**1. Data pre-processing (including cleansing, data splitting, identifying predictors) – 15%**

IDENTIFYING PREDICTORS

To identify predictors, I used each value as provided in the original data set, lagged by one day.

If I were to do it again, I would definitely look into CORRELATION for each daily flow value, especially as they are not equidistant from Skelton, and therefore would not necessarily have the same amount of lag time to affect Skelton’s daily flow, or even the same effect in after correct lagging, which could have been worked around by using weighted values for the inputs themselves. For example, MAP is much closer to Skelton than MAP, yet I have lagged both by the same amount of time.

Similarly, by taking a look at the map, we can see that is much in the watershed for a tributary that has its confluence with the Ouse further . However, for rainfall I believe it would have been more beneficial to produce a weighted moving average of all MAP locations’ rainfall, lagged by the appropriate amount of time for each input. For example, VALUES FOR RAINFALL.

CLEANSING AND SPLITTING

To begin with, I copied all of the data from the original Excel file into a .csv file, so that it was easier to read in a comma-delimited format. I processed the data set within my java program, within a separate class called dataPreprocessing.java. I processed it by first deleting the date from each row of inputs (deleteDates()), then casting all of the values into a two-dimensional array of doubles (castingToDouble()).

To cleanse the data, I then eliminated the non-numerical, negative and outlier values by deleting the row in which they were contained (eliminateOutliers()). My criteria for a value being an outlier was for it to be more than 2 standard deviations away from the mean of its other instances in the data.

With the outliers erased, I moved the Skelton daily flow value to the end of each row (repositionOutputToEnd()), so that it would not interfere with the indexing of the inputs. Keeping the inputs and outputs separate proved very helpful in navigating each row of data.

The next step was to move the next day’s Skelton daily flow to the output of each row, otherwise I would’ve been predicting values for the day it was, not the next day. I did this by taking the output value for each row and moving it to the previous day’s output value index.

For example (as the data is stored as a 2D array of doubles – referred to as DATA), if the output value for a row rested at DATA[n][m], that value would be replaced by DATA[n+1][m] (outputRepositionedFromNextDayArray()).

I needed to avoid issues from the data being trained in the order originally given, such as seasonal affections like training with more values from winter than summer leading to a model that was not able to adequately generalise for all seasons, for example. I did this by shuffling the DATA array randomly…… (shuffleArray()).

Moving on, I standardised all of the input and output values in the range [0.1, 0.9]. I chose this range because it allows the network to predict values outside of those it has been trained with. As mentioned in the lectures, if the range is [0, 1] in training at the highest observed value is 16, the system will not be able to cleanly predict a value for anything higher than this. For example, a value 19 will have a standardised value above one, which will cause problems such as CHECK LECTURE when destandardised. I had an issue with this, as I needed the minimum and maximum values to be returned so that I could destandardise the outputs later. To get around this, I changed the return type from a primitive type to a class which stored the minimum and maximum values for the outputs, as well as the entire data set (inputs and outputs) standardised. I did this by creating a class (standardisedPackager) within dataPreprocessing.java which had the standardised inputs and outputs as well as the minimum and maximum values for each value in the data set.

Next, I split the randomly shuffled data into a training, validation and testing set. As recommended by the lectures, I initially split this 60/20/20, but my code allows for any chosen distribution between the sets. I was able to do this by DATA SPLITTER JAVA RETURN CLASS

This concludes the pre-processing of the data, which was all done within my java program, called from the main method of backpropagationMain.java.

**2. Implementation of the MLP algorithm (including modifications / improvements) – 35%**

**The language used (and why you chose it); What libraries you have used.**

I used java, as it allowed me to use OOP to break the program into functional sections, which made debugging it much easier, as I knew where an issue would be likely to stem from, and most importantly, what else it could have an effect on. I used 3 classes, backpropagationMain.java, dataPreprocessing.java, and fileOperations.java. By using these classes, I was able to break the coursework into 3 major sections: the main algorithm, where training and testing was done (backpropagationMain.java), pre-processing of data (dataPreprocessing.java), and reading/writing files using data (e.g. amount of epochs, amount of hidden layers, learning parameter, mean squared error, etc.) so that I could procure graphs from them (and initially read the file containing all raw data - fileOperations.java).

One major advantage of using OOP in Java is that it allowed me to return multiple variables from a function by making use of Java’s ability to return classes. I did this by creating a class with the desired attributes and creating a function returning said class. For example, I was able to return an instance of the class trainingResults every time I ran training of a data set (backpropTraining()), returning the weights and biases in addition to the number of hidden nodes within the hidden layer. Then, using these attributes, I was easily able to perform validation and testing, which would have been much harder without having the aforementioned values readily available.

Another example is my function testing(), which takes as one of its inputs the aforementioned outputted class trainingResults, and returns its own instance of a class testingResults, which has mean squared error, DESTANDARDISED MODELLED OUTPUTS, and DESTANDARDISED OBSERVED OUTPUTS as attributes. By using a class as the return argument, testing() provides multiple measures I can use to draw graphs and evaluate the MLP with.

The other 2 examples of returning a class from a function are expanded upon in the pre-processing section.

Libraries used:

FileNotFoundException - Signals that an attempt to open the file denoted by a specified pathname has failed, which allows me to see where I have made a mistake, instead of the code simply crashing.

Random – generates random values, allowing me to start from many points in the weight space to find the global minima of the error function.

Scanner –

File - allows me to read from the excel file

ArrayList – allows me to make a new

List – data structure …..

**How you implemented it – e.g. OO approach with an MLP class and what methods it has, how the data are stored/structured, etc.**

**The MLP algorithm – what additions did you make – e.g. momentum, annealing, bold driver. Did you try different transfer functions? Alternative training algorithms – e.g. conjugate gradients?**

**Are there limits on your code (e.g. have things been hard-coded or can it create any MLP with any number of inputs, hidden layers, outputs, etc).**

Can have any amount of hidden layer nodes, but only 1 hidden layer and 1 output

**3. Training and network selection – 20%**

**4. Evaluation of final model (including comparisons between different modifications to the algorithm) – 20%**

GRAPHS AND TABLES

**5. Comparison with another data driven model – 10%;**

**Program listing (i.e. the code you have written)**

Avoid hard-coding things. In other words, don’t write the program for the given data set. It should be easily modifiable for other data sets, different numbers of inputs, different numbers of hidden nodes, etc.

I need to be able to read (with my eyes!) your code. So please submit it as a listing embedded in your report (e.g. as an appendix), or a separate text file or separate pdf (you can submit a zip file with all these in if you wish), so I can actually look at it.

I WILL NOT be running the code - so do not submit it as a file created from your IDE (as I may not have the same IDE you have so might not be able to open it). I simply want to see your program listing with the backpropagation algorithm highlighted (along with any improvements) - to see that it has been implemented correctly. The way I will see that your program works – is seeing all the ANNs you produce and evaluated.

This means I don't care about things like the user interface, how your program stores results, etc. You are using your program to evaluate how good an MLP is at solving problems - so the focus is the MLPs you produce. I don't need a user guide submitting or a discussion of HCI etc.

Make sure you highlight (e.g. with lots of comments) the actual backpropagation algorithm (and other enhancements) in your code so I can find it easily when marking. I need to see the algorithm(s) in your code to mark it (them).

I will also be looking at the code to see that is well structured, well commented, and decent variable names have been used.