Inthuch Therdchanakul

Loughborough University

ANN Implementation

# Introduction

This report discusses the implementation of multi-layer perceptron(MLP) in Python programming language. The MLP model attempts to predict the Index flood based on the predictors provided in the dataset.

# Data Pre-Processing

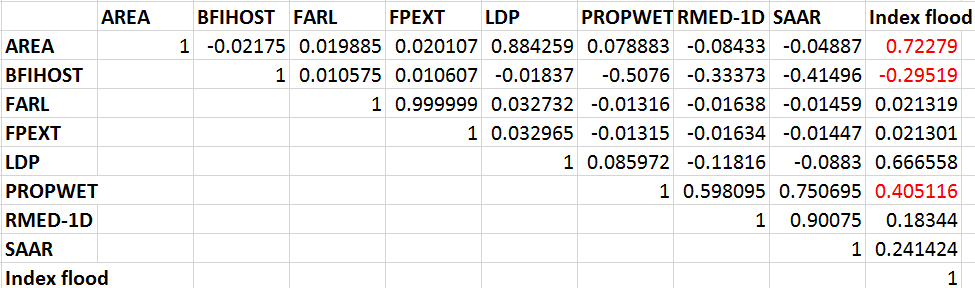


Figure 1: Correlation matrix computed in Microsoft Excel

In this section, the dataset is examined. The tasks performed include removing noisy data, handling missing values, removing outliers, standardizing values, and splitting data.

## Data Cleansing

The first step of data cleansing process is to remove noisy data. Removing noisy data reduces dimensionality which usually results in improvement in prediction accuracy and reduction in overall processing time. Since this dataset is unfamiliar to the author, it is not applicable to perform predictors selection based on deep understanding of the domain. The chosen method for predictors selection is to select the predictors based on ranking of coefficients, i.e. the predictors correlate well with the predictand. The correlation matrix in Figure 1 was computed by using CORREL in Microsoft Excel. The chosen predictors are highlighted in red (AREA, BFIHOST, and PROPWET). AREA is chosen because it is highly correlated with Index flood. However, LDP is left out despite being highly correlated with the predictand because it has lower coefficient ranking than AREA and highly correlated with AREA at the same time. This indicates that LDP can slow down the learning process of the MLP model (LeCun, et al., 2012) and it is therefore left out to reduce the size of the network and overall processing time. BFIHOST and PROPWET are kept as they follows AREA in the ranking of coefficients. The rest of the predictors are left out due to having correlation value close to zero.

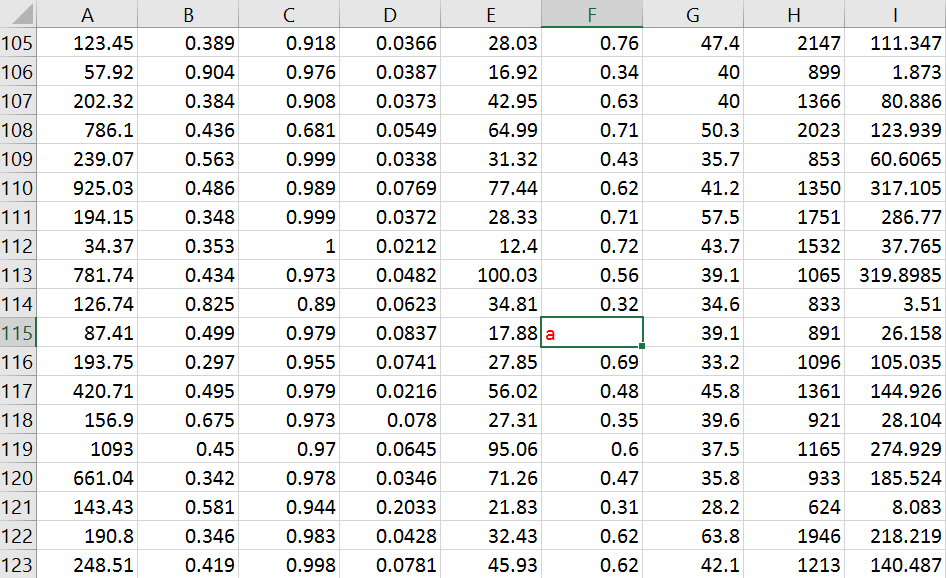
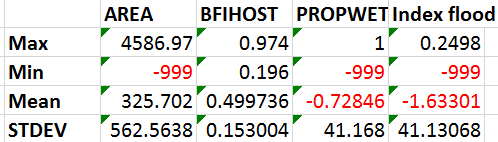


Figure 2: General statistic of the data set and an example of invalid data

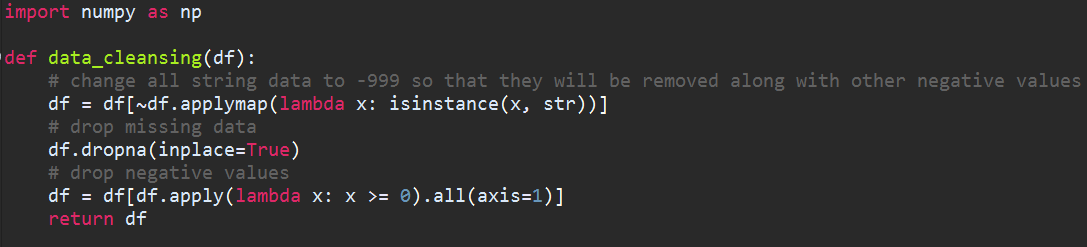


Figure 3: Data cleansing code in python

The statistic in Figure 2 shows that there are invalid values among in the columns which greatly affect their statistic. For example, PROPWET and Index flood column have negative mean values because these columns contain -999 values which can be considered invalid. In Figure 2, the cell highlighted in red shows that there is at least one invalid data in the form of string (a letter a in this case). To solve this problem, the python code in Figure 3 is used to convert all strings into negative values, drop the rows that contain missing values, and delete the rows with negative values.

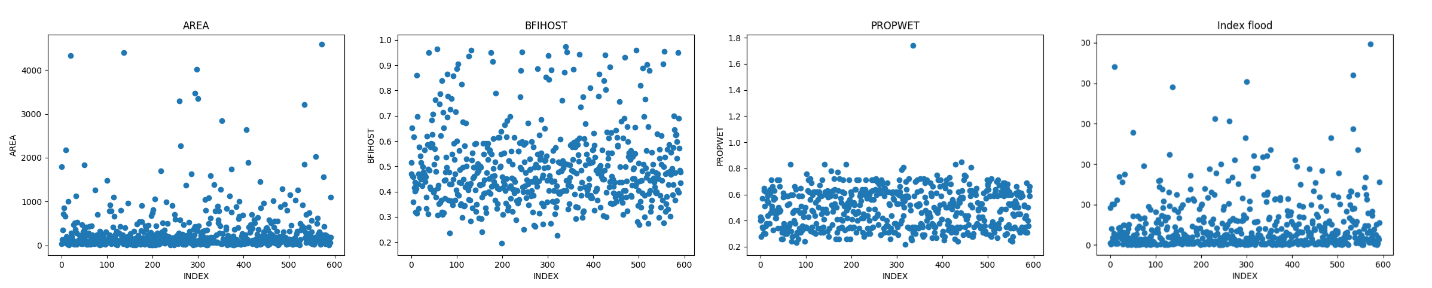


Figure 4: The plot of predictor and predictands before outliers are removed



Figure 5: The plot of predictors and predictands after outliers are removed

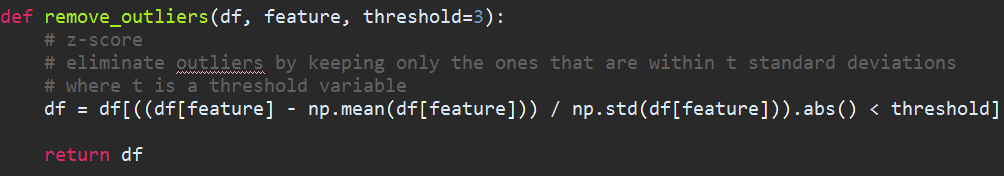


Figure 6: Python function used to remove outliers from the data set df

Outlier is a value that lies an abnormal distance from other values in the dataset (NIST SEMATECH, no date). The scatter plots in Figure 4 show that there are several outliers among the dataset. In this situation, the z score is used to detect and delete outliers from the dataset. The z score of a raw value x is:

(1)

Where µ is the mean of the population and *σ* is the standard deviation of the population. The z score of each value in the dataset is checked against a threshold t. The default value of t is usually 3. If the z score of a value exceed the threshold then it will be removed from the data set. As shown in Figure 5, the outliers are removed after applying algorithm in Figure 6 which remove outliers from the data set based on the z score of each raw value.

## Standardization

The raw dataset is comprised of attributes with varying scales. The learning process is usually faster if the average of each input to the training set is close to zero (LeCun, et al., 2012).In this case, the values in the dataset are standardized so that they have the range of [0.1, 0.9]. The standardized value Si of raw value Ri is:

(2)

Where Max is the maximum value of the population and Min is the minimum value of the population.

## Data Splitting

Before splitting the dataset, the data is shuffled so that the MLP is trained on the data from various part of the dataset. The data is then split into subsets as follows: 60% training set, 20% validation set, and 20% test set.

# Implementation of the MLP Algorithm



Figure 7: MLP structure

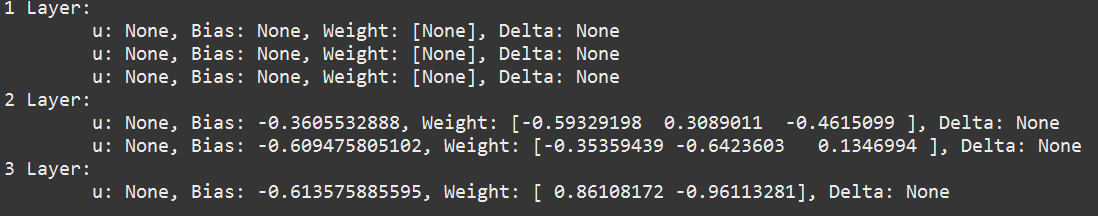


Figure 8: Console printing of MLP structure

The MLP structure illustrated in Figure 7 is implemented in the file MLP.py. In this case, we always assume that there is only one hidden layer and only one output perceptron in the output layer. The print out of the structure of the MLP is as shown in Figure 8. The MLP is an object containing layer objects (input layer, hidden layer, and output layer). A layer object contains node objects and each node object contain information such as output, weight (and bias weight), and delta value. The weights of the nodes are initialized to small random values at the start. The values are ranging from [] where n is the number of input to the MLP. The value 0.1 is also assigned to the learning rate before the training process starts.

## Backpropagation Algorithm – Standard

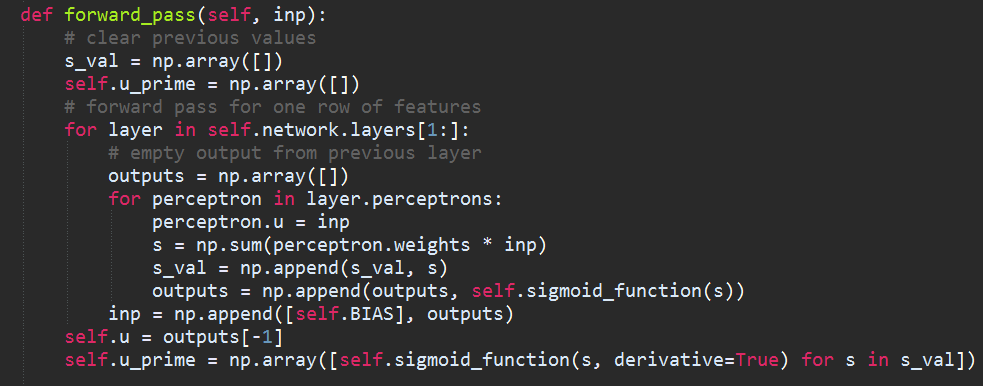


Figure 9: Python code extract of forward propagation function

Before the MLP can start making predictions, it must first be train on the training data. In this implementation, the backpropagation algorithm consists of three main stages: forward propagation, backward propagation, and weight update. Training involves feeding data through the MLP and continually update the weight based on how close the MLP output is in comparison to the expected output.

The first stage is forward propagation. As shown in Figure 9, each training data is fed through the input layer where it will pass forward through the network computing weighted sums (si) and sigmoid activation function (ui). The equation of weighted sums (3) and activation function (4) are shown below:

(3)

(4)



Figure 10: Python code for backward pass and weight update

Once each perceptron has been visited and the output calculated, the algorithm make a backward pass through the network calculating the error value (delta) of all nodes using the following formula:

(5)

Where is the weight between neuron i and m, is the delta value of neuron i and is the correct output. After calculating the delta value for each neuron, each weight is updated using the equation:

(6)

As shown in Figure10, the python code performs the steps mentioned earlier. The steps are repeated for each sample in the training set for n amount of times with n being the number of training cycles (epochs). The training stop when the error calculated on the validation set starts to increase.

## Backpropagation Algorithm – Momentum

One of the improvement made to the algorithm is momentum. The new weight is calculated using the term:

(7)

Where is the previous weight change and 0.9 is the momentum.

## Backpropagation Algorithm – Simulated Annealing

Another modification made to the algorithm is simulated annealing. This technique update the learning rate using the formula:

(8)

The python code extract in Figure 10 shows the used of equation (8) to update the learning rate.

# Training and Network Selection

## Training

The network used in training have hidden units ranges from [2, 10]. In this case, the root mean squared error is used to measure the performance of the network. The RMSE measures the accuracy of the network prediction against the observed data. The values of RMSE are positive with no upper limit. The closer the error is to zero, the more accurate the predictions are, indicating a more precise model. The equation (9) below shows the calculation of RMSE:

(9)

Where is the modelled value, Qi is the observed value, and n is the sample size.

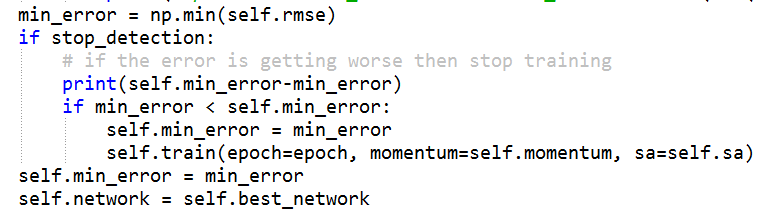


Figure 11: stop criteria of the MLP model

The training automatically stops when an increase in RMSE of the validation is observed. As illustrated in Figure 11, the observation is done every 1000 epoch to make sure that the error continues to increase. The training algorithm then return the network object saved when the minimum RMSE is observed. Based on the value of RMSE, the best network configuration was chosen to be tested on testing set.

## Network Selection

|  |  |
| --- | --- |
| Standard Backpropagation | |
| Hidden Units | **RMSE** |
| 2 | 32.7951418 |
| 3 | 30.9063654 |
| 4 | 30.7605183 |
| 5 | 30.9756278 |
| 6 | 30.9218789 |
| 7 | 30.9752338 |
| 8 | 30.7544307 |
| 9 | 30.8951517 |
| 10 | 30.6413938 |

Table 1: The table of network configurations and associated RMSE for standard backpropagation

|  |  |
| --- | --- |
| Backpropagation with momentum | |
| Hidden Units | **RMSE** |
| 2 | 30.86871 |
| 3 | 29.25023 |
| 4 | 30.14486 |
| 5 | 30.85834 |
| 6 | 29.53251 |
| 7 | 29.6940 |
| 8 | 30.43257 |
| 9 | 30.01853 |
| 10 | 29.4831 |

Table 2: The table of network configurations and associated RMSE for backpropagation with momentum

|  |  |
| --- | --- |
| Backpropagation with simulated annealing | |
| Hidden Units | **RMSE** |
| 2 | 30.4051 |
| 3 | 28.99233 |
| 4 | 30.45769 |
| 5 | 30.31347 |
| 6 | 30.20457 |
| 7 | 30.20325 |
| 8 | 30.04512 |
| 9 | 29.10592 |
| 10 | 28.51971 |

Table 3: The table of network configurations and associated RMSE for backpropagation with simulated annealing

From Table 1, Table 2, and Table 3, The best network configuration is the network with 10 hidden units trained using error backpropagation algorithm with simulated annealing. The MLP model’s RMSE value is 28.51971. The findings indicate that the performance of the MLP model can still be improved by training it with higher number of hidden units. This also shows that simulated annealing tends to perform better than standard backpropagation and backpropagation with momentum. However, due to time constraint and limited computational power, it is not possible to train the network with higher number of hidden units to see if the performance can be further improved or not. The current best network configuration is chosen as the final model.

# Evaluation of Final Model

As discussed in the previous section, the chosen network configuration is a network with 1 hidden layer, 10 hidden units, and 1 output unit trained using error backpropagation algorithm with simulated annealing.

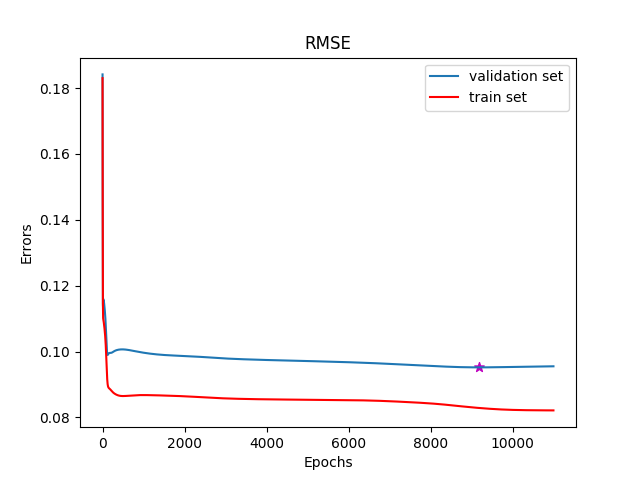


Figure 12: Final model RMSE

As shown in Figure 12, the final model, the network training stopped at 11000 epochs. However, the network that the backpropagation return is the network at 9191 epochs as marked by purple star symbol because this is the point where the error of the validation set starts to increase.

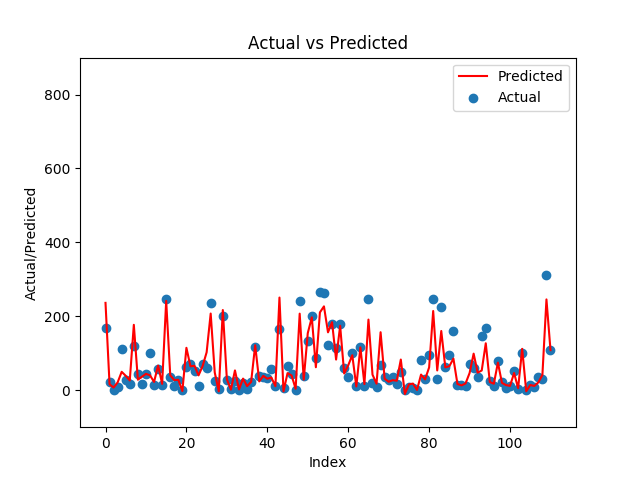


Figure 13: Prediction vs. expected output

The prediction of the testing is de-standardized with the equation:

(10)

The de-standardized final predictions of the model against test set is shown in Figure 13. This indicates that the stopping criteria of the training algorithm shown in Figure 11 managed to stop the learning process before overfitting occurs. The final model has the Root mean squared error of 28.51971 and the Correlation coefficient of 0.923396315718. Overall, the final MLP model is proven to be highly accurate at predicting the Index flood.

# Comparison with Other Data Driven Model

The section compares the MLP model with linear regression model and multi-layer perceptron model from machine learning software WEKA. The data passed into WEKA includes AREA, BFIHOST, PROPWET, and Index flood. Before training, the data are normalized and split into sunsets: 80% training set and 20% testing set.

The result of WEKA’s linear regression model is as follows:

Correlation coefficient 0.8605

Mean absolute error 50.3257

Root mean squared error 83.5505

Relative absolute error 48.2977 %

Root relative squared error 51.166 %

Total Number of Instances 119

The result of WEKA’s multi-layer perceptron model is as follows:

Correlation coefficient 0.8605

Mean absolute error 50.3257

Root mean squared error 83.5505

Relative absolute error 48.2977 %

Root relative squared error 51.166 %

Total Number of Instances 119

The result of MLP model is as follows:

Correlation coefficient 0.923396315718

Root mean squared error 28.51971

Total Number of Instances 111

The RMSE value of the MLP model is 28.51971 which means that the overall performance of MLP model is better than the data driven models from WEKA. It is possible that WEKA performs worse because the lack of customization. The software only offer basic functionality such as normalization, visualization tools, and several learning algorithms. However, it is still difficult to tell why WEKA’s performance is worse than the MLP model because we do not have access to WEKA source code.

# References

LeCun, Y., Bottou, L., Orr, G. B. & Müller, K.-R., 2012. Efficient BackProp. *Neural networks: Tricks of the trade,* pp. 9-48.

NIST SEMATECH, no date. *Enineering Statistics Handbook.* [Online]   
Available at: http://www.itl.nist.gov/div898/handbook/prc/section1/prc16.htm  
[Accessed 22 March 2017].

# Program Listing

## MLP.py

*'''*

*Created on Feb 20, 2017*

**@author:** *Inthuch Therdchanakul*

*'''*

import random

import numpy as np

class Perceptron():

def \_\_init\_\_(self, n\_inputs ):

self.n\_inputs = n\_inputs

self.set\_weights( np.array([random.uniform(-2./n\_inputs,2./n\_inputs) for \_ in range(0,n\_inputs+1)])) # +1 for bias weight

self.delta\_weights = np.zeros(n\_inputs+1)

self.delta = None

self.u = None

def set\_weights(self, weights ):

self.weights = weights

def \_\_str\_\_(self):

return *'u: %s, Bias: %s, Weight: %s, Delta: %s'* % ( str(self.u), str(self.weights[0]),str(self.weights[1:]), str(self.delta) )

class PerceptronLayer():

def \_\_init\_\_(self, n\_perceptrons, n\_inputs):

self.n\_perceptrons = n\_perceptrons

self.perceptrons = np.array([Perceptron( n\_inputs ) for \_ in range(0,self.n\_perceptrons)])

def \_\_str\_\_(self):

return *'Layer:\n\t'*+*'\n\t'*.join([str(perceptron) for perceptron in self.perceptrons])+*''*

class MLP():

def \_\_init\_\_(self, n\_inputs, n\_outputs, n\_perceptrons\_to\_hl, n\_hidden\_layers):

self.n\_inputs = n\_inputs

self.n\_outputs = n\_outputs

self.n\_hidden\_layers = n\_hidden\_layers

self.n\_perceptrons\_to\_hl = n\_perceptrons\_to\_hl

self.create\_network()

self.\_n\_weights = None

# end

def create\_network(self):

if self.n\_hidden\_layers>0:

# create the input layer

self.layers = [PerceptronLayer( self.n\_inputs,self.n\_inputs )]

# create hidden layers

self.layers += [PerceptronLayer( self.n\_perceptrons\_to\_hl,self.n\_inputs ) for \_ in range(0,self.n\_hidden\_layers)]

# hidden-to-output layer

self.layers += [PerceptronLayer( self.n\_outputs,self.n\_perceptrons\_to\_hl )]

else:

# If we don't require hidden layers

self.layers = [PerceptronLayer( self.n\_outputs,self.n\_inputs )]

self.layers = np.asarray(self.layers)

for perceptron in self.layers[0].perceptrons:

perceptron.set\_weights(np.array([None,None]))

def \_\_str\_\_(self):

return *'\n'*.join([str(i+1)+*' '*+str(layer) for i,layer in enumerate(self.layers)])

## Dataset.py

*'''*

*Created on Feb 14, 2017*

**@author:** *Inthuch Therdchanakul*

*'''*

import numpy as np

class datasets():

def \_\_init\_\_(self, df, features, label, max\_label, min\_label):

self.features = features

self.label = np.array(df[label])

self.feature\_names = df.drop(label, axis=1).columns.values

self.label\_name = label

self.max\_label = max\_label

self.min\_label = min\_label

def de\_standardise(self, label):

for i in range(0, len(label)):

label[i] = (((label[i] - 0.1)/(0.8)) \* (self.max\_label - self.min\_label))

+ self.min\_label

return label

## Preprocessing.py

*'''*

*Created on Feb 26, 2017*

**@author:** *Inthuch Therdchanakul*

*'''*

import numpy as np

def data\_cleansing(df):

# change all string data to -999 so that they will be removed along with other negative values

df = df[~df.applymap(lambda x: isinstance(x, str))]

# drop missing data

df.dropna(inplace=True)

# drop negative values

df = df[df.apply(lambda x: x >= 0).all(axis=1)]

return df

def remove\_outliers(df, feature, threshold=3):

# z-score

# eliminate outliers by keeping only the ones that are within t standard deviations

# where t is a threshold variable

df = df[((df[feature] - np.mean(df[feature])) / np.std(df[feature])).abs() < threshold]

return df

def standardise(df):

for col in df.columns.values:

df[col] = 0.8 \* ((df[col] - np.min(df[col]))/(np.max(df[col]) - np.min(df[col]))) + 0.1

return df

## Backpropagation.py

*'''*

*Created on Feb 20, 2017*

**@author:** *Inthuch Therdchanakul*

*'''*

import numpy as np

class BackPropagation():

def \_\_init\_\_(self, train\_set, val\_set, test\_set, network):

self.BIAS = 1

self.learning\_rate = 0.1

self.train\_set = train\_set

self.val\_set = val\_set

self.test\_set = test\_set

self.network = network

self.u = None

self.predictions = np.array([])

self.rmse = np.array([])

self.train\_rmse = np.array([])

self.momentum = False

self.sa = False

self.bold\_drv = False

self.current\_epoch = 0

self.epoch = 0

self.max\_lr = None

self.min\_lr = None

self.min\_error = 1

self.best\_network = network

def forward\_pass(self, inp):

# clear previous values

s\_val = np.array([])

self.u\_prime = np.array([])

# forward pass for one row of features

for layer in self.network.layers[1:]:

# empty output from previous layer

outputs = np.array([])

for perceptron in layer.perceptrons:

perceptron.u = inp

s = np.sum(perceptron.weights \* inp)

s\_val = np.append(s\_val, s)

outputs = np.append(outputs, self.sigmoid\_function(s))

inp = np.append([self.BIAS], outputs)

self.u = outputs[-1]

self.u\_prime = np.array([self.sigmoid\_function(s, derivative=True) for s in s\_val])

def backward\_pass(self, label):

# propagate deltas backward from output layer to input layer

self.network.layers[-1].perceptrons[0].delta = (label - self.u) \* (self.u\_prime[-1])

# update weights

self.network.layers[-1].perceptrons[0] = self.update(self.network.layers[-1].perceptrons[0])

output\_delta = self.network.layers[-1].perceptrons[0].delta

# calculate deltas in hidden layer

for i in range(len(self.network.layers[1].perceptrons),0,-1):

weight = self.network.layers[-1].perceptrons[0].weights[i]

self.network.layers[1].perceptrons[i-1].delta = np.sum(weight \* output\_delta) \* (self.u\_prime[i-1])

self.network.layers[1].perceptrons[i-1] = self.update(self.network.layers[1].perceptrons[i-1])

def update(self, perceptron):

# update every weight linked to the perceptron using deltas

if self.momentum:

# apply momentum

weight\_old = np.array([])

weight\_old = perceptron.weights

perceptron.weights = perceptron.weights + (self.learning\_rate \* perceptron.delta \* perceptron.u) + (0.9 \* perceptron.delta\_weights)

perceptron.delta\_weights = perceptron.weights - weight\_old

elif self.bold\_drv:

print(*"BOLD DRIVER DOES NOT WORK!"*)

elif self.sa:

r = 15000

exp = 10 - ((20\*self.current\_epoch)/r)

self.learning\_rate = self.min\_lr + (self.max\_lr - self.min\_lr) \* (1 - (1/(1 + np.e\*\*exp)))

weight\_old = np.array([])

weight\_old = perceptron.weights

perceptron.weights = perceptron.weights + (self.learning\_rate \* perceptron.delta \* perceptron.u)

else:

perceptron.weights = perceptron.weights + (self.learning\_rate \* perceptron.delta \* perceptron.u)

return perceptron

def train(self, epoch=1000, momentum=False, sa=False, max\_lr=0.9, min\_lr=0.01, stop\_detection=True):

self.momentum = momentum

self.sa = sa

self.epoch = self.epoch + epoch

self.max\_lr = max\_lr

self.min\_lr = min\_lr

for i in range(epoch):

self.current\_epoch = (self.epoch - epoch) + i

for feature, label in zip(self.train\_set.features, self.train\_set.label):

self.forward\_pass(feature)

self.backward\_pass(label)

# calculate training set error

self.predict(self.train\_set.features)

self.train\_rmse = np.append(self.train\_rmse, [self.calculate\_rmse(self.train\_set.label)])

# calculate validation set error

self.predict(self.val\_set.features)

rmse = self.calculate\_rmse(self.val\_set.label)

self.rmse = np.append(self.rmse, [rmse])

if np.min(self.rmse) == rmse:

self.best\_network = self.network

if ((i+1) % 200) == 0:

print(*"epoch: %s\t min\_rmse: %s\t hidden\_units: %s"* % (str(self.current\_epoch+1), str(np.min(self.rmse)), str(self.network.n\_perceptrons\_to\_hl)))

min\_error = np.min(self.rmse)

if stop\_detection:

# if the error is getting worse then stop training

print(self.min\_error-min\_error)

if min\_error < self.min\_error:

self.min\_error = min\_error

self.train(epoch=epoch, momentum=self.momentum, sa=self.sa)

self.min\_error = min\_error

self.network = self.best\_network

# make predictions using trained model

def predict(self, features):

self.predictions = np.array([])

for feature in features:

# clear previous values

s\_val = np.array([])

# forward pass for one row of features

for layer in self.network.layers[1:]:

# empty output from previous layer

output = np.array([])

for perceptron in layer.perceptrons:

perceptron.u = feature

s = np.sum(perceptron.weights \* feature)

s\_val = np.append(s\_val, s)

output = np.append(output, self.sigmoid\_function(s))

feature = np.append([self.BIAS], output)

self.predictions = np.append(self.predictions, output)

# calculate root mean squared error from validation set

def calculate\_rmse(self, data):

#return sqrt(np.mean((self.predictions - data)\*\*2))

return np.sqrt(np.mean((self.predictions-data)\*\*2))

# activation function

def sigmoid\_function(self, s, derivative=False):

if derivative:

return self.sigmoid\_function(s) \* (1 - self.sigmoid\_function(s))

else:

return 1/(1 + np.e\*\*-s)

## ANN\_Training.py

*'''*

*Created on Feb 20, 2017*

**@author:** *Inthuch Therdchanakul*

*'''*

import pandas as pd

import numpy as np

from Data.Datasets import datasets

from Data.Preprocessing import data\_cleansing

from Data.Preprocessing import remove\_outliers

from Data.Preprocessing import standardise

from Learning\_Algorithms.BackPropagation import BackPropagation

from ANN.MLP import MLP

import matplotlib.pyplot as plt

import pickle

from timeit import default\_timer as timer

import os

df = pd.read\_excel(*"Data.xlsx"*)

df = df[[*"AREA"*, *"BFIHOST"*, *"PROPWET"*, *"Index flood"*]]

df = data\_cleansing(df)

df = remove\_outliers(df, *"AREA"*, threshold=2.5)

df = remove\_outliers(df, *"BFIHOST"*)

df = remove\_outliers(df, *"PROPWET"*)

df = remove\_outliers(df, *"Index flood"*, threshold=2.5)

max\_label = np.max(df[*"Index flood"*])

min\_label = np.min(df[*"Index flood"*])

df = standardise(df)

df[*"BIAS"*] = 1

df = df[[*"BIAS"*, *"AREA"*, *"BFIHOST"*, *"PROPWET"*, *"Index flood"*]]

# split data set

df\_train, df\_val, df\_test = np.split(df, [int(.6\*len(df)), int(.8\*len(df))])

features\_train = np.array(df\_train.drop(*"Index flood"*, axis=1))

features\_val = np.array(df\_val.drop(*"Index flood"*, axis=1))

features\_test = np.array(df\_test.drop(*"Index flood"*, axis=1))

train\_set = datasets(df\_train, features\_train, *"Index flood"*, max\_label, min\_label)

val\_set = datasets(df\_val, features\_val, *"Index flood"*, max\_label, min\_label)

test\_set = datasets(df\_test, features\_test, *"Index flood"*, max\_label, min\_label)

y\_observed = test\_set.de\_standardise(test\_set.label)

# hidden unit value [2, 10]

start = timer()

network = MLP(3, 1, 7, 1)

clf = BackPropagation(train\_set, val\_set, test\_set, network)

training\_type = *"momentum"*

n\_hidden\_units = str(clf.network.n\_perceptrons\_to\_hl)

print(*"Trainning Started with %s hidden unit"* % str(n\_hidden\_units))

print(*"Training mode: "*, training\_type)

clf.train(momentum=True)

end = timer()

print(*"Tranining completed in %s with %s hidden unit"* % (str(end-start), str(n\_hidden\_units)))

y\_val = clf.rmse

y\_train = clf.train\_rmse

# make predictions on testing set

clf.predict(test\_set.features)

f1 = plt.figure()

f2 = plt.figure()

x = np.array([idx for idx in range(clf.epoch)])

ax1 = f1.add\_subplot(111)

ax1.set\_title(*"RMSE"*)

ax1.set\_xlabel(*"Epochs"*)

ax1.set\_ylabel(*"Errors"*)

ax1.plot(x, y\_val, label=*"validation set"*)

ax1.plot(x, y\_train, color=*"r"*, label=*"train set"*)

ax1.legend()

ax2 = f2.add\_subplot(111)

x = np.array([idx for idx in range(len(test\_set.label))])

y\_modelled = test\_set.de\_standardise(clf.predictions)

ax2.set\_title(*"Actual vs Predicted"*)

ax2.set\_xlabel(*"Index"*)

ax2.set\_ylabel(*"Actual/Predicted"*)

ax2.plot(x, y\_modelled, color=*"r"*, label=*"Predicted"*)

ax2.scatter(x, y\_observed, label=*"Actual"*)

ax2.set\_ylim(-100, 900)

ax2.legend()

# save the model for analysis later on

path = *"Simulations/"* + training\_type + *"\_"* + n\_hidden\_units + *"\_e"* + str(clf.epoch) + *"/"*

if not os.path.exists(path):

os.makedirs(path)

predictions\_filename = path + training\_type + *"\_"* + n\_hidden\_units + *"\_e"* + str(clf.epoch) + *"\_PREDICTIONS.pickle"*

rmse\_filename = path + training\_type + *"\_"* + n\_hidden\_units + *"\_e"* + str(clf.epoch) + *"\_RMSE.pickle"*

clf\_filename = path + training\_type + *"\_"* + n\_hidden\_units + *"\_e"* + str(clf.epoch) + *"\_MODEL.pickle"*

with open(predictions\_filename, *"wb"*) as f:

pickle.dump(ax2, f, pickle.HIGHEST\_PROTOCOL)

with open(rmse\_filename, *"wb"*) as f:

pickle.dump(ax1, f, pickle.HIGHEST\_PROTOCOL)

with open(clf\_filename, *"wb"*) as f:

pickle.dump(clf, f, pickle.HIGHEST\_PROTOCOL)

plt.show()

## Feature\_Analysis.py

*'''*

*Created on Feb 26, 2017*

**@author:** *Inthuch Therdchanakul*

*'''*

import matplotlib.pyplot as plt

import pandas as pd

import numpy as np

from Data.Preprocessing import data\_cleansing

from Data.Preprocessing import remove\_outliers

df = pd.read\_excel(*"Data.xlsx"*)

df = df[[*"AREA"*, *"BFIHOST"*, *"PROPWET"*, *"Index flood"*]]

df = data\_cleansing(df)

area = np.array(df[*"AREA"*])

bfihost = np.array(df[*"BFIHOST"*])

propwet = np.array(df[*"PROPWET"*])

idx\_flood = np.array(df[*"Index flood"*])

f1 = plt.figure()

f2 = plt.figure()

f3 = plt.figure()

f4 = plt.figure()

x = np.array([idx for idx in range(len(area))])

ax1 = f1.add\_subplot(111)

y = area

ax1.set\_title(*"AREA"*)

ax1.set\_xlabel(*"INDEX"*)

ax1.set\_ylabel(*"AREA"*)

ax1.scatter(x, y)

ax2 = f2.add\_subplot(111)

y = bfihost

ax2.set\_title(*"BFIHOST"*)

ax2.set\_xlabel(*"INDEX"*)

ax2.set\_ylabel(*"BFIHOST"*)

ax2.scatter(x, y)

ax3 = f3.add\_subplot(111)

y = propwet

ax3.set\_title(*"PROPWET"*)

ax3.set\_xlabel(*"INDEX"*)

ax3.set\_ylabel(*"PROPWET"*)

ax3.scatter(x, y)

ax4 = f4.add\_subplot(111)

y = idx\_flood

ax4.set\_title(*"Index flood"*)

ax4.set\_xlabel(*"INDEX"*)

ax4.set\_ylabel(*"Index flood"*)

ax4.scatter(x, y)

plt.show()

df = remove\_outliers(df, *"AREA"*, threshold=2.5)

df = remove\_outliers(df, *"BFIHOST"*)

df = remove\_outliers(df, *"PROPWET"*)

df = remove\_outliers(df, *"Index flood"*, threshold=2.5)

area = np.array(df[*"AREA"*])

bfihost = np.array(df[*"BFIHOST"*])

propwet = np.array(df[*"PROPWET"*])

idx\_flood = np.array(df[*"Index flood"*])

f1 = plt.figure()

f2 = plt.figure()

f3 = plt.figure()

f4 = plt.figure()

x = np.array([idx for idx in range(len(area))])

ax1 = f1.add\_subplot(111)

y = area

ax1.set\_title(*"AREA"*)

ax1.set\_xlabel(*"INDEX"*)

ax1.set\_ylabel(*"AREA"*)

ax1.set\_ylim(0,4000)

ax1.scatter(x, y)

ax2 = f2.add\_subplot(111)

y = bfihost

ax2.set\_title(*"BFIHOST"*)

ax2.set\_xlabel(*"INDEX"*)

ax2.set\_ylabel(*"BFIHOST"*)

ax2.scatter(x, y)

ax3 = f3.add\_subplot(111)

y = propwet

ax3.set\_title(*"PROPWET"*)

ax3.set\_xlabel(*"INDEX"*)

ax3.set\_ylabel(*"PROPWET"*)

ax2.set\_ylim(0,2)

ax3.scatter(x, y)

ax4 = f4.add\_subplot(111)

y = idx\_flood

ax4.set\_title(*"Index flood"*)

ax4.set\_xlabel(*"INDEX"*)

ax4.set\_ylabel(*"Index flood"*)

ax4.set\_ylim(0,1000)

ax4.scatter(x, y)

plt.show()

## model\_analysis.py

*'''*

*Created on Mar 18, 2017*

**@author:** *Inthuch Therdchanakul*

*'''*

import pickle

import matplotlib.pyplot as plt

import pandas as pd

import numpy as np

from Data.Datasets import datasets

from Data.Preprocessing import data\_cleansing

from Data.Preprocessing import remove\_outliers

training\_type = *"annealing"*

n\_hidden\_units = *"10"*

epoch = *"11000"*

df = pd.read\_excel(*"Data.xlsx"*)

df = df[[*"AREA"*, *"BFIHOST"*, *"PROPWET"*, *"Index flood"*]]

df = data\_cleansing(df)

df = remove\_outliers(df, *"AREA"*, threshold=2.5)

df = remove\_outliers(df, *"BFIHOST"*)

df = remove\_outliers(df, *"PROPWET"*)

df = remove\_outliers(df, *"Index flood"*, threshold=2.5)

max\_label = np.max(df[*"Index flood"*])

min\_label = np.min(df[*"Index flood"*])

df[*"BIAS"*] = 1

df = df[[*"BIAS"*, *"AREA"*, *"BFIHOST"*, *"PROPWET"*, *"Index flood"*]]

# split data set

df\_train, df\_val, df\_test = np.split(df, [int(.6\*len(df)), int(.8\*len(df))])

features\_train = np.array(df\_train.drop(*"Index flood"*, axis=1))

features\_val = np.array(df\_val.drop(*"Index flood"*, axis=1))

features\_test = np.array(df\_test.drop(*"Index flood"*, axis=1))

train\_set = datasets(df\_train, features\_train, *"Index flood"*, max\_label, min\_label)

val\_set = datasets(df\_val, features\_val, *"Index flood"*, max\_label, min\_label)

test\_set = datasets(df\_test, features\_test, *"Index flood"*, max\_label, min\_label)

path = *"Simulations/"* + training\_type + *"\_"* + n\_hidden\_units + *"\_e"* + epoch + *"/"*

predictions\_filename = path + training\_type + *"\_"* + n\_hidden\_units + *"\_e"* + epoch + *"\_PREDICTIONS.pickle"*

rmse\_filename = path + training\_type + *"\_"* + n\_hidden\_units + *"\_e"* + epoch + *"\_RMSE.pickle"*

clf\_filename = path + training\_type + *"\_"* + n\_hidden\_units + *"\_e"* + epoch + *"\_MODEL.pickle"*

with open(predictions\_filename, *"rb"*) as f:

predictions = pickle.load(f)

with open(rmse\_filename, *"rb"*) as f:

rmse = pickle.load(f)

with open(clf\_filename, *"rb"*) as f:

clf = pickle.load(f)

predictions.set\_ylim(-100,900)

test\_set = test\_set.label

rmse = clf.calculate\_rmse(test\_set)

correl = np.corrcoef(test\_set, clf.predictions)[0][1]

print(*"Instance: "*, len(test\_set))

print(*"Correlation coefficient: "*, correl)

print(*"RMSE: "*, rmse)

plt.show()