MANIPAL UNIVERSITY
JAIPUR

CESi recherche
LINEACT

A
Project Report
on

MACHINE LEARNING ALGORITHMS TO DETECT OUTLIERS

By

Shrey Mishra

(UNDER INTERNSHIP)

Roll No: 149107438

Department of Electronics and Communication Engineering

Manipal University, Jaipur

Under Guidance of

Dr. Nait Chabane Ahmed

Mar-May, 2018

Exia Cesi Saint Nazaire France

CERTIFICATE

This is to certify that the winter intern project report entitled, "***Machine learning algorithms to detect outliers***" submitted by **Shrey Mishra** in partial fulfillment of the requirement for the two-months (01-03-2018 to 01-05-2018) research internship program in Exia Cesi Saint Nazaire, France is carried out by him under my supervision and guidance.

Dr. Nait Chabane Ahmed

Professor,

Exia Cesi School of Engineering

Date: 01.05.2018

Saint Nazaire, France

ACKNOWLEDGEMENT

It is with a feeling of great pleasure that I would like to express my most sincere heartfelt gratitude to **Dr. Nait Chabane Ahmed**, Professor, Dept. of Engineering, Cesi Exia Saint Nazaire, France for suggesting the topic for my project work and for his ready and able guidance throughout my internship. I am greatly indebted to him for his constructive suggestion and criticism from time to time during the course of progress of my work.

I express my sincere thanks to **Dr. Rajveer Singh Shekhawat**, Director, Dept. of Computer Science & IT Engineering, Manipal University Jaipur for allowing me to do my internship as well as providing the necessary facilities in the department. I express my gratitude to **Mrs. Dumortier Stephanie, Dr. Vinayak Kulkarni,** for their advice, guidance and valuable suggestion for successfully completion of my intern project work.

I feel pleased and privileged to complete my internship in Cesi Exia school of Engineering.

Shrey Mishra

Roll No: 149107438

Date: 01.03.2018

Manipal University, Jaipur

ABSTRACT

The aim of the internship is to lay out a clear comparison between the different approaches that can be adopted to detect an outlier for a given data set.

There is a crisp comparison on accuracy of prediction on various algorithms (approaches that can be adopted to classify an outlier)

Keeping the idea in mind I propose various approaches that can be followed to implement a system to detect the outlier:

1-Linear regression approach

2-Classification (logistic regression approach)

3-Neural Networks

4-Self organized maps

The data sets I will be using are freely open for anyone else to use here is a link – data sets [1]

The idea's implemented in my work are heavily influenced from Andrew Ng's online course for machine learning, here is a link to the course page [2] and some of my own ideas.

TABLE OF CONTENT

LIST OF FIGURES

State of the Art

An *outlier* is an observation that lies an abnormal distance from other values in a random sample from a data. In a sense, This définition leaves it up to the analyst. Before abnormal observations can be singled out, it is necessary to characterize normal observations.

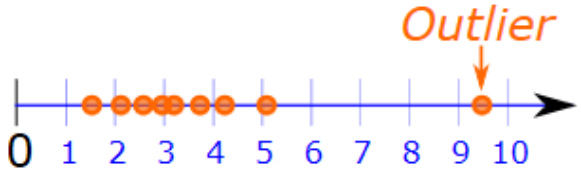


Figure (A) - An example of an anomly point in a given data[3]

A more exhaustive list of applications that utilise outlier detection is:

1) **Fraud detection** - Detecting fraudulent applications for credit cards

2) **Loan application processing** - to detect fraudulent applications or potentially problematical customers.

3) **Intrusion detection** - detecting unauthorised network access.

4) **Activity monitoring** - detecting mobile phone fraud by monitoring phone activity or suspicious trades in the equity markets.

5) **Network performance** - monitoring the performance of computer networks, for example to detect network bottlenecks.

6) **Structural defect detection** - monitoring manufacturing lines to detect faulty production runs for example cracked beams.

7) **Medical condition monitoring** - such as heart rate monitors, identifying novel molecular structures in pharmaceutical research.

8) **Detecting unexpected entries in databases** - for data mining to detect errors, frauds or valid but unexpected entries.

9) **Detecting mislabelled data** - in a training data set.

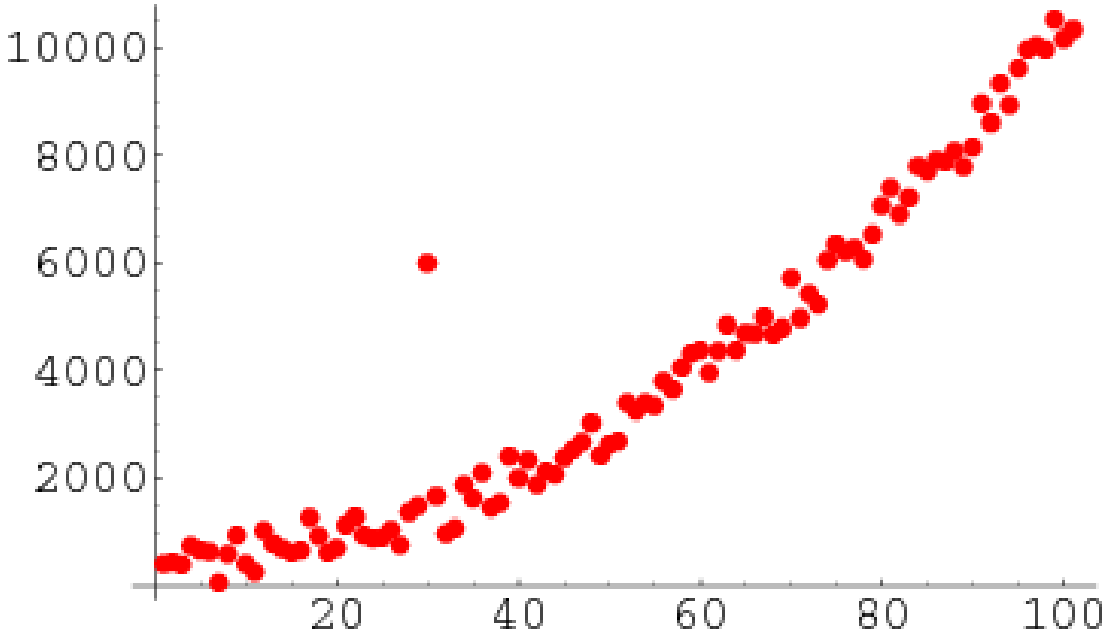


Figure (B) - Shows the idea of plotting out the outliers from a given data set

Effective intrusion detection is a difficult and elusivegoal for system administrators and information security researchers for which our outlier detection program can be used. The intrusion detection can be used with our program to identify intruder's based on some given data and with all the approaches that I will adopt, We can identify a possible intruder with some data to train with our algorithms. With this approach I propose a better solution and a crisp clear comparison with all the techniques (approaches) that I adpoted for the task. [4]

Chapter 1

Linear Regression

In supervised learning, we are given a data set and already know what our correct output should look like, having the idea that there is a relationship between the input and the output.

Supervised learning problems are categorized into "regression" and "classification" problems. In a regression problem, we are trying to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function.

In a classification problem, we are instead trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories.

The Hypothesis Function: -

Our hypothesis function has the general form:

$y\hat{} = h\vartheta(x) = \vartheta 0 + \vartheta 1 * x$

Note that this is like the equation of a straight line. We give to $h\vartheta(x)$ values for $\vartheta 0$ and $\vartheta 1$ to get our estimated output $y\hat{}$. In other words, we are trying to create a function called $h\vartheta$ that is trying to map our input data (the x's) to our output data (the y's).

Example:

Input (x)	Output (y)
0	4
1	7
2	7
3	8

Suppose we have the following set of training data:

Now we can make a random guess about our $h\vartheta$ function: $\vartheta 0=2$ and $\vartheta 1=2$. The hypothesis function becomes $h\vartheta(x)=2+2x$.

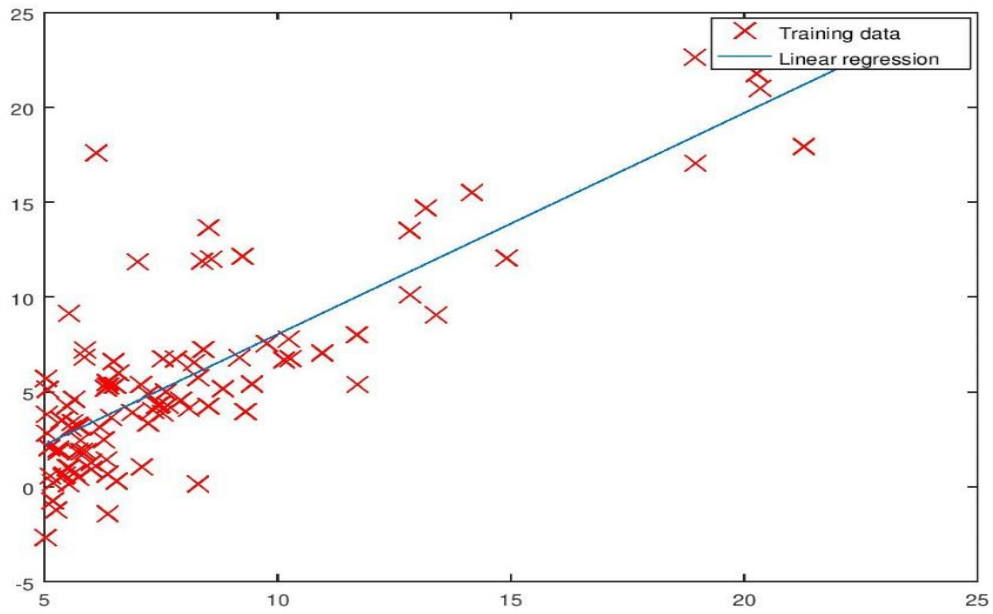


Figure 1.1 - An example of a linear hypothesis curve

So, for input of 1 to our hypothesis, y will be 4. This is off by 3. Note that we will be trying out various values of $\vartheta 0$ and $\vartheta 1$ to try to find values which provide the best possible "fit" or the most representative "straight line" through the data points mapped on the x-y plane.

Cost Function: -

We can measure the accuracy of our hypothesis function by using a **cost function**. This takes an average (actually an advance version of an average) of all the results of the hypothesis with inputs from x's compared to the actual output y's.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2$$

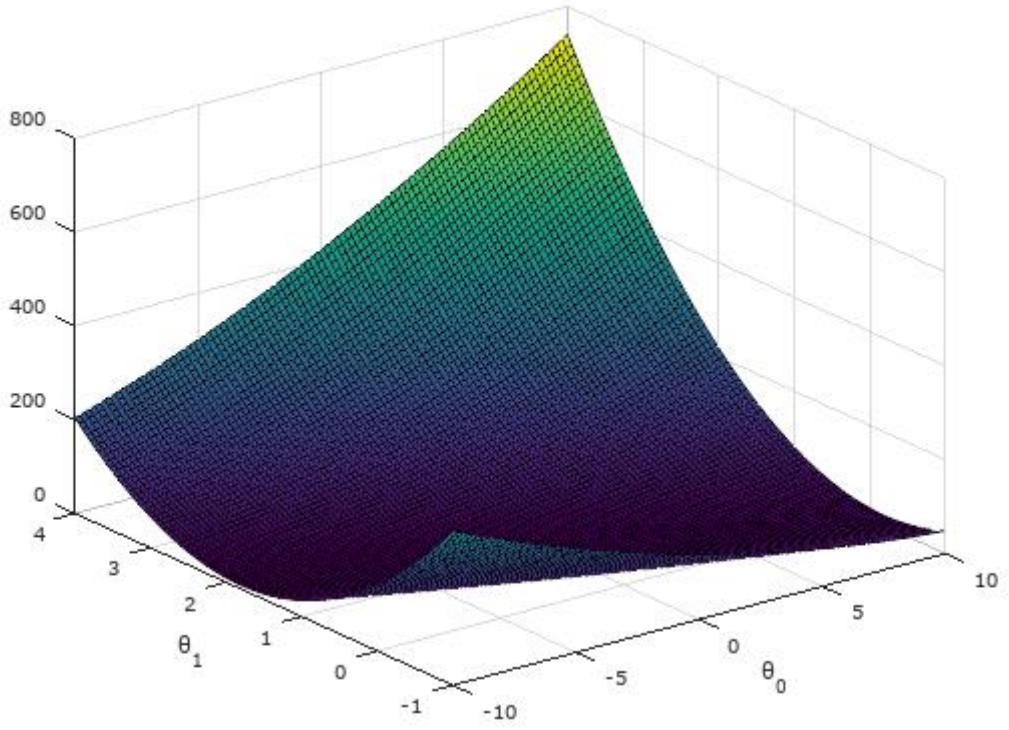


Figure 1.2(a)-The plot shows the varying Cost (J) is convex function

Here alpha can be defined as the learning rate which, if is too it much can overshoot on the convergence graph instead of converging and then we set it to repeat for n number of iterations such that the convergence graph gets straight or narrowed in the end as you can see in the picture below:

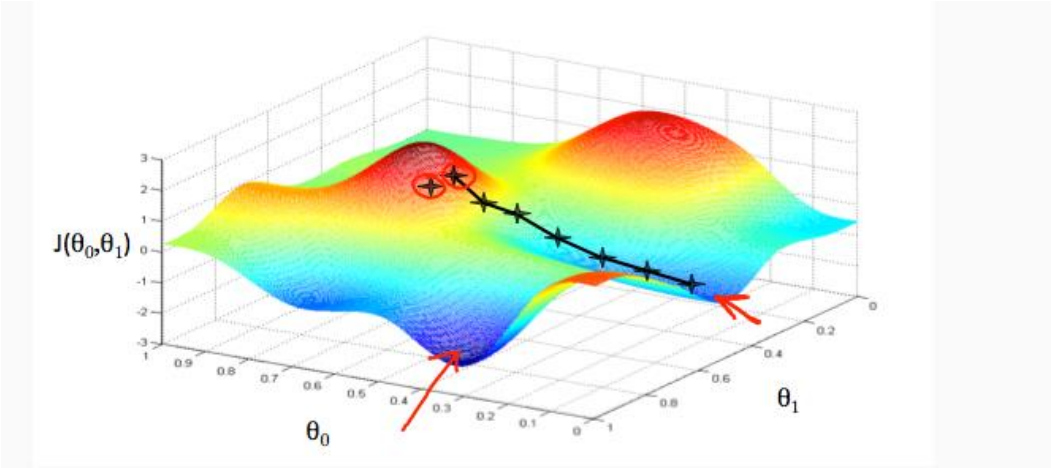


Figure 1.2 (b)-The plot shows the varying Cost (J) is convex function after every iteration

If we try to think of it in visual terms, our training data set is scattered on the x-y plane. We are trying to make straight line (defined by $h\vartheta(x)$) which passes through this scattered set of data. Our objective is to get the best possible line. The best possible line will be such so that the average squared vertical distances of the scattered points from the line will be the least. In the best case, the line should pass through all the points of our training data set. In such a case the value of J (ϑ0, θ1) will be 0.

Gradient Descent: -

We have our hypothesis function and we have a way of measuring how well it fits into the data. Now we need to estimate the parameters in hypothesis function. That's where gradient descent comes in.

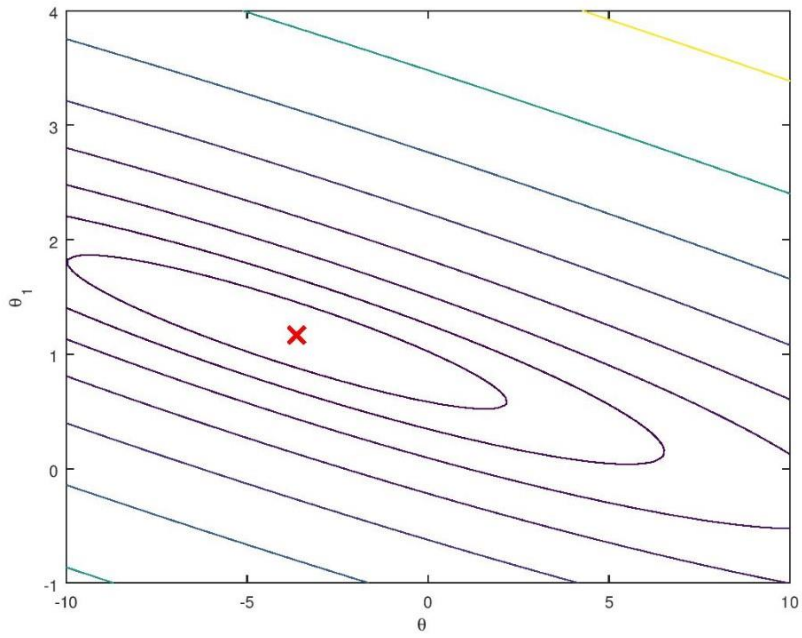


Figure 1.3-Shows how gradient converges to the global minima point

Imagine that we graph our hypothesis function based on its fields $\vartheta 0$ and $\vartheta 1$ (actually we are graphing the cost function as a function of the parameter estimates). This can be kind of confusing; we are moving up to a higher level of abstraction. We are not graphing x and y itself, but the parameter ranges of our hypothesis function and the cost resulting from selecting particular set of parameters.

We put $\vartheta 0$ on the x axis and $\vartheta 1$ on the y axis, with the cost function on the vertical z axis. The points on our graph will be the result of the cost function using our hypothesis with those specific theta parameters.

We will know that we have succeeded when our cost function is at the very bottom of the pits in our graph, i.e. when its value is the minimum.

The way we do this is by taking the derivative (the tangential line to a function) of our cost function. The slope of the tangent is the derivative at that point and it will give us a direction to move towards. We make steps down the cost function in the direction with the steepest descent, and the size of each step is determined by the parameter α, which is called the learning rate.

The gradient descent algorithm is:

Repeat until convergence: $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$

Where j=0,1 represents the feature index number.

Intuitively, this could be thought of as:

Repeat until convergence: $\vartheta j := \vartheta j - \alpha$

[Slope of tangent aka derivative in j dimension]

<u>Gradient Descent for Linear Regression</u>:

When specifically applied to the case of linear regression, a new form of the gradient descent equation can be derived. We can substitute our actual cost function and our actual hypothesis function and modify the equation to (the derivation of the formulas is out of the scope of this course, but a really great one can be found here):

repeat until convergence: {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} ((h_\theta(x_i) - y_i) x_i)$$

}

Where m is the size of the training set, $\vartheta 0$ a constant that will be changing simultaneously with $\vartheta 1$ and *xi*, *yi* are values of the given training set (data).

Note that we have separated out the two cases for ϑj into separate equations for $\vartheta 0$ and $\vartheta 1$; and that for $\vartheta 1$ we are multiplying *xi* at the end due to the derivative. The point of all this is that if we start with a guess for our hypothesis and then repeatedly apply these gradient descent equations, our hypothesis will become more and more accurate.

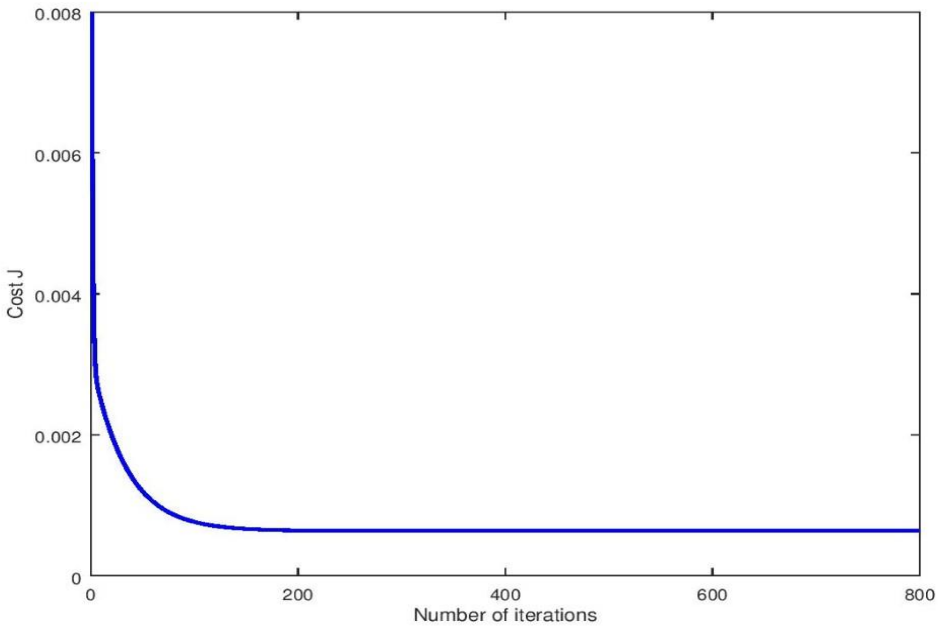


Figure 1.4-Shows the convergence of Cost (J) after a certain number of iterations

Chapter 2

APPLYING MODEL ON DATA SETS

Linear Regression with Multiple Variables

We now introduce notation for equations where we can have any number of input variables.

$$
\begin{aligned}
x_j^{(i)} &= \text{value of feature } j \text{ in the } i^{th} \text{ training example} \\
x^{(i)} &= \text{the column vector of all the feature inputs of the } i^{th} \text{ training example} \\
m &= \text{the number of training examples} \\
n &= \left|x^{(i)}\right|; (\text{the number of features})
\end{aligned}
$$

Now define the multivariable form of the hypothesis function as follows, accommodating these multiple features:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$$

This is a vectorization of our hypothesis function for one training example. The training examples are stored in X row-wise, like such:

$$X = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} \\ x_0^{(2)} & x_1^{(2)} \\ x_0^{(3)} & x_1^{(3)} \end{bmatrix}, \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

You can calculate the hypothesis as a column vector of size (m x 1) with:

$$h\vartheta(X) = X\vartheta$$

Gradient Descent for Multiple Variables

The gradient descent equation itself is generally the same form; we just have to repeat it for our 'n' features:

$$
\begin{aligned}
&\text{repeat until convergence: } \{ \\
&\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)} \\
&\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_1^{(i)} \\
&\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_2^{(i)} \\
&\cdots \\
&\}
\end{aligned}
$$

In other words,

$$
\begin{aligned}
&\text{repeat until convergence: } \{ \\
&\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \qquad \text{for j} := 0..\text{n} \\
&\}
\end{aligned}
$$

Features and Polynomial Regression

We can improve our features and the form of our hypothesis function in a couple different ways.

We can combine multiple features into one. For example, we can combine $x1$ and $x2$ into a new feature $x3$ by taking $x1 \cdot x2$.

Polynomial Regression

Our hypothesis function need not be linear (a straight line) if that does not fit the data well. We can change the behavior or curve of our hypothesis function by making it a quadratic, cubic or square root function (or any other form).

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2$$

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3$$

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3$$

One important thing to keep in mind is, if you choose your features this way then feature scaling becomes very important.

Normal Equation Method

The "Normal Equation" is a method of finding the optimum theta without iteration.

$$\theta = (X^T X)^{-1} X^T y$$

There is **no need** to do feature scaling with the normal equation. The following is a comparison of gradient descent and the normal equation:

Gradient Descent	Normal Equation
Need to choose alpha	No need to choose alpha
Needs many iterations	No need to iterate
O (*kn*2)	O (*n*3), need to calculate inverse of (*XTX)*
Works well when n is large	Slow if n is very large

With the normal equation, computing the inversion has complexity O(*n*3). So if we have a very large number of features, the normal equation will be slow. In practice, when n exceeds 10,000 it might be a good time to go from a normal solution to an iterative process.

Chapter 3

Regression Approach

Outlier prediction (Normal equation method):

To begin with the problem of optimization lets visualize the training points to begin with the problem as shown in the figure below

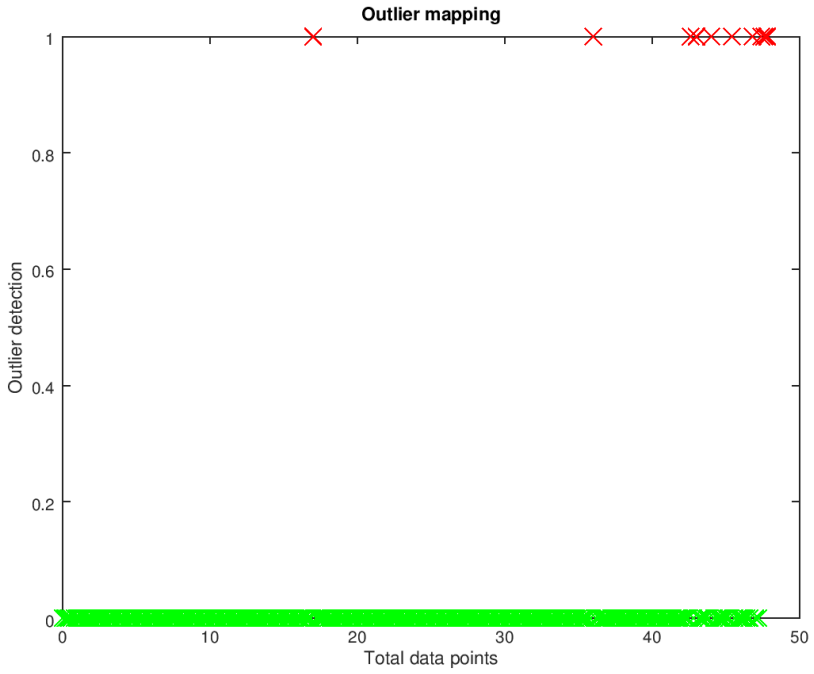


Figure 3.1-Shows the classified data (Vertebral.mat) before running the linear regression (Data points Vs Classification)

We run the Normal equation method for hypothesis to directly compute for the value of vector theta

(θ) and then overlay the predicted curve on the training samples. With this curve we can predict the value (Outlier or Not) at any point on the curve by multiplying Theta (θ) calculated for the predicted curve with the Parameters of X

While implementing this algorithm several points must be certain -

- The learning rate should be appropriate (not too high to overshoot the convergence graph/ nor too low to take a lot more number of iterations to find the global minima).
- The number of iterations set for the algorithm should be enough to straighten (Flat) the graph in the end so we know that there is not much change in cost if we change the theta parameter.
- The threshold set should be ideal (should not be too high/low) this can lead to false negative's and false positives to the data.

Normal equation result (Threshold=1.5*Average_Value):

```
Program paused. Press enter to continue.
Normalizing Features ...
Theta computed from the normal equations:
 0.034341
 0.073542
 0.136509
 0.085185
 0.101084
-0.030324
 0.039094
-0.061062
 0.053872
-0.012614
-0.020925
-0.029875
 0.034187

Program paused. Press enter to continue.
Outlier detection matrix:
Program paused. Press enter to continue.
outliers computed from the normal equations:
 185.000000

outliers actually there:
 50.000000

Program paused. Press enter to continue.
false_positives_number computed:
 135.000000
false negatives computed:
 0.000000
miss rate computed:
 9.271978
hit_rate computed:
 90.728022
outliers detection accuracy computed:
 370.000000
```

Figure 3.2 - Test results for (vowels.mat) with Normal equations

Outlier graph prediction (Gradient descent method):

To begin with this approach, we will use the same data set for testing that we used for Normal Equations method, we will try to calculate the same theta parameters that we calculated for the normal approach.

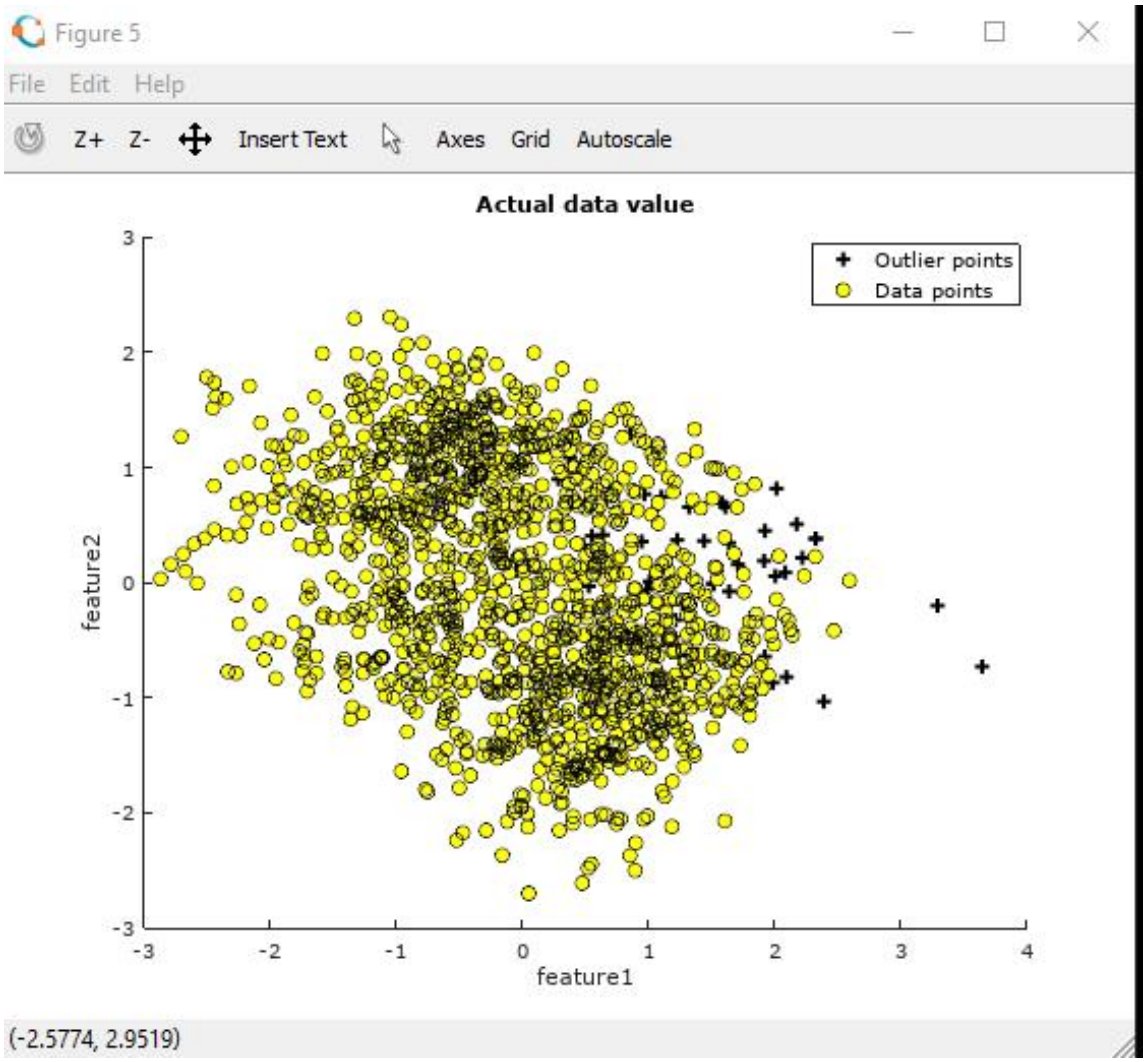


Figure 3.3 - Shows the various curves obtained from training samples

And thus, we take Gradient descent with multiple variables on polynomial regression model into account and hence compute the theta (θ) values for all the training samples in X (design matrix).

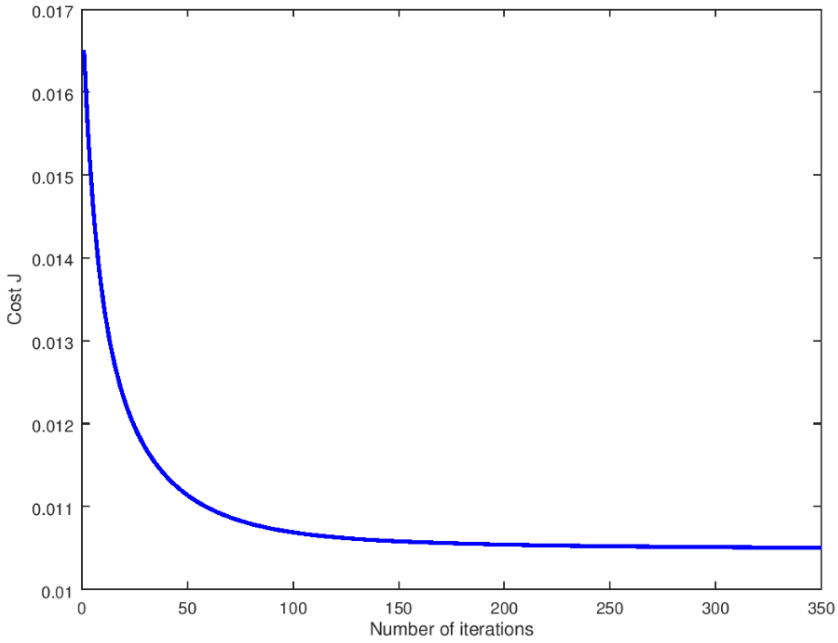


Figure 3.4-Shows the convergence of gradient descent for vowels data set

Similarly, we plot the predicted curve over the training samples and compute the accuracy

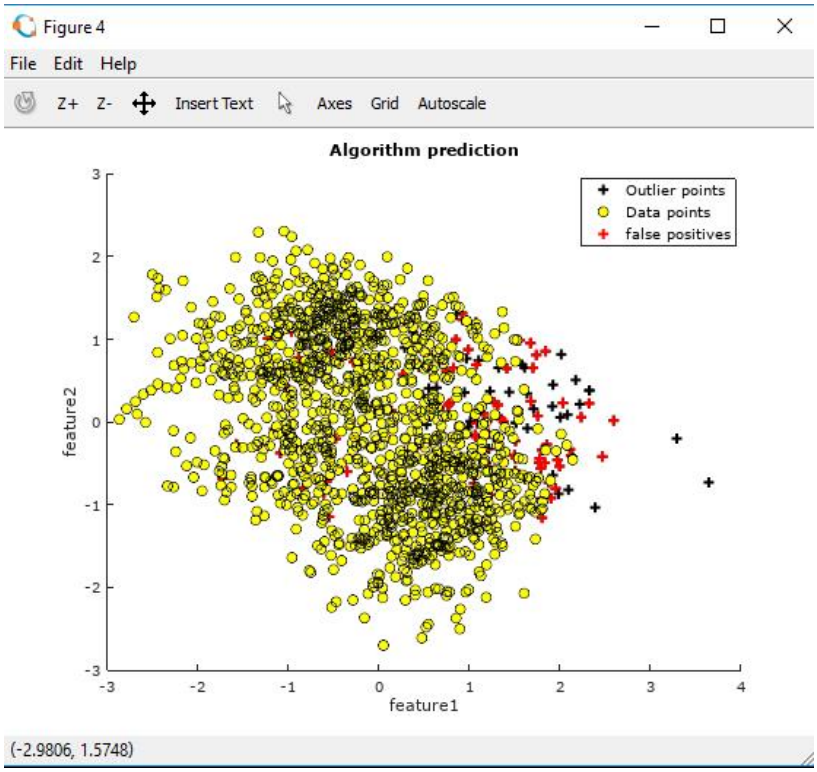


Figure 3.5 – Shows the comparison between experimental value and the prediction model

```
Running gradient descent ...
Program paused. Press enter to continue.
Theta computed from gradient descent:
 0.034341
 0.076608
 0.126243
 0.090889
 0.093730
 -0.019236
 0.035509
 -0.052427
 0.051571
 -0.000139
 -0.022963
 -0.025126
 0.032665

Program paused. Press enter to continue.
Outlier detection matrix:
outliers computed from the gradient descent:
 179.000000

outliers actually there:
 50.000000

false_positives_number computed:
 129.000000
false negatives computed:
 0.000000
miss rate computed:
 8.859890
hit_rate computed:
 91.140110
outliers detection accuracy computed:
 358.000000
```

Figure 3.6 - Test results for (vowels.mat) with Gradient Descent

Results: The hit rate of prediction was found to be nearly 91.140%

Note-The accuracy is subject to change for the data set on addition/removal of new values of alpha (0.1), iterations (350), Threshold (1.5 times the average) on various data sets**.**

Conclusion:

The given algorithm computed the values of theta and was able to successfully implement it with a working accuracy of 91.14% on various training samples. The Normal equation and the Gradient descent method had overshoot the result (135 false Positives)

Chapter 4

Classification model (Logistic regression)

To attempt classification, one method is to use linear regression and map all predictions greater than 0.5 as a 1 and all less than 0.5 as a 0. However, this method doesn't work well because classification is not actually a linear function.

The classification problem is just like the regression problem, except that the values we now want to predict take on only a small number of discrete values. For now, we will focus on the **binary classification problem** in which y can take on only two values, 0 and 1. (Most of what we say here will also generalize to the multiple-class case.) For instance, if we are trying to build a spam classifier for email, then $x(i)$ may be some features of a piece of email, and y may be 1 if it is a piece of spam mail, and 0 otherwise. Hence, $y \in \{0,1\}$. 0 is also called the negative class, and 1 the positive class, and they are sometimes also denoted by the symbols "-" and "+." Given $x(i)$, the corresponding $y(i)$ is also called the label for the training example.

Hypothesis Representation

We could approach the classification problem ignoring the fact that y is discrete-valued, and use our old linear regression algorithm to try to predict y given x. However, it is easy to construct examples where this method performs very poorly. Intuitively, it also doesn't make sense for $h\vartheta(x)$ to take values larger than 1 or smaller than 0 when we know that $y \in \{0, 1\}$. To fix this, let's change the form for our hypotheses $h\vartheta(x)$ to satisfy $0 \leq h\vartheta(x) \leq 1$. This is accomplished by plugging ϑTx into the Logistic Function.

Our new form uses the "Sigmoid Function," also called the "Logistic Function":

$$h_\theta(x) = g(\theta^T x)$$

$$z = \theta^T x$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

The following is the graph of logistic regression for (vowels.mat) just to understand what it looks like:

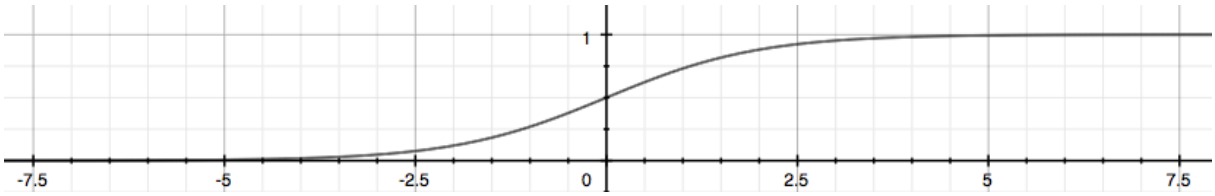


Figure 4.1- Sigmoid function

The function g(z), shown here, maps any real number to the (0, 1) interval, making it useful for transforming an arbitrary-valued function into a function better suited for classification.

hϑ(*x*) will give us the **probability** that our output is 1. For example, *hϑ*(*x*)=0.7 gives us a probability of 70% that our output is 1. Our probability that our prediction is 0 is just the complement of our probability that it is 1 (e.g. if probability that it is 1 is 70%, then the probability that it is 0 is 30%).

Cost Function & Optimization Algorithm

We cannot use the same cost function that we use for linear regression because the Logistic Function will cause the output to be wavy, causing many local optimas. In other words, it will not be a convex function Instead, our cost function for logistic regression looks like:

We can compress our cost function's two conditional cases into one case:

$$\text{Cost}(h_\theta(x), y) = -y \log(h_\theta(x)) - (1-y)\log(1-h_\theta(x))$$

But if we place (y=0 or y=1) one of the part of the expression gets nullified. To make this equation more general we represent this as :

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))]$$

And thus now our cost function has a convex shape so we can run our optimization algorithm, we will here be using fminunc (more advanced optimization algorithm) to calculate the optimum theta parameters.

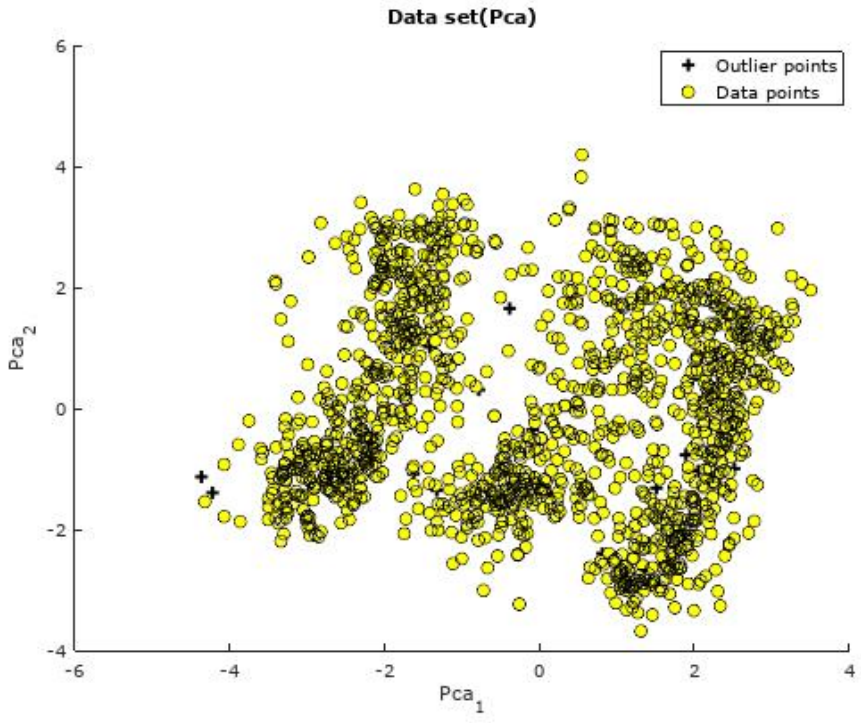


Figure 4.2 - PCA representation of the data on more contributing features

Testing Approaches -

1) ***Training and testing approach -*** We split our data set into 2 parts. The first part is called training set (contains randomly shuffled 70% of the data with atleast 70% of the outliers) and the second part is called testing data (contains randomly shuffled 30% of the data with atleast 30% of the outliers)

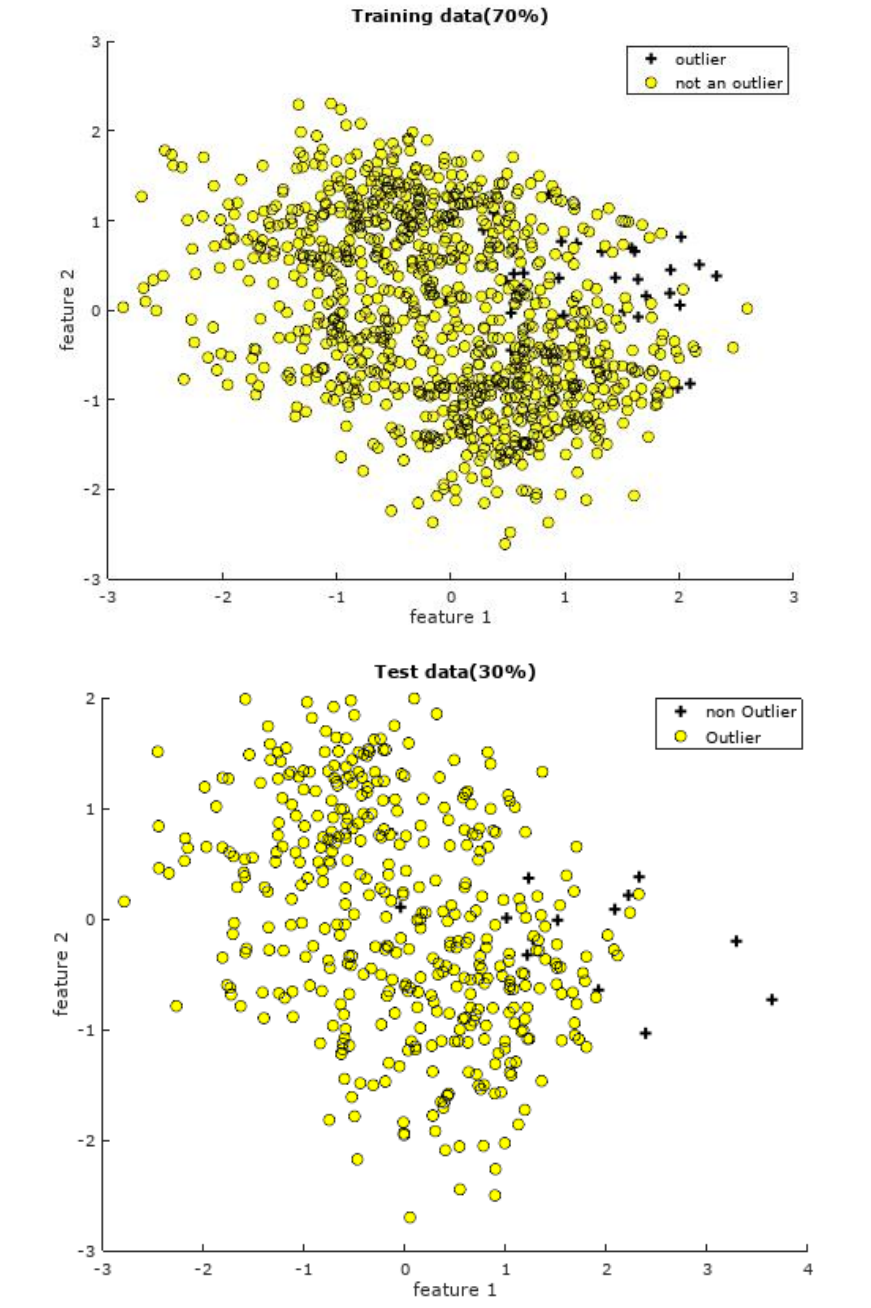
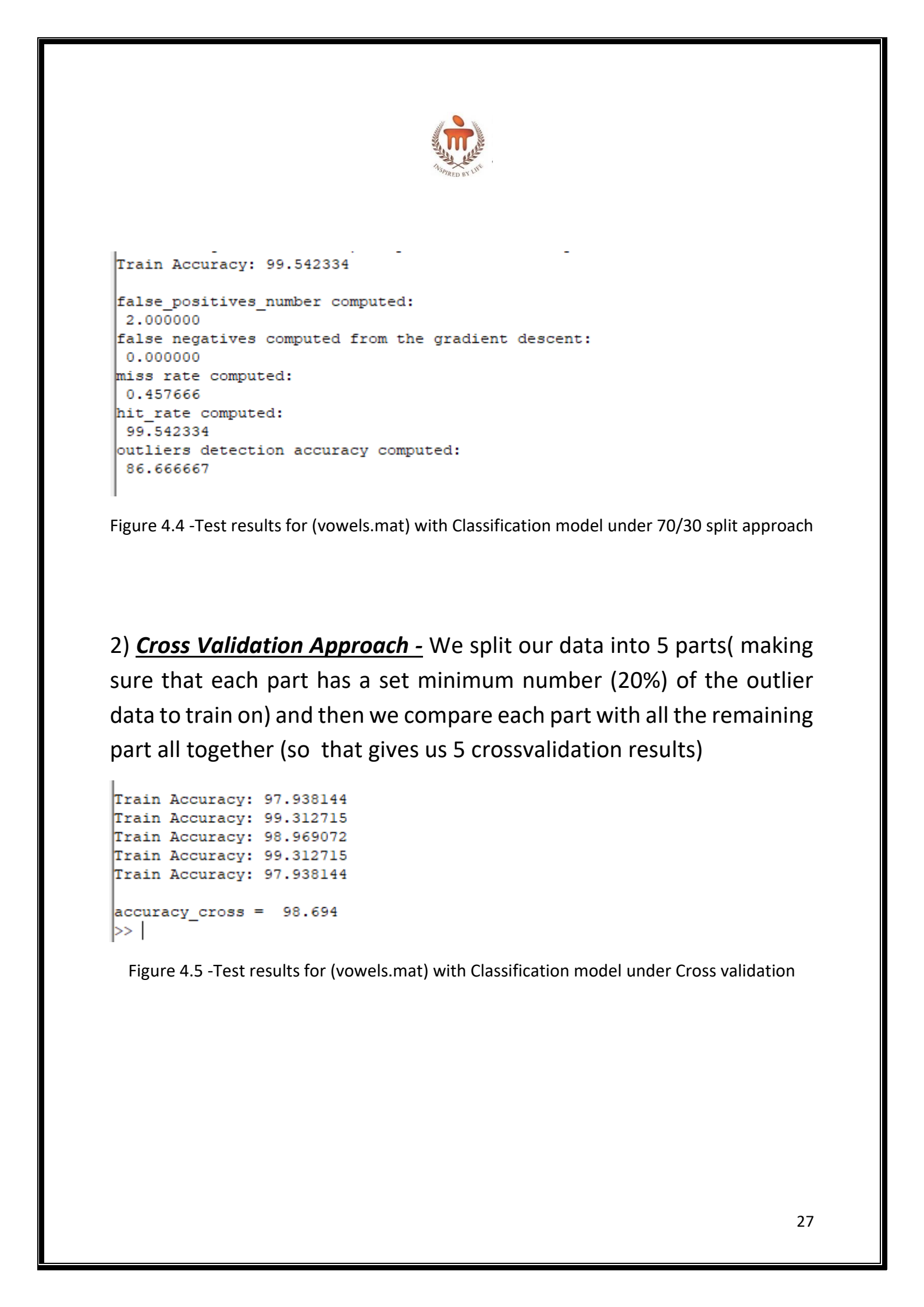


Figure 4.3 - Feature representation of the training & test data



```
Train Accuracy: 99.542334

false_positives_number computed:
 2.000000
false negatives computed from the gradient descent:
 0.000000
miss rate computed:
 0.457666
hit_rate computed:
 99.542334
outliers detection accuracy computed:
 86.666667
```

Figure 4.4 -Test results for (vowels.mat) with Classification model under 70/30 split approach

2) ***Cross Validation Approach -*** We split our data into 5 parts(making sure that each part has a set minimum number (20%) of the outlier data to train on) and then we compare each part with all the remaining part all together (so that gives us 5 crossvalidation results)

```
Train Accuracy: 97.938144
Train Accuracy: 99.312715
Train Accuracy: 98.969072
Train Accuracy: 99.312715
Train Accuracy: 97.938144

accuracy_cross =  98.694
>>
```

Figure 4.5 -Test results for (vowels.mat) with Classification model under Cross validation

Chapter 5

Neural Networks

Neural networks by far is a good approach to solve more complex hypothesis. It consist of three general types of layers-

1. Input layer – It's the first layer of the network to feed in the network with some input values (features or dimensions) our data has.
2. Hidden layers –consists of all the layers that come in between input layer and output layer (the more number of hidden layers you have the better is the network's prediction)
3. Output layer- consist of the number of nodes you want to classify for the various outliers in the given data. In the task of outlier detection, we will have two nodes (output can be {0,1})

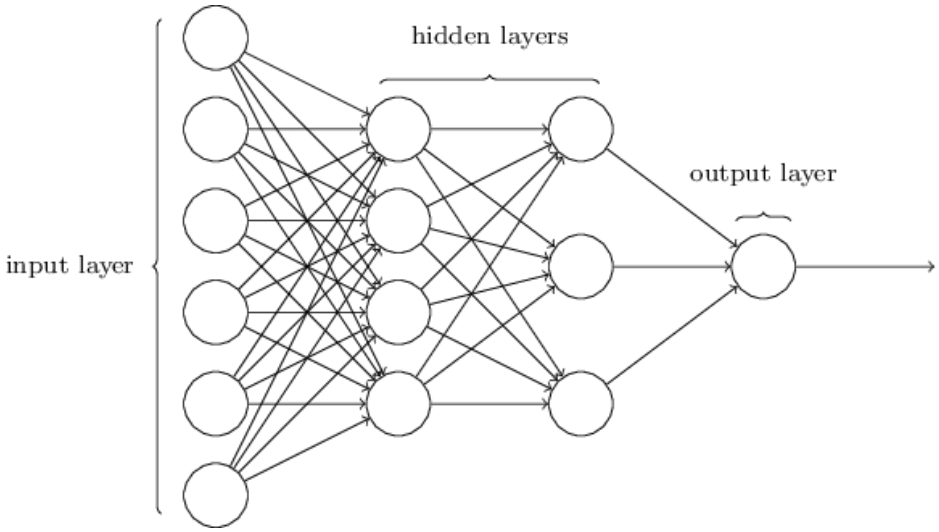


Figure 5.1 –shows a basic structure of a simple neural network

Every layer has some blocks to make the connection which are called as nodes and contain some value between 0 and 1 this is known as the activation function it's the same as the logistic function used in the classification approach.

For simplicity we will adopt this formula for calculation of nodes in the hidden layer [5] :

$$N_h = \frac{N_s}{(\alpha * (N_i + N_o))}$$

N_i = number of input neurons.
N_o = number of output neurons.
N_s = number of samples in training data set.
α = an arbitrary scaling factor usually 2-10.

Note- We have set the ideal value for alpha to be 5

After feeding the network with all the the input nodes equal to the number of dimensions that help in successful prediction of the task, we add one extra node to the layer know as the bias unit and it always outputs 1

Now before we move to the next layer we set up some weight parameters similar to theta parameters already used before in classification and regression approach but instead here we initialize these parameters as close to zero but not actually zero, once these parameters are initialized we will multiply the results with the weights of the layer and then again convert it back to some value between 0 and 1 (using the sigmoid function) before we actually feed forward to the next layer.

Upon reaching to the final layer (layer just before the output layer), We run the back propogation algorithm to find the error in our predictions starting from the last layer to the first hidden layer i.e. second layer in the network once we have the error calculated in the very first hidden layer we make an update in the theta parameter.

We train our weight parameter for every sample in the data (Repeat the same process every time) and at the end of all samples we will have set the optimum theta parameters (weights) by running back propogation for each algorithm and these weights are now tested on the test data. Here is a step by step walkthrough on implementing backpropogation algorithm:

Set $\Delta_{i,j}(l) := 0$ for all (l,i,j),

(hence you end up having a matrix full of zeros)

For training example t =1 to m:

1. Set $a(1):=x(t)$
2. Perform forward propagation to compute $a(l)$ for l=2,3,…,L

Gradient computation

Given one training example (x, y):

Forward propagation:

$$a^{(1)} = x$$
$$z^{(2)} = \Theta^{(1)} a^{(1)}$$
$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$
$$z^{(3)} = \Theta^{(2)} a^{(2)}$$
$$a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$
$$z^{(4)} = \Theta^{(3)} a^{(3)}$$
$$a^{(4)} = h_\Theta(x) = g(z^{(4)})$$

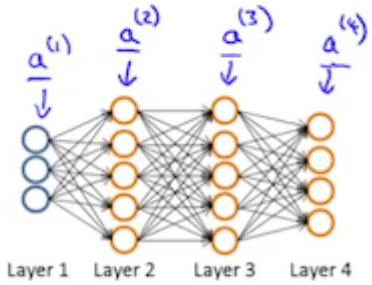


Figure 5.2 - shows the forward feeding of the network

3. Using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(t)}$

Where L is our total number of layers and $a(L)$ is the vector of outputs of the activation units for the last layer. So our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y. To get the delta values of the layers before the last layer, we can use an equation that steps us back from right to left.

4. Compute

$\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ using $\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) .* a^{(l)} .* (1 - a^{(l)})$

The delta values of layer are calculated by multiplying the delta values in the next layer with the theta matrix of layer l. We then element-wise multiply that with a function called g', or g-prime, which is the derivative of the activation function g evaluated with the input values given by *z*(*l*). The g-prime derivative terms can also be written out as:

$$g'(z^{(l)}) = a^{(l)} .* (1 - a^{(l)})$$

5. $\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ or with vectorization, $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

Hence we update our new Δ matrix.

- $D_{i,j}^{(l)} := \frac{1}{m} \left(\Delta_{i,j}^{(l)} + \lambda \Theta_{i,j}^{(l)} \right)$, if j≠0.
- $D_{i,j}^{(l)} := \frac{1}{m} \Delta_{i,j}^{(l)}$ If j=0

The capital-delta matrix D is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative. Thus we get :

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

Another thing to keep in mind is the regularization parameter (lambda) if it is too high it could lead to a very regularized decision boundry curve and which sometimes is not the best solution for very complex hypothesis curves, hence we will test the accuracy of lambda between (0 and 1) to see the best lambda value that fits well for the training data and implement it on the testing data.

The relation between the lambda and the accuracy:

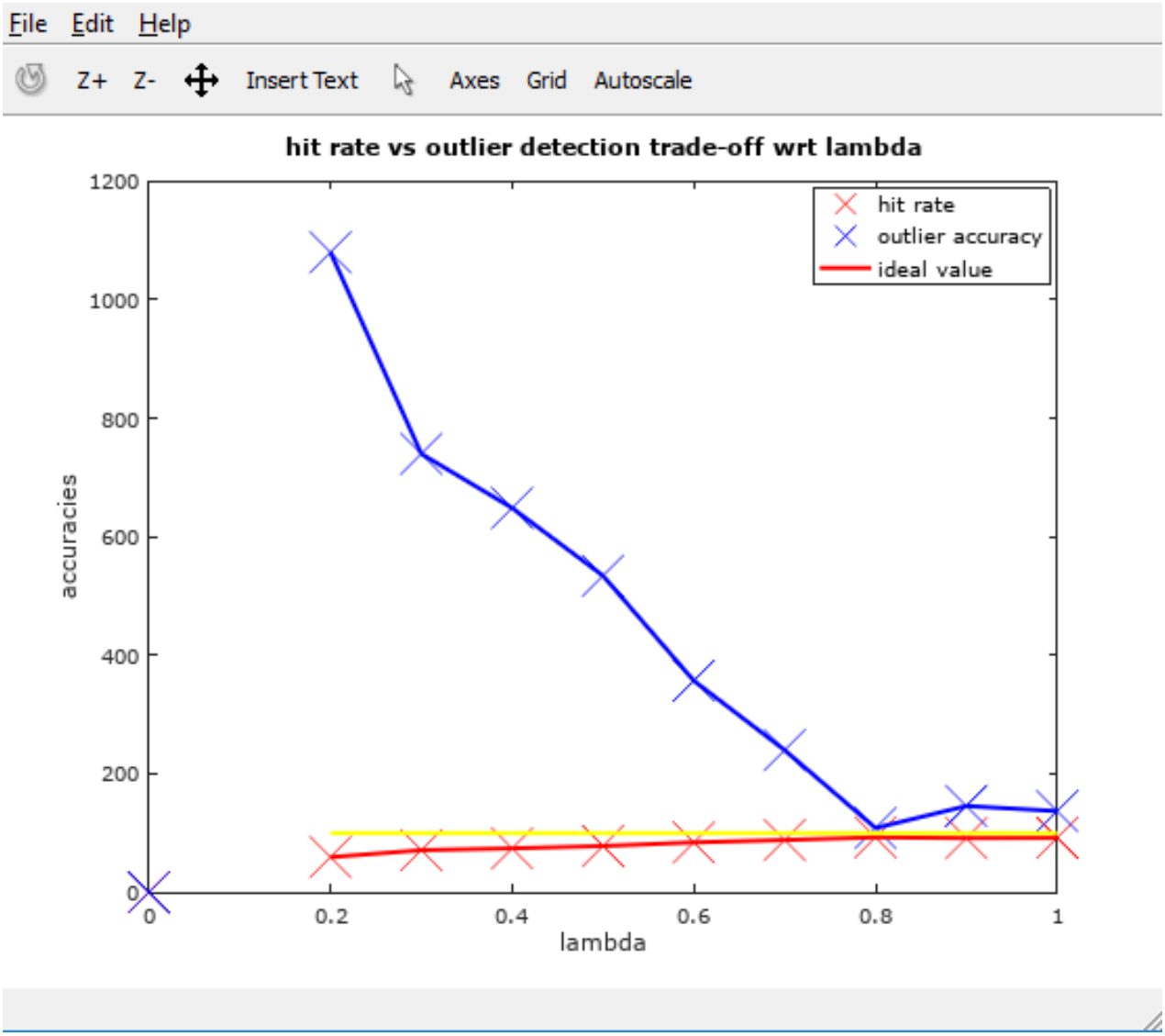


Figure 5.3 - show the relation between the lambda and accuracy on vowels.mat dataset

Note - here the outlier accuracy means the total number of outliers detected than there were actually in the data (in %) we see between 0.8 and 1.0 as the lambda this value approximates to maximum detection making sure the percentage of detection is not more than 100 and thus making the lambda ideal and not oveshooting the outliers detected in the data

```
ideal_lambda =  0.80000
results of the data(lambda hit_rate accuracy on train data ): 0.800000
results of the data(lambda hit_rate accuracy on train data ): 92.639843
results of the data(lambda hit_rate accuracy on train data ): 114.285714
Program paused. Press enter to continue.

Program paused. Press enter to continue.

Initializing Neural Network Parameters ...
Iteration   100 | Cost: 4.012263e-002
Program paused. Press enter to continue.
false_positives_number computed:
 31.000000
false negatives computed from the gradient descent:
 15.000000
miss rate computed:
 10.526316
hit_rate computed:
 89.473684
outliers detection accuracy computed:
 88.571429
>>
```

Figure 5.4 - shows the ideal value of lambda and the result obtained for vowels

The selection of lambda is based on a tradeoff between the hitrate and detection ratio where the most detection ratio from the various value of lamda has been given a higher preference over the hit rate (ideal value of lambda is that value where we have the closest detection ratio to 100 and the highest hit rate among all the test results of lamda)

Conclusion:

The given algorithm computed the values of weights (Theta parameters) and was able to successfully implement it with a working hit rate of 89.473% with outliers detection accuracy 88.57% with 31 (False positives) and 15 (False Negatives) with lambda value (0.8) on vowels.mat dataset.

Chapter 6

Self-Organized Map

Self-Organizing Map (SOM) was introduced as an unsupervised competitive learning algorithm of the artificial neural networks by Finnish Professor Teuvo Kohonen in the early 1980s. As such, SOM is also called the Kohonen map.

A SOM is made of a network of neurons that is usually one or two dimensions. At the beginning, the neurons will be initialized with random weight vectors that are of the same size as the dimensions (also called number of features or attributes) of the input sample. Input samples from the dataset will take turn to stimulate the neurons and cause their weight vectors to be tuned in such a way that the most stimulated (winner) neuron and its neighbors become more similar to the stimulating input sample. At the same time, the tuning effects for those neurons that neighbor the winner neuron gradually decrease inversely to their topological distances from the winner neuron. Gradually, this network of neurons will progress from an initially unordered map to a more stable topologically ordered map of clusters of neurons. Each cluster will contain those neurons that belong to a particular class of the input dataset [6].

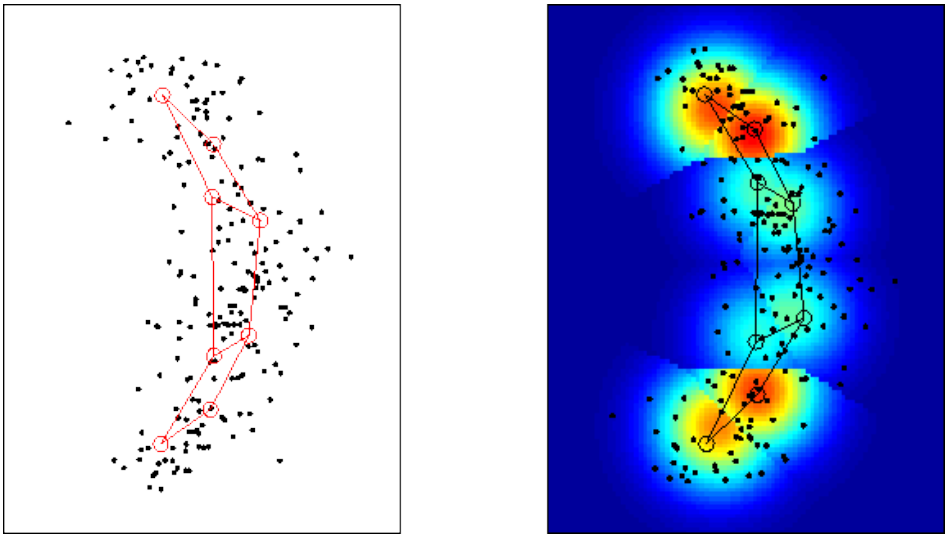


Figure 6.1 – explains a general representation of SOM algorithm

Overview for a SOM map to represent the distribution of a dataset, it has to go through an iterative training process using input samples from the dataset. The training process comprises the various phases as shown in figure below:

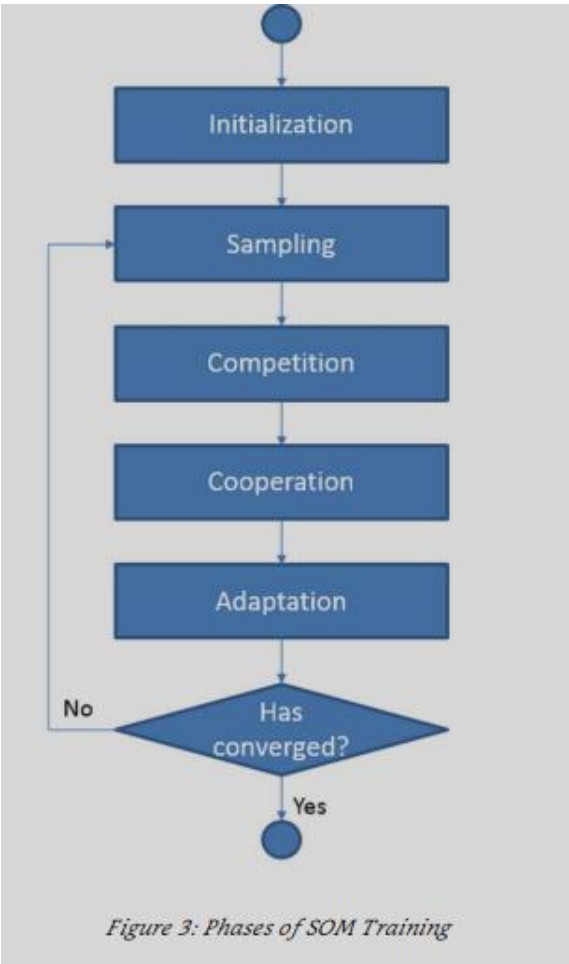


Figure 3: Phases of SOM Training

Figure 6.2 - explains the working architecture of SOM algorithm

1. **Initialization:**

At the beginning, all the neurons on the SOM map are initialized with random weight vectors of small values. Each neuron j in the map can be expressed as Wj = {wji : j = 1, ..., n; i = 1, ..., d} Where n is the total number of neurons and d is the number of dimensions of the input vector.

2. **Sampling:**

Once the map is initialized, it is set for an iterative process of automated self-organizational evolution. Sampling At each iteration, choose an input sample X from the training dataset. The input sample X is expressed as a vector, i.e. X = [x1, x2, ..., xd]. This will update the weights of the neuron with every sample.

3. **Competition:**

The chosen input sample X from the training dataset will begin to find its best-matching neuron from the neurons on the map. All the neurons on the map will compete to be the winner denoted as c(x) which is the most similar to the input sample X by: First, compute the Euclidean distances from the input sample to all the respective neurons on the map, i.e.

$$\|W_j - X\| = \sqrt{\sum (w_{ji} - x_i)^2}$$

Then, identifying the neuron that has the smallest Euclidean distance to be the winner, i.e.

$$c(x) = arg \min_j \|W_j - X\|$$

4. **Cooperation:**

In this phase, the winner of the competition will exert its influence over its neighboring neurons. In neurobiological studies, it is observed that a neuron that fires tends to excite the neurons in its immediate neighborhood more than those which are farther away from the winning neuron. In particular, the topological neighborhood is symmetric about the firing neuron (the winner neuron in the case of SOM), and the degree of excitement decreases monotonically with increasing lateral distance.

This phenomenon is best illustrated in Figure below:

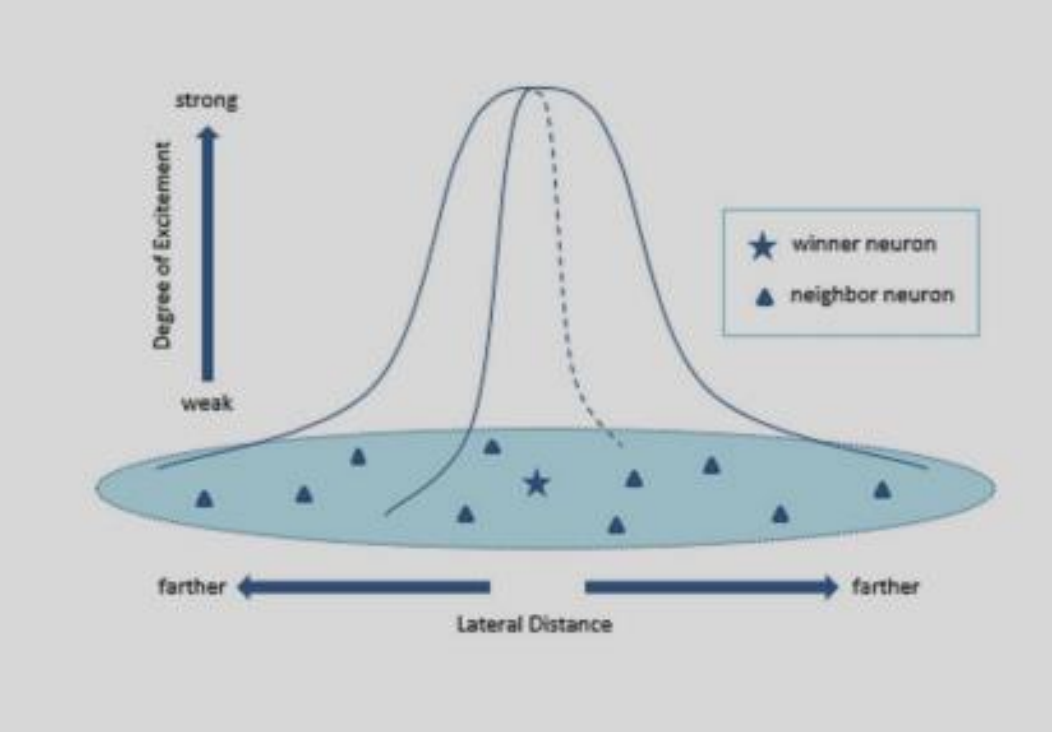


Figure 6.3 - Neighborhood Effect of Winner Neuron

It is the Gaussian function. As such, we can express this phenomenon in a Gaussian function as follows:

$$h_{j,c(x)} = exp(-\frac{d^2_{j,c(x)}}{2\sigma^2})$$

Where,

h j,c(x) is the neighborhood function, a measure of the influence (degree of excitement) of the winner neuron **c(x)** on neuron j.

d j,c(x) is the Euclidean distance from neuron j to the winner neuron **c(x)**.

σ is the standard deviation that affects the effective width of the topological neighborhood. When the neuron is **σ** away from the winner neuron, its effectiveness **h j,c(x)** is 0.61, as compared to 0.14 when it is 2**σ** away. Understandably, its effectiveness is at its maximum value of 1 at ground zero, where the winner neuron is situated. In this way, the neighborhood function can mimic the neurobiological phenomenon of lateral interaction within a set of excited neurons in the human brain.

In addition, SOM algorithm has a unique feature in which the size of the effective width shrinks with time. This can be achieved by introducing a time-varying component as follows:

$$\sigma(n) = \sigma_0 exp(-\frac{n}{\tau_1}) \quad , \quad n = 0, 1, 2, ... \quad , \quad \tau_1 = \frac{N}{\log \sigma_0}$$

Where, **σ0** is the initial effective width. The neighborhood function should initially include as many neurons as possible and gradually reduce the number over time. The initial effective width **σ0** can be set to the radius of the lattice. For example, if the size of the two dimensional lattice is **R x C**, then the initial effective width can be set as follows:

$$\sigma_0 = \sqrt{\frac{R^2 + C^2}{2}}$$

T1is the time constant.

N is the total number of iterations set for the SOM map.

The effect of this time-varying feature is illustrated in Figure below:

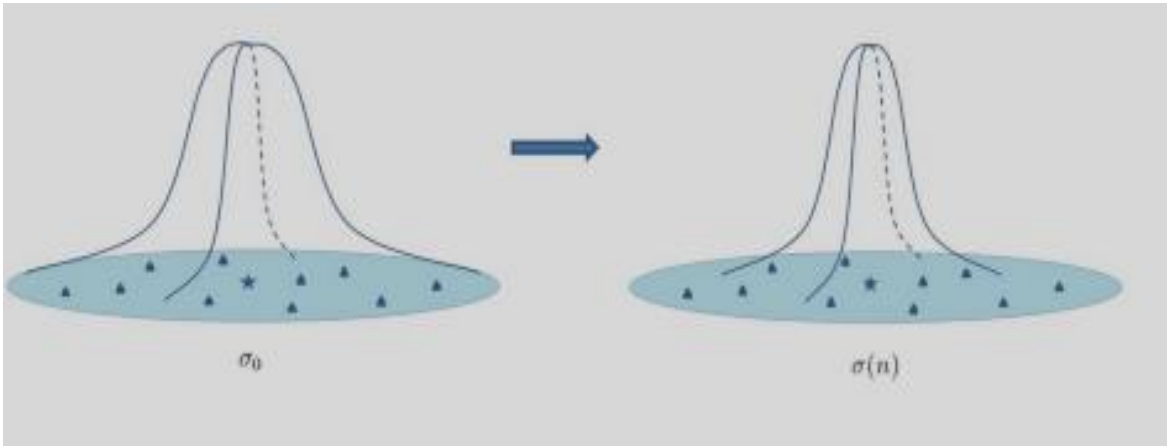


Figure 6.4 - Time-varying Neighborhood of a Winner Neuron

There is a shrink in the Gaussian curve with increase in time

This results in a time-varying neighborhood function as shown below:

$$h_{j,c(x)}(n) = exp(-\frac{d^2_{j,c(x)}}{2\sigma(n)^2})$$

In this way, the influence of the winner neuron on its neighbors decreases with time. This is a deliberate measure to prevent a particular neuron from monopolizing the process.

5. **Adaptation:**

Once the winner is found and its neighborhood functions are computed at each iteration, the topological change of the map will take place through updating of weights of its neurons. Specifically, the adaption phase is divided into two sub-phases – a **self-organizing phase** followed by a **convergence phase**.

Let's examine the weight updating process first before proceeding to the sub-phases for details. Generally, the weight vector of the winner **c(x)** is updated so that its new weight vector moves closer to the input vector. The weight vectors of all its neighbors are also updated in a similar manner, but the amount of update decreases inversely to the distance from the winner. Upon repeated feeding of the training data followed by updating of weight vectors of winners and their neighbors, the weight vectors of the neurons gradually follow the distribution of

the input vectors. This will eventually lead to a topological ordering of the map where neurons that are adjacent in the lattice tend to have similar weight vectors, a manifestation of **"Birds of a feather flock together"**. The weight updating equation is shown below:

$$W_j(n+1) = W_j(n) + \eta(n) h_{j,c(x)}(n)(X - W_j(n))$$

Where **h j,c(x) (n)** is the neighborhood function between neuron j and the winner neuron **c(x)** at nth iteration.

N(n) is the time-varying learning rate at nth iteration. The learning rate serves to moderate the learning step of each iteration. It will decrease with time so as to facilitate the convergence of the map.

The learning rate at nth iteration is computed as follows:

$$\eta(n) = \eta_0 \, exp(-\frac{n}{\tau_2}) \quad , \quad n = 0, 1, 2, ..., N$$

Where,

N0 is the initial learning rate.

T2 is a time constant which is set to the total number of iterations.

A) Self-organizing Phase:

The self-organizing phase may take as many as 20000 iterations, i.e. N = 20000.

Problem with too high/low learning rate:

The learning rate should begin with a value of close to 0.1, i.e. = 0.1, and gradually decreases, but remains above 0.025. Too wide a learning rate at the start will cause the learning to swing widely from one iteration to the next, virtually un-doing the previous learning, thus rendering it ineffective. On the contrary, too small a rate will unnecessarily prolong the convergence time. A properly selected learning rate should allow bigger updates at the earlier learning stages, followed by gradually slower updates nearing the end, and eventually settling down to a stable state.

At the end of the first phase, the winner can only affect its immediate neighbors.

B) Convergence Phase: This phase is introduced to fine tune the quality of the ordered map derived from the first phase of adaptation with the aim to provide a more accurate statistical quantification of the input dataset. As a general rule, the number of iterations in this phase must be at least 500 times the number of neurons available on the map. In this phase, the learning rate should be maintained at a small constant value like 0.01, and the neighborhood function should contain only the immediate neighbors of the winner neuron, which may eventually reduce to zero. However, the convergence phase may not be needed in applications where convergence of the parameters is not critical.

Getting the map to work:

Set the parameters for subsequent training of SOM as follows:

1. The size of the SOM map is 20x20.

2. The total iteration **N** is set at 20000 for the self-organizing phase.

3. The neighborhood function is:

$$h_{j,c(x)}(n) = exp(-\frac{d^2_{j,c(x)}}{2\sigma(n)^2})$$

Where **d j,c(x)** is the Euclidean distance from neuron j to the winner neuron **c(x)**.

σ(n) is the effective width of the topological neighborhood at nth iteration and it is computed as follows:

$$\sigma(n) = \sigma_0 exp(-\frac{n}{\tau_1}) \quad , \quad n = 0, 1, 2, ...$$

Where **σ0** is the initial effective width which is set to 15, i.e. the radius of the 10x10 map.

T1 is the time constant.

4. The weight updating equation is:

$$W_j(n+1) = W_j(n) + \eta(n) h_{j,c(x)}(n)(X - W_j(n))$$

Where N(n) is the time-varying learning rate at nth iteration and is computed as follows:

$$\eta(n) = \eta_0 \, exp(-\frac{n}{\tau_2}) \quad , \quad n = 0, 1, 2, ..., N$$

Where,

N0 is the initial learning rate and is set at 1.

T1 is a time constant which is set to the value of N.

Once the parameters are set, it is ready to test the results on datasets [6 Leow, Peter. Self-Organizing Map Demystified: Unravel the Myths and Power of SOM in Machine Learning Peter Leow. Kindle Edition.].

[Download the Som reference code used for the project and documentation- here]

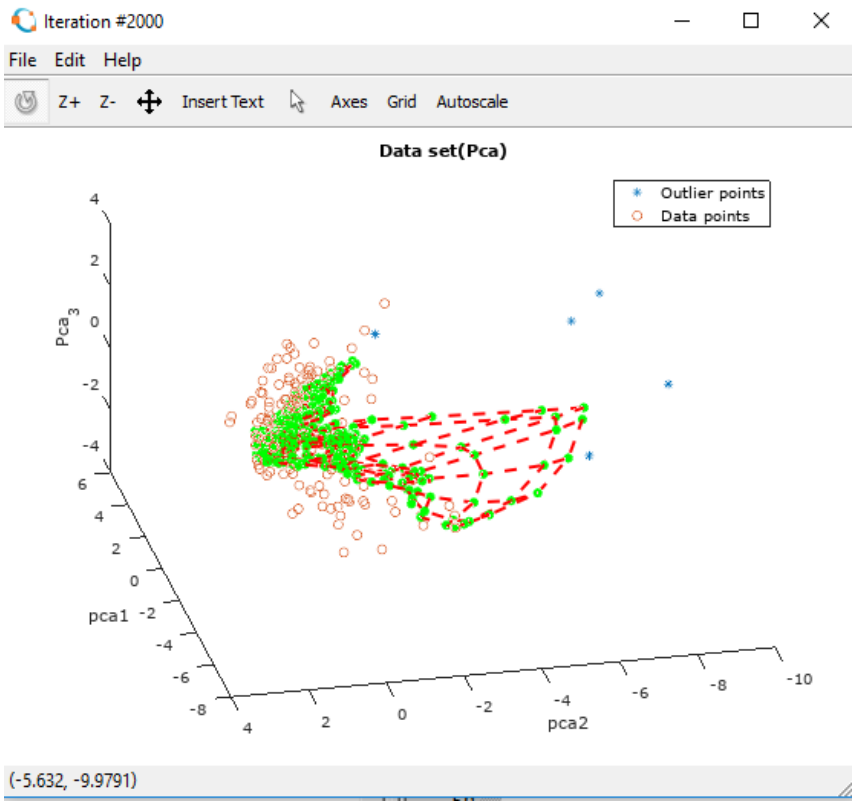


Figure 6.4 - shows the som result after only 2000 iterations

Conclusion:

The given algorithm computed the values of weights (Theta parameters) and was able to successfully implement it with a working hit rate of 100% with 0 (False positives) and 0 (False Negatives) on lympho.mat dataset.

<u>Visualization of Som (test with more iterations on other datasets):</u>

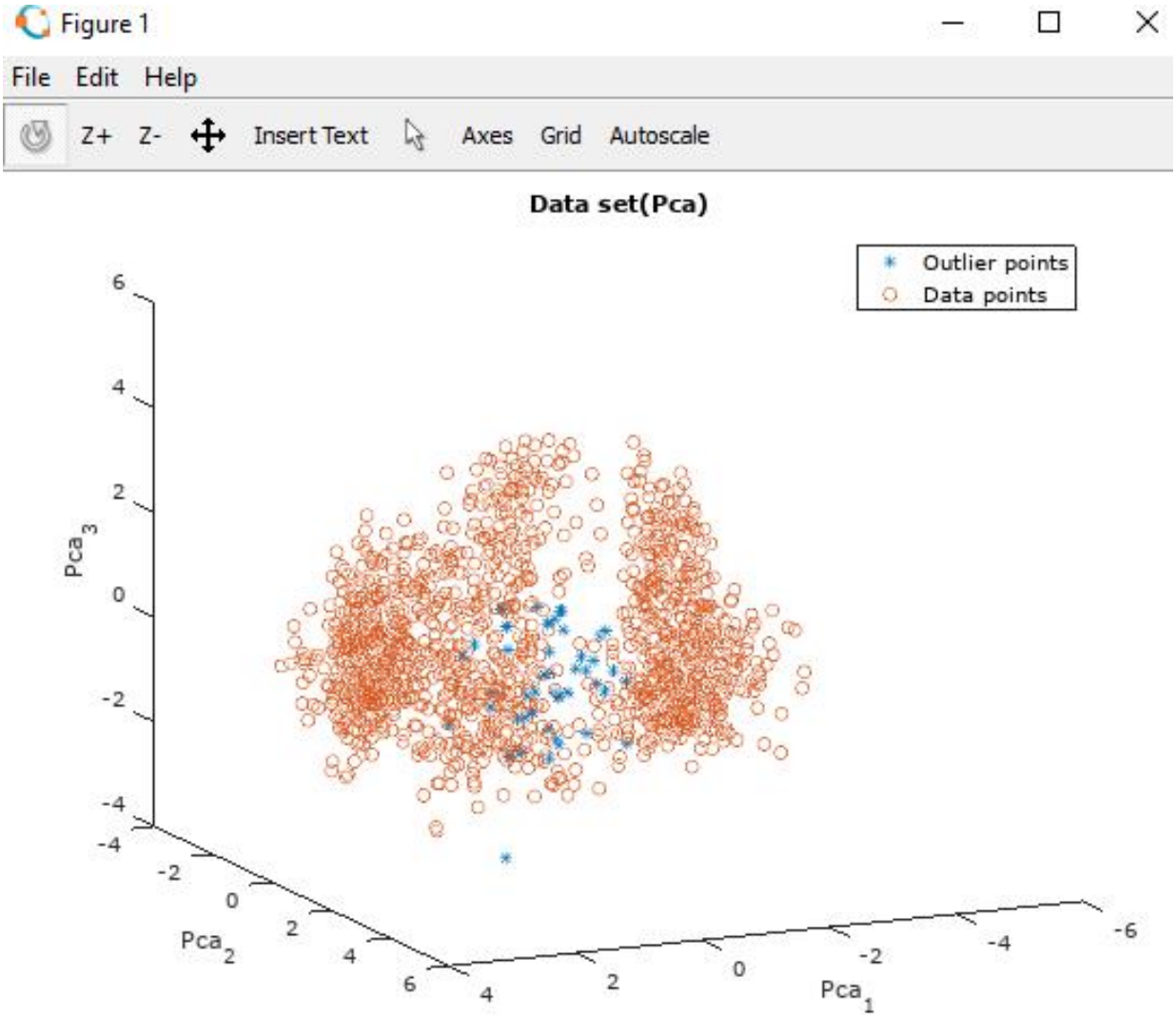


Figure 6.5 – shows the vowels data set in 3 dimensional plot along its most dominant PCA features (before training)

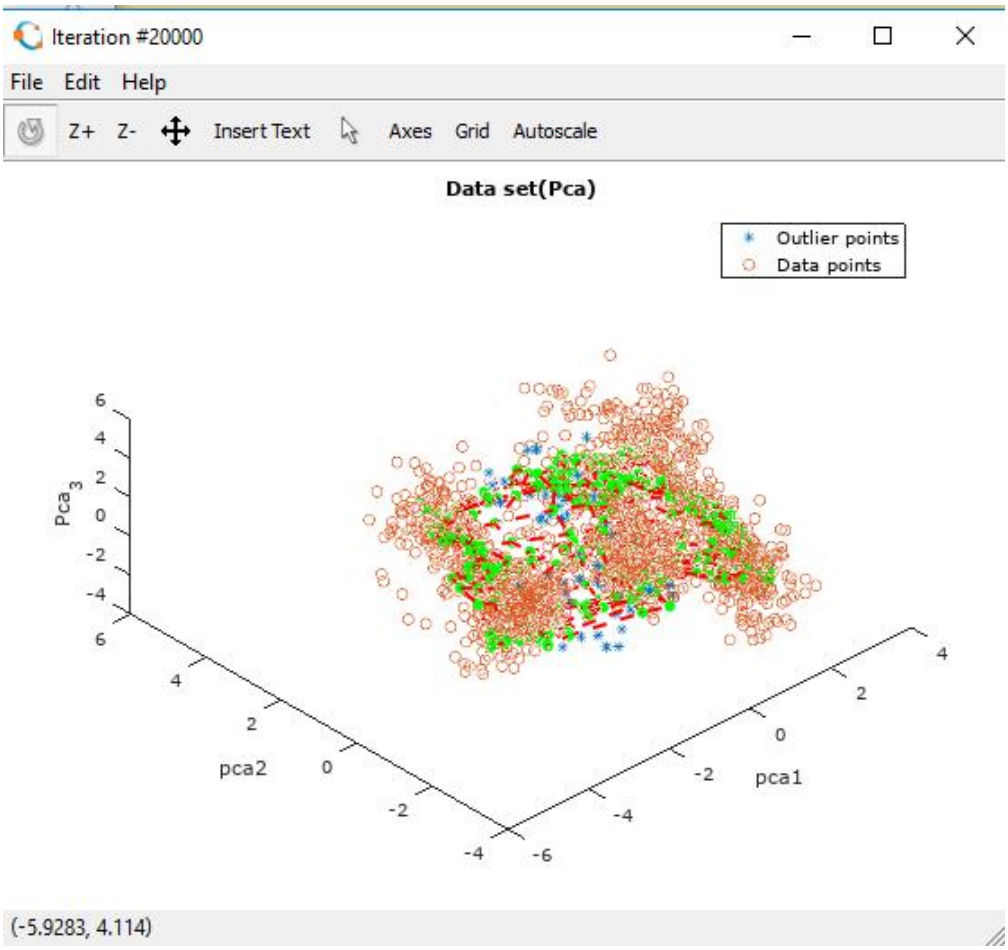


Figure 6.6 - shows the vowels data set in 3 dimensional plot along its most dominant PCA features (after 20000 iterations)

Visualizing 3D dataset across PCA (Principle component analysis):

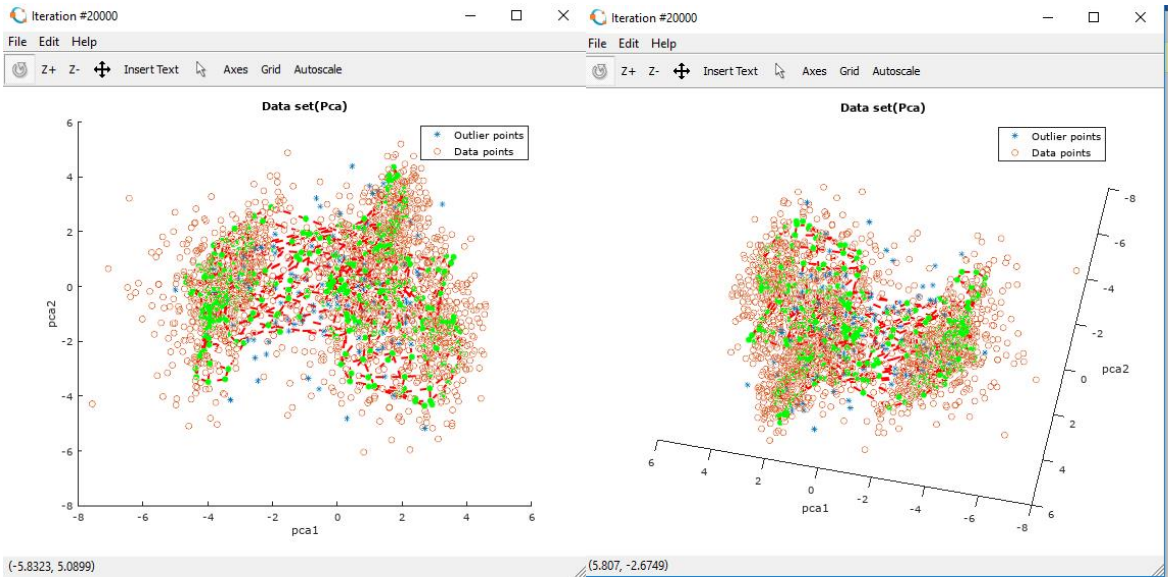


Figure 6.6 - shows the Visualizing a 3D dataset across its PCA (Principle component analysis)

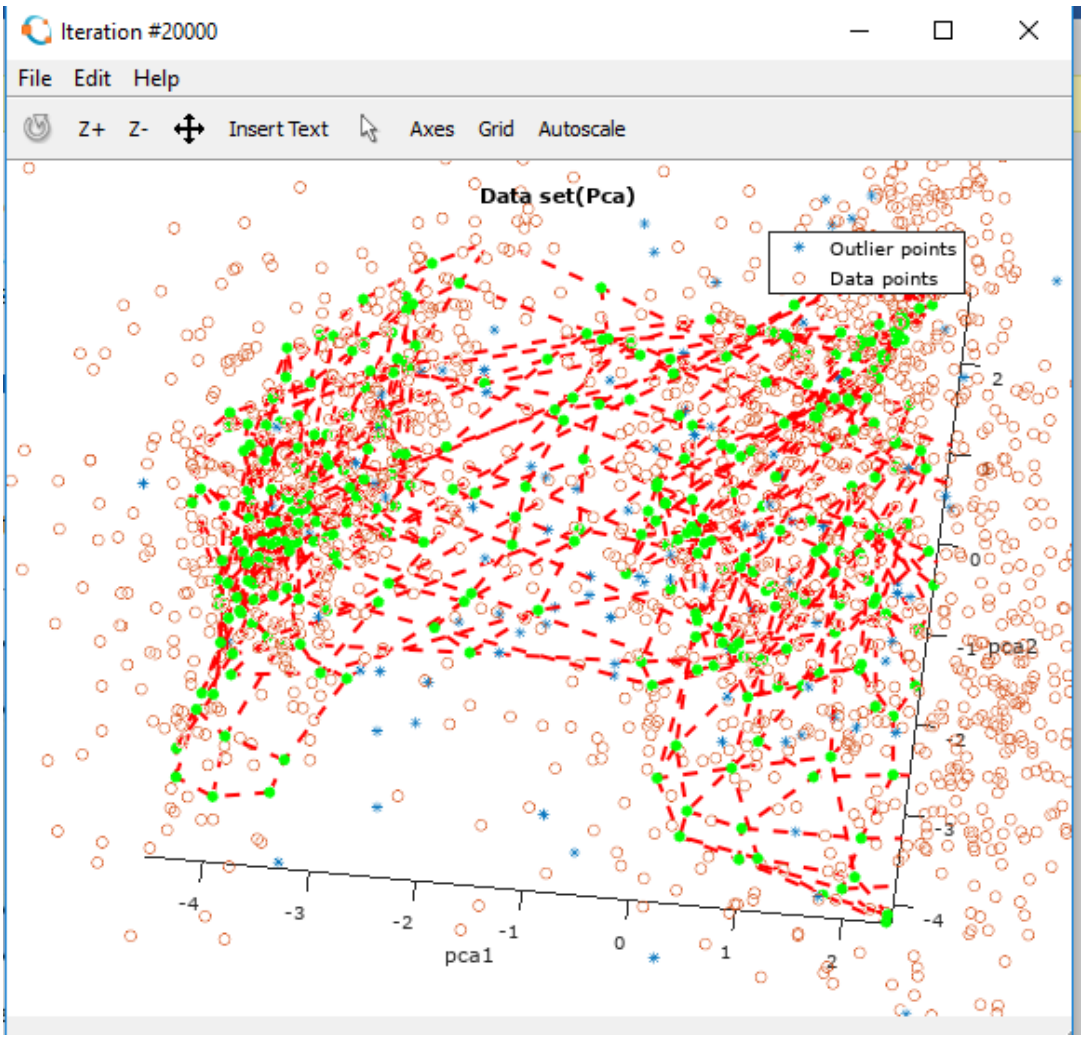


Figure 6.7 – Som map taking the shape of the letter data after 20000 iterations (in 3D)

Chapter 7

Testing and Comparison of result graphs and references

1) Regression:

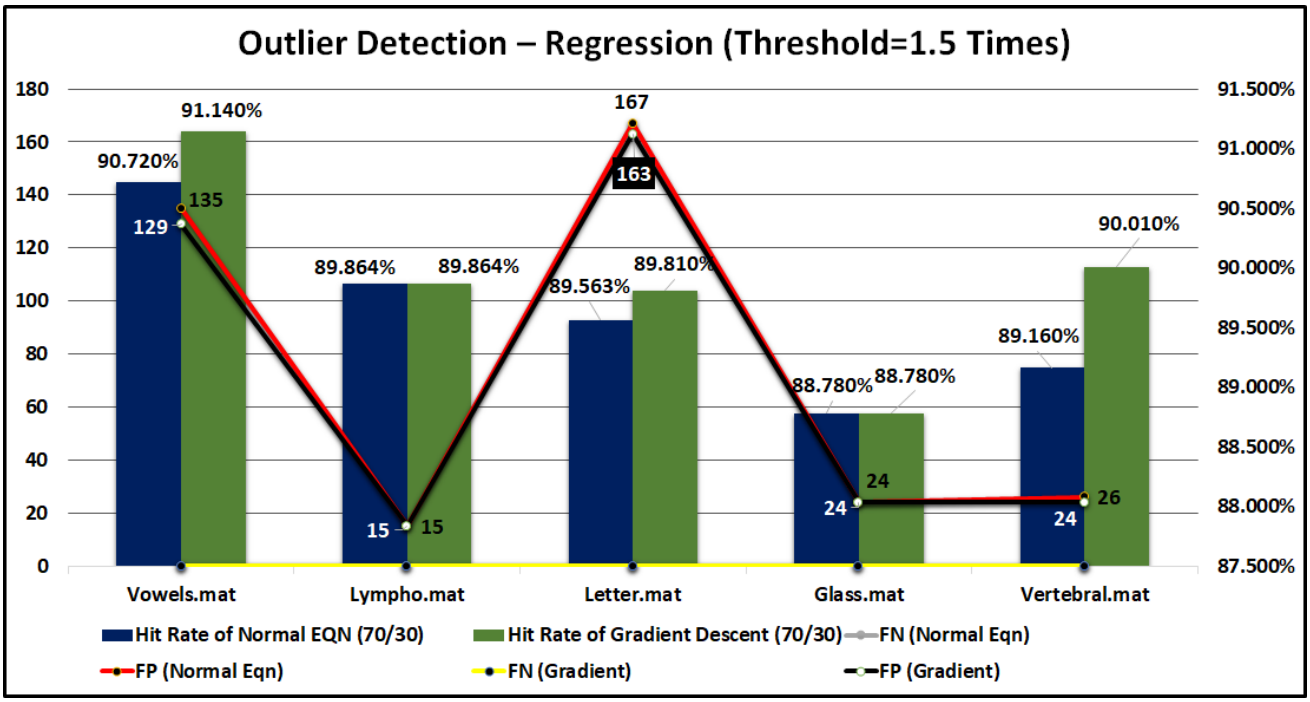


Figure 7.1 – Denotes results of regression on various data sets

2) Classification (without regularization):

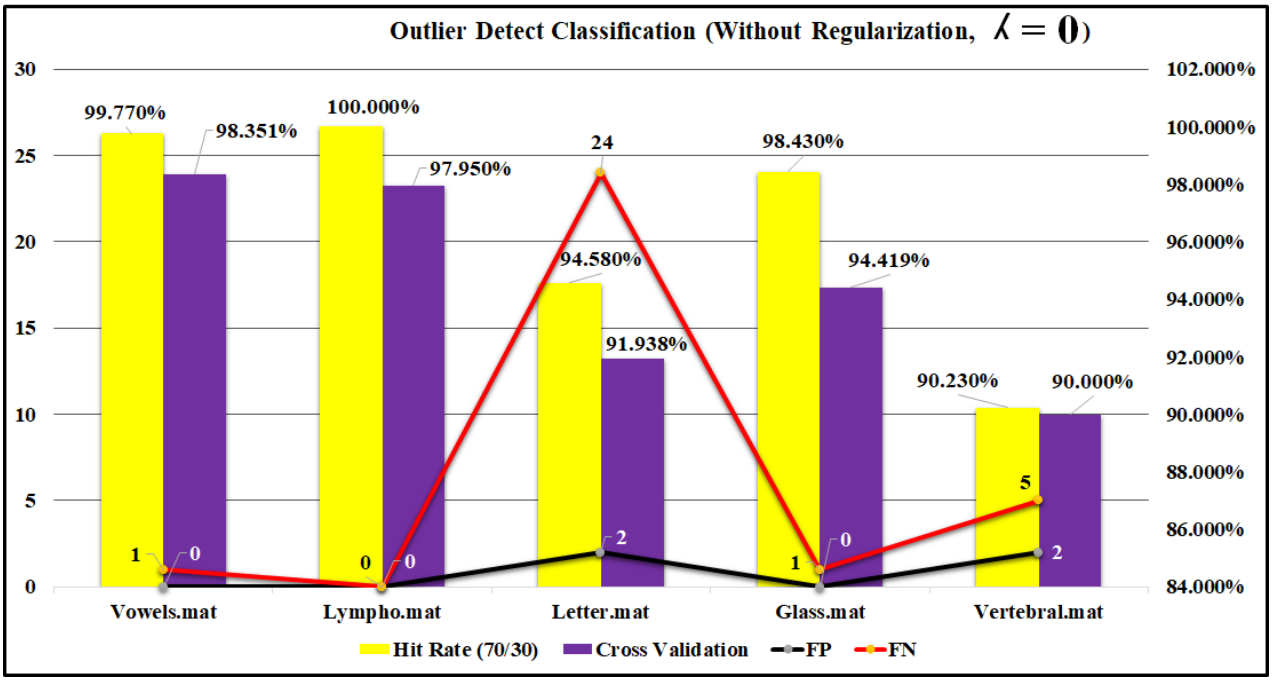


Figure 7.2 – Denotes Classification without regularisation

3) Classification (with regularization):

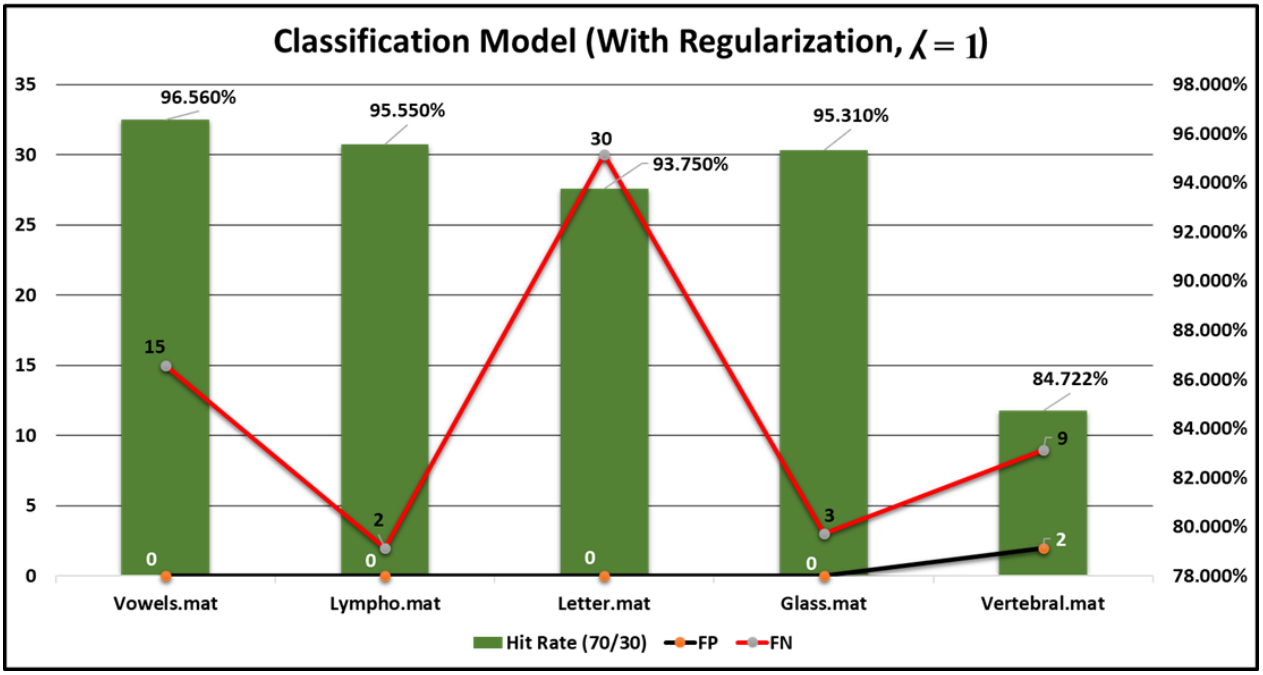


Figure 7.3 – Denotes Classification with regularisation

4) Neural Networks (ideal lambda parameters):

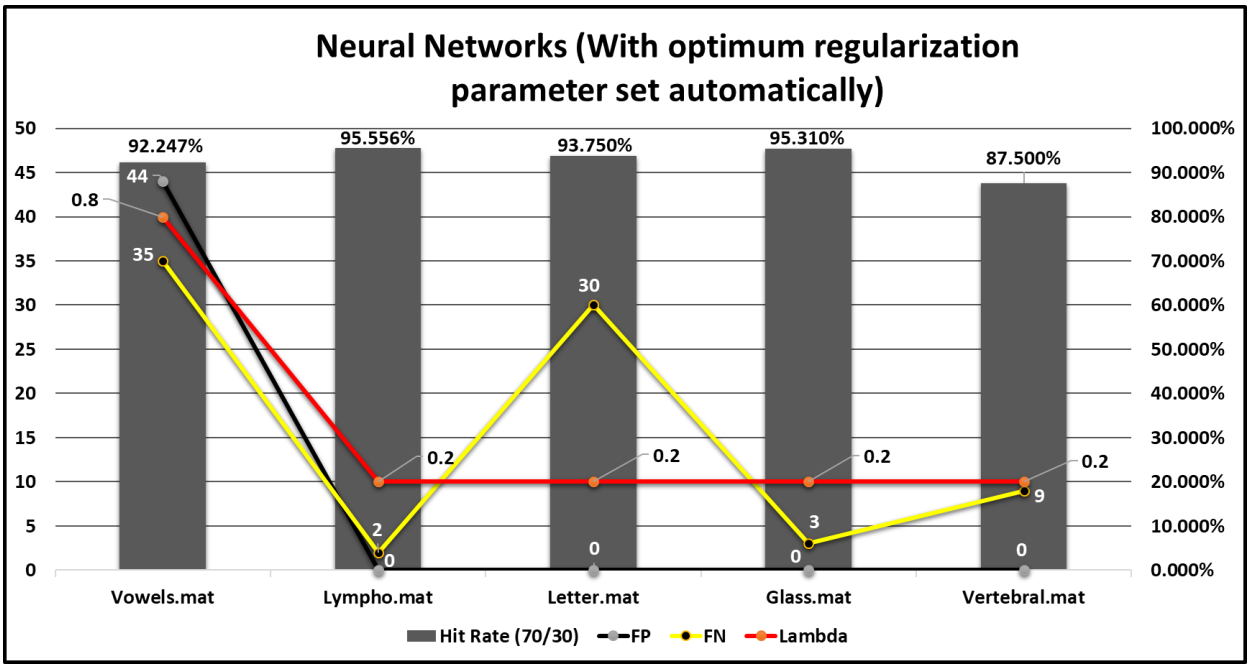


Figure 7.4 – Denotes results of Neural Networks

5) Self Organized Maps (Iterations = 20000):

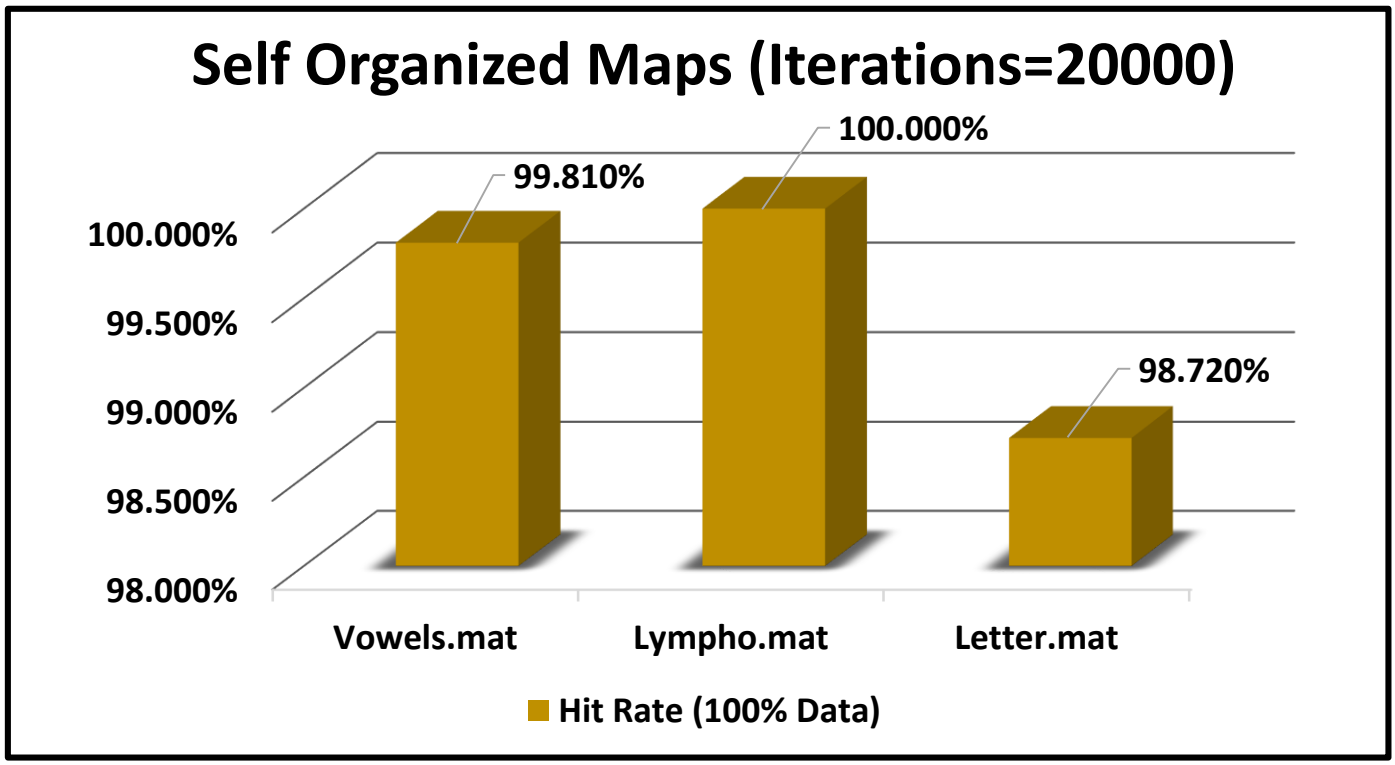


Figure 7.5 – Denotes results of Self Organised Maps

REFERENCES:

1. **Data sets for testing -** http://odds.cs.stonybrook.edu
2. **Coursera course referred for the writing material and algorithm implementation –** https://fr.coursera.org/learn/machine-learning
3. **Applications of outlier detection -** http://eprints.whiterose.ac.uk/767/1/hodgevj4.pdf
4. **Intrusion detection-** https://www.researchgate.net/profile/James_Cannady/publication/239064800_Multiple_Self_Organizing_Maps_for_Intrusion_Detection/links/5706400408aea3d28020cea8.pdf
5. **Reference node formula-** https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-network
6. **Self-organizing map content (book by peter leow)-** https://peterleowblog.com/self-organizing-map-demystified/
7. **Github repository used for reference** - https://github.com/zhuoyuzhu/Self-Organizing-Map
8. **Required software(Octave):** https://www.gnu.org/software/octave/download.html
9. **Test code files:** https://www.dropbox.com/s/60su3340tgub773/machine-learning%20-outlier%20detection.rar?dl=0

**** End of Report ****