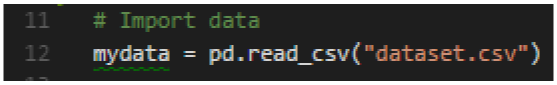
CMP3744M Assessment Item 2

Classification of Diabetic Retinopathy using K-Nearest Neighbour & Artificial Neural Networks with 1 hidden layer **Full code available at: (currently private, request access)** https://github.com/jallkay/datamining/blob/master/data_mining2.py

Section 1

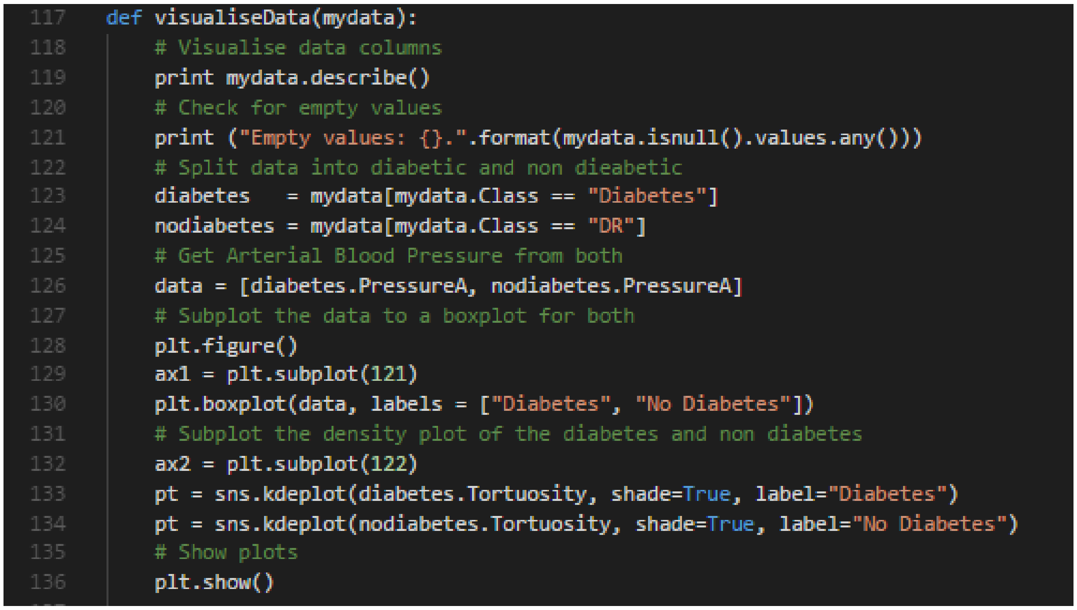
To import the data provided in CSV format, the decision was taken to use the library Pandas (2017). *Figure 1* shows the call to Pandas library (imported as pd) outside of any function or class definition so will be called when the script starts.



```
# Import data
mydata = pd.read_csv("dataset.csv")
```

Figure 1 - Importing data via pandas

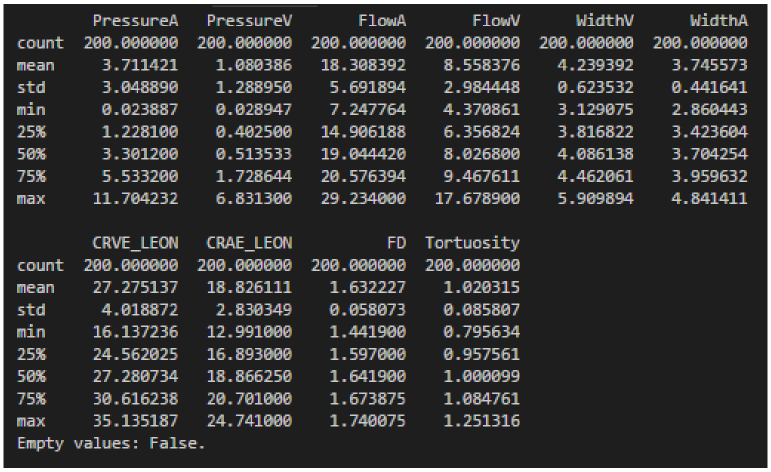
Once this has been completed, all other functions are ready to be called in the program, the first of which details the data visualisation section. One benefit to using Pandas DataFrame's to import the data is it can take advantage of the inbuilt 'describe' function, which will show key features about the dataset. Furthermore, it will make slicing and splitting the data much easier later, when showing the box plots. This functionality is contained in the *visualiseData* function, which is shown below, *Figure 2.*



```
117  def visualiseData(mydata):
118      # Visualise data columns
119      print mydata.describe()
120      # Check for empty values
121      print ("Empty values: {}.".format(mydata.isnull().values.any()))
122      # Split data into diabetic and non dieabetic
123      diabetes   = mydata[mydata.Class == "Diabetes"]
124      nodiabetes = mydata[mydata.Class == "DR"]
125      # Get Arterial Blood Pressure from both
126      data = [diabetes.PressureA, nodiabetes.PressureA]
127      # Subplot the data to a boxplot for both
128      plt.figure()
129      ax1 = plt.subplot(121)
130      plt.boxplot(data, labels = ["Diabetes", "No Diabetes"])
131      # Subplot the density plot of the diabetes and non diabetes
132      ax2 = plt.subplot(122)
133      pt = sns.kdeplot(diabetes.Tortuosity, shade=True, label="Diabetes")
134      pt = sns.kdeplot(nodiabetes.Tortuosity, shade=True, label="No Diabetes")
135      # Show plots
136      plt.show()
```

Figure 2 - visualiseData function

Line 119 displays the summary of the dataset, with *Line 121* checking to see whether there are any missing values in the dataset. The output of these lines is below, *Figure 3.* As identified below there are 10 columns (features) (outside of the identifying 'Class' column), all populated fully with 200 lines of data. There is a large spread of data, with column *CRVE_*LEON containing the largest value, 35.13. Comparatively, the smallest value in the dataset is in the *PressureA* column, with a value of 0.02. Additionally, there are no empty values as identified by *Line 121* on *Figure 2*.



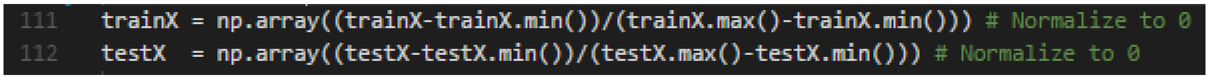
```
       PressureA   PressureV       FlowA       FlowV      WidthV      WidthA
count  200.000000  200.000000  200.000000  200.000000  200.000000  200.000000
mean     3.711421    1.080386   18.308392    8.558376    4.239392    3.745573
std      3.048890    1.288950    5.691894    2.984448    0.623532    0.441641
min      0.023887    0.028947    7.247764    4.370861    3.129075    2.860443
25%      1.228100    0.402500   14.906188    6.356824    3.816822    3.423604
50%      3.301200    0.513533   19.044420    8.026800    4.086138    3.704254
75%      5.533200    1.728644   20.576394    9.467611    4.462061    3.959632
max     11.704232    6.831300   29.234000   17.678900    5.909894    4.841411

       CRVE_LEON   CRAE_LEON          FD  Tortuosity
count  200.000000  200.000000  200.000000  200.000000
mean    27.275137   18.826111    1.632227    1.020315
std      4.018872    2.830349    0.058073    0.085807
min     16.137236   12.991000    1.441900    0.795634
25%     24.562025   16.893000    1.597000    0.957561
50%     27.280734   18.866250    1.641900    1.000099
75%     30.616238   20.701000    1.673875    1.084761
max     35.135187   24.741000    1.740075    1.251316
Empty values: False.
```

Figure 3 - dataset summary

Due to the varying values in each column, pre-processing in the form of data normalisation will be required. This is performed externally to the *visualiseData* function, instead being performed by the *cleanData* function. The full function performs additional tasks depending on the situation which will be explained later in this report, however the normalisation is performed on line 111 and 112, as shown below in *Figure 4*.

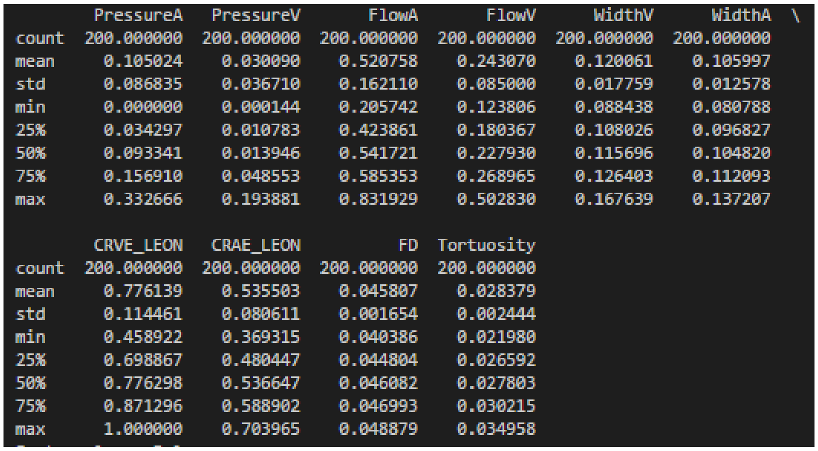
To perform the data normalisation, the values (in this case *trainX* and *testX,* as they have already been split into test and training sets by this point in the function) have the minimum value and maximum value calculated, to create a range between 0 and 1. This is all performed and then stored in a Numpy (2017) Array.



```
111    trainX = np.array((trainX-trainX.min())/(trainX.max()-trainX.min())) # Normalize to 0
112    testX  = np.array((testX-testX.min())/(testX.max()-testX.min())) # Normalize to 0
```

Figure 4 - Data normalisation

Post normalisation, the dataset summary in the same format to *Figure 3* is shown as *Figure 5*. As shown, the previous maximum value in *Figure 3* is now 1, and the previous minimum value is now 0. All other values in between that have been normalised to be in a range 0 and 1. This will help later in the creation and design of the Artificial Neural Network, and create a stable and clean dataset to test against.



```
       PressureA   PressureV       FlowA       FlowV      WidthV      WidthA  \
count  200.000000  200.000000  200.000000  200.000000  200.000000  200.000000
mean     0.105024    0.030090    0.520758    0.243070    0.120061    0.105997
std      0.086835    0.036710    0.162110    0.085000    0.017759    0.012578
min      0.000000    0.000144    0.205742    0.123806    0.088438    0.080788
25%      0.034297    0.010783    0.423861    0.180367    0.108026    0.096827
50%      0.093341    0.013946    0.541721    0.227930    0.115696    0.104820
75%      0.156910    0.048553    0.585353    0.268965    0.126403    0.112093
max      0.332666    0.193881    0.831929    0.502830    0.167639    0.137207

       CRVE_LEON   CRAE_LEON          FD  Tortuosity
count  200.000000  200.000000  200.000000  200.000000
mean     0.776139    0.535503    0.045807    0.028379
std      0.114461    0.080611    0.001654    0.002444
min      0.458922    0.369315    0.040386    0.021980
25%      0.698867    0.480447    0.044804    0.026592
50%      0.776298    0.536647    0.046082    0.027803
75%      0.871296    0.588902    0.046993    0.030215
max      1.000000    0.703965    0.048879    0.034958
```

Figure 5 - Post normalisation dataset summary

Plotting the two graphs required (a boxplot and a density plot) is also contained in the *visualiseData* function as shown in *Figure 2*, with the processing of the data required to plot beginning on *Line 123*. The library *Matplotlib* (2018) was used to plot the graphs, with an overlay library *Seaborn* (2017) to improve the looks of the graphs when shown. The actual plotting is performed between *Lines 128-134*, before outputting on *Line 136.*

The boxplot and density plot is shown below, *Figure 6*.

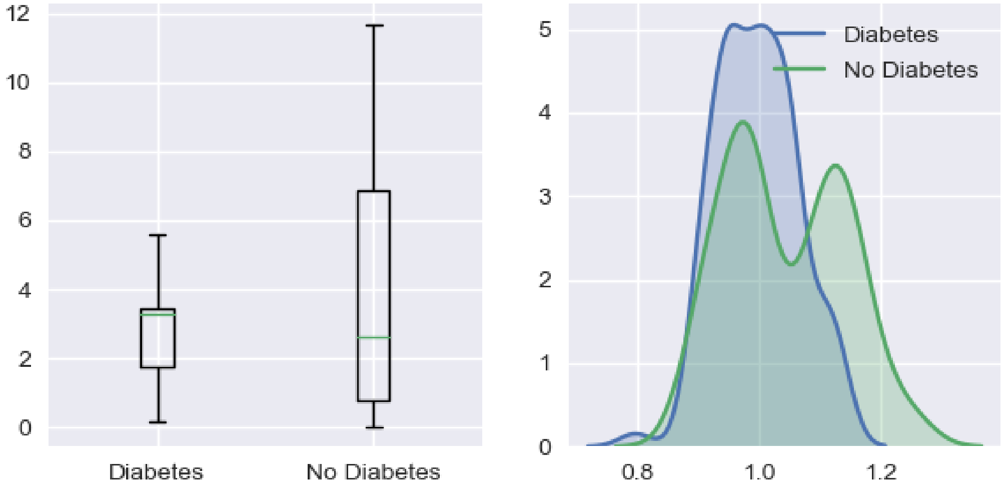


Figure 6 - Boxplot (left) & density plot (right)

Outliers can be evaluated against using these two plots, with the most obvious being in the 'No Diabetes' box plot. The highest value is 11.74 as identified in *Figure 3* under the column *PressureA*. All other figures in that dataset summary of the column show much lower values, so could indicate that the highest value is a lone outlier.

Section 2

The task that the student was given to select the best classifier across 10 models may well produce 90% accuracy, however the description does not detail several items that may affect the overall result. For example, if there were 10 different models ran, was there different parameters entered for each of the model? This may be necessary as if one of the models were to be an Artificial Neural Network, a

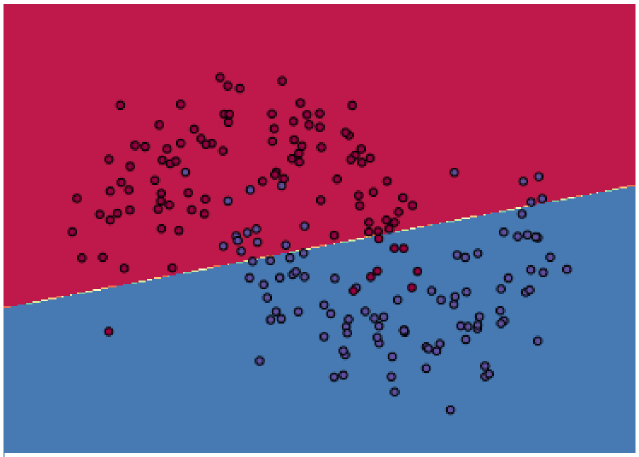


Figure 7 - Underfitted neural network (1 neuron in hidden layer)

different number of neurons could be entered in therefore affecting the result vastly. An approach could be undertaken therefore to work out the best parameters for each model, before comparing them overall. The reason why you may need to do this is because models can be underfitted or overfitted, and finding an optimal solution can take trial and error. For example, *Figure 7* above shows an underfitted neural network with 1 neuron in the hidden layer.

As is displayed above, the line classifying the red and blue points does not cover all the points, so there are many misclassified points which would therefore give a low accuracy percentage. If the student were to have tested a neural network in the 10 models but entered sub-optimal parameters if could give a result that does not indicate the full potential performance of an algorithm and therefore skew the overall results. Conversely, if the student had entered in a set of parameters it would produce an overfitted model to that specific dataset, and would not give an optimal result when using the validation/test set. An overfitted neural network is displayed below in *Figure 8*, which has 50 neurons in its hidden layer. For comparison, *Figure 9* shows an optimal solution for this dataset, having 5 neurons per in the hidden layer.

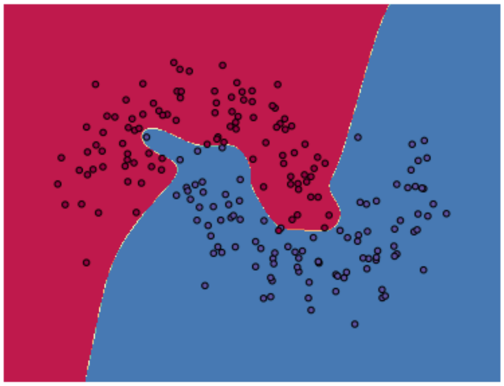


Figure 8 - Overfitted neural network (50 neurons in hidden layer)

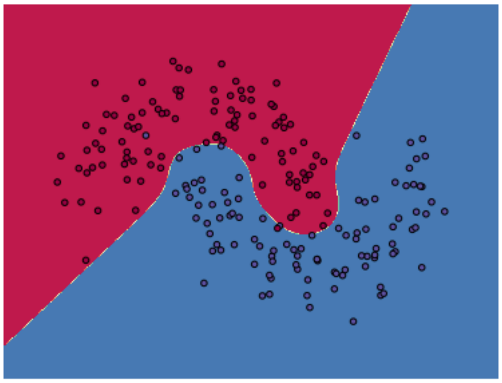


Figure 9 - Optimal fitted neural network (5 neurons in hidden layer)

An additional method which the description does not detail is whether the user randomised the order of the data in each model, or whether they performed any form of cross validation. The reason why you would perform this is in case any data that you have is swaying the overall dataset, and gives a good indication whether the model would work on an unseen test set. One method that can be performed to do this is 10-fold cross validation, a process which removes one tenth of the dataset and sets it as the test set, with the remaining data being the training set. The model is executed and its percentage of accuracy is recorded. This process is then ran another 9 times, each time selecting the test set further down the set of data, so the test set is always different. An overall percentage of accuracy is outputted, giving an indication of accuracy over many different sub-datasets. Examples of 10-Fold CV running will be shown later in this report in *Section 4*.

In conclusion the students model performing 90% accuracy may well be the best one, however they need to perform the additional steps as detailed above, by running additional parameters on each model, and performing cross validation on each model. From there, they will be in a better position to decide which classifier is the best.

Section 3

To create an artificial neural network in Python, the design of the network needs to be first confirmed. The description of the task details that there needs to be one fully connected hidden layer with 10 neurons in it initially. Visualising this in a diagram can be seen below in *Figure 10*. Code for generating the diagram provided by Harper (2018).

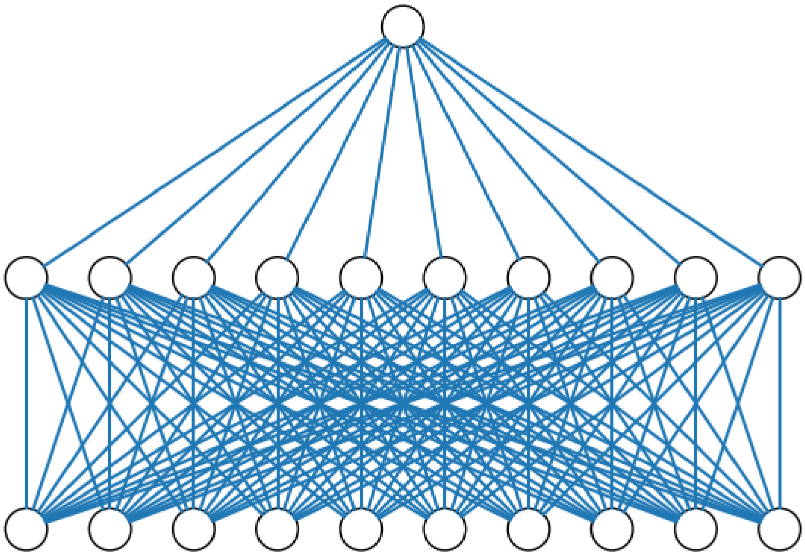
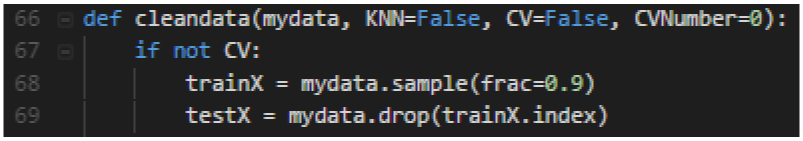


Figure 10 - Visualised neural network with 10 inputs, 10 neurons in the hidden layer and one output neuron

The number of connections between the first and second layer is many, as the data for each of the columns is connected to each other in each neuron.

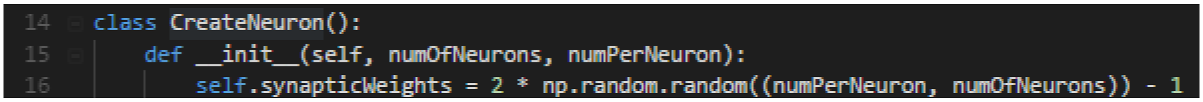
For the programmatic solution, the data needs to be initially split into the test and training datasets, outlined in the task description to be 90% for training and the remaining 10% for testing. This functionality is performed in the *cleanData* function, which splits the data using the Pandas inbuilt function *sample*. The function takes a random sample of the data, which in this case is defined as 0.9 (90%) *Figure 11* below shows the splitting of data.



```
def cleandata(mydata, KNN=False, CV=False, CVNumber=0):
    if not CV:
        trainX = mydata.sample(frac=0.9)
        testX = mydata.drop(trainX.index)
```

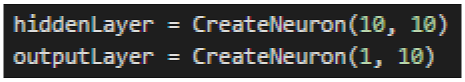
Figure 11 - cleanData function extract showing splitting data

Once the data has been cleaned and correctly assigned to training and test sets, a layer can be created and initialised with the correct number of neurons and inputs per neuron. To do this, a Python Class called *CreateNeuron* is called which assigns a set of synaptic weights at random initially using numpy, and similarly a bias set at random. *Figure 12* shows the class definition and initialisations, and *Figure 13* shows an example call to this Class to create the hidden and outer layers with the correct number of neurons.



```
class CreateNeuron():
    def __init__(self, numOfNeurons, numPerNeuron):
        self.synapticWeights = 2 * np.random.random((numPerNeuron, numOfNeurons)) - 1
```

Figure 12 - CreateNeuron Class initialisation

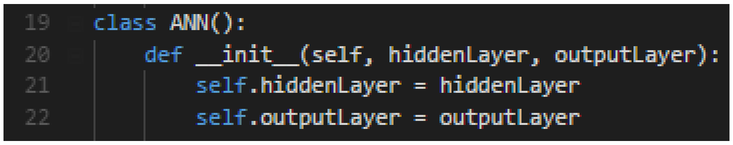


```
hiddenLayer = CreateNeuron(10, 10)
outputLayer = CreateNeuron(1, 10)
```

Figure 13 - CreateNeuron calls to create hidden layer and output layers

The calls to the CreateNeuron class in *Figure 13* show that the network in the hidden layer is being created with 10 neurons, with 10 inputs for each. Following that, the output layer has 1 neuron with 10 inputs, fed by the 10 neurons in the hidden layer. This mirrors *Figure 10*, which has each neuron receiving 10 inputs in the hidden layer.

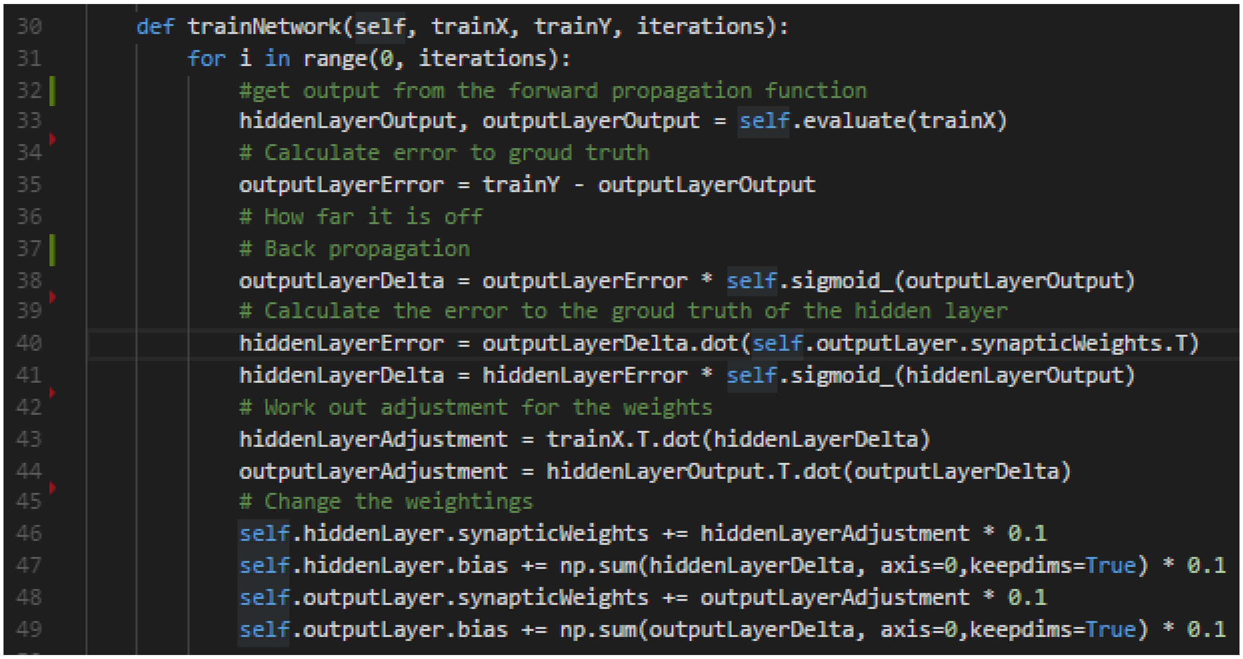
After creating the objects *hiddenLayer* and *outputLayer*, they need to be added as arguments to the *ANN* class instantiation, so you can begin calling functions within its class. The definition and initialisation of the objects is shown below in an extract of the full class, *Figure 14.*



```
class ANN():
    def __init__(self, hiddenLayer, outputLayer):
        self.hiddenLayer = hiddenLayer
        self.outputLayer = outputLayer
```

Figure 14 - ANN Class definition and initialisation of input layers

The next function that would be called from within the *ANN* class is *trainNetwork*, which takes an argument of the training dataset split into the columns of data, and the ground truth on whether the patient has diabetes or not. The final parameter is the number of iterations that the function will run for and adjust the weights and bias after each iteration. The full function is shown below in *Figure 15.*

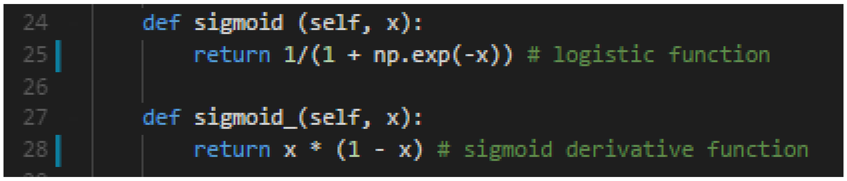


```
    def trainNetwork(self, trainX, trainY, iterations):
        for i in range(0, iterations):
            #get output from the forward propagation function
            hiddenLayerOutput, outputLayerOutput = self.evaluate(trainX)
            # Calculate error to groud truth
            outputLayerError = trainY - outputLayerOutput
            # How far it is off
            # Back propagation
            outputLayerDelta = outputLayerError * self.sigmoid_(outputLayerOutput)
            # Calculate the error to the groud truth of the hidden layer
            hiddenLayerError = outputLayerDelta.dot(self.outputLayer.synapticWeights.T)
            hiddenLayerDelta = hiddenLayerError * self.sigmoid_(hiddenLayerOutput)
            # Work out adjustment for the weights
            hiddenLayerAdjustment = trainX.T.dot(hiddenLayerDelta)
            outputLayerAdjustment = hiddenLayerOutput.T.dot(outputLayerDelta)
            # Change the weightings
            self.hiddenLayer.synapticWeights += hiddenLayerAdjustment * 0.1
            self.hiddenLayer.bias += np.sum(hiddenLayerDelta, axis=0,keepdims=True) * 0.1
            self.outputLayer.synapticWeights += outputLayerAdjustment * 0.1
            self.outputLayer.bias += np.sum(outputLayerDelta, axis=0,keepdims=True) * 0.1
```

Figure 15 - trainNetwork function

The function above in *Figure 15* trains the neural network using the training dataset by using both forward and backpropagation, with a result of a change in synaptic weights and bias for both layers, therefore directing the training towards the target of gaining a closely fitted neural network.

The function begins with iterating through each one of the iterations as outlined in the function inputs, and getting the outputs of the forward propagation function call *evaluate*, which is in *Figure 17,* and calculates the output of combining the input dataset with the weights and bias, before putting that output through the sigmoid logistic activation function (shown below in *Figure 16)*. This is then returned to the *trainNetwork* function, and further analysis can be performed. The initial error from the outputted data on *Line 33* is calculated on *Line 35*, and indicates how far off the currently trained network is from the ground truth values. This data is then passed through the back-propagation algorithm, calculating the delta by passing it through the sigmoid derivative activation function as shown below in *Figure 16.* The hidden layers error is calculated using the delta from the output layer, and then finally the hidden layer delta can be calculated using the sigmoid activation function. Adjusting the weights is taken care of between lines 43-49, which is calculated using the deltas from each layer. The weightings and bias adjustments are multiplied by 0.1 as a learning rate, to prevent the algorithm making jumps between iterations which would over correct itself.

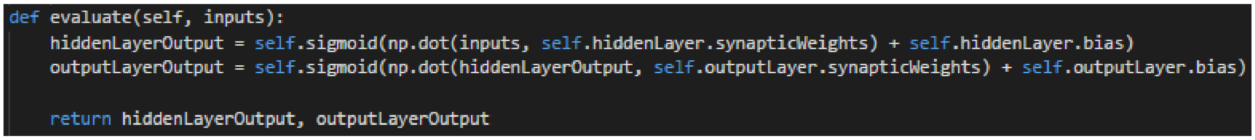


```
    def sigmoid (self, x):
        return 1/(1 + np.exp(-x)) # logistic function

    def sigmoid_(self, x):
        return x * (1 - x) # sigmoid derivative function
```

Figure 16 - Sigmoid and logistic activation functions

After this process has been completed, a fully trained network will be created with the weights stored as part of the class definition. This is then ready for running the *evaluate* function, which can be called with the test dataset to produce the predicted values of the test dataset. *Figure 17* shows the *evaluate* function.

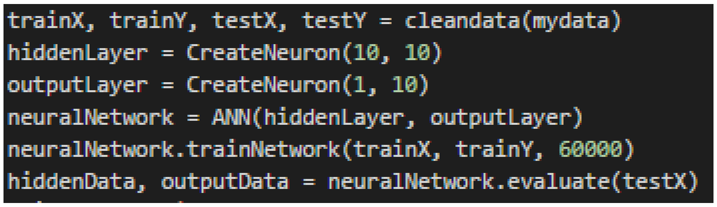


```
def evaluate(self, inputs):
    hiddenLayerOutput = self.sigmoid(np.dot(inputs, self.hiddenLayer.synapticWeights) + self.hiddenLayer.bias)
    outputLayerOutput = self.sigmoid(np.dot(hiddenLayerOutput, self.outputLayer.synapticWeights) + self.outputLayer.bias)

    return hiddenLayerOutput, outputLayerOutput
```

Figure 17 - evaluate function

As explained previously in this section, this function performs forward propagation to predict the values, this time using the weights and biases outputted from the fully trained network. An example process of going through each training function before calling this function is below, *Figure 18*.



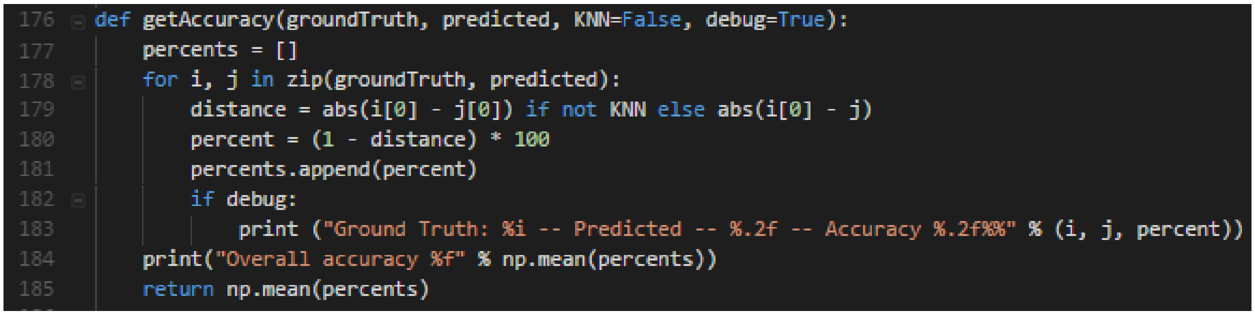
```
trainX, trainY, testX, testY = cleandata(mydata)
hiddenLayer = CreateNeuron(10, 10)
outputLayer = CreateNeuron(1, 10)
neuralNetwork = ANN(hiddenLayer, outputLayer)
neuralNetwork.trainNetwork(trainX, trainY, 60000)
hiddenData, outputData = neuralNetwork.evaluate(testX)
```

Figure 18 - Full process of cleaning data, training network and predicting test values

As the function is also used earlier in the training, the hidden layer is also outputted in the test function, however it is unused. The *outputData* variable is the meaningful output and has the predicted 0 (no diabetes) or 1 (diabetes) values. Evaluating this output is then passed into a function *getAccuracy*, which takes the ground truth values, the predicted values and flags to indicate whether the data is from the K-nearest-neighbour algorithm. The output of this function gives an overall percentage of accuracy from the predicted values to the ground truth ones, as well as giving options for logging in the console the individual accuracies for each predicted value. *Figure 19* shows this function.

The function itself calculates the distance between the ground truth and predicted values, and then makes sure that the value is always positive by calling the function *abs* which will ensure that. Once that has been completed the value is multiplied to turn it into a percentage and added to an internal array, which from that can calculate the mean percentage accuracy rate, which is returned. This works for both KNN and ANN implementations, and is one of the few shared functions.

Outputs of this function for both the ANN and KNN will be shown after explaining the implementation of the KNN algorithm.

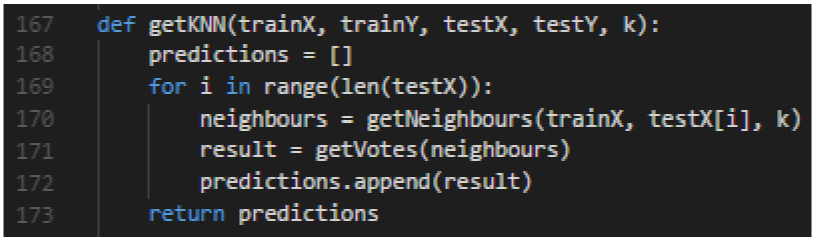


```
def getAccuracy(groundTruth, predicted, KNN=False, debug=True):
    percents = []
    for i, j in zip(groundTruth, predicted):
        distance = abs(i[0] - j[0]) if not KNN else abs(i[0] - j)
        percent = (1 - distance) * 100
        percents.append(percent)
        if debug:
            print ("Ground Truth: %i -- Predicted -- %.2f -- Accuracy %.2f%%" % (i, j, percent))
    print("Overall accuracy %f" % np.mean(percents))
    return np.mean(percents)
```

Figure 19 - getAccuracy function

The K-Nearest Neighbour implementation at a high level calculates the distance between a given patients data, and another patient to find which is the closest one. From that, it then uses the value of the closest neighbour as its output (whether the patient has diabetes or not). While the concept of this is simple, expressing this programmatically takes several small functions.

The first function which controls the whole operation is called *getKNN*, and takes an input of training and test data, as well as the number of K's that you want to evaluate against (how far you want to spread your search). *Figure 20* shows this function.



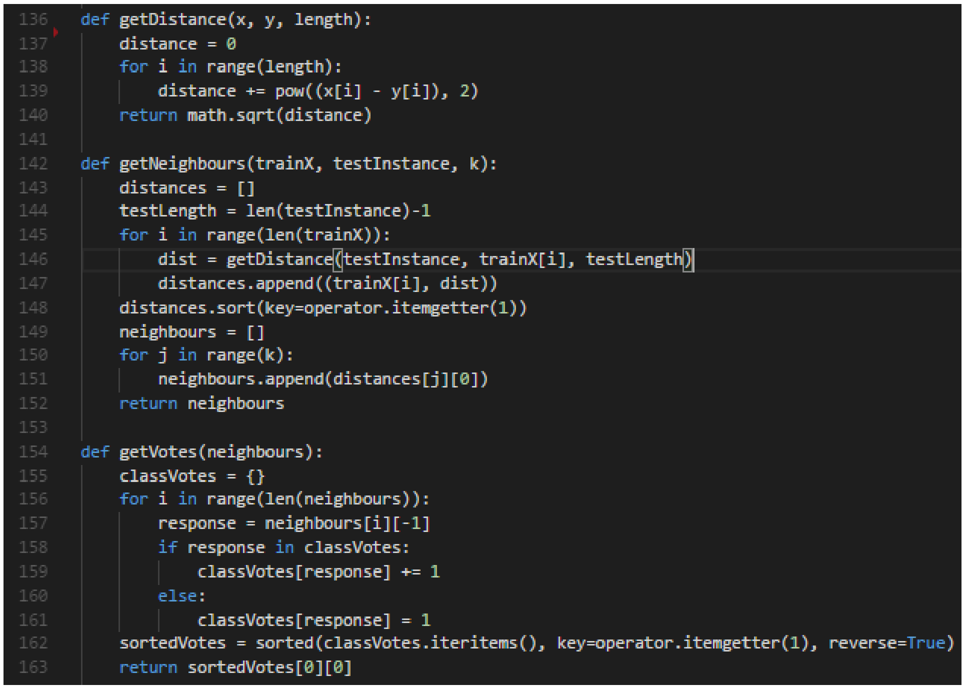
```
def getKNN(trainX, trainY, testX, testY, k):
    predictions = []
    for i in range(len(testX)):
        neighbours = getNeighbours(trainX, testX[i], k)
        result = getVotes(neighbours)
        predictions.append(result)
    return predictions
```

Figure 20 - getKNN function

The function begins by running a for loop going through each of the test values, which is then passed into the *getNeighbours* function, taking arguments of the training data, the test data and the number of

K’s that is being used for evaluation. The result of this is an array of neighbours around the inputted test data, and the closest one is found using the *getVotes* function. The output of this is added to the predictions array and the process is ran again until all the test values have been predicted.

Delving deeper into each function, *Figure 21* shows the *getDistance, getNeighbours* and *getVotes* functions. Due to the simplicity of the algorithm and the relatively easy to read code, no further explanation will be made for each specific function’s internal processing.



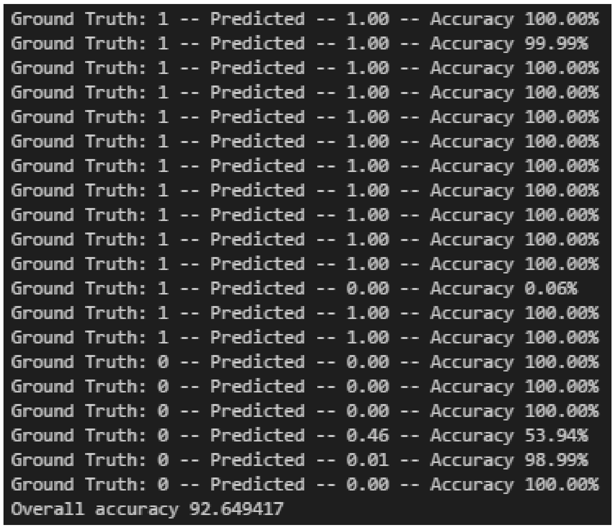
```python
def getDistance(x, y, length):
    distance = 0
    for i in range(length):
        distance += pow((x[i] - y[i]), 2)
    return math.sqrt(distance)

def getNeighbours(trainX, testInstance, k):
    distances = []
    testLength = len(testInstance)-1
    for i in range(len(trainX)):
        dist = getDistance(testInstance, trainX[i], testLength)
        distances.append((trainX[i], dist))
    distances.sort(key=operator.itemgetter(1))
    neighbours = []
    for j in range(k):
        neighbours.append(distances[j][0])
    return neighbours

def getVotes(neighbours):
    classVotes = {}
    for i in range(len(neighbours)):
        response = neighbours[i][-1]
        if response in classVotes:
            classVotes[response] += 1
        else:
            classVotes[response] = 1
    sortedVotes = sorted(classVotes.iteritems(), key=operator.itemgetter(1), reverse=True)
    return sortedVotes[0][0]
```

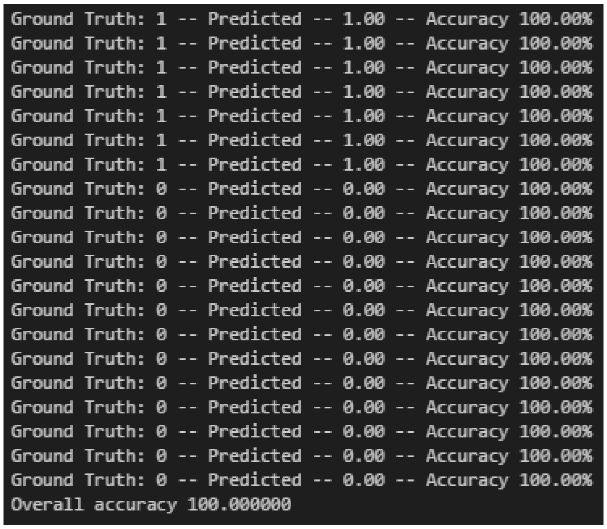
Figure 21 - getDistance, getNeighbours & getVotes functions for calculating KNN

The initial validation results for both algorithms are below, with the ANN using 10 neurons in the hidden layer with 10 inputs in each neuron. The KNN implementation using a K size of 5, as indicated in the task description.



```
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 99.99%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 0.00 -- Accuracy 0.06%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.46 -- Accuracy 53.94%
Ground Truth: 0 -- Predicted -- 0.01 -- Accuracy 98.99%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Overall accuracy 92.649417
```

Figure 22 - ANN Predicted results



```
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 1 -- Predicted -- 1.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Ground Truth: 0 -- Predicted -- 0.00 -- Accuracy 100.00%
Overall accuracy 100.000000
```

Figure 23 - KNN Predicted results

The initial results above in *Figure 22 & Figure 23* indicate that the KNN model gives better performance as it has 100% accuracy. However, as this is initial results and the dataset is random, it can't be evaluated full without performing 10-fold cross validation to estimate the output when presented with a new dataset to evaluate against. The ANN results are also very accurate at 92.6%, and more analysis will be required to pick an appropriate model overall.

Section 4

The rationale behind performing 10-fold cross-validation is to remove certain parts of the dataset at a time, to see how it affects the accuracy of the model. This can give the effect of testing the model with different datasets, and gives a better overall view of how it can perform in a new situation. *Figure 24 below shows how the different folds progress through the dataset.*

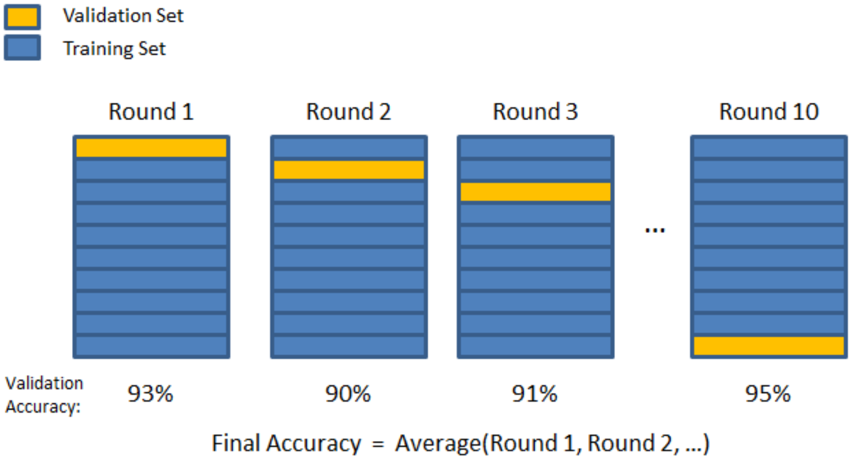
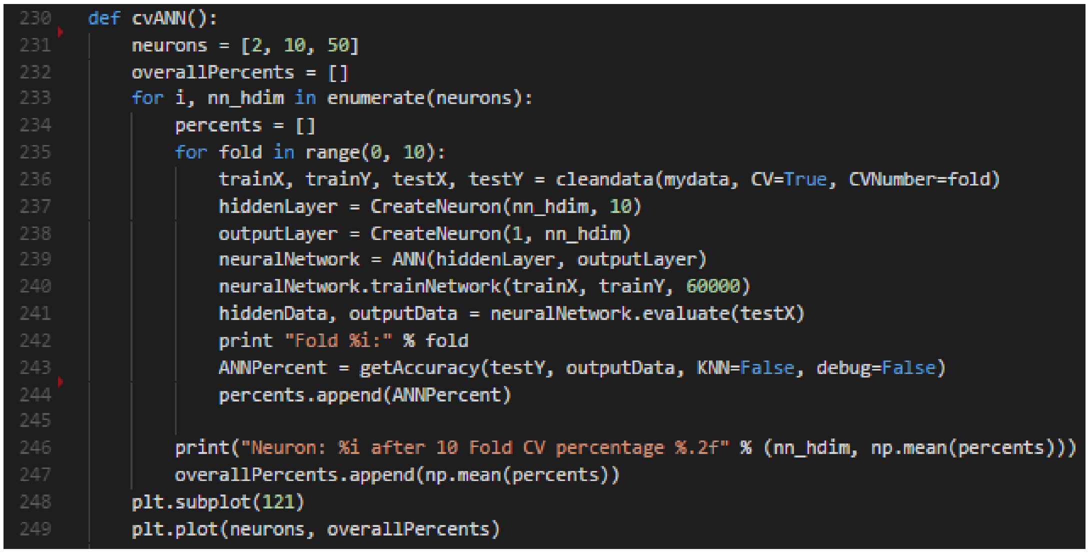


Figure 24 - 10-Fold CV diagram. McCormick (2013)

The specific steps for both the KNN and ANN for calculating 10-fold CV are similar, however accept different parameters. The implementation first to be discussed, ANN, is shown below in *Figure 25*

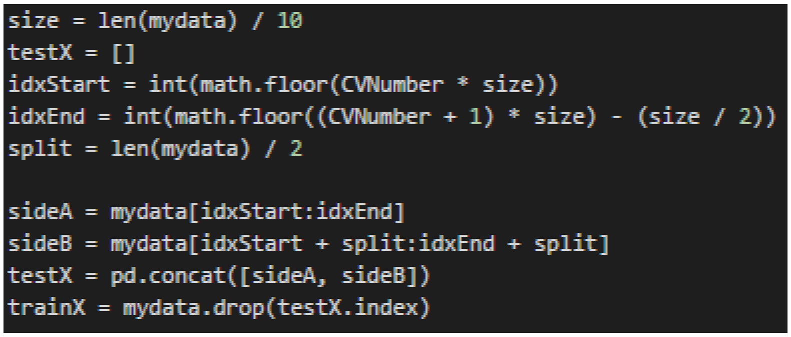


```python
def cvANN():
    neurons = [2, 10, 50]
    overallPercents = []
    for i, nn_hdim in enumerate(neurons):
        percents = []
        for fold in range(0, 10):
            trainX, trainY, testX, testY = cleandata(mydata, CV=True, CVNumber=fold)
            hiddenLayer = CreateNeuron(nn_hdim, 10)
            outputLayer = CreateNeuron(1, nn_hdim)
            neuralNetwork = ANN(hiddenLayer, outputLayer)
            neuralNetwork.trainNetwork(trainX, trainY, 60000)
            hiddenData, outputData = neuralNetwork.evaluate(testX)
            print "Fold %i:" % fold
            ANNPercent = getAccuracy(testY, outputData, KNN=False, debug=False)
            percents.append(ANNPercent)

        print("Neuron: %i after 10 Fold CV percentage %.2f" % (nn_hdim, np.mean(percents)))
        overallPercents.append(np.mean(percents))
    plt.subplot(121)
    plt.plot(neurons, overallPercents)
```

Figure 25 - 10-Fold Cross Validation for Artificial Neural Network

Similarly, to *Figure 18*, the bulk of the actual training of the network and evaluating the results are as before (*Lines 236-244)* however the 10-fold is performed in two main stages, the first being the for loop that encases the main bulk of the code, which iterates through 10 stages, passing in the current fold into the *cleanData* function, which calculates which fold to return. *Figure 26* below shows where this is performed in *cleanData*. The rationale between taking the data from two different sides of the dataset is that it is split into diabetes and non-diabetes, and each fold needs to have the same number of each.



```
size = len(mydata) / 10
testX = []
idxStart = int(math.floor(CVNumber * size))
idxEnd = int(math.floor((CVNumber + 1) * size) - (size / 2))
split = len(mydata) / 2

sideA = mydata[idxStart:idxEnd]
sideB = mydata[idxStart + split:idxEnd + split]
testX = pd.concat([sideA, sideB])
trainX = mydata.drop(testX.index)
```

Figure 26 - 10-Fold data splitting

After each fold the percentage is appended to an array, which is then outputted as the mean of itself in a graph. The process is very similar in the KNN implementation; however, the main bulk of the code is replaced with *getKNN* to get the predictions, and the neurons replaced with the number of K's that is being evaluated against. The output of both these functions when having a hidden layer consisting of 2, 10 and 50 neurons, and K's of 1, 5 and 10 produce the graphs as shown below in *Figure 27*.

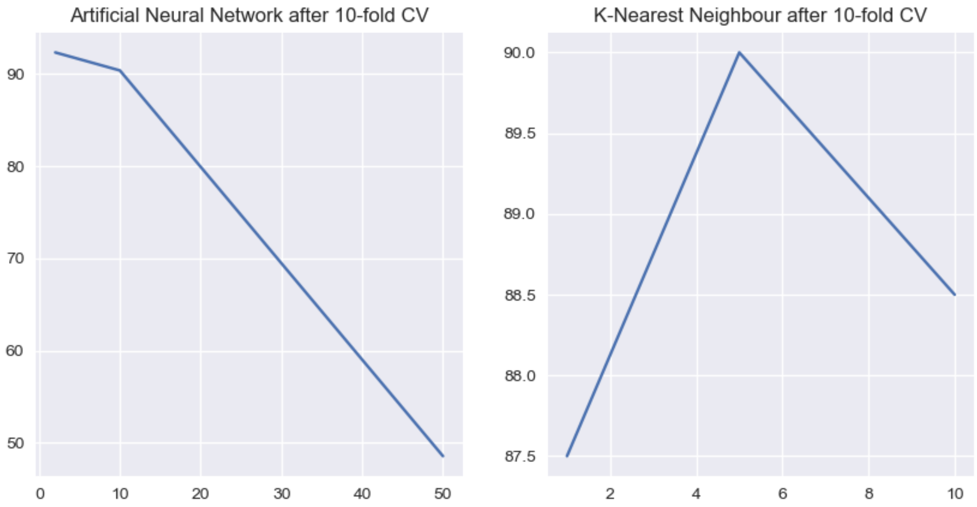


Figure 27 - ANN & KNN outputs after 10-fold CV

Looking at the results above in *Figure 27*, there is a slight advantage to using the ANN with 2 neurons in the hidden layer, with the best result being 92.33%. This is compared to the best result of the KNN, which was 90% accuracy at K number 5. Post these figures the models start to overfit, in a similar way to *Figure 8*. Therefore, finding a balance between an underfitted model (KNN where K = 1 for example), and an overfitted one (ANN where Neurons = 50 for example) is key.

These results would indicate that the Artificial Neural Network is a better classifier at their optimal parameters. Deeper results using more parameters have been evaluated against to confirm whether there are any data points that would have a higher accuracy that have not yet been evaluated. These are shown below in *Figure 28*.

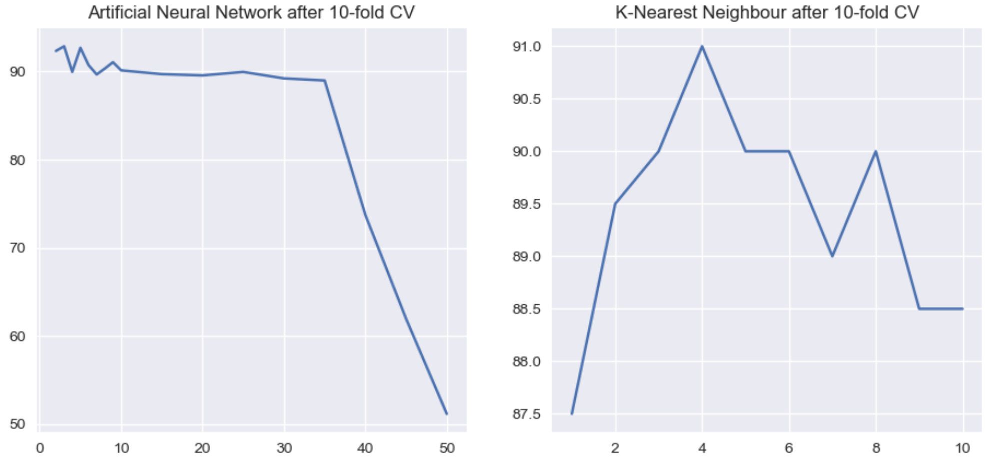


Figure 28 - More evaluated points on 10-Fold CV for ANN & KNN

These final results indicate that while ANN still has a higher overall accuracy, the best number of neurons is different to what was initially evaluated. The results above indicate that 3 neurons in the hidden layer give an overall accuracy of 92.86%. Additionally, the KNN results have improved, with 4 K's being most optimal, to give an improved accuracy of 91%.

This therefore indicates and further confirms that using 3 neurons in an Artificial Neural Network for this dataset would give the best accuracy and would be most appropriate.

References

Pandas. (2017). *Pandas Toolkit.* Available: http://pandas.pydata.org/pandas-docs/stable/pandas.pdf. Last accessed 18th April 2018.

Numpy. (2017). *Numpy Reference.* Available: https://docs.scipy.org/doc/numpy-1.13.0/numpy-ref-1.13.0.pdf. Last accessed 18th April 2018.

Matplotlib. (2018). *Matplotlib Reference.* Available: https://matplotlib.org/. Last accessed 18th April 2018.

Seaborn. (2017). *Seaborn: statistical data visualisation.* Available: https://seaborn.pydata.org/. Last accessed 18th April 2018.

Harper. (2018). *Visualise Neural Network.* Available: https://github.com/miloharper/visualise-neural-network. Last accessed 18th April 2018.

McCormick, Chris. (2013). *K-Fold Cross-Validation*. Available: http://mccormickml.com/2013/07/31/k-fold-cross-validation-with-matlab-code/**.** Last accessed 18th April 2018.