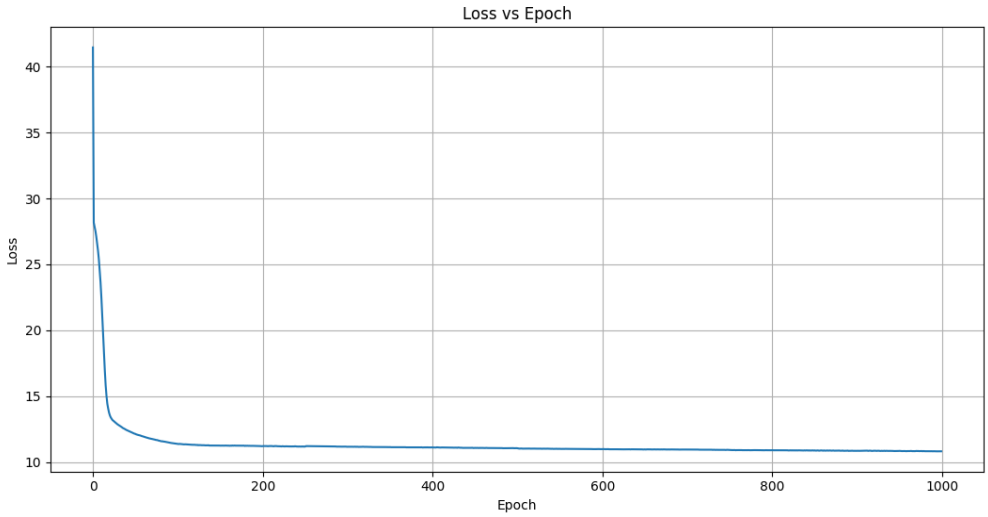
```
'''----------------TRAINING FUNCTION WITH BOLD DRIVER----------------'''
def train_with_bold_driver(self, x, y):
    self.loss_history = []  # Store loss for analysis
    max_learning_rate = 0.5
    min_learning_rate = 0.01

    learning_rate = self.learning_rate

    # Initialize bold driver parameters
    increase_factor = 1.1
    decrease_factor = 0.5
    prev_loss = float('inf')

    for epoch in range(self.epochs):  # Loop through epochs
        total_loss = 0  # Track total loss for this epoch

        for i in range(len(x)):  # Iterate over each training sample
            predictors = x[i].reshape(1, -1)  # Ensure correct shape (row vector)
            target = y[i].reshape(1, -1)  # Ensure correct shape (row vector)
```

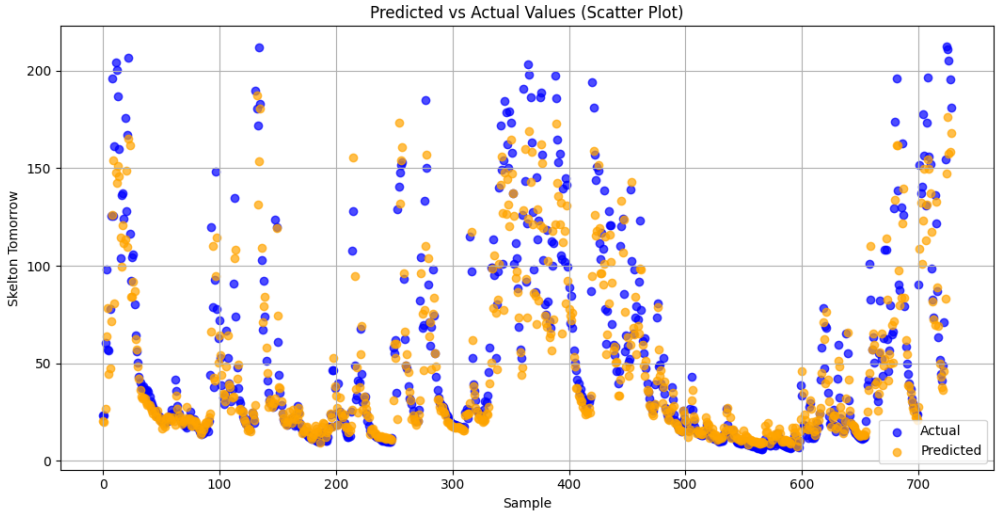
```
            # Forward pass
            output, output_sum, hidden_activation, hidden_sum = self.forward(predictors)

            # Compute loss and accumulate it
            loss = self.absolute_error(target, output).sum()

            total_loss += loss

            # Backward pass (compute gradients)
            output_delta, hidden_delta = self.backward(target, output,output_sum, hidden_sum)

            # Update weights and biases
            self.update_weights(predictors, output_delta, hidden_delta, hidden_activation)

        # Store and display average loss for the epoch
        avg_loss = total_loss / len(x)
        # **Apply Bold Driver every 50 epochs**
        if epoch % 250 == 0:
```

```
        if avg_loss < prev_loss:  # If loss decreased
            learning_rate = min(learning_rate * increase_factor, max_learning_rate)  # Increase learning rate
        else:  # If loss increased
            learning_rate = max(learning_rate * decrease_factor, min_learning_rate)  # Decrease learning rate

        # Update self.learning_rate after modifying it
        self.learning_rate = learning_rate

    # Update the previous loss for the next iteration
    prev_loss = avg_loss

    # Denormalize average loss
    denormalised_loss = denormalise(avg_loss, y_min, y_max)
    print(f"Epoch: {epoch+1}, Denormalised Loss: {denormalised_loss}")

    # Append denormalized loss to history
    self.loss_history.append(denormalised_loss)

return self.loss_history  # Return loss history for analysis
```

As you can see from the code above, all I had to do was add in a couple things to the training function. I set a maximum and a minimum learning rate to the learning rate must be in an interval. I also defined an increase and a decrease factor. Then I compared the new average loss to the previous average loss and if the new one was smaller than the new learning rate. Then the new average learning rate would be set by the old one multiplying the increase factor if that final answer is smaller than the maximum learning rate. This is the same but choose the greater of the learning rate multiplied by the decrease factor and the minimum learning rate if the new average loss is greater than the old average loss.

I then implemented it every 50, then 100, then 200, then finally 250 epochs, whilst increasing the number of epochs my MLP runs for to 1000. I then added a minimum and maximum learning rate to see if that was the issue and could hopefully stop the algorithm from becoming stuck at a local minimum. This decreased the meant error to below 10.8 which is the best yet.

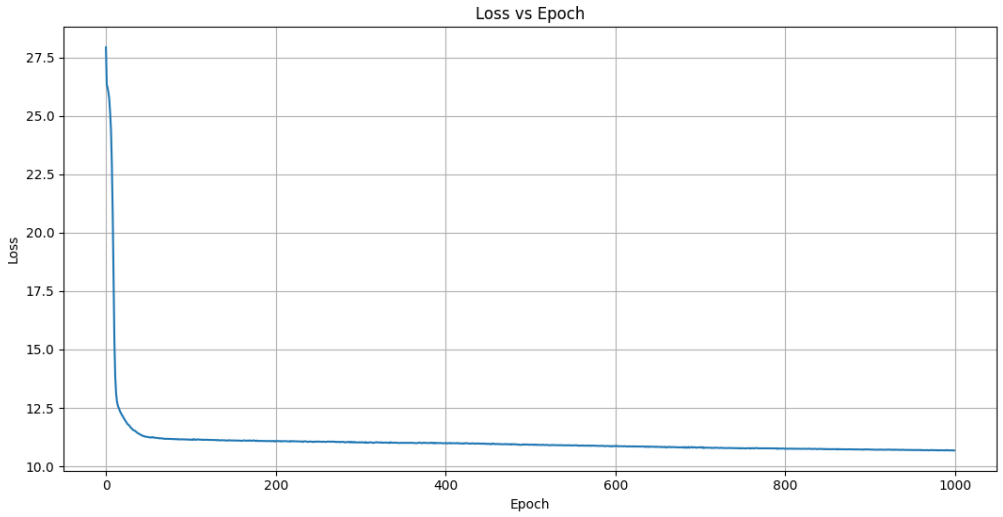


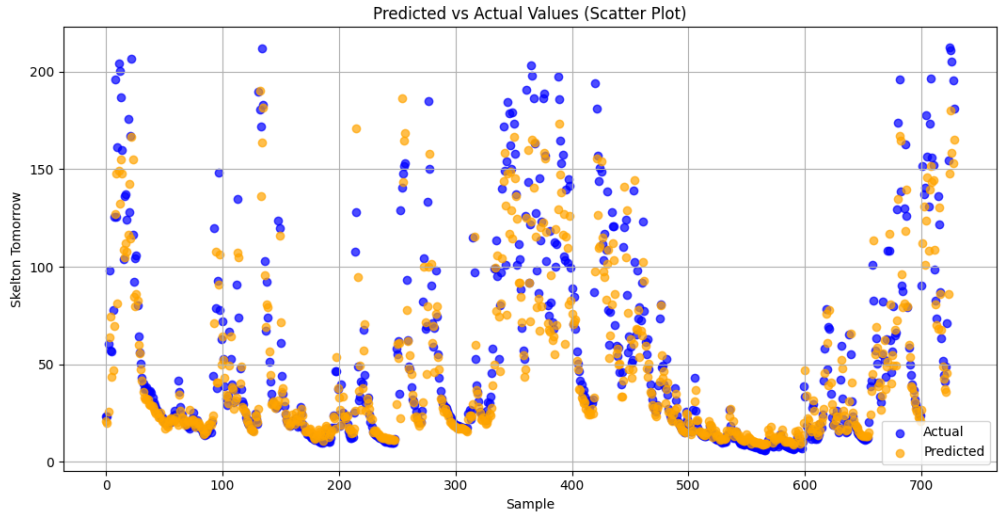


STEP 6 NEWTORK TRAINING - F325508

BOLD DRIVER AND MOMENTUM

Then I implemented bold driver with momentum with 1000 epoch and the bold driver checking every 250 and my final mean error went down to 10.67 which is even more of an improvement, and you can see the results below.



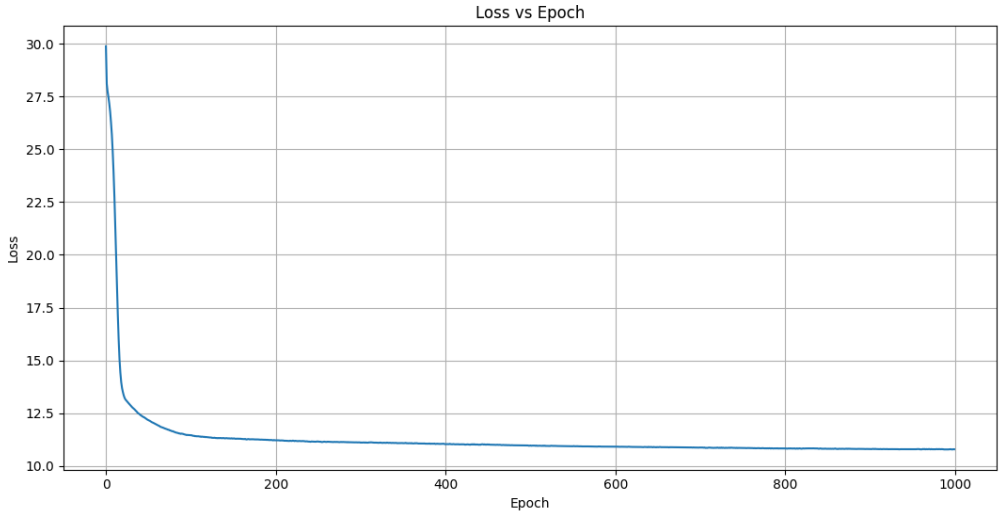


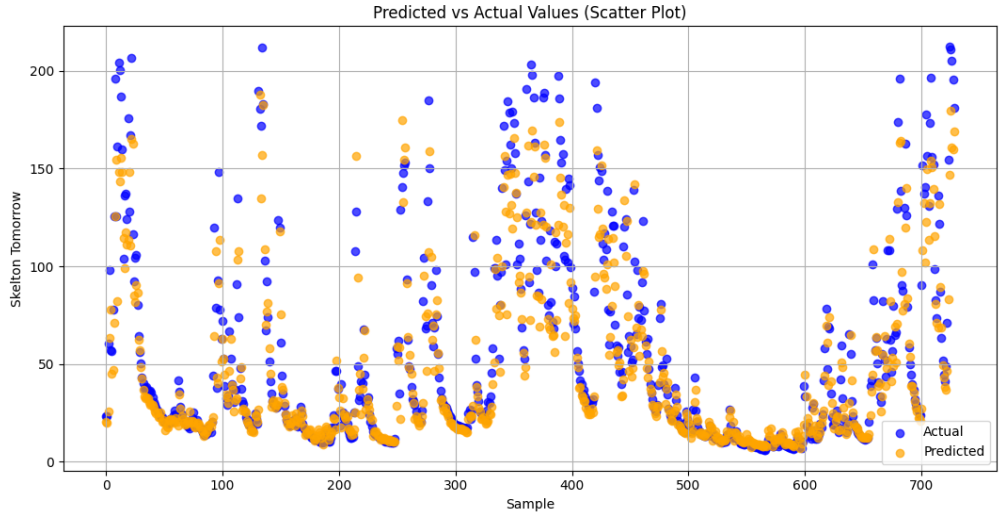
I did this through using the exact same training function for bold drive but then adding in the update weights with momentum function and from my results above I know it is correct.

WEIGHT DECAY

Then I implemented weight decay by itself in my MLP with 1000 epoch which meant the average error went down from 11.1 to 10.78 which is good. You can see the results below.

STEP 6 NEWTORK TRAINING - F325508





My code is below but all I changed in the train function was to call a new back pass function which you can see below:

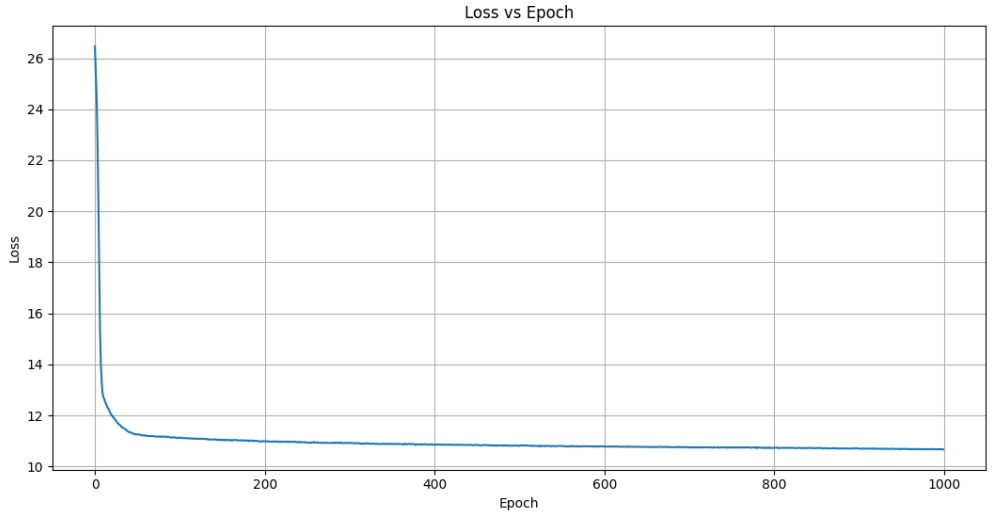
```
'''----------------BACKWARD WEIGHT DECAY-------------'''
def backward_weight_decay(self, epoch,y, output_layer_activation, output_layer_sum, hidden_layer_sum):
# Output node calculation
    hidden_omega = (1/(2*(self.weights_hidden_output.size)))*np.sum(self.weights_hidden_output**2)
    if epoch != 0:
        beta = 1/epoch
    else:
        beta = 0
    output_error = -self.absolute_error_derivative(y, output_layer_activation)+hidden_omega*beta
    output_delta = output_error * self.sigmoid_derivative(output_layer_sum)


    # Hidden node calculation
    hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
    hidden_delta = hidden_error * self.sigmoid_derivative(hidden_layer_sum)
    return output_delta, hidden_delta
```

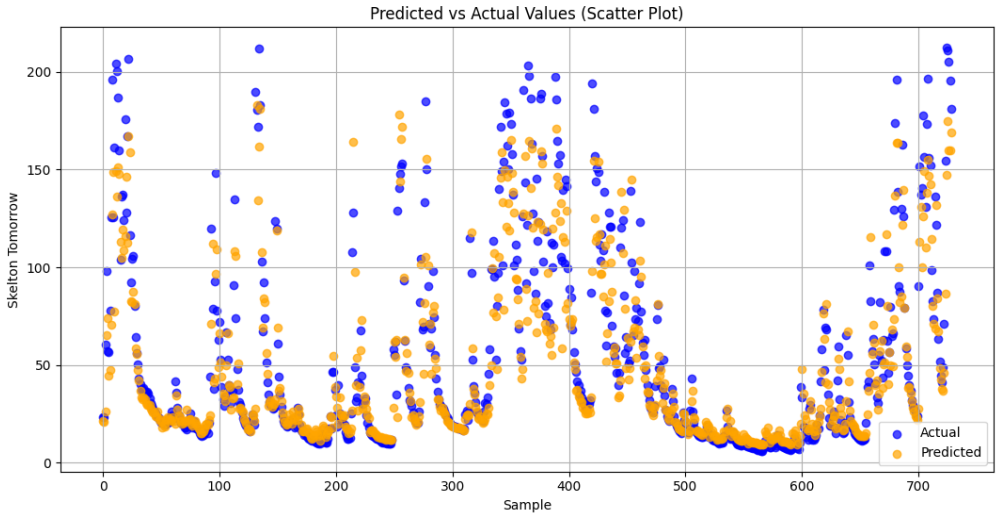
STEP 6 NEWTORK TRAINING - F325508

Here I took an extra parameter in which is the epoch. Then I defined hidden omega using the formula on the slide and then if the epoch is not 0 then a new beta is defined which is 1/epoch but if the epoch is equal to 0 then beta is defined at 0. This stops division by 0 error for the first epoch. Then I changed the output error to calculate the output error as normal but then add hidden omega multiplied by beta.

WEIGHT DECAY, BOLD DRIVER AND MOMENTUM

As annealing would counter act the bold driver, I decided to not include it and so below I decided to do the MLP with all 3 improvements (weight decay, bold driver and momentum which resulted in an error of 10.66.



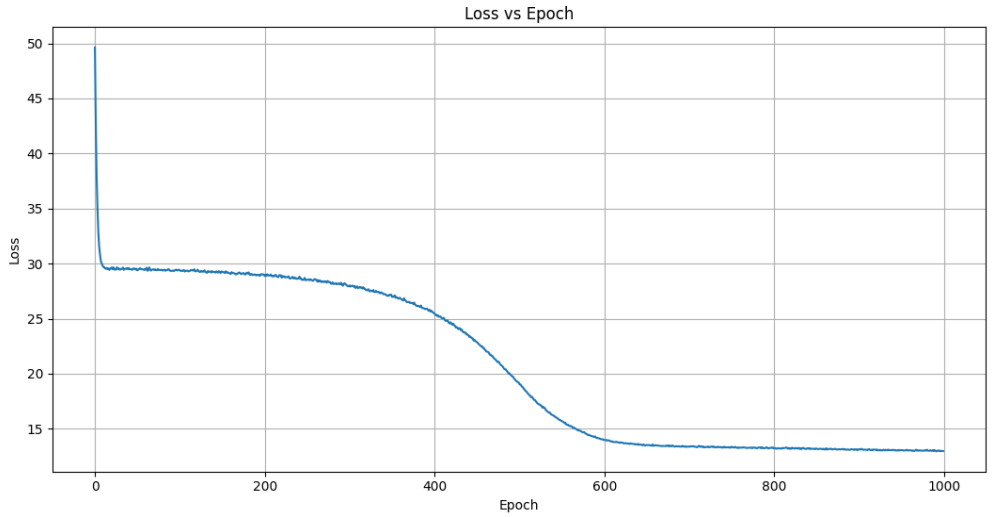


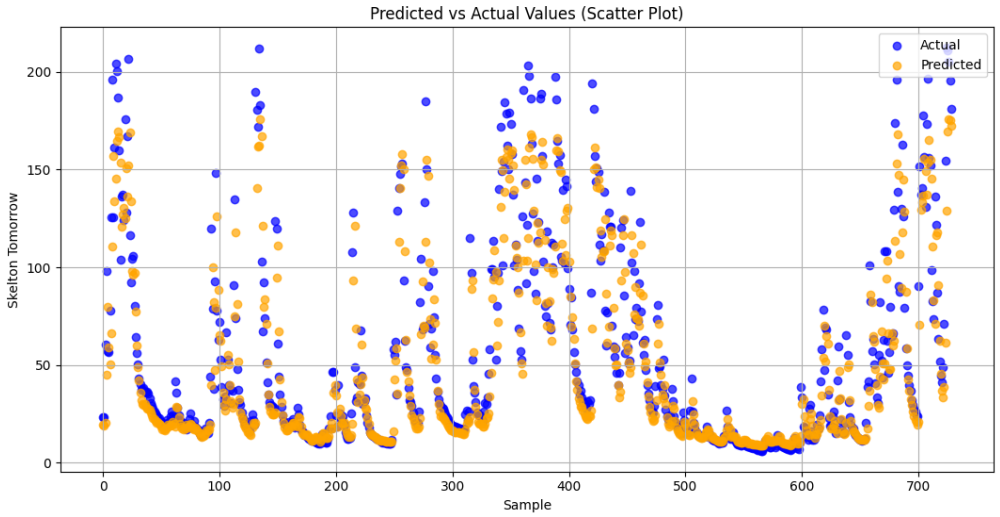
My new training function for all improvements was the same as the training function for momentum and bold driver except I changed the backward pass from the normal backwards pass to the backward pass that contained the backwards weight decay.

STEP 6 NEWTORK TRAINING - F325508

BATCH LEARNING

When applying batch learning my improvements with my MLP ran quite a lot faster but unfortunately was not quite as accurate due to not being able to fully use the improvements as well so the error was 12.9.





The only difference in code is this:

STEP 6 NEWTORK TRAINING - F325508

```
num_samples = len(x)

for epoch in range(self.epochs):  # Loop through epochs
    total_loss = 0  # Track total loss for this epoch

    # Shuffle dataset indices
    indices = np.arange(num_samples)
    np.random.shuffle(indices)

    for batch_start in range(0, num_samples, batch_size):  # Iterate over batches
        batch_indices = indices[batch_start:batch_start + batch_size]
        batch_x = x[batch_indices]
        batch_y = y[batch_indices]
```

So I iterated through the batches and instead of using x[i] and y[i] for the predictors and the predictand I made batch_x and batch_y variables using the batch indices.

This means that my function traversed through the data a lot faster.

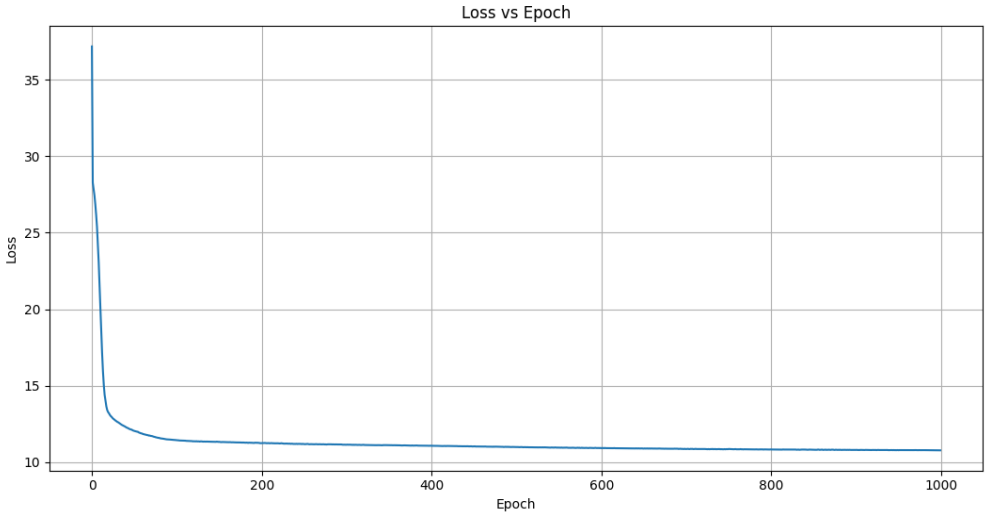
SUMMARY OF IMPROVEMENTS

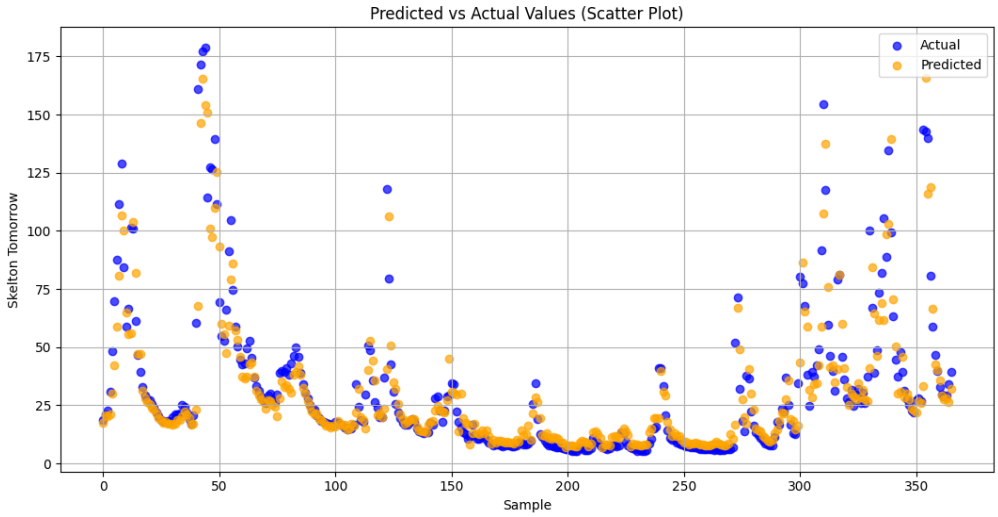
Whilst comparing all improvements and combination of improvements I found the best combination was the Momentum with Weight Decay and Bold Driver combined training function as they produced the best results by far with the training data so I believe the function will be better for the validation data.

STEP 7 EVALUATION OF THE FINAL MODEL

BASIC MODEL WITH VALIDATION DATA

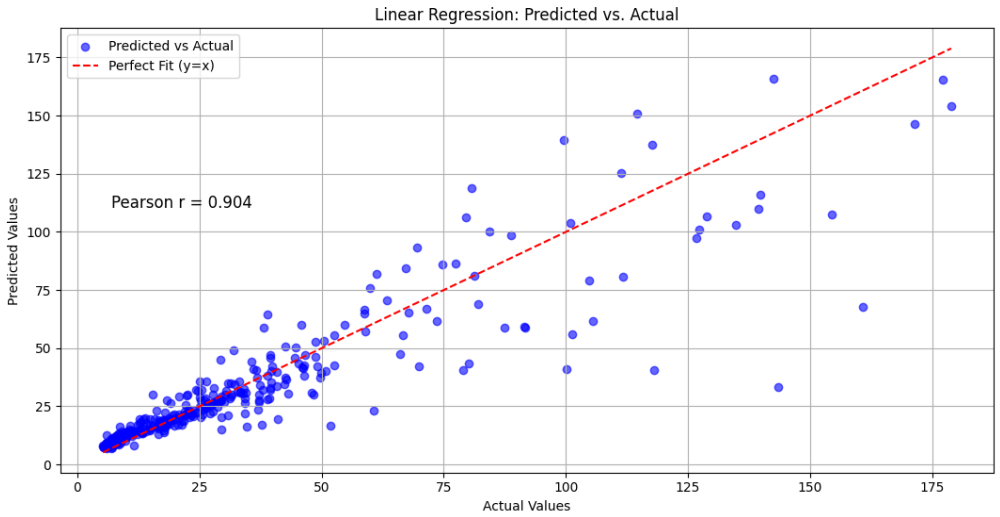
The mean absolute error 10.76 which I believe is quite good.





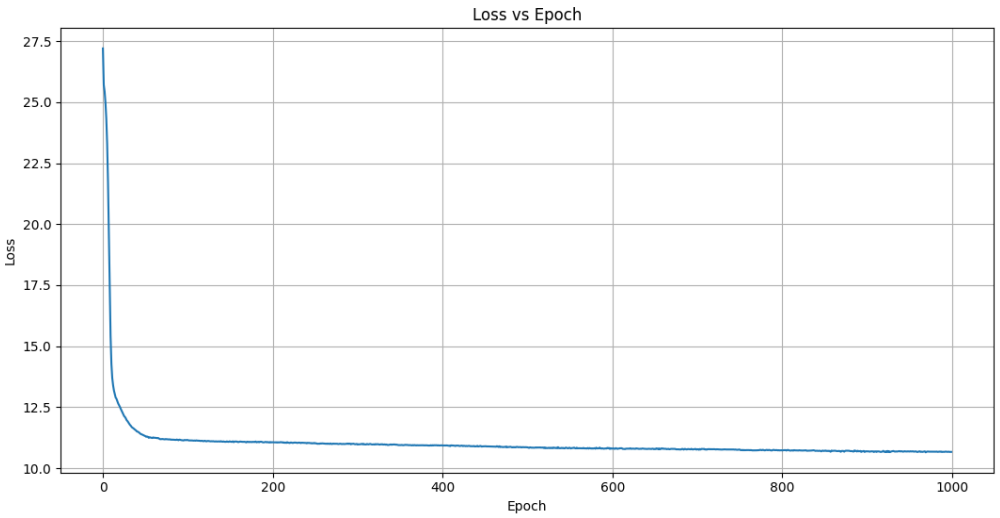
STEP 7 EVALUATION OF THE FINAL MODEL - F325508

I think that this model is very suited to the data, so I believe that my model is not over fitted to the training data.

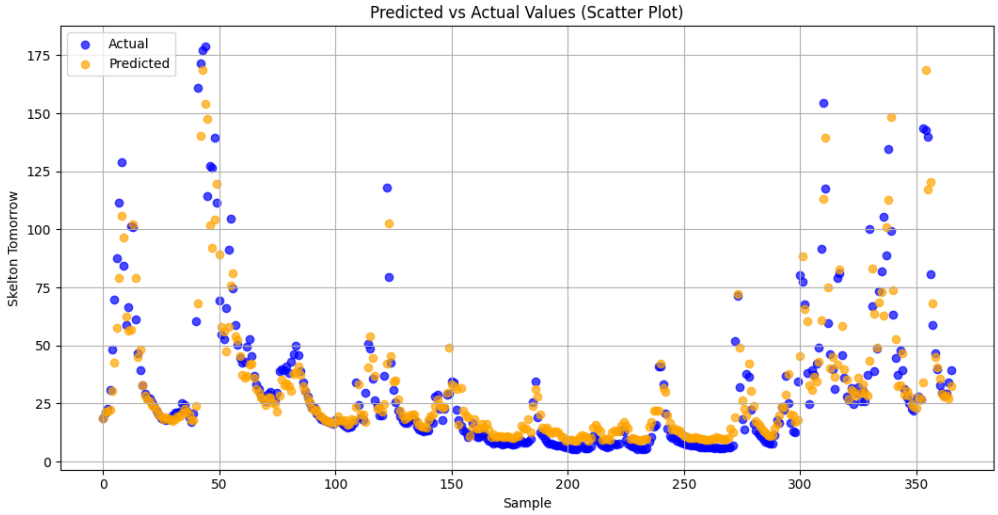


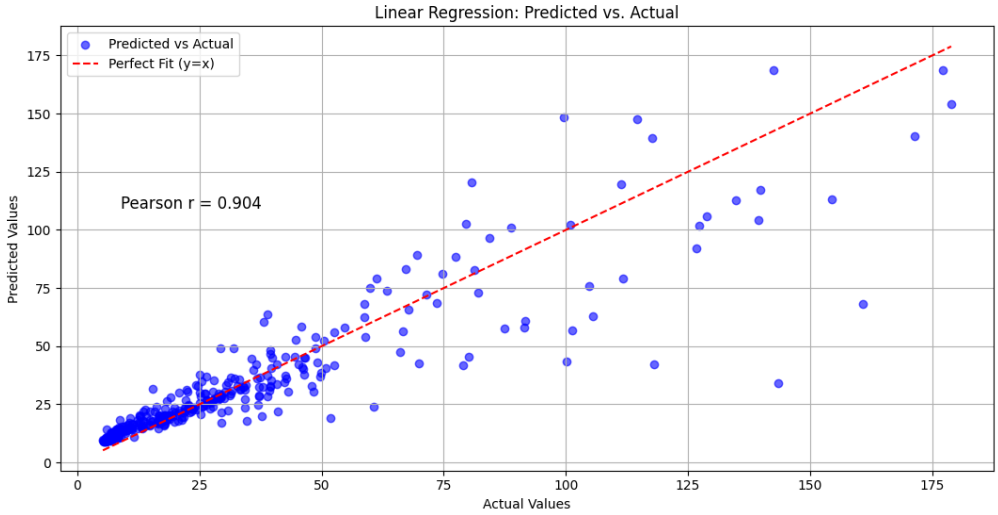
MODEL WITH ALL 3 IMPROVEMENTS WITH VALIDATION TEST DATA

This had a mean absolute error of 10.66, so the error decreases a bit which is good.



STEP 7 EVALUATION OF THE FINAL MODEL - F325508





The Pearson R value is still roughly the same which suggests despite there being less error predicted on average with the improvements, my MLP still predicts roughly the same which may mean that my improvements were possible more towards my training data. This may mean that my MLP with the improvements is a bit overtrained to my training data, however a Pearson R value score of 0.9 is good as I believe anything above 0.8 means that the MLP is doing well. So, I believe that my MLP is well trained.

STEP 7 EVALUATION OF THE FINAL MODEL - F325508

OTHER MODELS WITH VAL DATA

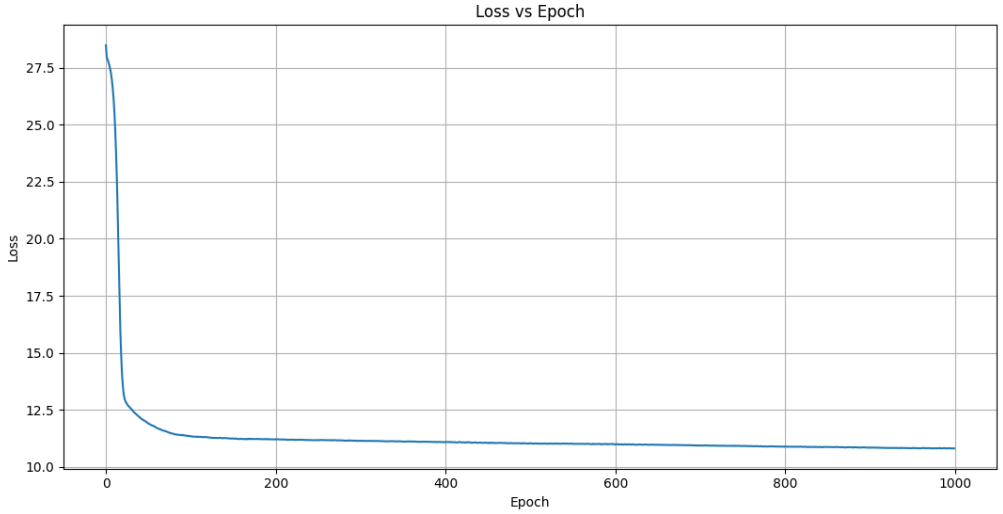
The Bold driver error was 10.78 which means this may not help my validation data as much as I expected.

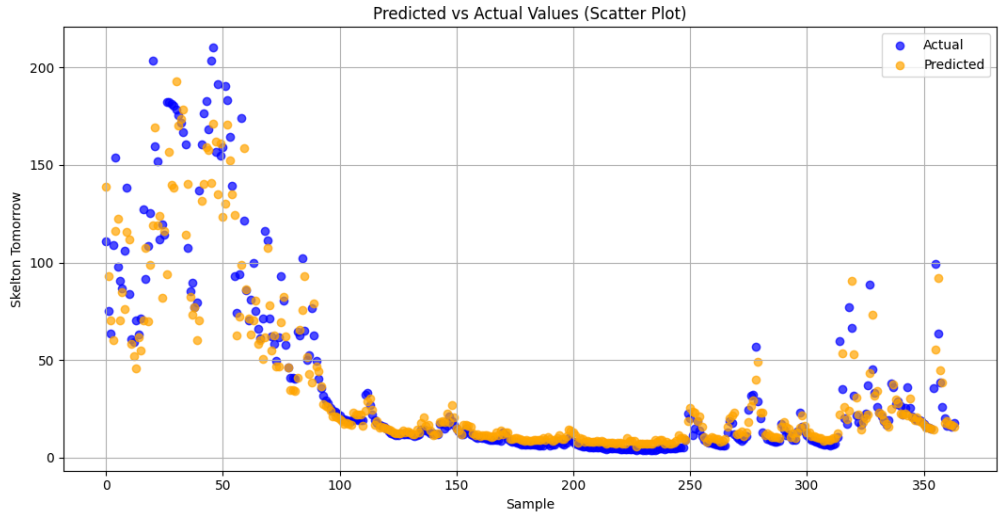
The Weight decay error was 10.8 so this is the same issue as bold driver, however I do think the random weights may have an effect, as if the weights were randomised close to the ideal then my weight decay may produce a better error.

The Bold driver with Momentum error was about 10.7 too which suggest that out of my three improvements momentum probably has the greatest effect and may help Bold driver improve my MLPs performance. This also suggest that Weight Decay may not help my MLP that much in comparison to Bold Driver and Momentum.

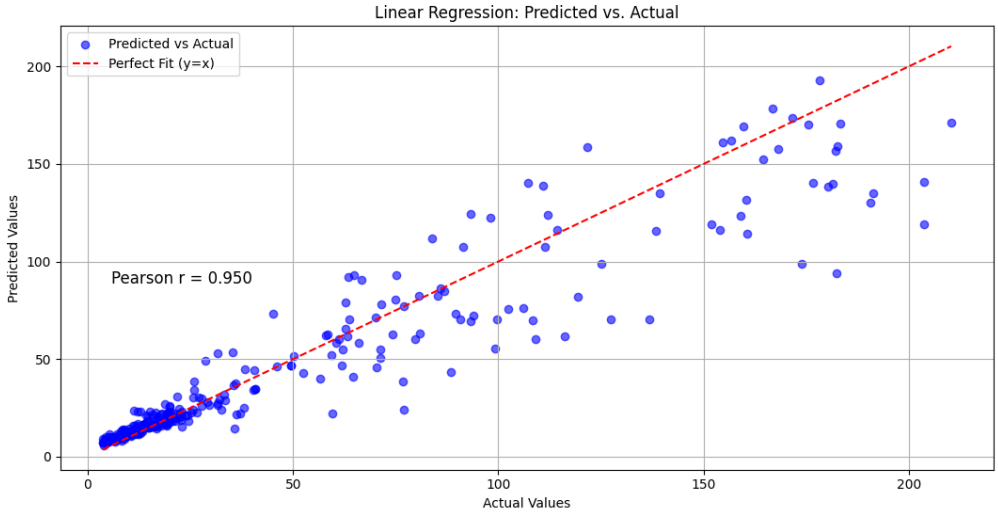
BASIC MODEL WITH TEST DATA

The Loss was 10.8 which is good.





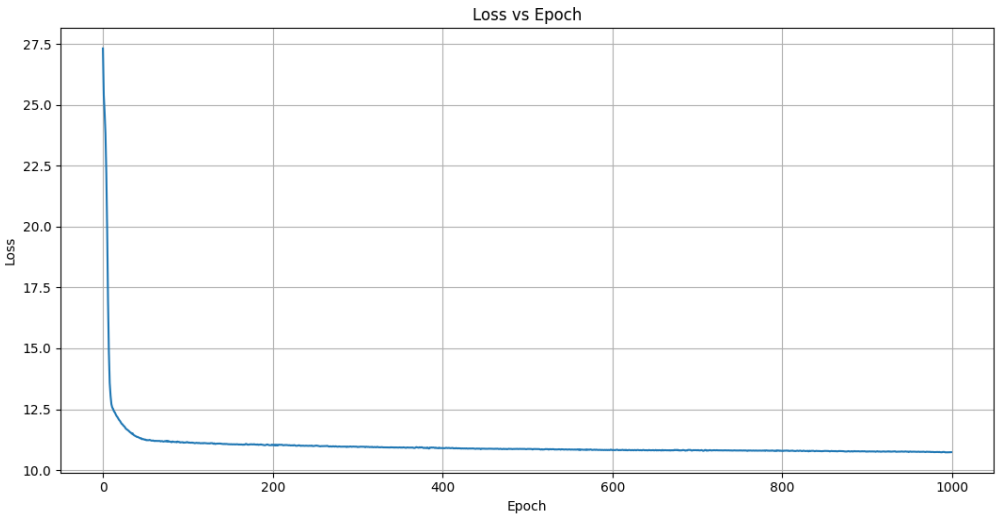
STEP 7 EVALUATION OF THE FINAL MODEL - F325508



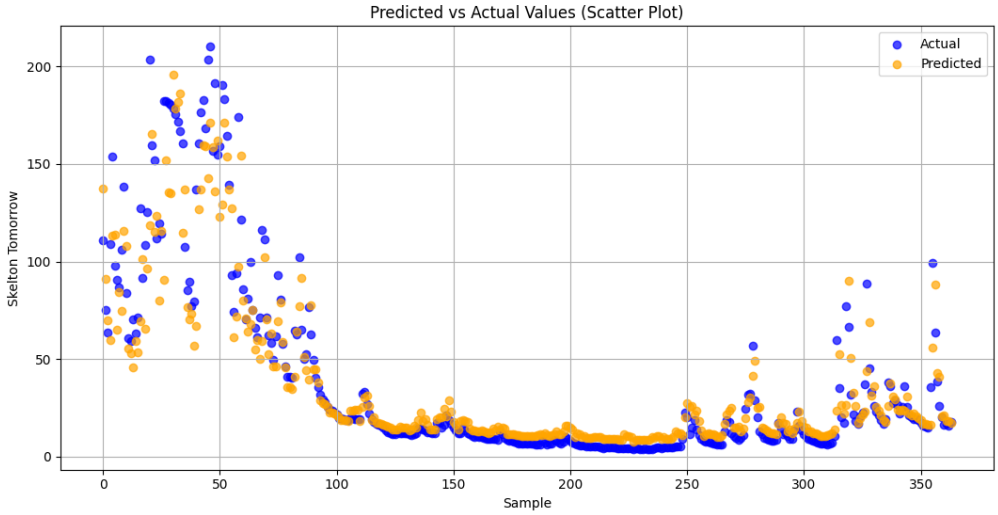
The Pearson R score of 0.95 is very impressive and shows that my MLP is not that overtrained with the training data as it can predict the test data well.

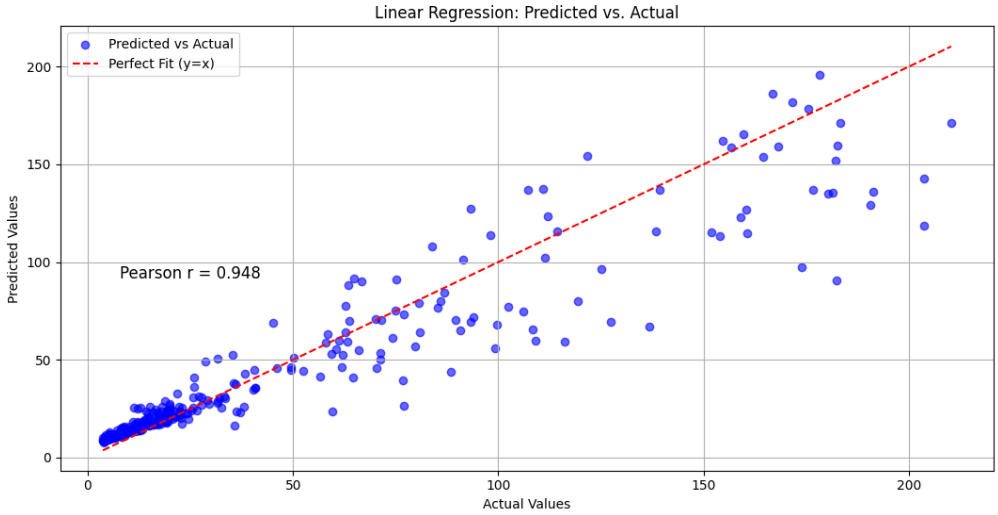
MODEL WITH ALL 3 IMPROVEMENTS WITH TEST DATA

The error was 10.72 which is good.



STEP 7 EVALUATION OF THE FINAL MODEL - F325508





Even though the average error decreased, the Pearson R value did not increase, it stayed roughly the same which again possibly suggests that my MLP improvements are overtrained with the training data but the MLP itself is not. However, the Pearson R coefficient is still good, so I do not think the MLP improvements are that much overtrained.

STEP 7 EVALUATION OF THE FINAL MODEL - F325508

THINGS I WOULD IMPROVE - FINAL COMMENTS

I would randomize my dataset to reduce outliers, which may improve the performance of my model during validation and testing, leading to more reliable results. One limitation of my code is that it currently uses only one hidden layer. I struggled with implementing additional layers, but I believe including multiple hidden layers would enhance the performance of my MLP and yield better outputs.

To improve code readability and maintainability, I plan to split my code into multiple files for easier navigation and understanding. Additionally, I aim to incorporate alternative training functions, such as conjugate functions, to optimize my model further.

Another area of focus is improving the model's ability to predict river flooding. Currently, river flow calculations might be inaccurate if water disperses on either side of the river during flooding events. Enhancing the model to account for such scenarios would significantly improve its predictions and accuracy in real-world applications.

I would also like to include alternative training functions like conjugate functions.

MY CODE - F325508

MY CODE

This is the main python file.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from cleaning import remove_outliers
import seaborn as sns
from scipy.interpolate import PchipInterpolator as pchip_interpolate
from scipy.stats import pearsonr


# Load the data from the Excel file
df = pd.read_excel('Mean water.xlsx', header=1)

# Rename columns correctly
df.columns = ["Date", "Crakehill", "Skip Bridge", "Westwick", "Skelton",
              "Arkengarthdale", "East Cowton", "Malham Tarn", "Snaizeholme",
              "Unused_1", "Unused_2", "Notes"]

# Drop unnecessary columns and convert date to datetime
df_water = df[["Date", "Crakehill", "Skip Bridge", "Westwick",
"Skelton"]].copy()
df_water["Date"] = pd.to_datetime(df_water["Date"], errors='coerce')

# Convert numeric columns to numeric types, forcing errors to NaN
for col in ["Crakehill", "Skip Bridge", "Westwick", "Skelton"]:
    df_water[col] = pd.to_numeric(df_water[col], errors='coerce')

#Plot original data individually against time
for col in ["Crakehill", "Skip Bridge", "Westwick", "Skelton"]:
   plt.figure(figsize=(10, 4))
   plt.scatter(df_water["Date"], df_water[col], label=f"{col} (Original)",
alpha=0.6)
   plt.xlabel("Date")
   plt.ylabel("Flow (Cumecs)")
   plt.title(f"Mean Daily Flow Before Cleaning - {col}")
   plt.legend()
   plt.grid(True)
   plt.show()



'''---Cleaning river flow: Outlier removal---'''
for col in ["Crakehill", "Skip Bridge", "Westwick", "Skelton"]:
    mean_value = df_water[col].mean()
    std_dev = df_water[col].std()
    df_water[col] = remove_outliers(df_water[col].values, mean_value, std_dev)
```

MY CODE - F325508

```
# Plot cleaned data individually against time
for col in ["Crakehill", "Skip Bridge", "Westwick", "Skelton"]:
    plt.figure(figsize=(10, 4))
    plt.scatter(df_water["Date"], df_water[col], label=f"{col} (Cleaned)",
alpha=0.6)
    plt.xlabel("Date")
    plt.ylabel("Flow (Cumecs)")
    plt.title(f"Mean Daily Flow After Cleaning - {col}")
    plt.legend()
    plt.grid(True)
    plt.show()


'''---------Interpolate missing values----------'''
for col in ["Crakehill", "Skip Bridge", "Westwick", "Skelton"]:
    mask = df_water[col].notna()

    # Ensure Date column has no NaN values by filling them with previous
values
    df_water["Date"] = df_water["Date"].ffill()  # Forward fill missing dates

    # Convert Date to ordinal (integer format) for interpolation
    x = df_water["Date"][mask].map(pd.Timestamp.toordinal)
    y = df_water[col][mask]
    x_interp = df_water["Date"].map(pd.Timestamp.toordinal)

    # Perform PCHIP interpolation
    interpolator = pchip_interpolate(x, y)
    df_water[col] = interpolator(x_interp)



# Apply moving averages
window_size = 4  # Define window size for moving average
df_water.loc[:, ["Crakehill", "Skip Bridge", "Westwick"]] = df_water.loc[:,
["Crakehill", "Skip Bridge", "Westwick"]].rolling(window=window_size,
min_periods=1).mean()

#Lag skelton values by 1 day
df_water["Skelton tommorow"] = df_water["Skelton"].shift(-1)
df_water["Predictor_1"] = df_water["Crakehill"] + df_water["Skip Bridge"] +
df_water["Westwick"]
df_water = df_water[["Date","Predictor_1", "Skelton"]]
```

MY CODE - F325508

```
# Round values to 3 decimal places
df_water = df_water.round(3)




# Save cleaned data to a new file
df_water.to_csv('Cleaned_Water_Data.txt', sep='\t', index=False)
print("Data cleaning complete. Cleaned file saved as
'Cleaned_Water_Data.csv'.")









#---- Process rainfall data------
df_rainfall = df[["Date", "Arkengarthdale", "East Cowton", "Malham Tarn",
"Snaizeholme"]].copy()
df_rainfall["Date"] = pd.to_datetime(df_rainfall["Date"], errors='coerce')
for col in ["Arkengarthdale", "East Cowton", "Malham Tarn", "Snaizeholme"]:
    df_rainfall[col] = pd.to_numeric(df_rainfall[col], errors='coerce')


# Plot original data individually against time
for col in ["Arkengarthdale", "East Cowton", "Malham Tarn", "Snaizeholme"]:
    plt.figure(figsize=(10, 4))
    plt.scatter(df_rainfall["Date"], df_rainfall[col], label=f"{col}
(Original)", alpha=0.6)
    plt.xlabel("Date")
    plt.ylabel("Flow (Cumecs)")
    plt.title(f"Mean Daily Flow Before Cleaning - {col}")
    plt.legend()
    plt.grid(True)
    plt.show()


'''------Cleaning rainfall data by removing averages-----'''
for col in ["Arkengarthdale", "East Cowton", "Malham Tarn", "Snaizeholme"]:
    mean_value = df_rainfall[col].mean()
    std_dev = df_rainfall[col].std()
    df_rainfall[col] = remove_outliers(df_rainfall[col], mean_value, std_dev)


# Plot cleaned data individually against time
```

MY CODE - F325508

```
for col in ["Arkengarthdale", "East Cowton", "Malham Tarn", "Snaizeholme"]:
    plt.figure(figsize=(10, 4))
    plt.scatter(df_rainfall["Date"], df_rainfall[col], label=f"{col}
(Cleaned)", alpha=0.6)
    plt.xlabel("Date")
    plt.ylabel("Flow (Cumecs)")
    plt.title(f"Mean Daily Flow After Cleaning - {col}")
    plt.legend()
    plt.grid(True)
    plt.show()


# Interpolate missing values in rainfall data using PCHIP
for col in ["Arkengarthdale", "East Cowton", "Malham Tarn", "Snaizeholme"]:
    mask = df_rainfall[col].notna()

    # Ensure Date column has no NaN values by filling them with previous
values
    df_rainfall["Date"] = df_rainfall["Date"].ffill()  # Forward fill missing
dates

    # Convert Date to ordinal (integer format) for interpolation
    x = df_rainfall["Date"][mask].map(pd.Timestamp.toordinal)
    y = df_rainfall[col][mask]
    x_interp = df_rainfall["Date"].map(pd.Timestamp.toordinal)

    # Perform PCHIP interpolation
    interpolator = pchip_interpolate(x, y)
    df_rainfall[col] = interpolator(x_interp)



'''------Apply moving averages to Rainfall data'''
df_rainfall.loc[:, ["Arkengarthdale", "East Cowton", "Malham Tarn",
"Snaizeholme"]] = (df_rainfall.loc[:, ["Arkengarthdale", "East Cowton",
"Malham Tarn", "Snaizeholme"]].rolling(window=window_size,
min_periods=1).mean())
df_rainfall = df_rainfall.round(3)


'''--------Merge Water and Rainfall data----'''
df_cleaned = pd.merge(df_rainfall, df_water, on="Date", how="inner")

# Creating predictand and predictors
df_cleaned['Skelton Tomorrow'] = df_cleaned['Skelton'].shift(-1)
df_cleaned['Skelton - 1'] = df_cleaned['Skelton'].shift(1)
df_cleaned['Average Rainfall'] = df_cleaned[['Arkengarthdale', 'East Cowton',
'Malham Tarn', 'Snaizeholme']].mean(axis=1)
```

MY CODE - F325508

```
df_cleaned['Average River flow'] = df_cleaned[['Crakehill', 'Skip Bridge',
'Westwick', 'Skelton']].mean(axis=1)
df_cleaned['Average Rainfall -1 day'] = df_cleaned['Average
Rainfall'].shift(1)
df_cleaned['Skelton + average Rain'] = df_cleaned['Skelton'] +
df_cleaned['Average Rainfall']
# + skelton today


# Save cleaned data to a new file
df_cleaned.to_csv('Cleaned_Data.txt', sep='\t', index=False)



# Lagging Rainfall Data by 1 to 7 Days
df_lagged_data = df_cleaned.copy()

for lag in range(1, 7):
    for col in ["Arkengarthdale", "East Cowton", "Malham Tarn", "Snaizeholme",
"Predictor_1", "Skelton"]:
        df_lagged_data[f"{col}_lag{lag}"] = df_cleaned[col].shift(lag)

df_rainfall = df_rainfall.round(3)
df_lagged_data = df_lagged_data.round(3)
df_lagged_data.to_csv('Lagged_Rainfall_Data.txt', sep='\t', index=False)



# Calculate correlation matrix
correlation_matrix_rain = df_lagged_data.copy().corr()


# Plot Heatmap of Skelton vs Rainfall Locations
correlation_data = df_cleaned
correlation_matrix = correlation_data.corr()

plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Correlation Heatmap: Finding Predictors")
plt.show()



# Save cleaned data to a new file
df_rainfall.to_csv('Cleaned_Rainfall_Data.txt', sep='\t', index=False)
print("Data cleaning complete. Cleaned file saved as
'Cleaned_Rainfall_Data.csv'.")
```

```
# Creates MLP class with matrix multiplication

class MLP:
    def __init__(self,x,y,input_size,output_size, hidden_size, learning_rate,
epochs):
        self.x = x
        self.y = y
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        # Weight initialization with uniform distribution
        self.weights_input_hidden = np.random.uniform(-2/input_size,
2/input_size, (input_size, hidden_size))
        self.weights_hidden_output = np.random.uniform(-2/hidden_size,
2/hidden_size, (hidden_size, output_size))

        # Bias initialization to go between 1 and 0 for sigmoid activation
        self.bias_hidden = np.random.uniform(-1, 1, (1, hidden_size))
        self.bias_output = np.random.uniform(-1, 1, (1, output_size))

        self.learning_rate = learning_rate
        self.epochs = epochs
        self.loss_history = []



    '''----------------------ACTIVATION FUNCTIONS----------------------'''
    #Sigmoid function
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    #Sigmoid derivative function
    def sigmoid_derivative(self, node_sum):
        activation = self.sigmoid(node_sum)
        return activation * (1 - activation)

    #Tanh function
    def tanh(self, x):
        return np.tanh(x)

    #Tanh derivative function
    def tanh_derivative(self, x):
```

```
        return 1 - np.tanh(x) ** 2

    #Relu function
    def relu(self, x):
        return np.maximum(0, x)

    #Relu derivative function
    def relu_derivative(self, x):
        return np.where(x <= 0, 0,
                        np.where(x > 0, 1, 0))

    '''----------------------Error FUNCTIONS----------------------'''
    #Mean Absolute error
    def absolute_error(self, y, y_pred):
        return np.mean(np.abs(y - y_pred))

    #Mean Absolute error derivative
    def absolute_error_derivative(self, y, output):
        return np.where(output > y, 1, -1)


    #Mean squared error
    def mean_squared_error(self, y, y_pred):
        return np.mean((y - y_pred) ** 2)

    #Mean squared error derivative
    def mean_squared_error_derivative(self, y, y_pred):
        return 2 * (y - y_pred) / len(y)


    #Root Mean squared error
    def root_mean_squared_error(self, y, y_pred):
        return np.sqrt(np.mean((y - y_pred) ** 2))

    #Root Mean squared error derivative
    def root_mean_squared_error_derivative(self, y, y_pred):
        return 2 * (y - y_pred) / len(y)

    '''---------------------- FORWARD and BACKWARD PROPAGATION----------------
------'''
    def forward(self, x):
        #Input to hidden layer

        hidden_layer_sum = np.dot(x, self.weights_input_hidden) +
self.bias_hidden
        hidden_layer_activation = self.sigmoid(hidden_layer_sum)

        #Hidden to output layer
```

MY CODE - F325508

```
        output_layer_sum = np.dot(hidden_layer_activation,
self.weights_hidden_output) + self.bias_output
        output_layer_activation = self.sigmoid(output_layer_sum)

        return output_layer_activation, output_layer_sum,
hidden_layer_activation, hidden_layer_sum


    def backward(self, y, output_layer_activation, output_layer_sum,
hidden_layer_sum):
    # Output node calculation
        output_error = -self.absolute_error_derivative(y,
output_layer_activation)
        output_delta = output_error *
self.sigmoid_derivative(output_layer_sum)


        # Hidden node calculation
        hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
        hidden_delta = hidden_error *
self.sigmoid_derivative(hidden_layer_sum)
        return output_delta, hidden_delta

    '''----------------BACKWARD WEIGHT DECAY-------------'''
    def backward_weight_decay(self, epoch,y, output_layer_activation,
output_layer_sum, hidden_layer_sum):
    # Output node calculation
        hidden_omega =
(1/(2*(self.weights_hidden_output.size)))*np.sum(self.weights_hidden_output**2
)
        if epoch != 0:
            beta = 1/epoch
        else:
            beta = 0
        output_error = -self.absolute_error_derivative(y,
output_layer_activation)+hidden_omega*beta
        output_delta = output_error *
self.sigmoid_derivative(output_layer_sum)


        # Hidden node calculation
        hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
        hidden_delta = hidden_error *
self.sigmoid_derivative(hidden_layer_sum)
        return output_delta, hidden_delta
```

MY CODE - F325508

```
    '''----------------------UPDATE WEIGHTS AND BIASES----------------------
'''
    def update_weights(self, x, output_delta,
hidden_delta,hidden_layer_activation):
        #update weights and biases

        x = x.reshape(1, -1)

        self.weights_hidden_output +=
np.dot(hidden_layer_activation.T,output_delta) * self.learning_rate
        self.weights_input_hidden += np.dot(x.T, hidden_delta) *
self.learning_rate

        self.bias_output += output_delta * self.learning_rate
        self.bias_hidden += hidden_delta * self.learning_rate

    '''-----------------UPDATE WEIGHTS AND BIASES WITH MOMENTUM---------------
-------'''
    def update_weights_with_momentum(self, x, output_delta, hidden_delta,
hidden_layer_activation):
        x = x.reshape(1, -1)

        # Initialise previous weight changes if they don't exist
        if not hasattr(self, 'prev_weight_change_hidden_output'):
            self.prev_weight_change_hidden_output =
np.zeros_like(self.weights_hidden_output)
            self.prev_weight_change_input_hidden =
np.zeros_like(self.weights_input_hidden)
            self.prev_bias_change_output = np.zeros_like(self.bias_output)
            self.prev_bias_change_hidden = np.zeros_like(self.bias_hidden)

        # Momentum parameter
        momentum = 0.9  # Feel free to adjust this value

        # Compute the current weight changes (gradients scaled by the learning
rate)
        weight_change_hidden_output = np.dot(hidden_layer_activation.T,
output_delta) * self.learning_rate
        weight_change_input_hidden = np.dot(x.T, hidden_delta) *
self.learning_rate
        bias_change_output = output_delta * self.learning_rate
        bias_change_hidden = hidden_delta * self.learning_rate

        # Apply momentum: combine the current weight changes with 0.9 *
previous changes
        self.weights_hidden_output += weight_change_hidden_output + momentum *
self.prev_weight_change_hidden_output
```

```
        self.weights_input_hidden += weight_change_input_hidden + momentum *
self.prev_weight_change_input_hidden
        self.bias_output += bias_change_output + momentum *
self.prev_bias_change_output
        self.bias_hidden += bias_change_hidden + momentum *
self.prev_bias_change_hidden

        # Save the current weight changes as the previous changes for the next
iteration
        self.prev_weight_change_hidden_output = weight_change_hidden_output
        self.prev_weight_change_input_hidden = weight_change_input_hidden
        self.prev_bias_change_output = bias_change_output
        self.prev_bias_change_hidden = bias_change_hidden


    '''--------------------UPDATE WEIGHTS WITH BATCH LEARNING------'''
    def update_weights_batch(self, X_batch, output_delta_batch,
hidden_delta_batch, hidden_activation_batch):

        batch_size = X_batch.shape[0]

        # Compute batch sum of weight updates
        delta_w_hidden_output = np.dot(hidden_activation_batch.T,
output_delta_batch) / batch_size
        delta_w_input_hidden = np.dot(X_batch.T, hidden_delta_batch) /
batch_size

        delta_b_output = np.sum(output_delta_batch, axis=0, keepdims=True) /
batch_size
        delta_b_hidden = np.sum(hidden_delta_batch, axis=0, keepdims=True) /
batch_size

        # Apply updates
        self.weights_hidden_output += self.learning_rate *
delta_w_hidden_output
        self.weights_input_hidden += self.learning_rate * delta_w_input_hidden

        self.bias_output += self.learning_rate * delta_b_output
        self.bias_hidden += self.learning_rate * delta_b_hidden

    '''---------------------TRAINING FUNCTION----------------------'''
    def train(self, x, y):
        self.loss_history = []  # Store loss for analysis

        for epoch in range(self.epochs):  # Loop through epochs
            total_loss = 0  # Track total loss for this epoch

            for i in range(len(x)):  # Iterate over each training sample
```

```
                predictors = x[i].reshape(1, -1)  # Ensure correct shape (row
vector)
                target = y[i].reshape(1, -1)  # Ensure correct shape (row
vector)



                # Forward pass
                output, output_sum, hidden_activation, hidden_sum =
self.forward(predictors)

                # Compute loss and accumulate it
                loss = self.absolute_error(target, output).sum()

                total_loss += loss

                # Backward pass (compute gradients)
                output_delta, hidden_delta = self.backward(target,
output,output_sum, hidden_sum)

                # Update weights and biases
                self.update_weights(predictors, output_delta, hidden_delta,
hidden_activation)

            # Store and display average loss for the epoch
            avg_loss = total_loss / len(x)

            # Denormalize average loss
            denormalised_loss = denormalise(avg_loss, y_min, y_max)
            print(f"Epoch: {epoch+1}, Denormalised Loss: {denormalised_loss}")

            # Append denormalized loss to history
            self.loss_history.append(denormalised_loss)

        return self.loss_history  # Return loss history for analysis




    '''----------------TRAINING FUNCTION WITH MOMENTUM----------------'''
    def train_with_momentum(self, x, y):
        self.loss_history = []  # Store loss for analysis

        for epoch in range(self.epochs):  # Loop through epochs
            total_loss = 0  # Track total loss for this epoch

            for i in range(len(x)):  # Iterate over each training sample
                predictors = x[i].reshape(1, -1)  # Ensure correct shape (row
vector)
```

```
                target = y[i].reshape(1, -1)  # Ensure correct shape (row
vector)

                # Forward pass
                output, output_sum, hidden_activation, hidden_sum =
self.forward(predictors)

                # Compute loss and accumulate it
                loss = self.absolute_error(target, output)

                total_loss += loss

                # Backward pass (compute gradients)
                output_delta, hidden_delta = self.backward(target,
output,output_sum, hidden_sum)

                # Update weights and biases
                self.update_weights_with_momentum(predictors, output_delta,
hidden_delta, hidden_activation)

            # Store and display average loss for the epoch
            avg_loss = total_loss / len(x)

            # Denormalize average loss
            denormalised_loss = denormalise(avg_loss, y_min, y_max)
            print(f"Epoch: {epoch+1}, Denormalised Loss: {denormalised_loss}")

            # Append denormalized loss to history
            self.loss_history.append(denormalised_loss)

        return self.loss_history  # Return loss history for analysis




    '''----------------TRAINING FUNCTION WITH BOLD DRIVER----------------'''
    def train_with_bold_driver(self, x, y):
        self.loss_history = []  # Store loss for analysis
        max_learning_rate = 0.5
        min_learning_rate = 0.01

        learning_rate = self.learning_rate

        # Initialize bold driver parameters
        increase_factor = 1.1
        decrease_factor = 0.5
        prev_loss = float('inf')
```

MY CODE - F325508

```
        for epoch in range(self.epochs):  # Loop through epochs
            total_loss = 0  # Track total loss for this epoch

            for i in range(len(x)):  # Iterate over each training sample
                predictors = x[i].reshape(1, -1)  # Ensure correct shape (row
vector)
                target = y[i].reshape(1, -1)  # Ensure correct shape (row
vector)

                # Forward pass
                output, output_sum, hidden_activation, hidden_sum =
self.forward(predictors)

                # Compute loss and accumulate it
                loss = self.absolute_error(target, output).sum()

                total_loss += loss

                # Backward pass (compute gradients)
                output_delta, hidden_delta = self.backward(target,
output,output_sum, hidden_sum)

                # Update weights and biases
                self.update_weights(predictors, output_delta, hidden_delta,
hidden_activation)

            # Store and display average loss for the epoch
            avg_loss = total_loss / len(x)
            # **Apply Bold Driver every 50 epochs**
            if epoch % 250 == 0:
                if avg_loss < prev_loss:  # If loss decreased
                    learning_rate = min(learning_rate * increase_factor,
max_learning_rate)  # Increase learning rate
                else:  # If loss increased
                    learning_rate = max(learning_rate * decrease_factor,
min_learning_rate)  # Decrease learning rate

                # Update self.learning_rate after modifying it
                self.learning_rate = learning_rate

            # Update the previous loss for the next iteration
            prev_loss = avg_loss

            # Denormalize average loss
            denormalised_loss = denormalise(avg_loss, y_min, y_max)
            print(f"Epoch: {epoch+1}, Denormalised Loss: {denormalised_loss}")

            # Append denormalized loss to history
```

```
            self.loss_history.append(denormalised_loss)

        return self.loss_history  # Return loss history for analysis




    '''----------------TRAINING FUNCTION WITH MOMENTUM AND BOLD DRIVER--------
--------'''
    def train_with_bold_driver_momentum(self, x, y):
        self.loss_history = []  # Store loss for analysis
        max_learning_rate = 0.5
        min_learning_rate = 0.01

        learning_rate = self.learning_rate

        # Initialize bold driver parameters
        increase_factor = 1.1
        decrease_factor = 0.5
        prev_loss = float('inf')

        for epoch in range(self.epochs):  # Loop through epochs
            total_loss = 0  # Track total loss for this epoch

            for i in range(len(x)):  # Iterate over each training sample
                predictors = x[i].reshape(1, -1)  # Ensure correct shape (row
vector)
                target = y[i].reshape(1, -1)  # Ensure correct shape (row
vector)

                # Forward pass
                output, output_sum, hidden_activation, hidden_sum =
self.forward(predictors)

                # Compute loss and accumulate it
                loss = self.absolute_error(target, output).sum()

                total_loss += loss

                # Backward pass (compute gradients)
                output_delta, hidden_delta = self.backward(target,
output,output_sum, hidden_sum)

                # Update weights and biases
                self.update_weights_with_momentum(predictors, output_delta,
hidden_delta, hidden_activation)

            # Store and display average loss for the epoch
```

```
            avg_loss = total_loss / len(x)
            # **Apply Bold Driver every 50 epochs**
            if epoch % 250 == 0:
                if avg_loss < prev_loss:  # If loss decreased
                    learning_rate = min(learning_rate * increase_factor,
max_learning_rate)  # Increase learning rate
                else:  # If loss increased
                    learning_rate = max(learning_rate * decrease_factor,
min_learning_rate)  # Decrease learning rate

                # Update self.learning_rate after modifying it
                self.learning_rate = learning_rate

            # Update the previous loss for the next iteration
            prev_loss = avg_loss

            # Denormalize average loss
            denormalised_loss = denormalise(avg_loss, y_min, y_max)
            print(f"Epoch: {epoch+1}, Denormalised Loss: {denormalised_loss}")

            # Append denormalized loss to history
            self.loss_history.append(denormalised_loss)

        return self.loss_history  # Return loss history for analysis




    '''---------------TRAINING WITH WEIGHT DECAY----------------'''
    def train_weight_decay(self, x, y):
        self.loss_history = []  # Store loss for analysis

        for epoch in range(self.epochs):  # Loop through epochs
            total_loss = 0  # Track total loss for this epoch


            for i in range(len(x)):  # Iterate over each training sample
                predictors = x[i].reshape(1, -1)  # Ensure correct shape (row
vector)
                target = y[i].reshape(1, -1)  # Ensure correct shape (row
vector)

                # Forward pass
                output, output_sum, hidden_activation, hidden_sum =
self.forward(predictors)

                # Compute loss and accumulate it
                loss = self.absolute_error(target, output).sum()
```

```
                total_loss += loss

                # Backward pass (compute gradients)
                output_delta, hidden_delta =
self.backward_weight_decay(epoch,target, output,output_sum, hidden_sum)

                # Update weights and biases
                self.update_weights(predictors, output_delta, hidden_delta,
hidden_activation)

            # Store and display average loss for the epoch
            avg_loss = total_loss / len(x)

            # Denormalize average loss
            denormalised_loss = denormalise(avg_loss, y_min, y_max)
            print(f"Epoch: {epoch+1}, Denormalised Loss: {denormalised_loss}")

            # Append denormalized loss to history
            self.loss_history.append(denormalised_loss)

        return self.loss_history  # Return loss history for analysis


    '''-------------TRAINING WITH ALL IMPROVEMENTS-----------------'''

    def train_with_bold_driver_momentum_weightdecay(self, x, y):
        self.loss_history = []  # Store loss for analysis
        max_learning_rate = 0.5
        min_learning_rate = 0.01

        learning_rate = self.learning_rate

        # Initialize bold driver parameters
        increase_factor = 1.1
        decrease_factor = 0.5
        prev_loss = float('inf')

        for epoch in range(self.epochs):  # Loop through epochs
            total_loss = 0  # Track total loss for this epoch

            for i in range(len(x)):  # Iterate over each training sample
                predictors = x[i].reshape(1, -1)  # Ensure correct shape (row
vector)
                target = y[i].reshape(1, -1)  # Ensure correct shape (row
vector)

                # Forward pass
```

```
                output, output_sum, hidden_activation, hidden_sum =
self.forward(predictors)

                # Compute loss and accumulate it
                loss = self.absolute_error(target, output).sum()

                total_loss += loss

                # Backward pass (compute gradients)
                output_delta, hidden_delta =
self.backward_weight_decay(epoch,target, output,output_sum, hidden_sum)

                # Update weights and biases
                self.update_weights_with_momentum(predictors, output_delta,
hidden_delta, hidden_activation)

            # Store and display average loss for the epoch
            avg_loss = total_loss / len(x)
            # **Apply Bold Driver every 50 epochs**
            if epoch % 250 == 0:
                if avg_loss < prev_loss:  # If loss decreased
                    learning_rate = min(learning_rate * increase_factor,
max_learning_rate)  # Increase learning rate
                else:  # If loss increased
                    learning_rate = max(learning_rate * decrease_factor,
min_learning_rate)  # Decrease learning rate

                # Update self.learning_rate after modifying it
                self.learning_rate = learning_rate

            # Update the previous loss for the next iteration
            prev_loss = avg_loss

            # Denormalize average loss
            denormalised_loss = denormalise(avg_loss, y_min, y_max)
            print(f"Epoch: {epoch+1}, Denormalised Loss: {denormalised_loss}")

            # Append denormalized loss to history
            self.loss_history.append(denormalised_loss)

        return self.loss_history  # Return loss history for analysis



    '''--------------BATCH LEARNING WITH IMPROVEMENTS------------------'''

    def train_with_batch_and_improvements(self, x, y, batch_size=32):
        self.loss_history = []  # Store loss for analysis
```

```
        # Learning rate parameters
        max_learning_rate = 0.5
        min_learning_rate = 0.01
        learning_rate = self.learning_rate

        # Bold driver parameters
        increase_factor = 1.1
        decrease_factor = 0.5
        prev_loss = float('inf')

        num_samples = len(x)

        for epoch in range(self.epochs):  # Loop through epochs
            total_loss = 0  # Track total loss for this epoch

            # Shuffle dataset indices
            indices = np.arange(num_samples)
            np.random.shuffle(indices)

            for batch_start in range(0, num_samples, batch_size):  # Iterate
over batches
                batch_indices = indices[batch_start:batch_start + batch_size]
                batch_x = x[batch_indices]
                batch_y = y[batch_indices]

                # Forward pass
                output, output_sum, hidden_activation, hidden_sum =
self.forward(batch_x)

                # Compute loss and accumulate it
                loss = self.absolute_error(batch_y, output).sum()
                total_loss += loss

                # Backward pass (compute gradients)
                output_delta, hidden_delta = self.backward_weight_decay(epoch,
batch_y, output, output_sum, hidden_sum)

                # Update weights and biases using batch updates
                self.update_weights_batch(batch_x, output_delta, hidden_delta,
hidden_activation)

            # Compute and store average loss for the epoch
            avg_loss = total_loss / (num_samples // batch_size)

            # **Apply Bold Driver every 250 epochs**
            if epoch % 250 == 0:
                if avg_loss < prev_loss:  # If loss decreased
```

```
                    learning_rate = min(learning_rate * increase_factor,
max_learning_rate)  # Increase learning rate
                else:  # If loss increased
                    learning_rate = max(learning_rate * decrease_factor,
min_learning_rate)  # Decrease learning rate

                # Update learning rate
                self.learning_rate = learning_rate

            # Update the previous loss for the next iteration
            prev_loss = avg_loss

            # Denormalize average loss
            denormalised_loss = denormalise(avg_loss, y_min, y_max)
            print(f"Epoch: {epoch+1}, Denormalised Loss: {denormalised_loss}")

            # Append denormalized loss to history
            self.loss_history.append(denormalised_loss)

        return self.loss_history  # Return loss history for analysis



    def get_weights(self):
        return self.weights_input_hidden, self.weights_hidden_output

    def get_loss_history(self):
        return self.loss_history

#Regression graph
def plot_regression_results(y_true, y_pred):

    residuals = y_true - y_pred  # Compute residuals

    # Compute Pearson correlation coefficient
    correlation, _ = pearsonr(y_true.flatten(), y_pred.flatten())

    plt.figure(figsize=(8, 6))

    # Scatter plot for actual vs predicted values
    plt.scatter(y_true, y_pred, color='blue', alpha=0.6, label="Predicted vs
Actual")

    # Reference line (Perfect Fit y = x)
    min_val = min(y_true.min(), y_pred.min())
    max_val = max(y_true.max(), y_pred.max())
    plt.plot([min_val, max_val], [min_val, max_val], color='red',
linestyle='dashed', label="Perfect Fit (y=x)")
```

```
    # Display correlation coefficient
    plt.text(min(y_pred), max(residuals), f"Pearson r = {correlation:.3f}",
fontsize=12, color='black')

    plt.xlabel("Actual Values")
    plt.ylabel("Predicted Values")
    plt.title("Linear Regression: Predicted vs. Actual")
    plt.legend()
    plt.grid(True)
    plt.show()




# Load the cleaned data
df_cleaned = pd.read_csv('Cleaned_Data.txt', sep='\t')

# Drop the Date column and unnecessary columns
drop_cols = ["Date", "Arkengarthdale", "East Cowton", "Malham Tarn",
"Snaizeholme", "Crakehill", "Skip Bridge", "Westwick"]
df_cleaned = df_cleaned.drop(columns=[col for col in drop_cols if col in
df_cleaned.columns])

# Drop rows with NaN values
df_cleaned = df_cleaned.drop(df.index[-1]).reset_index(drop=True)
df_cleaned = df_cleaned.drop(df.index[0]).reset_index(drop=True)

# Check to see if there are any NaN values
#print(df_cleaned.isnull().any().sum())

# Split the data into predictors and target variable
x = df_cleaned.drop(columns=["Skelton Tomorrow"])
y = df_cleaned["Skelton Tomorrow"]

# Normalize features (Standardization: mean = 0, std = 1)
#x = (x - x.mean()) / X.std()
#y = (y - y.mean()) / y.std()

x_min = x.min()
x_max = x.max()
y_min = y.min()
y_max = y.max()


# Min-Max Standardization
x = 0.8*(x - x_min) / (x_max - x_min)+0.1
y = 0.8*(y - y_min) / (y_max - y_min)+0.1

#denormalise error data
```

MY CODE - F325508

```
def denormalise(average_loss, y_min, y_max):
    return (average_loss) * (y_max - y_min)+ y_min




# Convert to NumPy arrays
x = x.to_numpy()
y = pd.Series(y).to_numpy().reshape(-1, 1)  # Ensure y is a column vector


# Split data into training (50%), testing (25%), validation (25%)

train_size = int(0.5 * len(x))   # 50% Training
test_size = int(0.25 * len(x))   # 25% Test
val_size = len(x) - (train_size + test_size)  # 25% Validation

x_train, x_test, x_val = np.split(x, [train_size, train_size + test_size])
y_train, y_test, y_val = np.split(y, [train_size, train_size + test_size])



# Perform backpropagation
learning_rate = 0.01
epochs = 1000 # Increased epochs for better training
# Initialize the backpropagation model
model = MLP(x_train, y_train,x_train.shape[1],y_train.shape[1],6,
learning_rate, epochs)

# Train the model
output = model.train_with_bold_driver_momentum_weightdecay(x, y)


# Get the weights and loss history
weights = model.get_weights()
loss_history = model.get_loss_history()



# Plot the loss curve
plt.figure(figsize=(8, 6))
plt.plot(loss_history)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss vs Epoch')
plt.grid(True)
plt.show()
```

```
# Plot the predicted vs actual values
y_pred = model.forward(x_test)[0]




# Denormalize the data
# Calculate min and max for y using training data

y_mean = df_cleaned["Skelton Tomorrow"].mean()
y_std = df_cleaned["Skelton Tomorrow"].std()

# Denormalize y_pred and y_test using training data's min and max
y_pred = (y_pred - 0.1) / 0.8 * (y_max - y_min) + y_min
y_test = (y_test - 0.1) / 0.8 * (y_max - y_min) + y_min
y_train = (y_train - 0.1) / 0.8 * (y_max - y_min) + y_min
y_val = (y_val - 0.1) / 0.8 * (y_max - y_min) + y_min

# Denormalize predictions and actual values before plotting
#y_pred = (y_pred * y_std) + y_mean
#y_test = (y_test.flatten() * y_std) + y_mean


# Plot Predicted vs Actual
plt.figure(figsize=(8, 6))
plt.scatter(range(len(y_test)), y_test, label='Actual', alpha=0.7,
color='blue')
plt.scatter(range(len(y_pred)), y_pred, label='Predicted', alpha=0.7,
color='orange')
plt.xlabel('Sample')
plt.ylabel('Skelton Tomorrow')
plt.title('Predicted vs Actual Values (Scatter Plot)')
plt.legend()
plt.grid(True)
plt.show()

plot_regression_results(y_test, y_pred)
```

This is the cleaning file

```
import numpy as np
```

MY CODE - F325508

```
import pandas as pd


#Function to remove the outliers
def remove_outliers(values, mean_value, standard_deviation):

    # Create a copy of input values to prevent modification of original data
    cleaned_values = values.copy()

    # Convert to numpy array if not already
    cleaned_values = np.array(cleaned_values, dtype=float)

    # Relace -999 or 'a' with NaN
    cleaned_values[(cleaned_values == -999) | (cleaned_values == "a")] =
np.nan

    # Replace values beyond 4 standard deviations from the mean with NaN
    cleaned_values[(cleaned_values < (mean_value - 3 * standard_deviation)) |
                   (cleaned_values > (mean_value + 3 * standard_deviation))] =
np.nan

    # Convert to pandas Series for further analysis
    cleaned_values = pd.Series(cleaned_values)

    # Calculate Interquartile Range (IQR) for further outlier detection
    Q1 = cleaned_values.quantile(0.25)
    Q3 = cleaned_values.quantile(0.75)
    IQR = Q3 - Q1

    # Define the lower and upper bounds for outliers
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Filter out values outside IQR bounds and replace with NaN
    cleaned_values[(cleaned_values <= lower_bound) | (cleaned_values >=
upper_bound)] = np.nan

    return cleaned_values
```